University of Waterloo
Department of Computer Science
Waterloo, Ontario N2L 3G1

June 17, 1988

INVOICE

Professor Michael Schwartz
University of Colorado at Boulder
Dept. of Computer Science
ECOT7-7 Engineering Center
Campus Box 430
Boulder, Colorado
80309-0430

REPORT(S) ORDERED CS-87-53 and CS-87-55

TOTAL COST $5.00

recd payment Cheque #1106 JUL 5 1988

Would you please make your cheque or international bank draft payable to the **Computer Science Department, University of Waterloo** and forward to my attention.

Thanking you in advance.

Yours truly,

Susan DeAngelis

Susan DeAngelis (Mrs.)
Research Report Secretary
Computer Science Dept.

/sd

Encl.

University of Colorado at Boulder

Department of Computer Science

ECOT 7-7 Engineering Center
Campus Box 430
Boulder, Colorado 80309-0430
(303) 492-3902

June 13, 1988

Attention Technical Reports
Department of Computer Science
University of Waterloo
Waterloo, Ontario N2L 3G1
Canada

I would like to obtain copies of the following papers:

1. W. H. Cheung, J. P. Black and E. G. Manning. A Study of Distributed Debugging. Research report CS-87-53, University of Waterloo. Department of Computer Science., 1987.
2. Y. Igarashi and D. Wood. Roughly Sorting: A Generalization of Sorting. Research report CS-87-55, University of Waterloo. Department of Computer Science., 1987.

If you could send me copies of these papers, I would appreciate it.

Sincerely,

Michael F. Schwartz
Assistant Professor

Your Invoice Reference	Our Reference Number	Purchase Order	Invoice Amount	Discount	Amount Paid
C273171 XRESREPOR	02-320-07		2.00		2.00

sent MAR 2 1988

THE UNIVERSITY OF CALGARY

NEW TELEPHONE # (403) 220-5707

Please Detach Before Presenting For Payment

Total Amount	2.00

273171.

Department of Computer Science
University of Waterloo
Research Reports 1987 (September to December) continued

Report No.	Title	Author	Cost
CS-87-49	Networks for Education at the University of Waterloo	D.D. Cowan S.L. Fenton J.W. Graham T.M. Stepien	2.00
CS-87-50	A Network Operating System For Interconnected LANS With Heterogeneous Data-Link Layers	D.D. Cowan T.M. Stepien R.G. Veitch	2.00
CS-87-51	Project ARIES A Network for Convenient Computing in Education	D.D. Cowan T.M. Stepien	2.00
CS-87-52	Naming of Objects in the Cluster System	F.C.M. Lau J.P. Black E.G. Manning	2.00
CS-87-53	A Study of Distributed Debugging	W.H. Cheung J.P. Black E.G. Manning	2.00
CS-87-54	Defaults and Conjectures: Hypothetical Reasoning for Explanation and Prediction	D.L. Poole	2.00
CS-87-55	Roughly Sorting: A Generalization of Sorting	Y. Igarashi D. Wood	2.00
CS-87-56	A Dynamic Fixed Windowing Problem	R. Klein O. Nurmi T. Ottmann D. Wood	2.00
CS-87-57	Explorations in Restricted Orientation Geometry	G.J.E. Rawlins	5.00
CS-87-58	A New Measure of Presortedness	V. Estivill-Castro D. Wood	2.00
CS-87-59	A Logical Framework for Default Reasoning	D.L. Poole	2.00
CS-87-60	On Consistent Equiarea Triangulations	R.B. Simpson	2.00

CS-87-61	Amalgamating Functional and Relational Programming through the Use of Equality Axioms	K. Yukawa	5.00
CS-87-62	Comparison of the Single Phase and Two Phase Numerical Model Formulation For Saturated-Unsaturated Ground Water Flow (in preparation)	P. Forsyth	2.00
CS-87-63	On the Existence of Speed-Independent Circuits	C-J. Seger	2.00
CS-87-64	Logic-based Program Transformation (in preparation)	M.H.M. Cheng M.H. van Emden P.A. Strooper	?
CS-87-65	An Algebra for Nested Relations	V. Deshpande P.-A. Larson	2.00
CS-87-66	On The Average Case of String Matching Algorithms	R. Baeza-Yates	2.00
CS-87-67	Perceptual Reasoning: A Logical Foundation for Computer Vision	J.D.Denis Gagne	2.00
CS-87-68	Searching with Uncertainty	R. Baeza-Yates J.C. Culberson G.J.E. Rawlins	2.00

If you would like to order any reports please forward your order, along with a cheque or money order payable to the **Department of Computer Science, University of Waterloo, Waterloo, Ont. N2L 3G1 to the Research Report Secretary.**

Please indicate your current mailing address.

SOFTWARE RESEARCH AND DEVELOPMENT GROUP
COMPUTER SCIENCE DEPT.
UNIVERSITY OF CALGARY
2500 UNIVERSITY DR. N.W.
CALGRY, ALBERTA
T2N 1N4

Research Report Secretary
Dept. of Computer Science
University of Waterloo
Waterloo, Ont. N2L 3G1

Please ~~send me~~ the following research report:

CS-87-53 A study of Distributed Debugging
by W. H. Cheung, J. P. Black, and E. G. Manning.

My address is:

Eyad Saad
5200 Anthony Wayne 1016
Detroit, MI 48202 USA

Thank you in advance.

Saad Eyad

Acct

Printing Requisition / Graphic Services

89464

1. Please complete unshaded areas on form as applicable.
2. Distribute copies as follows: White and Yellow to Graphic Services. Retain Pink Copies for your records.
3. On completion of order the Yellow copy will be returned with the printed material.
4. Please direct enquiries, quoting requisition number and account number, to extension 3451.

TITLE OR DESCRIPTION: CS-87-53

DATE REQUISITIONED: Nov. 7/8 | DATE REQUIRED: ASAP | ACCOUNT NO.: 1 1 2 6 6 0 1 2 1 4 1

REQUISITIONER - PRINT: T. BLACK | PHONE: 4459 | SIGNING AUTHORITY: T. Black.

MAILING INFO - NAME: Sue DeIngelis | DEPT.: C.S. | BLDG. & ROOM NO.: M+C 6081E | ☒ DELIVER ☐ PICK-UP

Copyright: I hereby agree to assume all responsibility and liability for any infringement of copyrights and/or patent rights which may arise from the processing of, and reproduction of, any of the materials herein requested. I further agree to indemnify and hold blameless the University of Waterloo from any liability which may arise from said processing or reproducing. I also acknowledge that materials processed as a result of this requisition are for educational use only.

NUMBER OF PAGES: 74 | NUMBER OF COPIES: 25

TYPE OF PAPER STOCK: ☒ BOND ☐ NCR ____ PT. ☒ COVER ☐ BRISTOL ☒ SUPPLIED ☐ ____

PAPER SIZE: ☒ 8½ x 11 ☐ 8½ x 14 ☐ 11 x 17 ☐ ____

PAPER COLOUR: ☒ WHITE ☐ ____ | INK: ☒ BLACK ☐ ____

PRINTING: ☐ 1 SIDE ___ PGS. ☒ 2 SIDES ___ PGS. | NUMBERING: FROM ____ TO ____

BINDING/FINISHING: ☒ COLLATING ☒ STAPLING ☐ ____ HOLE PUNCHED ☐ PLASTIC RING

FOLDING/PADDING | CUTTING SIZE

Special Instructions

Math fonts + tracks enclosed.

COPY CENTRE — OPER. NO. | BLDG. | MACH. NO.

DESIGN & PASTE-UP — OPER. NO. | TIME | LABOUR CODE

			D 0 1
			D 0 1
			D 0 1

TYPESETTING — QUANTITY

P A P 0 0 0 0 0				T 0 1
P A P 0 0 0 0 0				T 0 1
P A P 0 0 0 0 0				T 0 1

PROOF

P R F				
P R F				
P R F				

NEGATIVES — QUANTITY | OPER. NO. | TIME | LABOUR CODE

F L M				C 0 1
F L M				C 0 1
F L M				C 0 1
F L M				C 0 1
F L M				C 0 1

PMT

P M T				C 0 1
P M T				C 0 1
P M T				C 0 1

PLATES

P L T				P 0 1
P L T				P 0 1
P L T				P 0 1

STOCK

			0 0 1
			0 0 1
			0 0 1
			0 0 1

BINDERY

R N G				B 0 1
R N G				B 0 1
R N G				B 0 1
M I S 0 0 0 0 0 0				B 0 1

OUTSIDE SERVICES

$ ____ COST

TAXES — PROVINCIAL ☐ FEDERAL ☐

GRAPHIC SERV. OCT. 85 462-2

Printing Requisition / Graphic Services

1257

1. **Please complete unshaded areas** on form as applicable.
2. **Distribute copies** as follows: White and Yellow to Graphic Services. **Retain Pink Copies for your records.**
3. On completion of order the **Yellow copy will be returned** with the printed material.
4. **Please direct enquiries, quoting requisition number and account number**, to **extension 3451.**

TITLE OR DESCRIPTION: CS-87-53

DATE REQUISITIONED: Dec. 14/87 | DATE REQUIRED: ASAP | ACCOUNT NO.: 1 1 2 6 1 0 1 2 4 1

REQUISITIONER – PRINT: J. BLACK | PHONE: 2192 | SIGNING AUTHORITY: J. Black.

MAILING INFO – NAME: S. DeANGELIS | DEPT.: C.S. | BLDG. & ROOM NO.: M+C 6081E | ☑ DELIVER ☐ PICK-UP

Copyright: I hereby agree to assume all responsibility and liability for any infringement of copyrights and/or patent rights which may arise from the processing of, and reproduction of, any of the materials herein requested. I further agree to indemnify and hold blameless the University of Waterloo from any liability which may arise from said processing or reproducing. I also acknowledge that materials processed as a result of this requisition are for educational use only.

NUMBER OF PAGES: 71 | NUMBER OF COPIES: 25

TYPE OF PAPER STOCK: ☒ BOND ☐ NCR ____ PT. ☒ COVER ☐ BRISTOL ☒ SUPPLIED ☐ ____

PAPER SIZE: ☒ 8½ x 11 ☐ 8½ x 14 ☐ 11 x 17 ☐ ____

PAPER COLOUR: ☒ WHITE ☐ ____ | INK: ☐ BLACK ☐ ____

PRINTING: ☐ 1 SIDE ____ PGS. ☒ 2 SIDES ____ PGS. | NUMBERING: FROM ____ TO ____

BINDING/FINISHING: 3 down left side
☒ COLLATING ☒ STAPLING ☐ ____ HOLE PUNCHED ☐ PLASTIC RING

FOLDING/PADDING | CUTTING SIZE

Special Instructions

With fronts + backs enclosed.

COPY CENTRE — OPER. NO. | BLDG. | MACH. NO.

DESIGN & PASTE-UP — OPER. NO. | TIME | LABOUR CODE

		D01
		D01
		D01

TYPESETTING — QUANTITY

PAP000000				T01
PAP000000				T01
PAP000000				T01

PROOF

PRF				
PRF				
PRF				

NEGATIVES — QUANTITY | OPER. NO. | TIME | LABOUR CODE

FLM				C01
FLM				C01
FLM				C01
FLM				C01
FLM				C01

PMT

PMT				C01
PMT				C01
PMT				C01

PLATES

PLT				P01
PLT				P01
PLT				P01

STOCK

				001
				001
				001
				001

BINDERY

RNG				B01
RNG				B01
RNG				B01
MIS000000				B01

OUTSIDE SERVICES

$ ____ COST

TAXES – PROVINCIAL ☐ FEDERAL ☐ GRAPHIC SERV. OCT. 85 482-2

Printing Requisition/Graphic Services

86827

1. Please complete unshaded areas on form as applicable.
2. Distribute copies as follows: White and Yellow to Graphic Services. Retain Pink Copies for your records.
3. On completion of order the Yellow copy will be returned with the printed material.
4. Please direct enquiries, quoting requisition number and account number, to extension 3451.

TITLE OR DESCRIPTION: CS - 97-53

DATE REQUISITIONED: Oct 1/97 | DATE REQUIRED: ASAP | ACCOUNT NO.: 1216601241

REQUISITIONER – PRINT: S. DeIngelis | PHONE: 3112/6463 | SIGNING AUTHORITY: Tom Hancock

MAILING INFO – NAME: S. DeINGELIS | DEPT.: CS | BLDG. & ROOM NO.: MC 6081E | ☑ DELIVER ☐ PICK-UP

Copyright: I hereby agree to assume all responsibility and liability for any infringement of copyrights and/or patent rights which may arise from the processing of, and reproduction of, any of the materials herein requested. I further agree to indemnify and hold blameless the University of Waterloo from any liability which may arise from said processing or reproducing. I also acknowledge that materials processed as a result of this requisition are for educational use only.

NUMBER OF PAGES: 000 71 | NUMBER OF COPIES: 5

TYPE OF PAPER STOCK: ☑ BOND ☐ NCR ____ PT. ☒ COVER ☐ BRISTOL ☒ SUPPLIED ☐ ____

PAPER SIZE: ☑ 8½ x 11 ☐ 8½ x 14 ☐ 11 x 17 ☐ ____

PAPER COLOUR: ☑ WHITE ☐ ____ | INK: ☐ BLACK ☐ ____

PRINTING: ☐ 1 SIDE ____ PGS. ☑ 2 SIDES ____ PGS. | NUMBERING: FROM ____ TO ____

BINDING/FINISHING: ☑ COLLATING ☒ STAPLING ☐ ____ HOLE PUNCHED ☐ PLASTIC RING

FOLDING/PADDING | CUTTING SIZE

Special Instructions

With cover + back cover attached

COPY CENTRE — OPER. NO. | BLDG. | MACH. NO.

DESIGN & PASTE-UP — OPER. NO. | TIME | LABOUR CODE

			D01
			D01
			D01

TYPESETTING — QUANTITY

PAP000000				T01
PAP000000				T01
PAP000000				T01

PROOF

PRF				
PRF				
PRF				

NEGATIVES — QUANTITY | OPER. NO. | TIME | LABOUR CODE

FLM				C01
FLM				C01
FLM				C01
FLM				C01
FLM				C01

PMT

PMT				C01
PMT				C01
PMT				C01

PLATES

PLT				P01
PLT				P01
PLT				P01

STOCK

				001
				001
				001
				001

BINDERY

RNG				B01
RNG				B01
RNG				B01
MIS000000				B01

OUTSIDE SERVICES

$____ COST

TAXES – PROVINCIAL ☐ FEDERAL ☐ GRAPHIC SERV. OCT. 85 492-2

A Study of Distributed Debugging

W. H. Cheung
Computer Studies Department
University of Waterloo

James P. Black
Department of Computer Science
University of Waterloo

Eric G. Manning
Faculty of Engineering
University of Victoria

Research Report CS-87-53
October 1987

A Study of Distributed Debugging

W.H. Cheung J.P. Black

Department of Computer Science
University of Waterloo

E.G. Manning

Faculty of Engineering
University of Victoria

Abstract

No software system is perfect; errors and unanticipated behaviour always occur during the lifetime of a software system. Debugging is the process of detecting, locating and correcting such errors. Debugging and development are often more complicated for distributed programs than for sequential ones. The study of distributed debugging attempts to develop helpful techniques, methodologies and approaches for tackling the debugging process in a distributed environment.

