

# Printing Requisition / Graphic Services

77479

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-88-44

DATE REQUISITIONED

Aug. 23

DATE REQUIRED

ASAP

ACCOUNT NO.

1126601241

REQUISITIONER - PRINT

J. BLACK

PHONE

4459

SIGNING AUTHORITY

S. DeAngelis / J. Black

MAILING INFO -

NAME

DEPT.

C.S.

BLDG. & ROOM NO.

DC 2314

☒ 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	29	NUMBER OF COPIES	20
TYPE OF PAPER STOCK			
<input checked="" type="checkbox"/> BOND	<input type="checkbox"/> NCR	<input type="checkbox"/> PT.	<input checked="" type="checkbox"/> COVER
<input type="checkbox"/> BRISTOL	<input checked="" type="checkbox"/> SUPPLIED		
PAPER SIZE			
<input checked="" type="checkbox"/> 8 1/2 x 11	<input type="checkbox"/> 8 1/2 x 14	<input type="checkbox"/> 11 x 17	<input type="checkbox"/>
PAPER COLOUR		INK	
<input checked="" type="checkbox"/> WHITE	<input type="checkbox"/>	<input type="checkbox"/> BLACK	<input type="checkbox"/>
PRINTING		NUMBERING	
<input type="checkbox"/> 1 SIDE	<input checked="" type="checkbox"/> 2 SIDES	FROM	TO
BINDING/FINISHING			
<input checked="" type="checkbox"/> COLLATING	<input checked="" type="checkbox"/> STAPLING	<input type="checkbox"/> HOLE PUNCHED	<input type="checkbox"/> PLASTIC RING
FOLDING/PADDING		CUTTING SIZE	

Special Instructions

Math fronts & backs enclosed.

COPY CENTRE

OPER. NO. BLDG. NO. MACH. NO.

DESIGN & PASTE-UP

OPER. NO. TIME LABOUR CODE

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	QUANTITY	OPER. NO.	TIME	LABOUR CODE
P M T				C 0 1
P M T				C 0 1
P M T				C 0 1

PLATES	QUANTITY	OPER. NO.	TIME	LABOUR CODE
P L T				P 0 1
P L T				P 0 1
P L T				P 0 1

STOCK	QUANTITY	OPER. NO.	TIME	LABOUR CODE
				0 0 1
				0 0 1
				0 0 1
				0 0 1

BINDERY	QUANTITY	OPER. NO.	TIME	LABOUR CODE
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				B 0 1

OUTSIDE SERVICES

\$ COST



Carleton University  
Ottawa, Canada K1S 5B6

Feb. 14/89

Research Report Secretary  
Dept. of Comp Sci.  
Univ. of Waterloo.

I would appreciate your ~~returning~~ me the  
following reports (cheque Enclosed)

CS-88-36 "Fast String Matching with  
k Mismatches" — Baerza-Yates & Gonnet

CS-88-37 "New Algorithms for Pattern Matching  
with ~~not~~ or without mismatches"  
— Baerza-Yates & Gaston Gonnet

CS-88-44 A Study of Distributed Debugging  
— Chung, Block, Manning

received cheque  
& sent reports  
IUTS

Feb. 17/89

Yours Sincerely, Prof. B. PAGUREK

To

From

Date

memo

University of Waterloo

25 copies

88-44

Nancy Riley  
CCNG

DC 3561

sent Feb. 2/89

July 14, 1989

University of Waterloo  
Department of Computer Science  
Waterloo, Ontario N2L 3G1  
Attn: Technical Report Secretary

Dear Sir/Madam:

I would like to place an order for report #CS-88-44, "A Study of Distributed Debugging," by W.H. Cheung, J.P. Black, and E. Manning. Please send the order to the following address:

Marcie Palmer  
LSTC  
2100 E. St. Elmo Street  
B30E/9610  
Austin, TX 78744

I am enclosing a check in the amount of \$2.00 for the expenses. If there are any problems with my order, please don't hesitate to contact me. I can be reached at (512)448-5759.

Thank you,



Marcie Palmer

*Sent  
July 31*

REMITTANCE ADVICE

**PURDUE  
UNIVERSITY**

West Lafayette, Indiana

5-03-89 10609808

LI BR LB900-5550

INV.-REF DATE	INVOICE NUMBER	REFERENCE	VOUCHER NO.	AMOUNT		
				GROSS	DISCOUNT	NET
40589	INV040589		9140412	506		506
TOTAL				506		506

REFER TO CHECK AND VOUCHER NUMBER WHEN CORRESPONDING IN REGARD TO THIS PAYMENT  
DETACH THIS STUB BEFORE DEPOSITING CHECK

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

April 5, 1989

Purdue University Libraries  
Fiscal Dept.  
Stewart Center Rm. 264  
West Lafayette, IN  
47907

**INVOICE**

Purchase Order No. 9P08172

**REPORT(S) ORDERED**

CS-88-44 (2 copies)

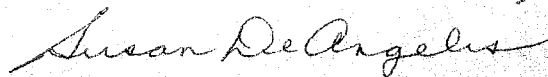
**TOTAL COST**

\$6.00 (This price includes postage)

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 (Mrs.)  
Research Report Secretary  
Computer Science Dept.