This paper provides an overview of current research in distributed debugging. We first look at the debugging process and discuss the issues in distributed debugging. Then, a simple model which provides a general framework for distributed debugging systems is presented. The model defines the role of a distributed debugging system and its relationship with its external environment, consisting of the program domain, the distributed domain and the human domain. Based on this model, we classify the research problems into three areas: the distributed debugging model, domain interaction and system support. Also, we discuss some solutions to these problems; the discussion is at three levels: basic debugging techniques, distributed debugging methodologies and general approaches. Finally, we suggest some research directions.

Contents

1 Introduction

Debugging is an essential step in developing a software system, since every non-trivial software system is imperfect; bugs always occur. However, no precise or elegant methodology has been developed for the debugging process; debugging is generally referred to as an art rather than a science. The introduction of a distributed environment makes the situation even worse, since this both complicates the debugging process and generates new types of bugs. In recent years, researchers have developed some helpful debugging techniques for concurrent or distributed environments. However, most papers on the debugging of distributed software discuss specific debuggers, or a particular class of bug. Few published papers look at the issues and solutions in a global sense, considering, for example, how debugging techniques apply to a general environment and how these techniques can be combined to form a useful debugging system. In this paper, we try to provide a global picture of current research in distributed debugging. It is not intended to be an exhaustive survey of the area; rather, we give our view of the issues and solutions based on a proposed framework for distributed debugging systems.

In Section 2, we give an overview of debugging. An understanding of classical debugging issues for sequential programming is essential for our discussion of distributed debugging, since we may consider the latter to be an extension of the former into more general multi-processing and multi-processor environments. As Smith stated [Smith85], some of the techniques used in multi-process debugging are similar to those provided by sequential program debuggers. For example, a user with multiple processes needs to monitor messages and their contents just as a user with single program needs to monitor procedure calls, the arguments of the calls and their results. Other techniques are unique to the multi-processing domain. For example, a user can specify the order in which suspended processes are to be resumed. When we move to a distributed environment which has multiple processors, no common time reference, and unpredictable communication delays, the situation is even worse.

Section 3 discusses issues in distributed debugging. We first look at typical errors occurring in a distributed system and then study the problems encountered in debugging distributed programs. The motivation for developing a distributed debugging system (DDS) is also discussed.

In Section 4, we introduce a simple DDS model. The model defines the role of a DDS and its relationship with its external environment, consisting of the program domain, the distributed domain and the human domain. Based on this model, we organize research problems into three areas: the distributed debugging model, domain interaction and system support. Of course, these areas are not totally independent; such a division is basically for convenience of discussion. While we consider solutions in all three areas, our emphasis is on the distributed debugging model. We arrange the solutions into three levels: basic debugging techniques, debugging methodologies and general approaches. Basic techniques refer to individual facilities that support distributed debugging. Methodologies refer to procedures for performing the debugging process. General approaches provide high level abstract models for constructing a complete debugging system. These three levels of solutions form a hierarchy, from basic techniques to abstract models, to deal with various problems in distributed debugging.

In Section 5, we present some basic techniques for distributed debugging. Although most of them have been used extensively to debug sequential programs, their use in distributed programs involves further considerations of effectiveness, implementation difficulties, and semantics. We provide a study of these techniques in the context of a distributed environment.

Section 6 presents some methodologies which have been used by researchers to debug distributed programs.

In Section 7, we present and discuss three general approaches to developing a DDS. These approaches are the database approach, the behavioural approach and the artificial intelligence (AI) approach. Some examples are given to illustrate their characteristics.

Section 8 looks at two aspects of domain interaction: graphics and integration. The study of

graphics emphasizes the use of computer graphics to support the human-computer interface for debugging. Integration attempts to integrate software development tools in order to provide a single, consistent programming environment. These two aspects have significant impact on the development of distributed debugging systems.

Finally, in Section 9, we summarize our discussion and discuss some future research directions.

2 Software Debugging

2.1 Failure, Error and Fault

At the program level, program *failure* is defined as a deviation of execution behaviour from that dictated by the program specification. Such a deviation results from an erroneous program state. An *error* is a part of an erroneous state which constitutes a difference from a valid program state. The cause of the invalid state transition which first establishes an erroneous state is referred to as a *fault*. Therefore, the manifestation of a fault produces errors in the program state which will lead to a failure. We refer to an error and its corresponding fault(s) as a *bug*.

The notions of failure, error, and fault are described in detail in [Anderson81] and [Laprie85]. These terms are usually used in the study of fault tolerance. However, their scope in debugging is different from their scope in fault tolerance; the difference can easily be appreciated with the help of Figure 1.

Normal behaviour N_e is expected correct behaviour. *Fault tolerant behaviour* is the incorrect behaviour which is handled properly by some fault tolerant facility in the program (*e.g.*, exception handler, recovery block). So, when execution yields fault tolerant behaviour, the program can recover from that incorrect behaviour and direct the execution back to some normal behaviour. However, not all causes of fault tolerant behaviour are known by the programmer; an

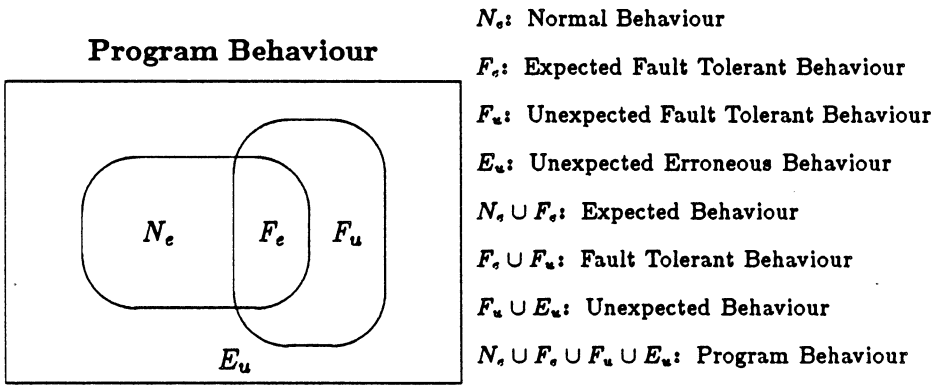

Figure 1: The Classification of Program Behaviour

error may result from different faults. Therefore, we refer to fault tolerant behaviour, the causes of which are known, as *expected fault tolerant behaviour* F_e, and to fault tolerant behaviour, the causes of which are unknown, as *unexpected fault tolerant behaviour* F_u. *Unexpected erroneous behaviour* E_u is the behaviour other than normal and fault tolerant behaviour. Since both normal and expected fault tolerant behaviour are expected by the programmer, we call them *expected behaviour*. Behaviour which is not expected is called *unexpected behaviour*.

In fault tolerance, failure is referred to as any behaviour which deviates from normal behaviour; that is, it includes F_e, F_u and E_u. In debugging, we refer to failure as the unexpected behaviour $F_u \cup E_u$. Debugging E_u is mandatory since the program fails to recover from the failure. However, debugging F_u is optional since the program returns to its normal execution and the programmer can debug it later or even ignore the failure (say, due to high debugging cost). More clearly, F_e is not considered a failure in debugging, because it is anticipated and properly handled by the program. Consequently, the corresponding 'error' and 'fault' are not considered to require debugging.

2.2 The Debugging Process

In a narrow sense, debugging is the process of locating the cause of an error. However, this task of *fault location* is only one step towards removing the error. Before this step, we must first expose and detect the error during testing or production. After this step, we should also be able to repair the fault and thus correct the error. People usually call the former step *error detection* and the latter step *fault repair*. Furthermore, the combination of error detection and fault location is generally called *diagnosis*. Because these steps are closely related, they should be considered at the same time. Therefore in a more general sense, we refer to *debugging* as a program development process which involves error detection, fault location and fault repair.

Usually, not one but several errors or faults appear in a program. Furthermore, a fault may be erroneously located or repaired. Hence, the debugging process is usually repeated until no further errors can be detected. After repairing a fault, not only should the failed test be repeated (to make sure the fault is fixed), but so should other tests that provide confidence in the effect of the repair (to verify that the repair does not generate other errors). However, we cannot generally guarantee that the program is error-free, since a non-trivial program can seldom be tested exhaustively. Moreover, the program is subject to continuous modification and enhancement to cope with changes of its external environment. Therefore debugging is not a single step, but rather a continuing iterative process covering the entire life cycle of the program.

2.3 Debugging during Software Development

Generally, software development involves the following steps: requirements analysis, software specification, program design, coding, testing, debugging, and maintenance. Errors, such as misinterpretation of requirements, misinterpretation of the design, incorrect use of design tools, bad algorithms, coding errors, faulty programming and change of requirements can occur at any step [Liffick85]. Hence, methods are developed to remove errors in each step; this leads to

terms such as *requirements debugging*, *design debugging*, *coding debugging* and *program debugging* [Mullerburg83].

We may try to debug errors before a program is actually executed. This kind of debugging is usually called *static debugging*. Design reviews, coding reviews, static analysis, and proofs of program correctness are some techniques for static debugging [Dunn84]. They are useful for reducing the number of faults in the program. However, bugs may still exist during execution, due to the incompleteness of these techniques and unanticipated changes in the execution environment [Myers76]. Consequently, there is always a need for *run-time debugging*. Some typical run-time debugging facilities for sequential programs are interactive symbolic debuggers, memory dumps and trace packages. In this paper, we concentrate our discussion on run-time debugging, and always refer to it simply as debugging.

2.4 Difficulties in Debugging

It is generally admitted that debugging is a difficult job. Programmers seem to suffer most from debugging, compared with other software development activities. What follows is a list of factors contributing to this problem.[1]

1. The necessity for debugging is an indication that programmers produce software which is less than perfect. Often, they find it emotionally disturbing to admit this fact.

2. Programmers tends to believe that their programs should work in one particular way, but in fact they do not [Weinberg71]. Such a belief hinders the programmers from viewing their programs in a different way and from looking at other reasonable assumptions.

3. Debugging is a mentally taxing activity, typically performed under a tremendous amount of pressure. Moreover, even intensive effort does not guarantee immediate return.

[1]Part of the list is extracted from [Myers79].

4. Debugging requires repeated switches between intuitive and analytic modes of thought. Research in cognitive psychology suggests that analytical thought occurs in the left hemisphere of the brain while intuitive thought occurs in the right hemisphere [Springer85]. Hence debugging may tend to make programmers feel uncomfortable.

5. Potentially, an error may appear anywhere in a software system. Therefore, locating a nontrivial error is a time-consuming and often boring job.

6. Compared with other software development activities, comparatively few research results, literature, or formal methods for debugging exist. The available debugging tools are also not adequate.

7. The semantics of a program may change when programmers locate and correct bugs, as the fixes introduce changes to the program. Thus debugging may introduce new bugs.

8. All software debugging facilities interfere with the execution of the system. They may either generate new errors, make the errors disappear or even make it impossible to observe existing errors. Debugging probes[2] can alter the program states in a way that makes bugs hard or impossible to find.

2.5 Debugging Methodologies

Bugs reveal themselves as failures. We may attempt to actively detect them by performing systematic tests on the program and thus discovering any discrepancy between the expected and observed behaviour. Or, we may monitor program execution passively in a production environment, to detect such erroneous behaviour as many occasionally appear. After a bug is detected, we begin debugging activities intended to determine its exact nature and location, and finally to correct it.

[2]Debugging probes are the instructions (*e.g.*, a code segment, a command) which are placed within a program for controlling the debugging process or capturing debugging information.

Some people tackle debugging by brute force. They just observe program behaviour and try to jump to a conclusion intuitively. They overlook systematic reasoning. Experience sometimes gives us the intuition necessary for fixing a bug, but we cannot rely on it. In fact, experience can be viewed as the condensed results of previous reasoning; we extract rules of thumb or heuristics to deal with bugs similar to those we have fixed. When the symptoms of a current bug match those of a previous one, we can use these heuristics to fix the bug quickly. However, there is no guarantee that the matching is correct, since a symptom may be due to various faults. Therefore, when experience fails us, or when we lack such experience, we must go through a reasoning process.

The following are methodologies to tackle debugging with the aid of reasoning [Myers79].

1. *Debugging by induction.* This method starts with clues from collected data and looks for relationships among the clues that lead to the error. We first locate pertinent data, and then organize them in order to search for any contradictions. After studying their inter-relationships, we devise a hypothesis about the bug. If we cannot devise such a hypothesis, we probably need to collect more data and to review their relationships again. If no hypothesis can be devised, the method ultimately fails. If one is obtained, we try to prove it by comparing it with the original clues. If we can prove it, we look for a solution and repair the fault. If the hypothesis cannot be proved, it may be incomplete or several faults may be present. We attempt to refine our hypothesis again and repeat the proof.

2. *Debugging by deduction.* The method hypothesizes a set of general suspicious faults. By the process of elimination and refinement, we decide which hypothesis should be proved. If the hypothesis is proved, we look for a solution and repair the fault. If the proof fails, we look for other hypotheses and if none can be found, the method fails. This method is similar to debugging by induction, except that it reasons from a set of general suspicious faults, rather than from a set of specific clues obtained from collected data.

3. *Debugging by state-sequence backtracking.* This is based on the proposition that if S' was the state of the program at time T', then S must have been the state of the program at time T. A programmer backtracks the production of the incorrect results through the logic of the program until he discovers the point where the logic went astray. Usually, this method is most successful for small programs, but too complicated for large programs.

4. *Debugging by testing.* This uses test cases to locate the fault. The software executes a sequence of test cases, and we formulate a hypothesis about the fault based on the outcomes of these tests. Then the hypothesis is tested. This method is usually used in conjunction with the inductive or deductive methods for reasoning about the hypotheses.

To reduce the complexity of program design and development, two general methodologies, called *top-down* and *bottom-up*, are generally considered [Dunn84]. A program is decomposed into a set of modules, forming a hierarchical structure. Top-down methodology is an incremental procedure to assemble, test and debug the modules by moving downward through the hierarchy, beginning with the main control module. On the other hand, bottom-up methodology begins assembly, testing and debugging with modules at the lowest level of the hierarchy and moves upward towards the main control module. While there are numerous discussions of these two methodologies for software development (such as [Pressman82], [Dunn84] and [Birrell85]), few discussions of them for production-time debugging can be found in the literature. Clearly, they are very useful to localize bugs in a complex program. Their use in debugging should be encouraged and supported by a debugger.

2.6 High Level versus Low Level Debugging

It is quite difficult to come up with definitions for low level and high level debugging in general [Gramlich83]. Many people agree that low level debugging studies the machine states with tools like memory and register dumps, while high level debugging is anything that is not low level

debugging. However, some people suggest that high level debugging is debugging in the *problem domain* while low level debugging is in the *solution domain*. Others suggest that high level debugging consists of tools added to a low level debugger to make debugging easier. Finally, still others suggest that high level debugging occurs at the same level as the programming language while low level debugging is below the programming language level.

In fact, the level of debugging depends on the objects in which we are interested. In very low level debugging, we look at machine-level objects (*e.g.*, memory locations and processor status) and execution sequences of machine instructions. In very high level debugging, we look at application level objects (*e.g.*, a list of records in a sorting program, a matrix in a numerical analysis program) and functional tasks (*e.g.*, sorting, solving an equation). In between these two levels, there could be various levels of debugging, such as the programming language level and the operating system level. Each level presents its own set of objects and operations to programmers for debugging.

Many people choose the programming language level as the boundary between low level and high level debugging. Low level debugging requires knowledge below the level of the programming language. High level debugging only requires knowledge at or above the level of the programming language. Low level debugging relies on the user to translate low level data into higher level abstractions. This step requires a high degree of mental effort and usually makes debugging very difficult and time-consuming. However, system programmers who are implementing low level software have to debug their programs at the level of hardware architecture and memory management. In general, high level debugging is desirable for the great majority of programmers who debug general application programs, while low level debugging should be used only when necessary, for example, by system programmers to debug system software.

Most debuggers work at a low level of abstraction. High level debuggers have been viewed as being expensive, because of slow speed, large storage requirements and implementation difficulties. As hardware costs continue to drop and software development costs—particularly the

costs of human labour—continue to rise, high level debuggers become more and more attractive.

2.7 Hard Bugs versus Easy Bugs

Every programmer knows that some bugs are easy to locate while others are very difficult to locate. Although such a distinction depends to a degree on debugging experience and on knowledge of the implementation environment, it also depends on some other factors. The following are the comments of some people in this matter [Gramlich83] [Gross83]:

- Weyuker asserted that most bugs are easily fixed (80%) in approximately an hour. The hard bugs (20%) are those which take on the order of a day to find and correct. This classification is based on debugging time.
- Smith (Carnegie-Mellon University) stated that a bug is hard if the tools you have are not capable of helping you find it. Therefore, there will always be hard bugs. This classification is based on the capability of debugging tools.
- Powell (University of California, Berkeley) remarked that the hardest bugs are those that appear consistently through all levels of the software development hierarchy: the specification, the algorithm and the implementation. For example, a single specification bug may cause design bugs and then program bugs. The programmer needs to trace from the program source up to the specification document to locate the original bug. This classification is based on the location of a bug.

In our opinion, there is no absolute division between easy and hard bugs. A bug which is hard for one programmer may be located easily by another programmer. However, the above classifications give us some indicators in different dimensions. Moreover, a simple classification of easy and hard bugs is not very useful. It is also necessary to know the nature of these bugs

and their frequencies of occurrence. Then, we can decide what features might be put into a debugger to tackle such hard bugs.

3 Issues in Distributed Debugging

3.1 Distributed Debugging

According to Enslow's definition [Enslow78], a distributed system has at least three dimensions of decentralization: hardware, control and database, and it should meet five criteria:

1. a multiplicity of physical and logical resources,
2. a physical distribution of these resources interacting through a communication network,
3. a high-level operating system (we call it a *distributed operating system* (DOS)) that integrates the control of the resources into a functioning whole,
4. system transparency for system services and resource accesses,
5. cooperative autonomy for operations of all components or resources.

Some researchers treat the terms *distributed system* and *concurrent system* as the same in their discussions [Baiardi83] [Foudriat85]. However, we refer to a concurrent system as a *single* processor system with multiple processes running concurrently and cooperatively. A distributed system should have a multiplicity and distribution of processors (criteria 1 and 2).

Le Lann gives another definition of distributed system from the view of its characteristics; a distributed system is any *non-perfect multireference* system with distributed control for some of the operating functions [Le Lann77]. Here, *multireference* refers to multiple physical spaces (*i.e.*, processing entities) and time references, and *non-perfect* refers to variable and unpredictable (*i.e.*, not known with absolute accuracy) communication delays between entities. Le

Lann also discusses the principles of time nondeterminacy, relativistic observation and state non-determinancy, and then concludes that the precise global state of a non-perfect system does not exist. The conclusion states an inherent constraint for developing and debugging distributed software; we are forced to deal with an environment that lacks precise global states due to multiple processors, lack of common time reference, and variable, unpredictable communication delays.

Distributed debugging is the process of debugging *distributed programs* on a distributed system; it does not refer to distributed implementation of a debugger which debugs sequential programs, like the Blit debugger [Cargill83]. The term distributed program refers to a collection of related, cooperative program modules which will be instantiated as processes at runtime in a distributed system. The collection of these running processes is called a *process cluster* (or simply, a *cluster*). A process cluster runs on top of a DOS which provides at least the functions of process manipulation and *interprocess communication* (IPC). An *interaction* of the processes is an activity that involves more than one process, or a process and the DOS. The interactions among processes are via IPC, which we assume to be done exclusively with messages. Most of the work done to date in distributed debugging emphasizes the IPC level. Here, messages and processes are the crucial objects to be manipulated, while IPC-based interactions are the important activities to be monitored.

The complexity of distributed programs is largely due to concurrent interactions among communicating processes; errors can be subtle, even transient, making their detection and correction very difficult. Also, the development of distributed software is still a new discipline and this brings additional problems to the study of distributed debugging, related to communications, system transparency and concurrency modelling. In general, there are more hard bugs in distributed systems than in single processor systems.

If we assume that ideal interactions (*i.e.*, fault-free and with accurately known communication delays) among processes can be achieved, and that global information about the cluster can

be obtained, we might consider distributed debugging to be the same as multi-process debugging in a single machine (*i.e.*, concurrent program debugging). This could be achieved if the underlying DOS can achieve processor transparency; that is, a user can view the distributed system as a "virtual uniprocessor", not as a collection of distinct machines. Unfortunately, processor transparency is not often fully implemented in existing distributed systems [Tanenbaum85], and may be impossible (refer to Le Lann's arguments above). Furthermore, for debugging at the DOS level, a programmer usually needs knowledge of the actual implementation. Therefore, problems raised by a distributed system *per se* should be considered, in addition to those raised by a multi-process environment. In summary, debugging distributed programs is generally more difficult than debugging concurrent ones.

3.2 Bugs in Distributed Programs

A distributed program is basically a set of programs. Therefore, bugs which occur in sequential programs still occur in distributed programs. In addition, new kinds of bugs are introduced due to the particular features of process clusters, such as concurrency, synchronization, and cooperative communication. These bugs include message omission, unanticipated messages, arrival of messages in an unexpected order, deadlock among processes, untimely process death, faulty synchronization, partition due to communication problems, communication overload, and protocol faults. We should notice that these bugs are not unrelated. For example, a message omission error may lead to a deadlock error if the receiver must receive the lost message before continuing its execution. Communication overload may lead to message omission. Moreover, if the sender tries to retransmit a possibly lost message and the receiver does not anticipate this duplicated message, an error occurs. On the other hand, certain errors occur in a particular IPC paradigm. For example, in non-blocking IPC, a process may inadvertently receive a message into a buffer before using the message last received into that buffer, or a process can inadvertently change information in a message buffer which is shared by two consecutive send operations.

However, these errors do not occur in blocking IPC.

For a distributed program, we must consider not only a larger number of errors, but also a much larger number of possible causes of the errors—faults. An error may possibly be caused by several faults, and a fault may generate several errors. This many-to-many mapping makes fault location very complicated. A random guess or trial-and-error method is generally ineffective. Faced with this complexity, we need systematic methods to analyze bugs and find the relationships between bugs and causes. Based on the mapping, we could design appropriate algorithms to avoid or locate certain bugs. For example, Ezhilchelvan *et al* [Ezhilchelvan86] classify errors in system components using two parameters, time and value, and argue that such a classification can help us to develop simpler algorithms to deal with certain errors. In [Gordon85], Gordon investigates distributed program errors that depend on the relative order between events. These ordering errors are divided into two classes, timing errors and misorderings. Timing errors are time-dependent orderings of events that lead to errors. Misorderings are time-independent events that always lead to errors. The author characterizes these errors, shows necessary conditions for their occurrence, and gives methods for their prevention and detection. He also implemented a postmortem debugger called TAP, based on the use of *timing graphs*, to help a programmer find the causes of detected timing errors. A timing graph is a directed acyclic graph representing the partial order of events for a distributed program. TAP assumes that the underlying DOS has a monitoring mechanism to collect enough information at runtime to construct timing graphs. Basically, TAP is only a facility to allow a programmer to study timing graphs manually; the programmer is required to locate the faults by his own analysis, based on the study.

3.3 Problems Encountered in Distributed Debugging

In this section, we look at the problems in distributed debugging. As distributed debuggers faces different clock times on multiple processors, management of the system state space and control of time are two major considerations [Wittie85]. In addition, interaction complexity and human-debugger communication raise further problems. A discussion of some typical problems follows:[3]

1. *Maintenance of Global States.* Global clock synchronization is a classical research problem in distributed systems, and it is nontrivial to have a global synchronous clock to support distributed debugging [Curtis82] [Wittie85]. This implies that it is very difficult for a debugger to provide services which depend tightly on exact timing, such as checkpointing. In a general sense, it is hard or even impossible to obtain global information about a cluster at a precise instant in time [Le Lann77]. Moreover, an immediate change of control for all parts of a computation on different machines at the same time is very difficult, due to unpredicatable communication delays and different processor states. Consequently, it is very difficult to arrange for actions to occur precisely at an agreed time, or to freeze a cluster at some point to obtain an instantaneous picture or snapshot of the whole cluster.

2. *Large State Space.* The execution state of a cluster includes machine state on each processor and a record of interactions among processors. Generally, the state space is very large, which raises the problem of manipulating such a huge volume of state data at execution time. For example, how do we select useful data for later analysis, given limited disk storage and main memory? How do we integrate distributed data from individual processors? How do we display useful information extracted from the data for a programmer? On the other hand, distributed systems can grow incrementally and have a tendency to have large numbers of processes and processors. As we know, the larger a system is, the more

[3]Some points are from [Garcia-Molina84].

information we should be able to handle. Furthermore, the complexity of interactions increases when the number of cooperative processes or processors increases.

3. *Interaction Among Multiple Asynchronous Processes.* Bugs occurring among processes in a cluster are usually complex or sporadic, probably caused by improper synchronization among the processes or by race conditions. In many cases, these bugs are hard to reproduce, since they depend not only on input data, but also on the relative timing of interactions between processes. Worse still, some clusters are dynamic enough (*i.e.*, numerous process creations and terminations) that even identifying the relevant set of processes is very difficult.

4. *Multiple Autonomous Processors.* It is not easy to locate erroneous behaviour and to manage debugging operations on multiple processors. We assume that processors run autonomously in a truly concurrent fashion, and only synchronize and communicate with each other via messages. When a cluster fails, we shall have a hard time just discovering which process is responsible for the error and what is the immediate cause of the failure. To do so, we must examine states of processes in the cluster and a partial ordering of their interactions. Multi-threaded executions make the examination more difficult than singe-threaded execution, as we need to collect and to assemble state information from different time and state references. On the other hand, when a processor halts due to an error, it takes some time to stop other processors. By the time all processors are stopped, the critical information may have been destroyed and it becomes difficult to discover the error. Furthermore, if the cluster runs on a single processor, many hardware (or even system software) failures will be obvious to us. However, with multiple processors, we may not be aware that one of the processors is down or faulty, causing the cluster to behave in an erroneous way.

5. *Significant Communication Delays.* The machines may be geographically dispersed so that we must collect debugging information remotely. Significant, variable and unpredictable communication delays, and limited communication bandwidth make some typical debugging techniques, such as central manipulation of cluster state information, impractical.

6. *Error Latency.* When a processor behaves abnormally, other operational processors may not discover the error immediately. Usually, there is a time lag between the time when the error actually occurs and the time when the error is discovered. This problem also exists in sequential programs, but is worse in distributed ones. Due to significant communication delays, the time lag may be quite large in a distributed system. During this time period, the error may propagate widely. Via communications, the faulty processor affects the processes on another processor which passes the effect to other processors and so on. This domino effect may generate a burst of additional failures, often making the original problem very difficult to locate. We need to find ways to limit the propagation of errors.

7. *Human Problems.* It is generally agreed that people find it harder to handle concurrent events than sequential events. Also, the temporary memory of our brains is very limited; excessive debugging information is simply ignored [Model79]. Therefore, there are problems of presenting data about a large number of concurrent events to a programmer in an elegant and concise way, so that he can easily understand the program behaviour and obtain the required information.

3.4 The Need for Distributed Debugging Systems

Although programmers face a lot of problems and difficulties in developing distributed programs, they have been obliged to work in very primitive development environments. Few tools are available to help them in distributed software development, especially in the debugging phase. The need for an elegant distributed debugging system is obvious and urgent. In particular, we

mention the following reasons for providing such a debugging system [Bei85]:

1. *Programming complexity.* It is complicated to develop distributed programs. Unfortunately, research on distributed programming languages and other programming tools is relatively new. We lack experience and methodologies for distributed programming. Therefore, most distributed programs developed nowadays are still immature, and require a lot of testing, debugging and evaluation.

2. *Human errors.* A bug-free program is an abstract theoretical concept. We can reduce errors by using good programming methods and by using formal correctness proofs, but we still cannot avoid human errors completely in significant programs. Therefore, debugging is, and will probably remain, an unavoidable step in program development. Debugging is especially important for distributed programs because programming complexity and the lack of appropriate tools for distributed programming both make human errors more likely.

3. *Debugging difficulties.* As we mention in Section 2.4, debugging is a difficult job. We also mention in Section 3.3 that a distributed environment makes the situation even worse. An elegant distributed debugging system which can address these difficulties will be a definite improvement.

4. *Debugging costs.* Debugging is often costly in terms of time, manpower and computing resources. Estimates of the fraction of total effort spent on debugging vary from 50% to 90% over the software life cycle. Clearly, the costs of debugging distributed software will lie at the high end of the range; a useful debugging system could reduce such costs.

4 A Framework for Studying Distributed Debugging Systems

4.1 A Model of Distributed Debugging Systems

As the problems in distributed debugging are complex and involve a lot of issues, it is essential to consider them in a systematic way. One way to do so is to decompose the debugging environment into particular domains, and study the specific issues and corresponding solutions in each domain. Following this idea, we introduce a model depicted in Figure 2 to define the role of a typical *distributed debugging system* (DDS) in the *debugging environment.* A DDS is a collection of software modules to facilitate the debugging process. These modules may be in the form of a process cluster, a code segment in each process, or supporting routines in the kernel. The debugging environment (or simply, the environment) is a collection of entities which are useful to the DDS or which interact with the DDS.

In the model, the environment is partitioned into three domains: the program domain, the human domain and the distributed domain. The program domain includes static descriptions of the distributed program being debugged, such as its source code and expected behaviour as specified by the programmer. The human domain includes the attributes of a typical programmer who uses the DDS to debug clusters. The distributed domain includes the characteristics of the underlying DOS on which the cluster executes, and the execution behaviour of the cluster.

The DDS is considered to be an agent which manages the interactions among these three domains. It collects behaviour data from the distributed domain, and imposes appropriate control on the domain. Also, it consults the program domain to interpret the collected data and to forward meaningful information to the human domain. The DDS should interact with the human domain effectively, in order to allow the programmer to control the DDS easily. Furthermore, the DDS itself may analyze information from the domains and make some conclusions about a bug. This implies that it provides some automatic facilities to help the programmer debug a

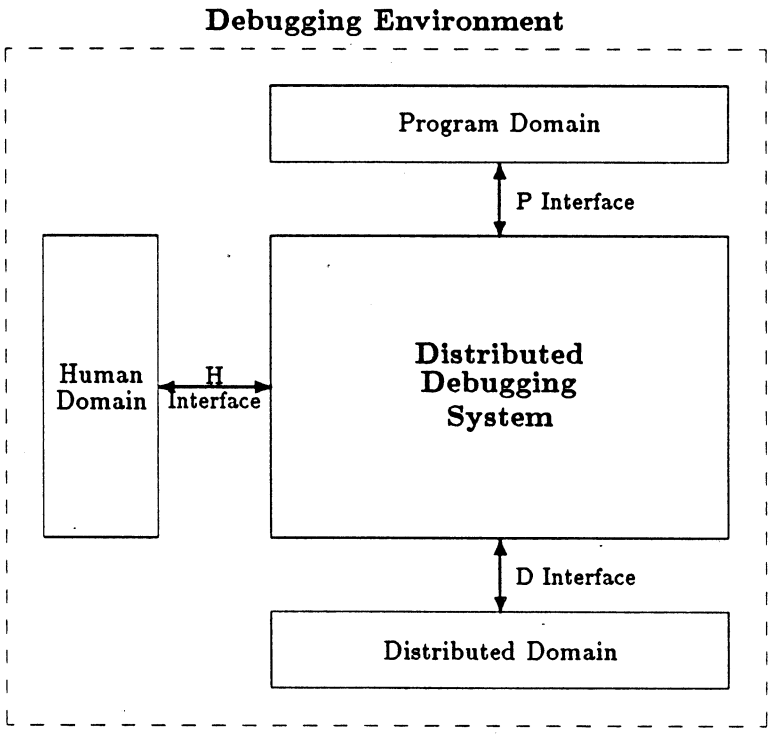

Figure 2: A Model of Distributed Debugging System

distributed program. For example, it may help the programmer to formulate hypotheses about a bug and to verify them.