/sd

Encl.

## PURCHASE ORDER



**PURDUE UNIVERSITY LIBRARIES**  
FISCAL DEPT. STEWART CENTER RM. 264  
WEST LAFAYETTE, IN 47907

TO:

University of Waterloo  
Dept. of Computer Science  
Waterloo, Ontario N2L 3G1, CANADA

03/07/89

PURCHASE ORDER NUMBER	9908172
THIS NUMBER MUST APPEAR ON ALL INVOICES, PACKERS & CORRESPONDENCE	

PURCHASE ORDER NO.	ORDER DATE	VENDOR CODE	EST. PRICE
9908172	03/07/89	QNW	\$4.00

## DESCRIPTION

Study of Distributed Debugging, by W.H. Cheung, J.P. Black, E. Manning  
CS-88-44  
2 copies

DEST DEPT CS DEST STAFF Spafford/Denill

S Purdue University Libraries 9908172  
H Acquisitions Dept  
I Stewart Center  
P West Lafayette, IN 47907  
T  
O

S PURDUE UNIVERSITY LIBRARIES  
I FISCAL DEPT. STEWART CENTER RM. 264  
L WEST LAFAYETTE, IN 47907  
T PHONE 317-494-2908  
O

EXEMPT FROM INDIANA SALES TAX  
CERTIFICATE # 85-60020410047

FORM LB-1

VENDOR

# Printing Requisition / Graphic Services

60814

- 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 3431.

TITLE OR DESCRIPTION <b>CS-88-44</b>		DATE REQUISITIONED <b>April 12/89</b>	DATE REQUIRED <b>ASAP</b>	ACCOUNT NO. <b>1126601241</b>
REQUISITIONER - PRINT <b>J. BLACK</b>		PHONE <b>4459</b>	SIGNING AUTHORITY <b>J. Black</b>	
MAILING INFO - <b>SUE DEANGELIS</b>	NAME	DEPT. <b>C.S.</b>	BLDG. & ROOM NO. <b>DC 2314</b>	<input checked="" type="checkbox"/> DELIVER <input type="checkbox"/> 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 <b>26</b>	NUMBER OF COPIES <b>30</b>
TYPE OF PAPER STOCK <input checked="" type="checkbox"/> BOND <input type="checkbox"/> NCR <input type="checkbox"/> PT. <input checked="" type="checkbox"/> COVER <input type="checkbox"/> BRISTOL <input checked="" type="checkbox"/> SUPPLIED <input type="checkbox"/>	
PAPER SIZE <input checked="" type="checkbox"/> 8 1/2 x 11 <input type="checkbox"/> 8 1/2 x 14 <input type="checkbox"/> 11 x 17 <input type="checkbox"/>	
PAPER COLOUR <input checked="" type="checkbox"/> WHITE <input type="checkbox"/> <input checked="" type="checkbox"/> BLACK <input type="checkbox"/>	
PRINTING <input type="checkbox"/> 1 SIDE <input checked="" type="checkbox"/> 2 SIDES <input type="checkbox"/> PGS. FROM TO	
BINDING/FINISHING <b>3 down left side</b> <input checked="" type="checkbox"/> COLLATING <input checked="" type="checkbox"/> STAPLING <input type="checkbox"/> PUNCHED <input type="checkbox"/> PLASTIC RING	
FOLDING/PADDING	CUTTING SIZE

**Special Instructions**

*Math fronts & backs enclosed*

<b>COPY CENTRE</b>	OPER. NO.	BLDG.	MACH. NO.
<b>DESIGN &amp; PASTE-UP</b>	OPER. NO.	TIME	LABOUR CODE
			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	QUANTITY
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	QUANTITY	OPER. NO.	TIME	LABOUR CODE
P M T				C 0 1
P M T				C 0 1
P M T				C 0 1

PLATES	QUANTITY	OPER. NO.	TIME	LABOUR CODE
P L T				P 0 1
P L T				P 0 1
P L T				P 0 1

STOCK	QUANTITY	OPER. NO.	TIME	LABOUR CODE
				0 0 1
				0 0 1
				0 0 1
				0 0 1

BINDERY	QUANTITY	OPER. NO.	TIME	LABOUR CODE
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				B 0 1

**OUTSIDE SERVICES**

COST \$

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

# Printing Requisition / Graphic Services

15104

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

**A Study of Distributed Debugging CS-88-44**

DATE REQUISITIONED

DATE REQUIRED

ACCOUNT NO.

Oct. 31/88

ASAP

1 12 16 6 10 11 12 4 1 1

REQUISITIONER - PRINT

PHONE

SIGNING AUTHORITY

J. Black

4459

J. Black

MAILING  
INFO -

NAME

DEPT.

BLDG. & ROOM NO.

☒ DELIVER

Sue DeAngelis

CSS.

DC

2314

☐ 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 **26** NUMBER OF COPIES **50**

TYPE OF PAPER STOCK

☒ BOND ☐ NCR ☐ PT. ☒ COVER ☐ BRISTOL ☒ SUPPLIED ☐

PAPER SIZE

☒ 8 1/2 x 11 ☐ 8 1/2 x 14 ☐ 11 x 17 ☐

PAPER COLOUR

☒ WHITE ☐ ☒ BLACK ☐

PRINTING

☐ 1 SIDE ☐ PGS. ☒ 2 SIDES ☐ PGS. FROM TO

BINDING/FINISHING

☒ COLLATING ☒ STAPLING ☐ PUNCHED ☐ PLASTIC RING

FOLDING/  
PADDING

CUTTING  
SIZE

Special Instructions

Math fronts and backs enclosed.

COPY CENTRE

OPER. NO. BLDG. NO. MACH. NO.

DESIGN & PASTE-UP

OPER. NO. TIME LABOUR CODE

TYPESETTING

QUANTITY

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

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

P A P 0 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. 85 482-2



A Study of Distributed Debugging

W.H. Cheung

J.P. Black

C.S. Dept., University of Waterloo

Eric Manning

Fac. of Eng., University of Victoria

Research Report CS-88-44

October, 1988

**Faculty**  
**of**  
**Mathematics**

University of Waterloo  
Waterloo, Ontario, Canada

N2L 3G1

# A Study of Distributed Debugging

W.H. Cheung  
J.P. Black  
C.S. Dept., University of Waterloo

Eric Manning  
Fac. of Eng., University of Victoria

Research Report CS-88-44  
October, 1988



# **A Study of Distributed Debugging**

W.H. Cheung     J.P. Black

Department of Computer Science  
University of Waterloo

Eric Manning

Faculty of Engineering  
University of Victoria

## **Abstract**

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 a general view of current research in distributed debugging. We first look at the issues in distributed debugging. Then, a simple framework for studying distributed debugging systems is presented. Based on this model, we classify research problems into three areas—the distributed debugging model, domain specification and system support. We focus our discussion on the distributed debugging model. This leads us to present and discuss some research results on debugging techniques, methodologies and approaches. Finally, we draw conclusions and suggest some research directions.



# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
<b>2</b>	<b>Issues in Distributed Debugging</b>	<b>3</b>
<b>3</b>	<b>A Framework for Studying Distributed Debugging Systems</b>	<b>4</b>
3.1	A Model of Distributed Debugging Systems . . . . .	4
3.2	A Decomposition of the Problem of Developing a DDS . . . . .	6
<b>4</b>	<b>Basic Debugging Techniques</b>	<b>8</b>
4.1	Output Debugging . . . . .	9
4.2	Tracing . . . . .	9
4.3	Breakpoints . . . . .	10
4.4	Assertion Execution . . . . .	11
4.5	Controlled Execution . . . . .	12
4.6	Replay . . . . .	12
4.7	Monitoring . . . . .	13
<b>5</b>	<b>Debugging Methodologies</b>	<b>14</b>
5.1	Debugging Reasoning . . . . .	14
5.2	Top-Down versus Bottom-Up Debugging . . . . .	15
5.3	Two-Phase Debugging . . . . .	16
<b>6</b>	<b>Three General Approaches for Distributed Debugging</b>	<b>16</b>
6.1	The Database Approach . . . . .	17
6.2	The Behavioural Approach . . . . .	18
6.3	The AI Approach . . . . .	19
<b>7</b>	<b>Concluding Remarks</b>	<b>20</b>



# 1 Introduction

Debugging is an essential step in developing a software system, since every non-trivial software system contains bugs. However, no precise or elegant method 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 gives rise to new types of bugs. In recent years, researchers have developed some helpful debugging techniques for distributed environments. However, most papers on the debugging of distributed software discuss a specific debugger or a particular technique. Few published papers look at the issues and solutions in a general 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 general 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 our discussion, we concentrate on run-time debugging, and always refer to it simply as debugging. Nevertheless, we emphasize that static debugging and run-time debugging complement each other, and neither of them should be overlooked. (See “Terminology” box.) Furthermore, formal specification and development techniques for distributed software are becoming more well-understood, and can be well suited to the behavioural approach described in Section 6. However, since little existing literature makes explicit links between specification techniques and debugging, we consider it outside our scope.

In a distributed system, we deal with an environment that lacks precise global states because of multiple processors, lacks a common time reference, and suffers from variable, unpredictable communication delays. This imposes an inherent constraint on the development and debugging of distributed programs. Moreover, concurrent interactions among communicating processes constitute a significant part of the complexity of distributed programs. Bugs due to these interactions can be subtle, even transient, making their location very difficult. Unfortunately, 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, it is much more difficult to debug a distributed program than a sequential program.