Relative to communication with each domain, our model specifies three *interfaces*: D, P, and H. These interfaces are associated with the distributed domain, the program domain and the human domain, respectively.

- The **D interface** is for communication with the underlying distributed execution environment. Normally, it is the DDS-DOS interface. More specifically, it may involve the kernel-process interface, exception handling, remote operations and coordination with various parts of the DDS on different machines. More importantly, it provides a precise view of the distributed computational model supported by the distributed operating system. Based on this interface, the DDS observes and possibly influences the behaviour of a cluster in the distributed domain. On the other hand, the DDS itself is a distributed program. Its software implementation is based on this interface.

- The **P interface** is for communication with the distributed program to be debugged. The communication not only involves the program itself, but may also involve representations of the "expected" behaviour of the program and perhaps specifications of anticipated errors. In high-level debugging, it may also involve the representation of the problem itself, in addition to the program. The interface involves the language-debugger interface and the mechanism for detecting bugs according to the representations. It also conveys necessary information about the program (*e.g.*, the source code, the objects manipulated) to the DDS and allows the flow of debugging information from the DDS to the program domain (*e.g.*, a modification of the program due to a bug, the recording of a bug for documentation).

- The **H interface** is for communication with a human programmer. First of all, it defines the allocation of debugging activities between the DDS and the programmer. With this interface, we can clearly define for which activities the DDS is responsible, which

activities the programmer performs by himself and which activities both the DDS and the programmer should consider. Then, based on the allocation, this interface defines the human-debugger interaction. Human factors for debugging activities and mastering parallel events are the main concerns, since they have a profound effect on the usefulness of the DDS. Moreover, since graphical displays and pointing devices have become popular, their impact on this interface should not be overlooked.

Of course, a programmer can bypass the DDS and interact directly with the program domain and distributed domain for debugging. However, such interactions indicate that the DDS is not effective. With a good DDS human users want to debug through the H interface alone.

In the model, the D, P and H interfaces completely define the interaction between the DDS and its external environment. If the interfaces are designed carefully and precisely in a domain-independent way, it is possible to construct a DDS which is portable to other implementations. Portability is a very attractive feature as it allows us to transfer a DDS from one implementation to another implementation with little effort. Furthermore, we can maintain a well-developed DDS despite changes to the external environment. However, a portable design is nontrivial, since the interfaces are complicated and they may have to cope with specific pre-existing environments.

In general, this model can also be applied to the sequential and concurrent environments. The sequential environment is a special case of the model in which the cluster has only one process. There are no process interactions and thus we simplify the debugging environment to deal only with the internal behaviour of the process. The concurrent environment is also a special case, in which all processes in the cluster run on only one processor. Again, this simplifies the debugging environment to deal with the interactions within a processor only, and hence allows us to obtain and control global states of the cluster more easily.

4.2 A Decomposition of the Problem Space

The model identifies three main domains in a typical debugging environment. Now, we look at the problems of developing the DDS and its interactions with the external environment. For the sake of our study, we divide the problem space into three main areas and then further decompose them into smaller problem components. First of all, we should have a clear picture of the DDS itself before we can implement it. Therefore, the first problem area is to design a distributed debugging model. As the DDS needs to interact with the domains, the second problem area is domain interaction. The third problem area concerns the implementation issues. It looks at what kind of system facilities are required to support the distributed debugging model and domain interactions.

1. **Distributed debugging model (DD Model)**

   A DD model is a conceptual framework which governs debugging activities in general. It also establishes assumptions about the environment. We divide the issues of a DD model into three major components. The first one is a general *approach* to problems in distributed debugging. The second one is a set of guidelines or *methodologies* to govern debugging activities. The third one is a set of *techniques* to facilitate the debugging process. In order to develop a complete DDS, we should have an appropriate design for all of these components. Concisely, we can express a DD model as:

   DD model = Approach + Methodologies + Techniques.

2. **Domain interaction**

   This area considers the interactions between the DDS and each domain in the DDS model. There are two major factors to be considered: *representation* and *communication*. Representation concerns the description of a domain. More precisely, it studies representations

of objects (*e.g.*, errors, monitoring information, notations on a graphical display) or activities (*e.g.*, program behaviour, human interaction) in the domains. Communication concerns the design of a domain interface. Only when the objects and activities of the domains are described precisely can we define the interaction between the domains and the DDS. Usually, the representations are expressed in appropriate languages. It is important to understand the limitations of these languages and their ability to express all the objects and activities in the domains. Also, the interfaces should be defined precisely and effectively. The flow of information across the interface must be controlled. Protocols are needed to govern the cooperation between the domains and the DDS. Basically,

$$\textbf{Domain interaction} = \textbf{Representation} + \textbf{Communication.}$$

3. **System support**

   This area considers implementation issues, and can be further divided into two parts: basic support and model-specific support. Basic support includes the common requirements for the DDS, such as the D, P and H interfaces. Model-specific support provides facilities to support a particular DD model. This separation allows us to first provide a basic facility and then develop various models based on it, either for experimental purposes or for different debugging requirements. Therefore, system support can be considered as:

$$\textbf{System support} = \textbf{Basic support} + \textbf{Model-specific support.}$$

These three areas for research can be considered to be different aspects of constructing a DDS. The DD model deals with the design issues, the domain interaction deals with the external communication issues, and the system support deals with the implementation issues. In summary, we give the overall expression of a DDS as:

$$\textbf{DDS} = \textbf{DD model} + \textbf{Domain interaction} + \textbf{System support.}$$

Using this decomposition as a guideline, we are able to address the issues and develop corresponding solutions in a clear and systematic way. While the DDS model gives us the role of a DDS relative to its external environment, this decomposition helps us to look at the DDS in terms of design, interaction and implementation. Based on this framework, we can study current research in distributed debugging in a more systematic way.

Notice that these three areas are not independent. The DD model is the main theme of considerations directly related to debugging. However, the DD model must eventually be related to the issues in the area of domain interaction for external communication and the area of system support for implementation. Therefore, in our discussion in the next few sections, we concentrate on the DD model and discuss the issues in the other areas whenever appropriate.

5 Basic Debugging Techniques

5.1 Introduction

This section presents some effective debugging techniques. As we shall see, most of them are extensions to classical debugging techniques for single programs.

As mentioned in Section 2.3, one technique for debugging a distributed program is *static debugging*, which refers to locating possible bugs in a program before the program actually runs [Mullerburg83]. While not ignoring static debugging, we consider it to be beyond the scope of this paper and will not discuss it further.

All existing DDSs are based on some assumptions, explicit or otherwise. The following are two common ones:

1. The necessary system software (so-called *hard core*), such as the DOS, the communications subsystem and the debugging tools, is debugged[4], and programmers can therefore concen-

[4]Precisely, there is a low probability that bugs occur in hard core software.

trate on the debugging of application software. More importantly, in reasoning about the problems and solutions, we can assume that the debugger is correct.

2. The main focus of a DDS is the interaction among processes, rather than the internal logic of each process. There exist sequential debuggers to deal with bugs internal to a single process. The DDS works at the process level of granularity and is interested in process events and IPC; the assumption is made that internal bugs of the processes are or can be fixed by the use of a sequential debugger. In Schwan's terminology, a distributed debugger deals with *programming in-the-large* [Schwan86] in which processes in a cluster are considered as the basic building blocks. A sequential debugger deals with *programming in-the-small*; that is, the internal behaviour of an individual process.

   Of course, there are some DDSs designed for lower levels of debugging which may involve internal process behaviour. For example, Multibug is intended for designers of kernels [Corsini86]. It allows the designers to manipulate objects on each processor through the console of a host. The level of granularity is down to processor objects, such as memory locations, I/O ports and CPU registers.

Although many DDSs assume a sequential debugger to deal with internal bugs, there is little literature discussing the interface between a DDS and a sequential debugger. In many cases, we do not know whether a failure is caused by an internal bug or an interaction bug. It is definitely useful to consider the interface between distributed and sequential debuggers, in order to exchange debugging information and to switch easily between the two. The authors of Bugnet plan to explore this matter. They want to add a Bugnet interface to the UNIX *dbx* sequential debugger so that a programmer can closely monitor a small subset of processes running in Bugnet [Jones86].

In the following discussion, we keep the two general assumptions listed above. That is, the DDS focuses on process interactions and assumes that it itself is debugged.

5.2 Output Debugging

This is the most primitive debugging technique. A programmer inserts debugging probes, usually output statements, at carefully selected places in the program. Using the output data, the programmer tries to understand the execution behaviour in order to find bugs. The advantages of this technique are that it is independent of debugging tools, as no tool is required except output statements, and the programmer sees only output selected by himself. Clearly, it has disadvantages, otherwise we would not need to develop other debugging tools. It relies completely on the programmer to select appropriate places in the distributed program to insert debugging statements. This resembles an art and is based on the programmer's experience and thinking. Also, the technique requires modifications to the software and hence may confuse the existing program structure or even introduce new bugs. More importantly, it can easily change the behaviour of a cluster. Thus, this technique is not effective for locating time-dependent bugs.

5.3 Tracing

This differs from the previous technique in that instead of allowing manual tracing with output debugging, the operating system provides a standard trace facility to print selected tracing information. The programmer turns on the trace in the software when necessary, and turns it off during normal execution. The trace facility keeps track of execution flow or object modification, and reports relevant changes at certain times.

This technique has the advantages of output debugging and also eases the task of inserting output traces. However, it still relies completely on the programmer to specify appropriate actions. Also, if traces are enabled for multiple processors, the programmer or debugger has to assemble them to obtain a global trace. In either case, global times for all trace information are necessary in order to form the global trace. In other words, we need a global clock

synchronization facility before we can construct global traces from distributed local traces. If there is no such facility, then a selected processor should be responsible for forming the global trace, according to the order in which trace information is received from all other processors (*e.g.*, IDD [Harter85]). This implementation is based on the assumption that trace information generated at time T arrives at the selected node before any trace information generated at time T' later than time T. Due to variable communication delays and DOS operations, the validity of this property may not be guaranteed. Also, the selected node becomes a bottleneck for the collection of trace information.

5.4 Breakpoints

A breakpoint is a point in the execution flow where normal execution is suspended and cluster state information is saved. At a breakpoint, a programmer can interactively examine and modify parts of cluster states, such as execution status and data values, or control later execution, such as single step execution. Execution can continue at the breakpoint when requested by the programmer.

Using this technique, no extra code is added to the program. Therefore, it avoids some of the effects of debugging probes on distributed programs. Also, the programmer can control cluster execution and select display information interactively. The main disadvantage is that a programmer must understand the internal execution flow and cluster states very well and must be knowledgeable enough to set breakpoints at appropriate places and to examine relevant data.

Also, the technique raises particular problems in a distributed environment. First of all, it is hard to define a breakpoint *precisely* in terms of global states in distributed environment, since the process cluster runs in an asynchronous manner. Thus, people usually define a breakpoint in terms of events in a process (*e.g.*, IDD [Harter85], Pilgrim [Cooper87]) or interactions among processes (*e.g.*, CGDS [Bei85], the Jade Monitoring System [Joyce87]).

Second, the semantics of single step execution are no longer clear. Some define it to be the execution of a single machine instruction or a statement of source code on a local processor (*e.g.*, CBUG [Gait85]). Others may consider it to be a single statement on each processor involved (*e.g.*, IDD [Harter85]). Some people treat an event, such as message transmission, message reception or process termination, as a single step (*e.g.*, Radar [LeBlanc85]). With single instruction execution, we can know exactly where the execution will stop, as an instruction is usually well-defined in a processor. However, executing a single instruction may not be very productive. To find bugs which result from the interaction of processes, it is more effective to run the cluster until a significant event occurs.

Third, there is the problem of how to stop the process cluster at a breakpoint or after a single step. Due to the difficulty of maintaining global states and the existence of significant variations in communication delays, we cannot halt the entire cluster at a precisely defined point. People can only expect the cluster to halt at a point "close to" the desired point, where "close to" is hard to define.

When a breakpoint is triggered, the process cluster should be stopped. One simple way to do so is to broadcast "pause" messages to all processors [Garcia85] [Cooper87]. A processor suspends its execution entirely when it receives the pause message. In many cases, we only want to stop the process cluster in which we are interested rather than the entire distributed system, because some other process clusters are running on the system. In such cases, pause messages with a cluster identification are sent out. When a processor receives such a message, it only suspends the execution of those processes which belong to the specified process cluster. Through the debugger, the programmer then examines the states of the various processes. Later, the programmer can set other breakpoints and resume the execution of the cluster. This is achieved by broadcasting "continue" messages to all processors.

Now, the problem is how to record the global state information consistently at the time when the process cluster is suspended. Chandy and Lamport introduced a distributed algorithm

to obtain the global states or *distributed snapshot* of a cluster [Chandy85]. The algorithm is intended to capture only the global states with *stable properties.* A stable property is one that, once it becomes true, remains true thereafter. Examples of stable properties are process termination, deadlock and token loss in a token ring network.

The algorithm is divided into two independent phases. During the first phase, local state information is recorded at each process. This phase ends when it is determined that all information, both in processes and in transit over communication channels, has been taken into account. In the second phase, local state information of each process is collected into a snapshot by the process(es) which initiated the snapshot. Spezialetti and Kearns modified this algorithm to make the second phase simpler or more efficient [Spezialetti86]. There are two important assumptions made in the use of this algorithm. First, a process must have at least one incoming and one outgoing communication channel. The algorithm cannot collect local information of a process which has no communication channel or only one communication channel connected to other processes. Second, the messages must be received in the order in which they are sent. Hence, the algorithm cannot be based directly on a datagram-type communication facility.

5.5 Assertion Execution

An assertion is a predicate statement which specifies invariant conditions at the point of execution where it is placed. More powerful assertions allow the specification of invariant conditions to hold over intervals of program execution. Assertions may be inserted dynamically by a programmer at a breakpoint, like the idea of incremental assertion generation in IDD [Harter85]. This allows the programmer to set up assertions to guard against some of the failures that have been seen previously.

Assertions behave like conditional breakpoints. They are inserted into the program by a programmer. During execution, whenever an assertion is violated, the system reveals the situa-

tion to the programmer and usually halts the execution. Therefore, it can save the programmer from examining large traces in order to detect the violation. Also, assertion execution is more powerful than normal breakpoints. The former can detect certain errors based on the assertions while the latter leaves detection entirely to the programmer.