A distributed program is basically a set of sequential programs. Therefore, bugs which occur in sequential programs still occur in distributed programs. In addition, new kinds of bugs are introduced due to the characteristics of distribution, 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

## Terminology

**Program failure** is defined as a deviation of execution behaviour from that dictated by the program specification. An **error** is an erroneous state which leads to a failure. The cause of this error is referred to as a **fault**. A **bug** is a fault in the program.

**Debugging** is the process of *locating* and *correcting* detected bugs. The user may try to detect bugs by analyzing the program, by performing systematic tests, or by monitoring program execution. After a bug is detected, the user determines its exact nature and location, and finally corrects it.

A **distributed program** is a collection of related, co-operating program modules which will be instantiated as processes at run-time on multiple processors in a distributed system.

A **cluster** is a set of processes of interest, typically an instantiation of a distributed program. A cluster runs on top of a distributed operating system which provides at least the functions of process manipulation and interprocess communication.

**Distributed debugging** is the process of debugging distributed programs on a distributed system; it does not refer to distributed implementation of a sequential debugger.

**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.

**Run-time Debugging** is debugging based on execution information of the program. Typical run-time debugging tools for sequential programs are interactive symbolic debuggers, memory dumps and trace packages.

**Static debugging** involves techniques applied to a program before it is actually executed. Design reviews, coding reviews, and formal proofs of program correctness are well-known examples. While these techniques are very useful for reducing the number of bugs in the program, bugs may still appear during execution, due to the incompleteness of these techniques and unanticipated changes in the execution environment.

synchronization, misuse of interprocess communication (IPC) primitives, partition due to communication problems, communication overload, and protocol faults. Notice that these bugs are not unrelated. For example, a message omission may lead to a deadlock if the receiver must receive the lost message before continuing to execute, and communication overload may lead to message omission.

This paper is organized as follows. Issues in distributed debugging are discussed in Section 2. In Section 3, we introduce a simple model of distributed debugging systems. The model defines the role of a distributed debugging system and its relationship with the debugging environment. Based on this model, we decompose the problems of developing a distributed debugging system into three areas: the distributed debugging model, domain specification and system support. While we consider the problems in all three areas, our main theme of discussion is the distributed debugging model which is divided into three major subareas: basic techniques, methodologies and approaches. In Sections 4, 5 and 6, we present and discuss some research results in these three subareas respectively. Finally, in Section 7, we conclude our discussion and discuss some future research directions.

## 2 Issues in Distributed Debugging

It is generally admitted that debugging is a difficult job. It is a mentally taxing activity, typically performed under pressure. Unfortunately, even intense effort does not guarantee immediate return. Psychologically, programmers find it emotionally disturbing to admit that their programs are imperfect and require debugging. Furthermore, a programmer often settles on a particular view of a debugging problem. Such a view hinders him from viewing the problem in a different way, or from looking at other reasonable assumptions. Moreover, the semantics of a program may change when a programmer locates and corrects bugs, as fixes introduce changes to the program. Debugging probes can also alter program states in a way that makes bugs hard or impossible to identify. These make debugging a very annoying job.

The concurrency and complexity of distributed programs make the situation even worse. In general, 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. These inherent limitations make it difficult for the programmer to master and thus to debug a distributed program.

Technically, difficulties arise from the simultaneous use of multiple processors, each having its own physical time reference. Thus, control of time and management of state space are two major considerations, as are problems arising from interprocess communication. Here are some typical difficulties.

1. *Maintenance of Precise Global States.* Global clock synchronization is a classical research problem in distributed systems [1], and it is nontrivial to have an accurate global clock. In a general sense, even with accurately synchronized clocks, it is impossible to obtain global information about a cluster at a precise instant in time: unpredictable communication delays, various speeds and different states on multiple machines will cause the operations of collecting global information to proceed at different rates. For the same reasons, an immediate change of control for all parts of a computation on different machines at the same time is impossible, unless all machines know the time beforehand. Thus, we can only expect an approximate global state, such as local states on each machine plus states of communication channels within a certain time period. Alternatively, we could confine ourselves to global states with *stable properties*, that is, those which remain true once they become true.
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 large quantities of state data for debugging purposes 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 data from individual processors? How do we display useful information extracted from the data? Moreover, distributed systems can grow incrementally and tend to have large numbers of processes and processors. The larger a system is, the more information must be manipulated during debugging. Furthermore, the complexity of interactions increases as the number of cooperating processes or processors increases.