Assertion methods used in sequential debuggers depend on the notion of a global state. In a distributed environment, it is hard or impossible to obtain a precise snapshot at arbitrary points in execution for checking assertions. Thus, assertions in distributed debugging can only describe states local to an individual processor, or focus on sequences of transmitted messages.

5.6 Controlled Execution

In a distributed system, the behavior of a cluster depends not only on input data, but also on the relative speeds of processors and on communication delays. Hence, special kinds of control, such as changing the relative speeds of the machines, simulating various delays in the communication paths and altering the order of message passing events are necessary. Controlled execution allows a programmer to analyze concurrent events at the speed and in the order he expects at execution time. However, some errors may disappear due to the changes of speed and interaction sequence.

This technique is very useful for revealing bugs in the testing phase. When the behaviour of a cluster is under control, the programmer can alter the order of interactions in order to perform a set of pre-defined tests. However, exhaustive testing of all possible interactions is almost always impossible. We need to find ways to prune the number of tests, such as identifying equivalence classes of interactions.

5.7 Replaying

Replaying simulates the history of a process cluster in a controlled environment. The debugger continually captures execution information for the cluster and saves the information in a history log. When an error occurs, the program is suspended. Using the history log, the debugger replays the program execution in an artificial environment. The programmer can control the speed and direction of the replay via the debugger. By close examination of the program behaviour before the failure, the programmer may locate the fault. For example, the Radar debugger mainly provides support for replaying [LeBlanc85]. During execution, every event and every message of a cluster are recorded in disk storage. After execution is complete, Radar is invoked to perform a graphical replay of the execution. It displays the events one at a time, allowing the programmer to watch the communication traffic among the processes.

One problem of replaying is how to record cluster behaviour for later replays. Leblanc and Mellor-Crummey propose a general solution to this problem, termed *Instant Replay* [LeBlanc87]. During cluster execution, the relative order of significant events, rather that the data associated with such events, is saved. As a result, the technique requires less time and space to save the information for replaying. An experiment shows that the technique only imposes a small overhead on performance (less than 1%). Therefore, it can be used to record events of a production system for debugging. However, the replaying is not a simulation run. Instead, it is an execution with the same input from the external environment and with a mechanism to enforce the event order that occurs in the original execution. Instant Replay assumes that each process is deterministic; that is the process does not contain nondeterministic statments or allow asynchronous interrupts. Also, the replaying must involve the whole cluster; we cannot replay a subset of the processes in the cluster. This also implies that we cannot speed up the replay or just examine a part of the cluster behaviour.

Bugnet allows a programmer to replay part of cluster execution by using a technique called a

checkpoint algorithm [Jones86]. The algorithm is based on a global clock synchronization facility (*e.g.*, the TEMPO time service on UNIX [Gusella83] [Gusella85]). All processes in a cluster start execution synchronously with reference to the global clock. During execution, Bugnet collects IPC events of the cluster. After a period of time (currently 30 seconds), execution is halted globally, and the state of each process is captured. During the checkpoint pauses, the only events that may occur are the arrivals of messages that were transmitted just before a pause. If such pending messages were lost, inconsistent checkpoints would result: the sender knows the messages were sent but the receiver never receives them; this issue is similar to the problem of recording consistent states in a distributed snapshot, as mentioned in Section 5.4. These messages are saved and then presented to the receivers during the next run period. The halting period is the same for all process (0.5 second), so that all continue execution at the same time. This run, stop, checkpoint, and continue cycle repeats until an error occurs or the programmer decides to quit or replay. For replay, the latest checkpoint point before the requested start of replay is located, and a common clock time as well as necessary replay information (*e.g.*, process status) are sent to the agents which monitor the replay process. Due to the accuracy of the global clock facility, Bugnet can only replays execution sequences with an accuracy of 0.2 seconds.

5.8 Monitoring

Most existing distributed debuggers adopt this technique, or from another viewpoint, many monitoring systems are developed to help debugging. Examples include Miller's Distributed Monitoring System [Miller85] and the Jade Monitoring System [Joyce87]. Besides debugging, these distributed monitoring systems are also used for performance evaluation and to study cluster behaviour.

The idea of monitoring is to capture useful information during execution, manipulate it

and display it to the programmer. Sometimes, the information is saved for later use, such as replaying. The captured information describes all the events and interactions of the execution. A debugger may analyze the captured information in some fashion, detect an error, or even locate the fault. Since the execution is monitored continuously, the occurrences of infrequent, unpredictable and irreproducible bugs should not be missed, and the recorded events will provide valuable information for diagnosing the bugs.

However, keeping a copy of every event and every message is not cheap. Full monitoring of a cluster produces large volumes of data and requires tremendous amounts of processing time and storage to manipulate. Furthermore, the examination of the traces is a very tedious activity and requires a high degree of expertise. Therefore, people use filtering and clustering techniques to reduce the amount of information which needs to be displayed or kept.

- **Filtering:** For handling large volumes of monitoring data, this technique attempts to filter or ignore data which is irrelevant to the current debugging interest. Although it introduces extra filtering time during monitoring, it can save time needed to process such data. We may filter monitoring data in two ways. The first one is to show only selected data to a programmer, but to save all data in storage (*e.g.*, IDD [Harter85], the Jade Monitoring System [Joyce87]). The second one is to save only relevant or selected data in storage; all other data is discarded (*e.g.*, Miller's Distributed Monitoring System [Miller85]). We may call the former *display filtering* and the latter *data filtering*. Display filtering does not reduce the amount of storage to keep the data, but avoids flooding the display with irrelevant data. Data filtering takes a step further to discard the irrelevant data in order to save a large amount of storage. However, it may not save sufficient data to completely reconstruct the original cluster behaviour (say, for replaying).

- **Clustering:** In order to reduce the amount of storage for monitoring data, clustering is used to group and condense a designated set of events into a single composite event.

Monitoring systems using this technique include Bates' Monitor [Bates83b] and the Jade Monitoring system [Joyce87]. To use this technique, a programmer should first define composite events in terms of previously defined events or primitive events,[5] by using an event definition language such as EDL [Bates82]. For example,[6] a collection of producer, buffer and consumer processes may form a simple component of a larger distributed program. While we see several events to transfer a message from the producer to the consumer, we can cluster them into a composite event called "a resource being consumed." Then during execution, by recognizing a sequence of events as a composite event, the latter is recorded and the former is discarded. The clustering process can be exercised in a hierarchical manner, so that more abstract and higher level information is maintained. In addition, the clustered information is more suitable than the raw data for presentation to programmers and thus facilitates high level debugging. However, there are some problems involved with recognizing composite events out of a low level event stream, such as filtering out noise, handling the relative timing of processes, distributing event information to other processors, and sharing a primitive event among several composite events [Bates83a].

6 Debugging Methodologies

This section presents the methodologies used by some distributed debuggers to organize overall debugging activities and the application of the lower level techniques mentioned above. The sequential debugging methodologies discussed in Section 2.5 still apply; in addition, several methodologies which have been considered in distributed debugging are presented.

[5]Primitive events are the events that can directly be obtained from monitoring data.

[6]This example is taken from [Joyce87].

6.1 Top-Down versus Bottom-Up Debugging

Many distributed programs are composed of a large number of processes. If designs of such distributed programs are modular, their processes are generally grouped into various functional modules, and several modules are merged again to form a bigger module and so on [Lau86]. Many debuggers provide a facility to support this kind of process grouping, such as Hierarchical Process Groups in Bugnet [Jones86], Groups in the Jade Monitoring system [Joyce87] and Clusters in CGDS [Bei85]. To deal with the debugging of these grouped processes, researchers have proposed two quite familiar methodologies: top-down debugging and bottom-up debugging.

- **Top-down debugging** [Hamlet83] [Baiardi83]

  The methodology is illustrated in Figure 3. Initially, the behaviour of the entire program is considered and a distinction is made between erroneous and correct modules. Then, debugging focuses on the erroneous modules and more detailed behaviour within the modules is considered. Again, the distinction among erroneous and correct submodules is made, and so on until the process responsible for the bug is located.

- **Bottom-up debugging** [Garcia-Molina84]

  Figure 4 shows the procedure of bottom-up debugging. In the first step, each process is debugged separately in an artificial environment. The detailed behaviour of each process is considered. Later, several processes are merged together as a module and debugging then concentrates on the interactions among the processes within the module. Again, the debugged modules are grouped together for further debugging and so on until the entire distributed program is debugged.

Basically, these two methodologies can reduce the complexity of the debugging process. There is no general agreement on which methodology is better; this may depend on the situation. For example, top-down debugging may be more convenient in a developed system, as all processes

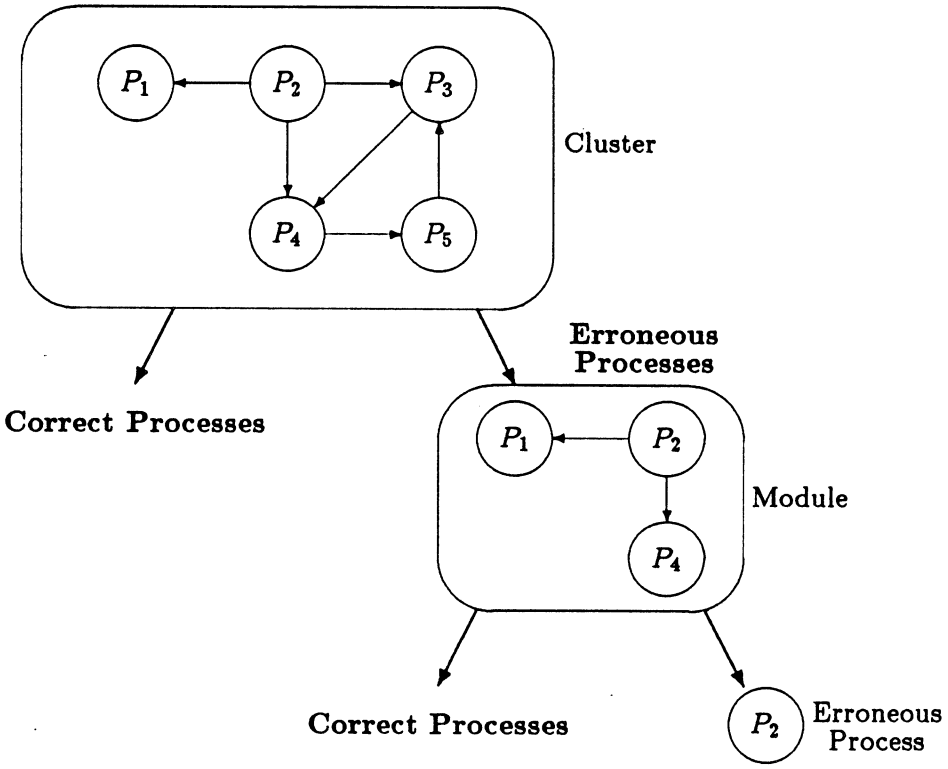

Figure 3: Top-Down Debugging

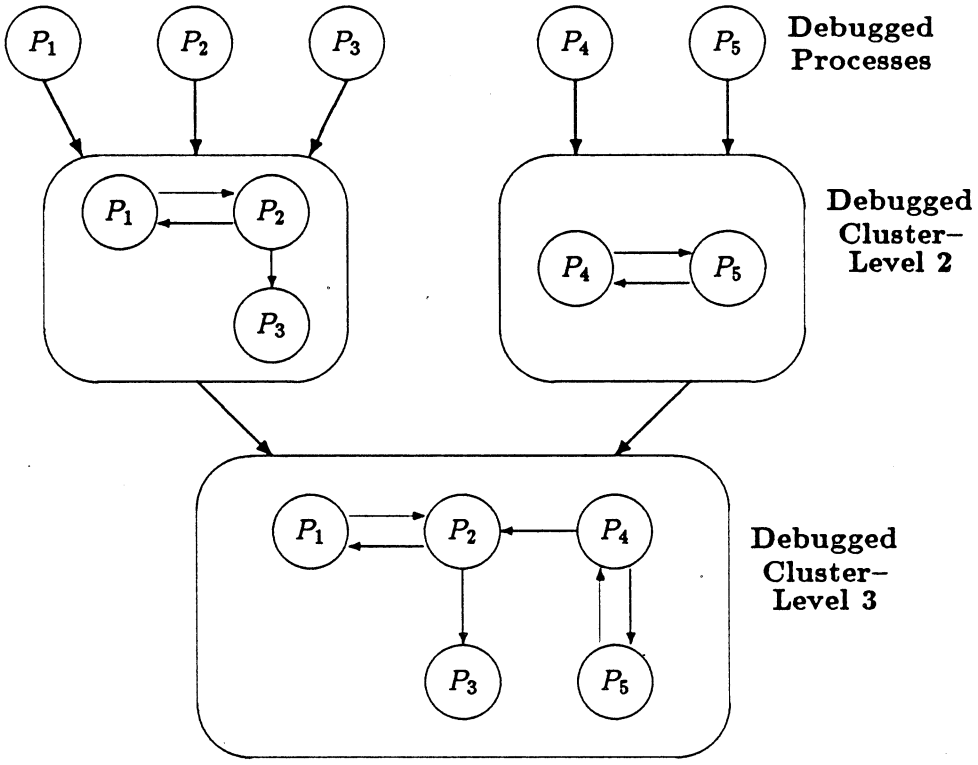

Figure 4: Bottom-up Debugging

are fully implemented. Bottom-up debugging is more appropriate for a newly-developed system since we can test and debug the system gradually from small modules up to the whole cluster.

Actually, the use of these methodologies in distributed debugging is an extension of their use in sequential debugging. We manipulate processes instead of program segments as basic module units. We deal with messages flow among processes instead of data flow among program procedures. However, the control flow is now in multiple threads instead of a single thread. Also, the communication structure of a cluster may change due to dynamic process creation and termination. Therefore, we must not only group the processes into functional modules, but also restrict message interactions between those modules in order to use the methodologies effectively.

6.2 Two-Phase Debugging

One basic idea of debugging is to monitor the events of a program during execution, and hope that the accumulated trace will exhibit an abnormality which may help in localizing the fault. Experience shows that once the fault is localized to within a relatively small part of the system, its identification is not particularly difficult.

Two-phase debugging applies this idea in distributed debugging. The debugging process which is shown in Figure 5 goes through two phases:

1. *Phase one.* The software continues to execute until an error is observed. During execution, the debugging system makes trace entries for truly significant or important process events. When an error occurs, the trace is examined and the buggy processes are identified.