3. *Interaction Among Multiple Asynchronous Processes.* Bugs occurring among processes in a cluster are often complex or sporadic, 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 among processes. Worse still, some clusters are dynamic enough (*i.e.*, numerous process creations and terminations) that even identifying the set of processes currently belonging to a cluster can be difficult.
4. *Communication Limitations.* Significant, variable and unpredictable communication delays, and limited communication bandwidth may make some typical debugging techniques impractical, such as central manipulation of cluster state information.
5. *Error Latency.* Usually, there is a time lag between the occurrence of an error and its discovery. This time lag exists in sequential programs, but is worse in distributed ones. Due to significant communication delays and autonomous operations, the time lag may be quite large in a distributed system. During this time period, the error may propagate widely. Via communications, the erroneous process affects the processes on another processor which pass the effect to other processors and so on. This domino effect may generate a burst of additional failures, often making the original bug very difficult to locate.

In spite of the difficulties programmers face in developing distributed programs, few effective 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.

### **3 A Framework for Studying Distributed Debugging Systems**

#### **3.1 A Model of Distributed Debugging Systems**

As the problems in distributed debugging are complex and involve a number 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 in each domain. This motivates the model depicted in Figure 1, which defines the role of a typical *distributed debugging system* (DDS) in the *debugging environment*. A DDS is a collection of software modules that facilitate the debugging process. These modules may be in the form of a process cluster, a code segment in each process, supporting routines in the kernel, or some combination of them. The debugging environment (or simply, the environment) is a collection of domains which interact with the DDS.

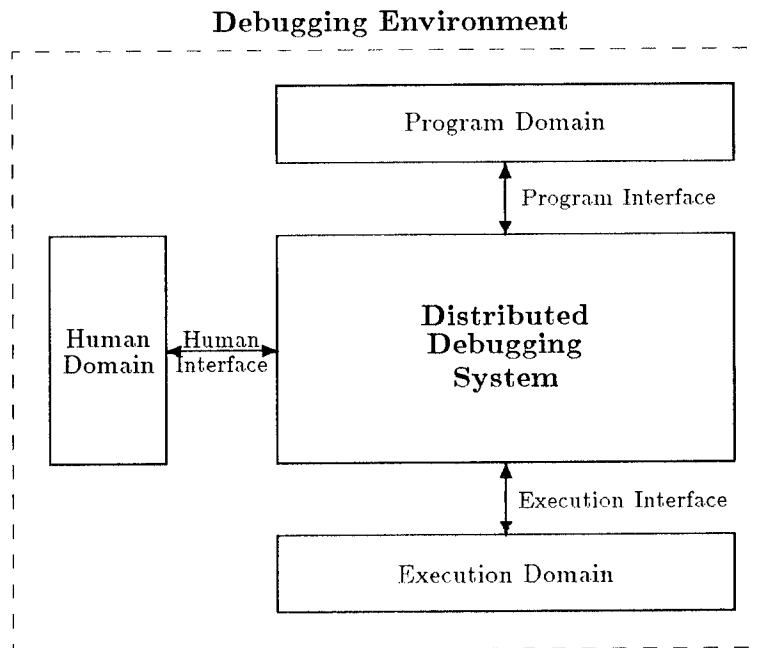


Figure 1: A Model of Distributed Debugging System

In the model, the environment is partitioned into three domains and corresponding interfaces. The *program domain* includes static descriptions of the distributed program being debugged, such as its source code and specification of expected behaviour. The *human domain* includes the attributes of the programmer who uses the DDS to debug clusters. The *execution domain* includes the characteristics of the underlying distributed operating system and the run-time behaviour of the cluster.

The DDS is viewed as an agent which manages the interactions among these three domains. It collects behavioural data from the execution domain, and may impose appropriate control on the execution domain. In addition, it consults the program domain to interpret behavioural 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 draw some conclusions about a bug.

The *execution interface* involves 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 computational model supported by the distributed operating system. Based on this interface, the DDS observes and possibly controls the behaviour of a cluster in the execution domain. As well, the DDS itself is a distributed program, and its implementation is based on this interface.

The *program interface* conveys necessary information about the program 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 communication not only involves the program itself, but may also involve representation of the “expected” behaviour of the program and perhaps even specification of anticipated errors. The interface involves the language-debugger interface and the mechanism for detecting errors with respect to the specification.

The *human interface* defines the allocation of debugging activities—some to the DDS, some to the programmer, and some to both. Human factors in debugging activities and mastery of parallel events are the main concerns, since they have a profound effect on the usefulness of the DDS. For example, good use of graphical displays and pointing devices can have dramatic results on the usability of a DDS. Of course, a programmer can bypass the DDS and interact directly with the program and execution domains for debugging. However, such interactions indicate that the DDS is not effective. Given a good DDS, human users will want to debug through the human interface alone.

### 3.2 A Decomposition of the Problem of Developing a DDS

Now, we look at the problems of developing a DDS and defining its interactions with the environment. For the sake of our study, we divide the problem into three main areas and then further decompose them into smaller components, illustrated in Figure 2.

A *distributed debugging model* (DD Model) is a conceptual framework for developing a DDS. We divide the framework into three major subareas, each focusing on a particular set of instruments. The first is a set of *basic techniques* to capture run-time information and control the execution environment. The second is a set of guidelines or *methodologies* to organize overall debugging activities. The third is general *approaches* which provide high-level abstract models for tackling the debugging process. In each subarea, there are various alternatives. A designer may first select an approach and a set of methodologies, and then develop suitable basic techniques. During debugging,

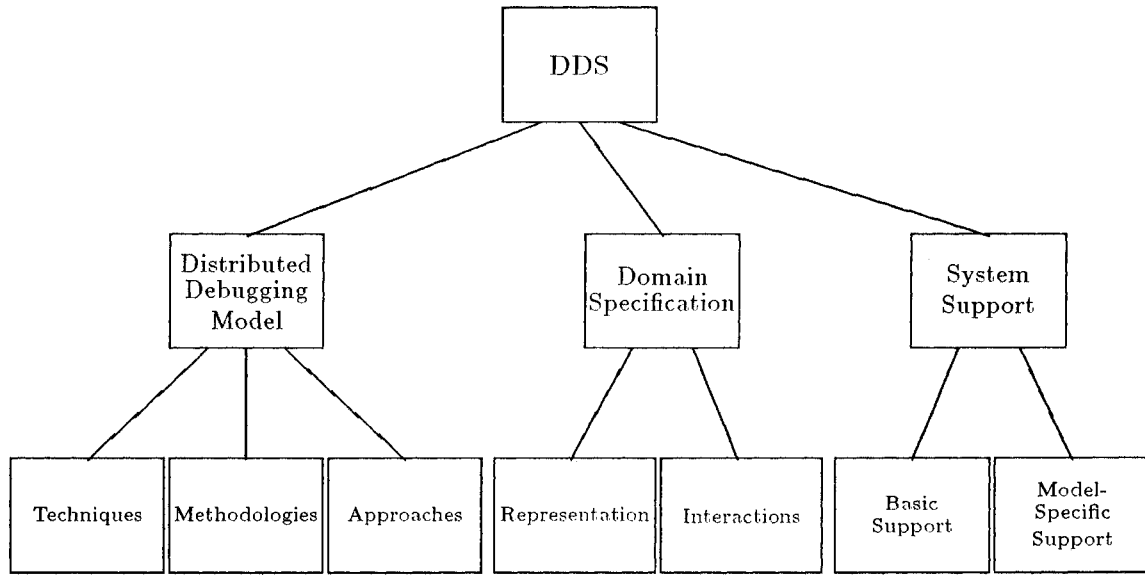


Figure 2: A Decomposition of the Problem Space for DDS Development

a programmer uses the techniques to obtain information, follows the approach to tackle a bug, and adopts one or several methodologies to structure the debugging activities.

The area of *domain specification* considers the interactions between the DDS and each domain in the model. There are two major factors to be considered: *representation*, or description of the domains, and *interactions* between the DDS and the domains. More precisely, it involves representations of objects (*e.g.*, errors, monitoring information) or activities (*e.g.*, program behaviour, human interaction) in the domains. Only when the objects and activities of the domains are described precisely can we define the interactions between the domains and the DDS.

*System support* involves implementation issues, and can be further divided into *basic support* and *model-specific support*. A good example of basic support is a global clock synchronization facility to order events. An example of model-specific support might be that a particular methodology requires a driver process to interact with the process being debugged. This separation between these two types of support allows us to first provide a basic framework and then develop various models based on it, either for purpose of model evaluation or for different debugging requirements.

We use this decomposition to study the development of a distributed debugging system. Notice that these three areas are not independent. The DD model must eventually be related to the issues in the area of domain description for external communication and the area of system support for implementation. However, because of its overall importance in the design of a DDS, we concentrate

our discussion on the DD model; nevertheless, the issues in the other areas are discussed whenever appropriate.

## 4 Basic Debugging Techniques

Although most of the techniques presented in this section have been used extensively to debug sequential programs, their use in distributed programs involves further considerations of effectiveness, implementation difficulties and semantics.

All existing distributed debuggers are based on explicit or implicit assumptions. The following are three common ones:

1. The necessary system software (so-called *hard core*), such as the distributed operating system, the communications subsystem and the debugging tools, is debugged<sup>1</sup>, and programmers can therefore concentrate on the debugging of application software. By excluding pieces of system software from the hard core, we can think about debugging them too.
2. The main focus of a DDS is the interaction among processes, rather than the internal logic of each process. We assume that internal bugs of the processes are or can be removed by the use of a sequential debugger. In other words, a distributed debugger deals with programming-in-the-large, in which processes 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.
3. All interactions among processes are based on the use of message passing for IPC.

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 develop the interface between distributed and sequential debuggers, in order to exchange debugging information and to switch easily between the two.

In the following discussion, we keep the general assumptions listed above. That is, the DDS assumes the existence of a hard core of debugged software and focuses on process interactions based on message passing.