2. *Phase two.* Erroneous processes are then tested in an artificial environment which attempts to recreate the conditions under which the original bug was observed. The original conditions such as input data, message exchange sequence and auxiliary test processes, are

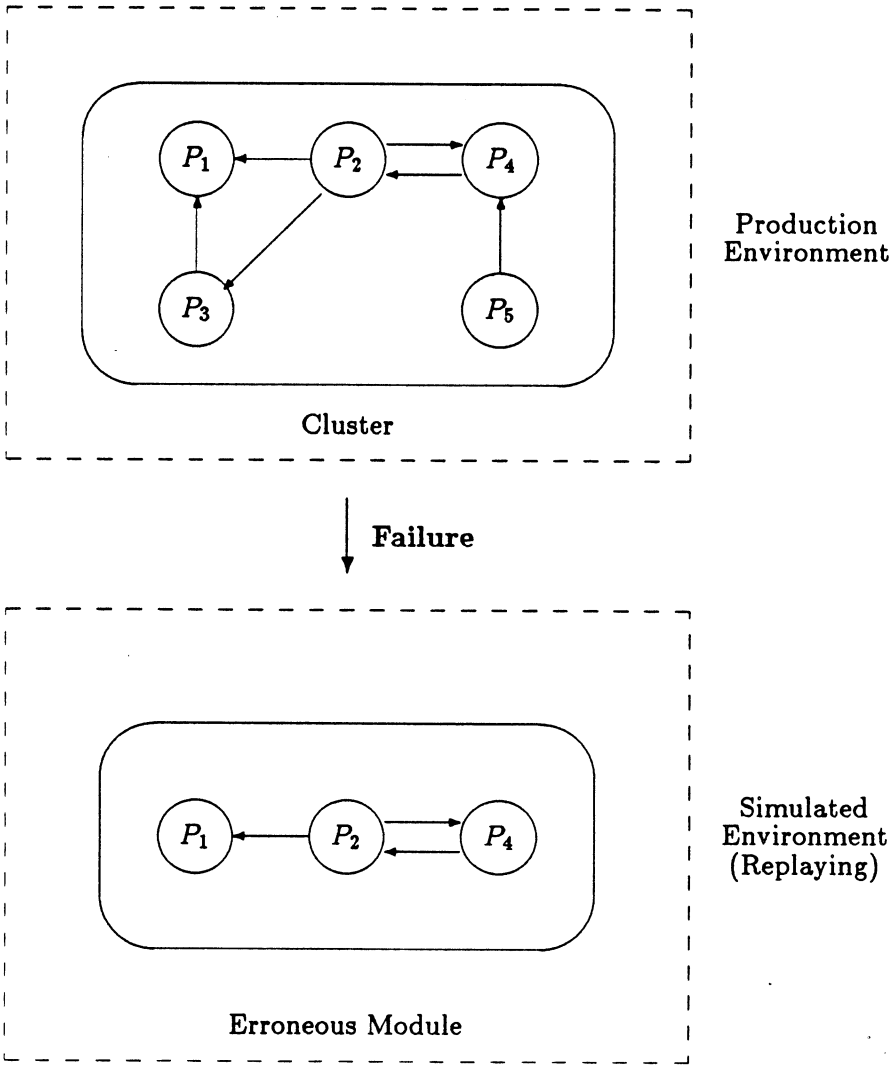

Figure 5: Two-phase Debugging

reconstructed and execution is simulated using the techniques of replaying and controlled execution. This simulation is needed in many cases, since it is not feasible to collect all the necessary information during phase one. Various tests may be needed to reveal the bug. After study of the test results, it should be possible to locate the fault.

Creating the artificial environment for the second phase may be a difficult and time-consuming task, especially if the error is one of synchronization. Also, this methodology encounters problems similar to those of monitoring. Garcia-Molina [Garcia-Molina84] suggested the use of *wraparound tracing* to keep the trace storage of reasonable size. That is, the storage can be written in a circular fashion; the newest data is written over the oldest data. High level tracing, like the clustering technique in monitoring, was suggested to reduce the processing time.

7 Three General Approaches for Distributed Debugging

A general approach provides a high-level abstract model to guide the development of a complete DDS. It also gives guidelines on how to use the basic techniques and methodologies to capture and manipulate debugging information. We describe three general approaches in this section. The first one is the database approach, which emphasizes information manipulation. The second one is the behavioural approach, which emphasizes cluster behaviour manipulation. The third one is the AI approach, which considers automation of the debugging process, using some sort of knowledge base.

7.1 The Database Approach

This approach views debugging as performing queries and updates on a database that contains program information (*e.g.*, system specification, source code and symbol table) as well as execution information. Figure 6 shows how a DDS which adopts this approach relates to

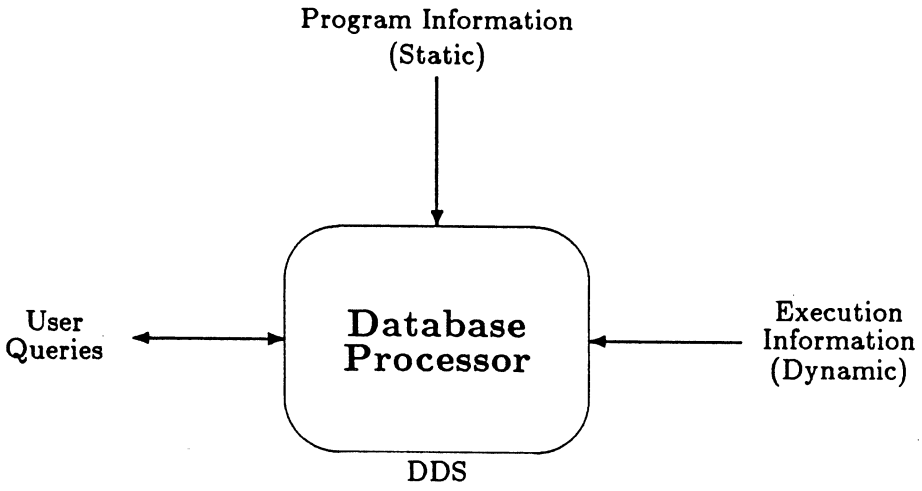

Figure 6: The Database Approach

the debugging environment. Program information is stored in a database. Execution information is collected and forwarded to the DDS as database update entries. A programmer tries to understand the program and its execution by making database queries. The intent is that answers to the queries provide clues to the fault. Thus, in this approach, debugging is a matter of understanding the execution as well as the specification of a distributed program.

The use of the database paradigm offers several advantages. First, we can make use of the functions available in a typical database system such as storage management, data retrieval and concurrent access control. Therefore, a debugger which adopts this approach is conceptually rather simple. It basically monitors the execution and forwards run-time information to the database system. Second, the database system usually allows a rich set of queries on the values of and relationships among the data. Third, the query interface of the database system can be used as the user interface of the debugger. Fourth, since program information and execution information are kept in a database, the programmer can access them in an integrated fashion for debugging. Fifth, execution information in various processors actually constitutes a distributed database. Therefore, we can rely on the techniques that have been developed for managing and querying a distributed database to manipulate it.

Some examples of this approach are Bugnet [Curtis82], OMEGA [Powell83], Garcia-Molina's debugger [Garcia-Molina84], Snodgrass' monitor [Snodgrass84] and the Program Visualization System [Schwan86].

In Bugnet, an event monitor in each processor captures local execution information and then sends it to a central node. The central node stores this execution information in a database for later study, mainly replaying. However, the database is only a central repository for information, not an actual database system supporting queries.

In OMEGA, program information is placed in a program database and execution information is captured by a program monitor. Through a central database system, OMEGA provides a

uniform relational query facility for programmers to access all the information. Although the debugger deals mainly with sequential programs, its designer also considered the possibility of managing distributed data.

Garcic-Molina's debugger incorporates the idea of using a relational database system to store and examine run-time traces, although the facility was not implemented. The use of a database query language to extract useful information was also presented.

Snodgrass' monitor uses a relational database, extended to incorporate time, to capture execution information. It provides a query language, called TQuel, for programmers to issue high-level queries about the program behaviour. The system is sophisticated and was actually implemented to monitor programs on a tightly-coupled multiprocessor system. However, due to central manipulation of the execution information, the maximum number of processes is bounded (say, approximately fifty).

The Program Visualization System is implemented using the INGRES relational database. It represents a distributed program as a set of objects and relationships among these objects. An object is an abstraction of a program block at any level of granularity. A relationship is a binary relation between two objects. This representation is stored in a database. A view of the program is a sequence of queries regarding the objects and relationships available in the database. A sophisticated user interface which employs multiple windows and bitmapped graphics is provided for view construction and display. Multiple, alternative and complementary views of a single distributed program can be constructed. In this way, a programmer usually has a better understanding of the characteristics of the distributed program. However, program views are static, in contrast to basic database views. While the database will change during execution, constructed program views remain invariant with respect to those changes. Therefore, a requirement for consistency between each program view and the database would force the programmer to re-derive the program views whenever the database is updated. This feature has the advantage of obtaining a series of "snapshot" views of the program. However, it relies on

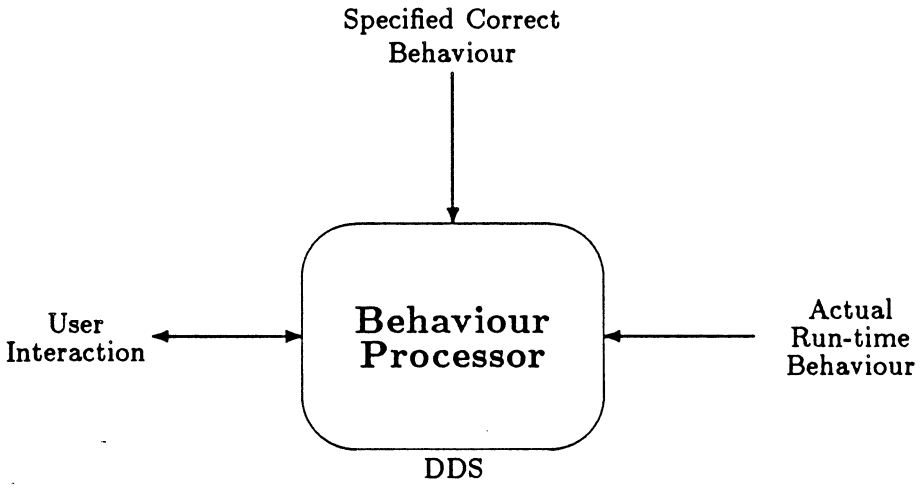

Figure 7: The Behavioural Approach

the programmer to update the program views to avoid inconsistency.

7.2 The Behavioural Approach

In this approach, debugging is viewed as the process of observing a sequence of actual events from the activity of a cluster, and comparing the actual behaviour to the expected behaviour as defined by the user. Monitoring is used to capture the execution events. Then, the debugger deduces the actual behaviour from the events and compares it with the expected behaviour. If there is a discrepancy between them, the debugger notifies the programmer of the possible occurrence of an error, or performs some actions predefined by the programmer. Finally, the discrepancy is presented to the programmer who attempts to identify the fault. The general structure of a DDS which employs this approach is shown in Figure 7.

Current research also emphasizes the development of machine-based tools to aid the programmer in understanding program behaviour. This is achieved by providing for the creation of multiple viewpoints of a distributed program and the ability to view the program at different levels of abstraction. The notion of *behavioural abstraction* introduced by Bates *et al* [Bates83b] is one example. A set of primitive events constitutes the lowest level of program behaviour. This may provide a particular viewpoint of the program. Behavioural abstraction permits the debugger to define alternative, high-level viewpoints of the program and then to observe the program behaviour from those viewpoints.

Usually, this approach provides a language facility for specifying events, behaviour and actions. It may include a *behavioural definition language* to define expected behaviour (or abnormal behaviour), based on pre-defined or user-defined events, and an *action language* to specify actions. For behavioural abstraction, it may also include a mechanism in the behavioural definition language to specify alternative high level viewpoints. This idea is similar to that of multiple views in the Program Visualization System, mentioned in the database approach. The main difference is that behavioural abstraction is based on a behavioural description of a cluster, while multiple views are constructed from objects and relationships of the corresponding distributed program.

Using a behavioural definition language, we can use a debugger to impose a redundant specification of the expected behaviour on the program that is being debugged. The debugger then checks when the program has reached some point in its execution history, at which point other debugging tools can be applied to report when the behaviour deviates from the expected behaviour. The reporting mechanism may be invoked selectively, thus avoiding the need for programmers to analyze and filter large volumes of output from the debugger.

The behavioural definition language is more powerful than traditional debugging techniques. For example, a breakpoint can only be used to report when a computation has reached some specific state. However, the programmer must determine by himself where to place a breakpoint.

Also, several execution paths may lead to the same breakpoint, and not all of them are necessarily useful to the programmer. By contrast, in a behavioural definition language, it is possible to express an execution history that can be arbitrarily fine-grained, thus allowing very selective debugging. Another example is that of an assertion statement which can test for state violations during execution. Assertions only test for state violations at some single points in the execution history. Behavioural definition languages, on the other hand, are able to express the allowed sequences of operations and the flow of information.

The purpose of the action language is to specify the operation to be done after incorrect behaviour of the cluster has been detected, for example, the actions may include halting the cluster, logging the event or reporting the event to the programmer, etc.

In recent years, there have been a number of studies in distributed debugging using this approach. Some of the examples are Bates' Debugging Monitor [Bates82], the Path Rules Debugger [Bruegge83], the ECSP Debugger [Baiardi86], and the Task Graph Language and Token Lists in Mininet [Livesey83].

Bates' Debugging Monitor uses a language called EDL to describe the behaviour of a cluster. An entry in EDL is an *event definition* which describes how an instance of an event might occur and what the attributes of the instance are. Each event definition is composed of a heading (to identify the event) and three types of clauses: the *is* clause, which defines an event expression (regular expression) over previously defined events; the *cond* clause, which places constraints on the events described in the *is* clause; and the *with* clause, which defines attributes (such as process identifier) for each instance of the event. A programmer can define hierarchical events from primitive events or defined events. However, it does not include a construct to explicitly show such an event hierarchy; the whole description is only a collection of definitions in any order as long as an event definition does not have a forward reference to other event definitions.

The Path Rules Debugger introduces a modified version of path expressions called *Path*

Rules to monitor the behaviour of a computation and also trigger actions to be performed (halt the execution, assign a value to a debugger variable, or a sequence of debugger commands) in the presence or absence of a certain event. A path rule has an event sequence recognition part called a *generalized path expression* (GPE) to describe an event. A programmer can manipulate path variables in GPE in order to describe attributes of an instance of the event at runtime. A path rule also has an action part called a *path action* to define actions associated with the event. The path rules can detect violations of the execution sequence automatically on the level of abstraction currently used by the programmer. A GPE can be encapsulated within another GPE. However, The Path Rules language does not have a notion of grouping processes into clusters.

The ECSP Debugger has the notion of grouping processes in a hierarchical structure. In order to support a description of the expected behaviour at various levels of abstraction, the description of a process P hides the interactions between $P_1, ..., P_n$ activated by a parallel command of P (ECSP is based on the Communicating Sequential Processes model). The description is given by a set of *Event Specifications*, such as communications on an IN/OUT port and process termination. A *Behaviour Specification* defines a partial order on the events of a process to describe the allowed sequences of interactions. It also includes a set of assertions (such as process state, the values of counters associated with event specifications), each to be evaluated after the occurrence of a given sequence of interactions. The main features of the debugger are the association of a behaviour specification with a process, the feasibility of defining events at various levels of abstraction and the strong connection between the debugger and the semantics of ECSP. However, the latter feature precludes porting the debugger to other languages.

The Task Graph Language (TGL) is based on a general communication model in which IPC is done only by message exchanges. It is intended for expressing expected IPC patterns in a task graph. Using TGL, a programmer can specify the connectivity (which tasks can communicate), sequencing (which messages must precede or follow one another) and mutual exclusion (which

message sequences must not interfere). A compiler parses a program written in TGL to generate a set of *Token Lists*. A token list is a distributed representation of the task graph. Each process receives a list of tokens, containing information needed to synchronize the IPC operations of that process with those processes with which it exchanges messages. A runtime package, called the *Token List Mechanism*, enforces the constraints specified in TGL by checking the token lists; a *send* token must match a *receive* token before the token list mechanism allows the IPC operation to proceed. The TGL is easy to use and the token list mechanism is not complicated. However, it does not check the content of messages, and also has no construct to describe various levels of abstraction of behaviour.

7.3 The AI Approach

This approach uses artificial intelligence techniques to help in detecting bugs, suggesting possible causes of bugs, and even proposing corrections. It tries to release people from some of the intellectual work of debugging. Figure 8 shows the basic structure of a DDS using the AI approach. One idea is to use an expert system for debugging. The intent is to capture the expertise of an experienced programmer in a knowledge base and to make it available to all programmers. The main potential advantage is that it can serve as an intelligent assistant to the programmer for reasoning and developing fault hypotheses. The programmer is relieved of the burden of tracing the execution of the program or of carrying out symbolic execution.

There are apparently no existing debuggers which use AI techniques directly to debug distributed programs. However, there are some systems which have been developed for typical program debugging at the source level, such as the Fault Localization System (FALSOY) [Sedlmeyer83], a knowledge-based programming assistant [Harandi83], and PROUST [Johnson84]. Seviora provides a good general study of such knowledge-based program debugging systems [Seviora87].

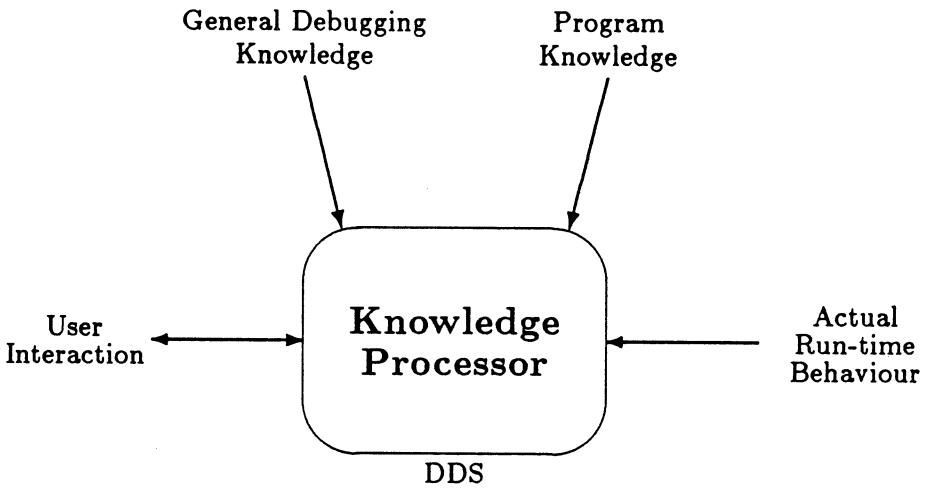

Figure 8: The Artificial Intelligence Approach

An example closer to our interests is the Message Trace Analyzer (MTA) developed by Gupta and Seviora [Gupta84]. MTA is knowledge-based system designed for debugging a concurrent real-time system in which all activities are encapsulated in processes and their interactions are only via message exchanges. Given a correctly integrated and static trace of message events for an execution, MTA consults a knowledge base to detect any illegal message sequence and its cause. Once MTA concludes its work and communicates its findings to a user, the user can ask MTA to explain its reasoning in various levels of detail.

The knowledge base consists of two types of knowledge. The first type is the domain knowledge of process-structured systems in general. The second type is system-specific knowledge which is an abstract functional description of the system being debugged. In an implementation mentioned in [Gupta85], MTA requires about 20 minutes of VAX-11/780 CPU time to analyze a trace of 2300 messages. MTA works on the assumption that an accumulated trace would exhibit an abnormality which would help to localize the fault. Once the fault is localized to within a relatively small part of the system, it can be identified at source level and removed. Thus, MTA only localizes a failure rather than the fault itself. Also, it assumes that failures are known at the system level. These limitations restrict MTA to only well-known as well as well-defined system failures. Moreover, if we consider a distributed environment rather than the centralized concurrent environment assumed by MTA, a globally integrated message trace is very difficult to obtain, thus invalidating the basic assumption of MTA. However, this work is impressive as it demonstrates the capability of AI techniques for debugging complicated concurrent systems.

7.4 Comments

Generally speaking, these three approaches are based on different divisions of the debugging task between a programmer and the debugger. The database approach deals only with program information and leaves the programmer to observe the error and locate the fault, by formulating the

queries and by analyzing the answers. The behavioural approach is able to detect errors and to show discrepancies between actual and expected behaviour. It is a step away from the database approach towards provision of automatic error detection based on behaviour description. The event sequence and the discrepancy also give more clues for fault location. The programmer is required to locate and repair the fault by himself. The AI approach is very attractive as it has the potential to release the programmer from much debugging work. However, current research in AI is mainly on diagnosis, and automatic fault repair is an open area. Also, research on the diagnosis problem is still primitive. More importantly, the evaluation speed of such AI implementations seems generally poor. This limits their application to off-line analysis, rather than on-line real-time debugging. In other words, further research in AI is required before we can consider this approach seriously. Actually, none of the approaches suggests any way to locate and repair a fault. Clearly, much research will be required to develop a systematic approach for fault location and then fault repair.

Some debuggers do not adopt any approach for debugging. They simply provide sets of basic debugging tools based on the techniques mentioned in Section 5. For example, CBUG [Gait85] only provides a set of basic tools for supporting snapshot dumps, conditional breakpoints, single-stepping, monitoring and tracing. No approach is suggested to deal with the debugging process itself.

8 Domain Interactions

8.1 The Graphical Environment

Studies of the human visual system and memory indicate that graphical representations are often better than textual or numeric representations. In recent years, graphical hardware has become quite cheap, and current I/O technologies can provide adequate data transfer rates. So-

phisticated workstations with high-resolution color graphics displays create multiple dimensions for presenting information interactively; colour and multiple window facilities introduce extra dimensions to displaying information in an elegant way.

Distributed debugging must present a large amount of system state information to programmers in a concise way. Such multi-dimensional display capabilities meet this requirement and so graphical representation has become more and more important for human-debugger interaction. Almost all debuggers developed recently on such workstations have graphical interfaces. Naturally, the idea of *graphical debugging* arises.

Graphical debugging is a debugging style in which all concepts and objects are displayed using a graphical representation. By using a graphical debugger, a programmer can easily observe cluster behaviour and detect bugs. To design a graphical debugger, the following criteria should be addressed [Bei85]:

1. *Naturalness:* The debugger should only use diagrams to display that information which can be naturally represented by diagrams.

2. *Simplicity:* The debugger should use simple graphs to display information so that users can easily understand the information.

3. *Precision:* The debugger should display information precisely, so that the users are not confused in detecting and locating bugs.

4. *Clarity:* The debugger should display information clearly and highlight important information.

The superiority of graphical display to textual display is based on an assumption that a programmer employs mental imagery to understand system activities [Joyce87]. A line of text describing an event is harder to understand, because it must first be translated into the programmer's mental model. In contrast, graphical depiction of such an event can be closer to

the user's mental model and require little or no translation. Also, a graphic display is more effective for representing the structure and dynamic behaviour of a cluster. In other words, it seems easier for the programmer to assimilate the information presented by graphics than to assimilate the same information presented by the text console. However, textual display still has its strong points. A textual display can present a sequence of events with accurate and detailed description, while a graphical display does not generally display all events in detail and provides no indication of the sequence of events that lead to a particular state. Therefore, even though we emphasize the effectiveness of graphical display, we should not discard textual display in graphical debugging.

Although many people claim that graphical debugging tools are better than textual debugging tools, little empirical evidence exists to support this claim. Moran and Feldman show in an experiment that, for simple bugs which occur early in an event sequence within a concurrent Ada program, the type of debugger display (graphical or textual) has no statistically significant impact on debugging time [Moran85]. The lack of use of debugger visualization could account for the lack of statistically differences in debugging time; the bugs and the programs were so simple that subjects in the experiment did not fully utilize the debuggers. Further experiments are necessary to determine if a graphical debugger is more advantageous than a textual debugger for hard bugs in a complex, distributed program.

Some examples of graphical debuggers are CGDS [Bei85], IDD [Harter85] and the Jade Monitoring System [Joyce87].

CGDS uses a process and message graph to show dynamic IPC activities at runtime. It uses interactive graphic techniques, such as multiple windows, pop-up menus, and pointing devices to provide the programmer with a convenient interface. The use of default settings makes its use simple. CGDS adopts a master-slave scheme. A master debugger is responsible for handling user interaction (breakpoint setting, single step execution, behaviour modification) and graphical display (zoom and shrink a cluster window). The slave debuggers on other processors capture

local IPC activities and report them to the master debugger. No textual display is supported.

IDD provides a formated display, like a grid, for viewing message traffic. The horizontal axis of the display is time and the vertical axis is process number. An IPC activity is represented by a line connecting two points on the display. Thus, we can easily observe a sequence of IPC activities. However, it does not show the structure of the cluster. IDD also uses multiple windows, pop-up menus and pointing devices at the human-debugger interface. A number of graphics operators are used to focus on messages of interest. For example, the 'Display' operator allows the user to adjust the focus along the time axis as well as the process axis. The 'Local' operator creates a separate window for displaying internal process information in a textual format.

The Jade monitoring system provides both textual and graphical displays. The textual display reports each event in the event stream with one or two lines of text. The graphical display, which is called Mona, shows the structure of a cluster with an animated graphical view of the event stream. Interactive graphical techniques similar to CGDS are used to deal with programmer interaction. Furthermore, it provides an event line display, which is similar to the IDD display with two main differences. First, while the IDD display shows the real time of events along the time axis, the horizontal axis of the event line display is divided into a number of event intervals. An event interval demarcates adjacent events in the event stream; it has no relationship to the passage of real time. Second, instead of drawing a line, the Jade monitoring system places corresponding events in appropriate time-event locations.

8.2 The Integrated Environment

A significant change is beginning to occur in the programmer's environment and therefore also in debugging. Nowadays, software development is highly interactive, with extensive use of graphics. Workstations allow powerful cost-effective computing tailored to particular tasks. Expert

systems, as software consultants, provide information about software development tasks. The most significant change may come if knowledge engineering and functional programming replace present procedure-oriented software engineering techniques. To accommodate such concepts for debugging as well as other phases of software development, integrated environments have been introduced. In other words, a debugger is not developed as an independent module. Rather, the debugger is integrated with other software development tools.

An integrated environment can be characterized as follows [Fairley83]:

1. It is a versatile collection of analysis, design, implementation, testing, debugging, and maintenance tools.
2. Its tools are integrated, with frequent and easy interactions among them.
3. It provides a consistent user interface among the tools.
4. It has common representations of all program and execution information.
5. It encourages good software development practice.

Perhaps the most important feature is that there is no need for explicitly switching among various programming tasks, such as editing, compiling, testing and debugging. Thus, this allows a programmer to manage these tasks simultaneously with ease. In addition, with an information base, an integrated environment can store and retrieve information about the program and its previous development tasks (this resembles the database approach). This helps debugging by providing a central information source for the programmer to understand the program better and to trace the impact of intended program changes easily. While there are some existing integrated environments for sequential programs, such as Smalltalk [Byte81] and the Cornell Program Synthesizer [Teitelbaum81], there is apparently no such environment for distributed programs. As distributed programs are more complex than sequential ones, it is particularly important to provide an integrated environment for distributed program development.

9 Concluding Remarks

9.1 Summary

In this paper, we first address the issues in sequential and distributed debugging. While many issues in sequential debugging are still relevant to distributed debugging, the latter has some unique problems due to the nature of distributed environments. The basic problems are lack of global knowledge, and management of a large system state space with multiple foci of control. We then discuss the need for distributed debugging systems. Because distributed debugging is a very complicated process and because there is a lack of appropriate tools to help programmers debug their distributed programs, the need for an elegant distributed debugging system is obvious and urgent.

We introduce a framework for studying distributed debugging in a systematic and less complicated way. This framework consists of two parts. The first part is a DDS model which describes the role of a DDS and its interaction with the debugging environment. Three major domains of the debugging environment are identified—the program domain, the human domain and the distributed domain. The model views the relationships between the DDS and each domain as a communication interface. The second part of the framework is a decomposition of problems in distributed debugging. We divide the problem space into three areas. The main area is DD model development. The other two are domain interaction and system support. These areas are further decomposed into smaller components.

Then we concentrate on research in DD model development. The DD model consists of three main parts: basic techniques, methodologies and approaches. Basic techniques, such as tracing, breakpoints and monitoring, are the practical techniques which can be implemented in a debugging tool. Methodologies guide us to decompose the debugging process into manageable steps and hence facilitate the process. Top-down, bottom-up and two-phase debugging are

three methodologies proposed by researchers. Approaches refer to the philosophical views of the debugging process as a whole. They provide abstract models of debugging and give guidelines for capturing and analysing debugging information. In the literature, we can find three typical approaches: the database approach, the behavioural approach and the AI approach.

Finally, we briefly discuss two aspects of domain interaction: the graphical environment and the integrated environment. Clearly, people play a very important role in debugging. Therefore the debugger should communicate with them effectively. A graphical environment can provide an excellent interface between human and debugger. The power of graphics is especially useful in distributed debugging. On the other hand, we should consider distributed debugging as a part of the software development life cycle. The study of integrated environments provides us with some guidelines for developing a distributed debugging system which can be integrated with other software development tools.

9.2 Research Directions

The main objective of distributed debugging is to provide an effective and efficient DDS for programmers to debug distributed programs. To do so, we should study the issues described in the previous sections and find appropriate solutions. Also, we should integrate individual debugging facilities in a systematic way by considering their functions and interfaces. Our framework for studying DDS is intended to serve these purposes. Based on this framework, we suggest that researchers explore distributed debugging in at least three main areas. In the following discussion, we first briefly describe possible research in each area and then discuss some related considerations.

The first area is to develop a DD model and study various approaches, methodologies and basic techniques to tackle the process of distributed debugging. It focuses on the design issues of distributed debugging. Although there are many studies on different aspects of distributed

debugging, we do not have general principles for designing DD models and evaluating them systematically.

The second area concerns the interaction between a DDS and its external environment. Basically, the communication issues are emphasized. This area is particularly important for the human domain and the H interface. Research here includes the role of people in distributed debugging, human behaviour during the debugging activity, human-oriented representations of debugging objects, effective communication between human and DDS, and human factors related to the understanding of concurrent dynamic execution. While there are some interesting studies on the psychological aspect of sequential debugging [Gould75], the psychological study of distributed debugging is an unexplored subject. On the other hand, the D interface has been recognized as difficult to study. The issues are basically time, delay, synchronization, large state space and no global view of process clusters. These problems are only partially solved, and there is still much room for improvement. In the program domain, many researchers have concentrated on developing languages to express errors, faults, system behaviour and assertions. We notice that a large portion of work on the behavioural approach is in this category. Although a number of languages have been introduced, there is still no general principle to guide the development of such languages and to evaluate them.

The third area is to look at the issues in implementing a DDS. The issues may include provision of global state management, consistent breakpoints, handling monitoring data, information filtering and information abstraction. Also, the architecture for a DDS should be considered. For example, a master-slave scheme may generate performance bottlenecks at the master node, but permits centralized processing of debugging information. The study of DDS architecture may include the interface between local and distributed debuggers, decomposition of the DDS into functional modules, and coordination of these modules in a distributed environment.

Before we develop a DDS, we should define the characteristics of its external environment clearly. The DDS model we have introduced divides the environment into three domains. It

is useful to study the characteristics of these domains and formulate appropriate models. For example, a distributed computational model gives us a picture of how the components of a distributed program compute and communicate. Based on this picture, we can make assumptions about the execution behaviour and also design the D interface effectively. The study of the D, P and H interfaces may give interesting results. If system-independent elements are isolated from all the interfaces, we may be able to construct a portable DDS module. The design and implementation also have a significant impact on the DDS. For example, a poor design of the H interface clearly discourages a programmer from using it and hence degrades its usefulness. An implementation of the D interface which requires a lot of remote accesses definitely reduces the performance level of the DDS.