---

<sup>1</sup>That is, it is assumed that no bugs are found in hard core software.

## 4.1 Output Debugging

This is the most primitive debugging technique, but is also the easiest to implement. Indeed, it is often the only technique available. 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 only simple output statements are required, and that the programmer sees only the data he selects. However, it has disadvantages. First, the programmer needs to observe output of processes on multiple processors at the same time. When the number of processes is large, such observation becomes infeasible. Also, the technique relies completely on the programmer to select appropriate places in the program to insert output statements. This art is guided by the programmer's experience and thinking. Furthermore, the technique requires modifications to the program and hence may alter the existing program structure or even introduce new bugs. More importantly, it can easily change the behaviour of a cluster, and thus is not effective for locating time-dependent bugs. However, it is often combined with other techniques to make the debugging job easier.

## 4.2 Tracing

This differs from output debugging in that the operating system provides a standard trace facility to display selected tracing information. The programmer turns the trace on and off in the program when necessary. 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 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 timestamps (either physical or logical time) for all trace information are necessary. In other words, we need a clock synchronization facility. Although the development of a clock synchronization facility is feasible, many existing distributed operating systems do not provide such a facility. When there is no such facility available, a selected processor can be made responsible for forming the global trace, according to the order in which trace messages are received from all other processors. Due to variable communication delays and the nondeterminism of processor scheduling, the trace messages may not arrive at the processor in the order they were generated. Also, the selected node can become a bottleneck for the collection of trace information. Thus, the provision of a clock synchronization facility is a better solution.

### 4.3 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, by requesting single-step execution or setting further breakpoints. Execution continues after the breakpoint when requested by the programmer.

Using this technique, no extra code is added to the program by the user. 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 be knowledgeable enough to set breakpoints at appropriate places in the program and to examine relevant data.

The technique raises particular problems in a distributed environment. First of all, it is impossible to define a breakpoint in terms of precise global states. Thus, people usually define a breakpoint in terms of events in a process or interactions among processes.

Second, the semantics of single step execution are no longer obvious. Some define it to be the execution of a single machine instruction or a statement of source code on a local processor. Others consider it to be a single statement on each processor involved. Some people treat an event, such as message transmission, message reception or process termination, as a single step. 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 halting the process cluster at a breakpoint or after a single step. When a breakpoint is triggered, the whole cluster must be stopped. One simple way to do so is to broadcast “pause” messages to all processors. 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. 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 cluster. To resume the execution, a “continue” message is broadcast to all processors. However, the “pause” and “continue” operations are not so simple. For example, when a process which is subject to a timeout request halts for debugging, the real time clock is still running. At the time the process resumes execution, it will encounter a much shorter timeout interval, and its behaviour may change significantly. Cooper introduced a logical clock mechanism to maintain correct timeout intervals, and thus to provide transparent halting [2].

There is also a problem of how to halt the cluster in a consistent global state. Chandy and Lamport introduced a distributed algorithm to obtain *distributed snapshots* of a cluster [3]. The algorithm is intended to capture *only* global states with *stable properties* which, once they become true, remain true thereafter (*e.g.*, deadlock, process death). 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.

There are two important issues in the use of this algorithm. First, halting a process depends on the interactions between itself and other processes. The algorithm cannot collect local information about a process which has no or only one communication channel connected to other processes. Furthermore, consider a process which has only infrequent interactions with other processes: that process would halt long after all other processes have halted. Miller and Choi deal with this issue by introducing two additional control channels between a debugger process and each process in the cluster [4]. The second issue is that 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.

#### 4.4 Assertion Execution

An assertion is an executable predicate which specifies invariant conditions of execution at the point where it is placed in a distributed program. Note that this involves run-time verification, and is not the same as a “logical assertion” in a program proof. 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. This allows the programmer to introduce assertions to detect malfunctions of interest.

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 situation to the programmer and usually halts execution. This 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. Because it is impossible to obtain precise snapshots at arbitrary points in execution for checking, assertions can only describe states local to an individual processor, or focus on sequences of transmitted messages, or deal with approximate or stable global states.

## 4.5 Controlled Execution

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 are useful, such as changing the relative speeds of the processes, simulating delays in the communication paths (perhaps for race detection) and altering the order of message-passing events. Controlled execution allows a programmer to analyze concurrent events at the speed and in the order expected. However, some errors may disappear due to changes of relative speed and interaction sequence.

This technique is very useful for revealing bugs in the testing phase; the programmer can alter the order of interactions to perform a set of pre-defined tests. However, exhaustive testing of all possible interactions is almost always impossible. Techniques are needed to prune the number of tests, such as by identifying equivalence classes of interaction sequences.

## 4.6 Replay

Replay simulates the history of a process cluster in a controlled environment. The debugger continually captures relevant execution information (*e.g.*, input, messages) for the cluster and saves it 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 be able to locate the bug.