As we know, debugging generally goes through three steps: error detection, fault location and fault repair. However, some existing distributed debuggers only provide facilities to detect errors, forcing the programmer to take appropriate actions to locate and repair the corresponding fault. Some provide more facilities to localize the possible range of a fault. Very few debuggers actually locate a fault automatically. Needless to say, there is no debugger automated enough to perform the three steps completely. Generally, fault location and repair are difficult and complicated. An error may possibly be associated with multiple faults. The error space and fault space are very large, especially in a distributed environment. Therefore, an extensive search is required for the possible faults, which may consequently lead to the use of AI techniques to deal with the problems. Although there have been some fruitful results on the diagnosis problem [Davis84] [Reiter85] [Poole86], we still have no experience of applying AI techniques to distributed debugging. Furthermore, intelligent fault repair is still a challenging and open area for research.

There are too many types of possible bugs in distributed programs. In order to restrict our scope to a reasonable range, we must consider particular types of bugs in a debugging environment. To define a certain debugging environment, we must construct models for the three

domains of the DDS and make reasonable assumptions. However, even though we may simplify the debugging environment by making appropriate assumptions, there are still a large number of possible bugs in distributed software. Hence, besides the provision of an error detection mechanism which can hopefully capture all bugs, it is a definite advantage to classify them into several classes and consider them one by one. Timing bugs are an interesting class of bugs. They are the bugs which depend upon time or the order of a sequence of events. Definitely, algorithms or procedures to diagnose these bugs provide us with a real challenge. Gordon has done some analytic work for timing bugs [Gordon85], but there is still much room for improvement and generalization.

References

[Anderson81] T. Anderson and P.A. Lee, *Fault Tolerance: Principles and Practice*, Prentice Hall, 1981.

[Baiardi83] F. Baiardi, N. De Francesco, E. Matteoli, S. Stefanini, and G. Vaglini, "Development of a Debugger for a Concurrent Language", *Proceedings of the SIGSOFT/SIGPLAN Software Engineering Symposium on High-Level Debugging, SIGPLAN Notices*, ACM, Vol. 18, No. 8, pp. 98–106, August 1983.

[Baiardi86] F. Baiardi, N.D. Francesco, and G. Vaglini, "Development of a Debugger for a Concurrent Language", *IEEE Transactions on Software Engineering*, Vol. SE-12, No. 4, pp. 547–553, April 1986.

[Bates82] P.C. Bates and J.C. Wileden, "EDL: A Basis for Distributed System Debugging Tools", *The 15th Hawaii International Conference on System Science*, ACM, pp. 86–93, 1982.

[Bates83a] P.C. Bates and J.C. Wileden, "High-Level Debugging of Distributed Systems: The Behavioural Abstraction Approach", *The Journal of System and Software*, Elsevier Science Publicing Co., Vol. 3, pp. 255–265, 1983.

[Bates83b] P.C. Bates, J.C. Wileden, and R. Lesser, "A Debugging Tool for Distributed Systems", *Proceedings of Phoenix Conference on Computers & Communications*, IEEE, pp. 311–315, 1983.

[Bei85] J.N.W. Bei, "CDGS: A Graphical Debugger for Distributed Software", *ICR Report 85-10*, Institute for Computer Research, University of Waterloo, June 1985.

[Birrell85] N.D. Birrell and M.A. Ould, "A Practical Handbook for Software Development", Cambridge University Press, 1985.

[Bruegge83] B. Bruegge and P. Hibbard, "Generalized Path Expressions: A High Level Debugging Mechanism", *Proceedings of the SIGSOFT/SIGPLAN Software Engineering Symposium on High-Level Debugging, SIGPLAN Notices*, ACM, Vol. 18, No. 8, pp. 34–44, August 1983.

[Byte81] Byte, "A Series on Smalltalk", *BYTE: The Small Computer*, Vol. 6, No. 8, August 1981.

[Cargill83] T.A. Cargill, "The Blit Debugger", *The Journal of Systems and Software*, Elsevier Science Publishing Co., Vol. 3, pp. 277–284, 1983.

[Chandy85] K.M. Chandy and L. Lamport, "Distributed Snapshots: Determining Global States of Distributed Systems", *ACM Transaction on Computer Systems*, Vol. 3, No. 1, pp. 63–75, February 1985.

[Cooper87] R. Cooper, "Pilgrim: A Debugger for Distributed Systems", *Proceedings of the 7th International Conference on Distributed Computing Systems*, IEEE, pp. 458–465, September 1987.

[Corsini86] P. Corsini and C.A. Prete, "Multibug: Interactive Debugging in Distributed Systems", *IEEE Micro*, Vol. 6, No. 3, pp. 26–33, June 1986.

[Curtis82] R. Curtis and L. Wittie, "BUGNET: A Debugging System for Parallel Programming Environment", *The 3rd International Conference on Distributed Computing Systems*, IEEE, pp. 394–399, October 1982.

[Davis84] R. Davis, "Diagnostic Reasoning Based on Structure and Behavior", *Artifical Intellgence*, Elsevier Science Publishers, Vol. 24, No. 1-3, pp. 347–410, December 1984.

[Dunn84] R.H. Dunn, *Software Defect Removal*, McGraw-Hill, 1984.

[Enslow78] P.H. Enslow, "What is a 'Distributed' Data Processing System?", *IEEE Computer*, Vol. 11, pp. 13–21, January 1978.

[Ezhilchelvan86] P.D. Ezhilchelvan and S.K. Shrivastava, "A Characterisation of Faults in Systems", *Proceedings of the 5th Symposium on Reliability in Distributed Software and Database Systems*, IEEE, pp. 215–222, January 1986.

[Fairley83] R.E. Fairley, "Integrated Environments (Session Summary)", *Proceedings of the SIGSOFT/SIGPLAN Software Engineering Symposium on High-Level Debugging, SIGPLAN Notices*, ACM, Vol. 18, No. 8, pp. 60–62, August 1983.

[Foudriat85] E.C. Foudriat, "Panel Discussion: Debugging in Distributed Systems", *Proceedings of the 5th International Conference on Distributed Computing Systems*, IEEE, pp. 548, May 1985.

[Gait85] J. Gait, "A Debugger for Concurrent Programs", *Software—Practice and Experience*, John Wiley & Sons, Vol. 15, No. 6, pp. 539–554, June 1985.

[Garcia85] M.E. Garcia and W.J. Berman, "An Approach to Concurrent Systems Debugging", *Proceedings of the 5th International Conference on Distributed Computing Systems*, IEEE, pp. 507–514, May 1985.

[Garcia-Molina84] H. Garcia-Molina, F. Germano Jr., and W.H. Kohler, "Debugging a Distributed Computing System", *IEEE Transactions on Software Engineering*, Vol. SE-10, No. 2, pp. 210–219, March 1984.

[Gordon85] A.J. Gordon, "Ordering Errors in Distributed Programs", *Ph.D Thesis, Technical report 611*, Computer Science Department, University of Wisconsin–Madison, August 1985.

[Gould75] J.D. Gould, "Some Psychological Evidence on How People Debug Computer Programs", *International Journal of Man–Machine Studies*, Academic Press, Vol. 7, pp. 151–182, 1975.

[Gramlich83] W.C. Gramlich, "Debugging Methodology (Session Summary)", *Proceedings of the SIGSOFT/SIGPLAN Software Engineering Symposium on High-Level Debugging, SIGPLAN Notices*, ACM, Vol. 18, No. 8, pp. 1–3, August 1983.

[Gross83] T. Gross, "Session Summary: Distributed Debugging", *Proceedings of SIGSOFT/SIGPLAN Software Engineering Symposium on High-Level Debugging, ACM SIGPLAN Notices*, Vol 18, No. 8, August 1983.

[Gupta84] N.K. Gupta and R.E. Seviora, "An Expert System Approach to Real Time System Debugging", *ICR Report 84-27*, Institute for Computer Research, University of Waterloo, November 1984.

[Gupta85] N.K. Gupta, "An Expert System Approach to Debugging Process Structured System", *Master's Thesis*, Department of Electrical Engineering, University of Waterloo, 1985.

[Gusella83] R. Gusella and S. Zatti, "TEMPO—Time Services for the Berkeley Local Network", *Technical Report UCB/CSD 83/163*, Computer Science Division (EECS), University of California, Berkeley, December 1983.

[Gusella85] R. Gusella and S. Zatti, "An Election Algorithm for a Distributed Clock Synchronization Program", *Technical Report UCB/CSD 86/175*, Computer Science Division (EECS), University of California, Berkeley, December 1985.

[Hamlet83] D. Hamlet, "Debugging Level: Step-wise Debugging", *Proceedings of the SIGSOFT/SIGPLAN Software Engineering Symposium on High-Level Debugging, SIGPLAN Notices*, ACM, Vol. 18, No. 8, pp. 4–8, August 1983.

[Harandi83] M.T. Harandi, "Knowledge-Based Program Debugging: A Heuristic Model", *Proceedings of 1983 Softfair*, pp. 282–288, July 1983.

[Harter85] P.K. Harter, D.M. Heimbigner, and R. King, "IDD: An Interactive Distributed Debugger", *Proceedings of the 5th International Conference on Distributed Computing Systems*, IEEE, pp. 498–506, May 1985.

[Johnson84] W.L. Johnson and E. Soloway, "Intention-Based Diagnosis of Programming Errors", *Proceedings of AAAI 1984*, pp. 162–168, August 1984.

[Jones86] S.H. Jones, R.H. Barkan, and L.D. Wittie, "A Real Time Distributed Debugging System", *Technical Report 86/35*, Department of Computer Science, State University of New York at Stony Brook, December 1986.

[Joyce87] J. Joyce, G. Lomow, K. Slind, and B. Unger, "Monitoring Distributed Systems", *ACM Transactions on Computer Systems*, Vol. 5, No. 2, pp. 121–150, May 1987.

[Laprie85] J.-C. Laprie, "Dependable Computing and Fault Tolerance: Concepts and Terminology", *Digest of the 15th Annual International Symposium on Fault-Tolerant Computing*, IEEE, pp. 2–11, June 1985.

[Lau86] Francis C.M. Lau, "Policies and Mechanisms for Distributed Clusters", *Ph.D. Thesis, ICR Report 86-19*, Department of Computer Science, University of Waterloo, November 1986.

[LeBlanc85] R.J. LeBlanc and A.D. Robbins, "Event-Driven Monitoring of Distributed Programs", *Proceedings of the 5th International Conference on Distributed Computing Systems*, IEEE, pp. 515–522, May 1985.

[LeBlanc87] T.J. LeBlanc and J.M. Mellor-Crummey, "Debugging Parallel Programs with Instant Replay", *IEEE Transactions on Computers*, Vol. C-36, No. 4, pp. 471–482, April 1987.

[Le Lann77] G. Le Lann, "Distributed Systems—Towards A Formal Approach", *Information Processing 77*, IFIP, pp. 155–160, 1977.

[Liffick85] B.W. Liffick, *The Software Developer's Sourcebook, From Concept to Completion: The Essential Reference*, Addison-Wesley, 1985.

[Livesey83] J. Livesey and E. Manning, "Protection and Synchronisation in a Message-Switched System", *Computer Networks*, Elsevier Science Publishers, Vol. 7, No. 4, pp. 253–267, August 1983.

[Miller85] B.P. Miller, C. Macrander, and S. Sechrest, "A Distributed Programs Monitor for Berkeley UNIX", *Proceedings of the 5th International Conference on Distributed Computing Systems*, IEEE, pp. 43–54, May 1985.

[Model79] M.L. Model, "Monitoring System Behaviour in a Complex Computational Environment", *Ph.D Thesis*, Computer Science Department, Stanford University, January 1979.

[Moran85] M. Moran and M.B. Feldman, "Toward Graphical Animated Debugging of Concurrent Programs in Ada", *Proceedings of International Symposium on New Directions in Computing*, IEEE, pp. 344–351, August 1985.

[Mullerburg83] M.A.F. Mullerburg, "The Role of Debugging within Software Engineering Environments", *Proceedings of the SIGSOFT/SIGPLAN Software Engineering Symposium on High-Level Debugging, SIGPLAN Notices*, ACM, Vol. 18, No. 8, pp. 81–90, August 1983.

[Myers76] G.J. Myers, "Software Reliability: Principles and Practices", John Wiley & Sons, 1976.

[Myers79] G.J. Myers, "The Arts of Software Testing", John Wiley & Sons, 1979.

[Poole86] D.L. Poole, "Default Reasoning and Diagnosis as Theory Formation", *Technical Report CS-86-08*, Department of Computer Science, University of Waterloo, March 1986.

[Powell83] M.L. Powell and M.A. Linton, "A Database Model of Debugging", *The Journal of System and Software*, Elsevier Science Publicing Co., Vol. 3, pp. 295–300, 1983.

[Pressman82] R.S. Pressman, "Software Engineering: A Practitioner's Approach", McGraw-Hill, 1982.

[Reiter85] R. Reiter, "A Theory of Diagnosis form First Principles", *Technical Report No.187/86*, Department of Computer Science, University of Toronto and The Canadian Institute for Advanced Research, December 1985.

[Schwan86] K. Schwan and J. Matthews, "Graphical Views of Parallel Programs", *SIGSOFT Software Engineering Notes*, ACM, Vol. 11, No. 3, pp. 51–64, July 1986.

[Sedlmeyer83] R.L. Sedlmeyer and W.B. Thompson, "Knowledge-based Fault Localization in Debugging", *Proceedings of the SIGSOFT/SIGPLAN Software Engineering Symposium on High-Level Debugging, SIGPLAN Notices*, ACM, Vol. 18, No. 8, pp. 25–31, August 1983.

[Seviora87] R.E. Seviora, "Knowledge-Based Program Debugging Systems", *IEEE Software*, Vol. 4, No. 3, pp. 20–32, May 1987.

[Smith85] E.T. Smith, "A Debugger for Message-based Processes", *Software—Practice and Experience*, John Wiley & Sons, Vol. 15, No. 11, pp. 1073–1986, November 1985.

[Snodgrass84] R. Snodgrass, "Monitoring in a Software Development Environment: A Relational Approach", *Proceedings of the SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, SIGPLAN Notices*, ACM, Vol. 19, No. 5, pp. 124–131, May 1984.

[Spezialetti86] M. Spezialetti and P. Kearns, "Efficient Distributed Snapshots", *The 6th International Conference on Distributed Computing Systems*, IEEE, pp. 382–388, May 1986.

[Springer85] S.P. Springer and G. Deutsch, "Left Brain, Right Brain", *Revised Edition, Chapter 11*, W.H. Freeman and Company, 1985.

[Tanenbaum85] A.S. Tanenbaum and R.V. Renesse, "Distributed Operating Systems", *ACM Computing Surveys*, Vol. 17, No. 4, pp. 419–470, December 1985.

[Teitelbaum81] T. Teitelbaum, T. Reps, and S. Horwitz, "The Why and Wherefore of the Cornell Program Synthesizer", *ACM SIGPLAN Notices*, Vol. 16, No. 6, pp. 8–16, June 1981.

[Weinberg71] G.M. Weinberg, "The Psychology of Computer Programming", Van Nostrand Reinhold, 1971.

[Wittie85] L. Wittie and R. Curtis, "Time Management for Debugging Distributed Systems", *Proceedings of the 5th International Conference on Distributed Computing Systems*, IEEE, pp. 549–550, May 1985.