One problem of replay is recording cluster behaviour. LeBlanc and Mellor-Crummey proposed a general solution to this problem, termed *Instant Replay* [5]. During cluster execution, the relative order of significant events (calls to monitors, in their case), rather than the data associated with such events, is saved. As a result, the technique requires little time overhead (less than 1%) and less space to save the information for replay. Therefore, it can be used to record events of a production system. However, the replay 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 occurred in the original execution. Instant Replay assumes that each process is deterministic; that is, the process does not contain nondeterministic statements or allow asynchronous interrupts. Also, the replay must involve the whole cluster, rather than a subset of the processes. This implies that the system can neither speed up the replay nor be confined to a part of the cluster behaviour.

Bugnet allows a programmer to replay part of cluster execution by using a *checkpoint algorithm* [6]. It also aims to replay events in physical time, rather than just in their correct relative order. All processes in a cluster start execution synchronously with reference to a global clock. During

execution, Bugnet collects IPC events of the cluster. After a period of time, the whole cluster is halted, 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 senders know the messages were sent but the receivers never receive them. Thus, these messages must be saved and then presented to the receivers during the next run period. All processes are synchronized so that they 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 is located, and a common clock time as well as necessary replay information are sent to the agents which monitor the replay process. While the checkpoint algorithm facilitates physical time replay, it imposes significant overhead during normal execution: two extra processes for each application process and large storage for keeping checkpoints. Moreover, the halting period (0.5 seconds) may be unacceptable for time-critical applications.

## 4.7 Monitoring

Many existing distributed debuggers adopt this technique, or from another viewpoint, many monitoring systems are developed to aid in debugging. The idea of monitoring is to capture useful data during execution, and then display it to the programmer. Sometimes, the data is saved for later use, such as replay. A debugger may also analyze the captured data in some fashion to detect an error. Since the execution is monitored continuously, the occurrences of infrequent, unpredictable and irreproducible bugs are not missed, and the recorded data may provide valuable information for diagnosing those bugs.

However, keeping a copy of every event 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 monitoring records is a very tedious activity and requires a high degree of expertise. Therefore, filtering and clustering techniques are used to reduce the amount of information to be examined.

*Filtering* attempts to ignore data which is irrelevant to the current debugging interest. Although it introduces extra filtering time, it can save time needed to process such data. This monitoring data may be filtered in two ways. In *display filtering*, while all data is stored, only selected data is shown to a programmer. In *data filtering*, only relevant or selected data is saved in storage and all other data is discarded. Display filtering does not reduce the amount of storage to keep the data, but avoids flooding the display with irrelevant data. Data filtering goes 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 replay).

*Clustering* is used to group a designated set of events into a single composite event. To use this technique, a programmer first defines composite events in terms of previously-defined events or primitive events obtained directly from monitoring data. 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. However, there are problems associated with recognizing distributed composite events, such as filtering out irrelevant events, handling the relative timing of processes, distributing event information to other processors, and sharing a primitive event among several composite events.

## 5 Debugging Methodologies

Distributed programming is still a young discipline. We have difficulty understanding the additional complexity of distributed programs compared with sequential programs. This makes the use of an explicit process of reasoning especially important in distributed debugging. In addition, since a large distributed program is usually too complex to debug in a single piece, several methodologies have been proposed to simplify the job. We identify three in the literature: top-down, bottom-up, and two-phase debugging. Notice that they can supplement one another. For example, two-phase debugging can be embedded into one step of top-down or bottom-up debugging.

### 5.1 Debugging Reasoning

Experience sometimes gives us the intuition necessary to debug a program, but cannot always be relied on. There is no guarantee that the intuition is correct, since a symptom may be due to more than one bug. Therefore, when experience fails, or is lacking, the only recourse is reason.

Some general methodologies have been proposed to tackle debugging with the aid of reasoning [7]. In *debugging by induction*, one tries to look for relationships among clues from collected data which lead to the error and to devise hypotheses about the bug. The hypotheses are refined and proven or disproven until the bug is located. In contrast, *debugging by deduction* first hypothesizes a set of suspicious faults. Then, by elimination and refinement, the programmer attempts to prove that one of the suspicious faults is actually the bug. Clearly, these two methodologies may fail when the correct hypothesis cannot be devised. Another methodology is *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 an earlier 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 workable for small programs, but too complicated for large programs. However, the technique of program slicing [8] may be used to simplify the work by isolating relevant program slices for backtracking. A program slice is a simplified form of the program that still produces certain specified behaviour. The programmer may backtrack the program slices instead of the original program to identify the bug. Finally, *debugging by systematic testing* can help the user of inductive and deductive methods to devise hypotheses. The programmer executes a sequence of test cases, and formulates a hypothesis about the bug based on the outcomes of these tests. Then the hypothesis is verified.

## 5.2 Top-Down versus Bottom-Up Debugging

A large distributed program is composed of a large number of processes. If it is modular in design, the processes are generally grouped into various functional modules, several modules are merged again to form a larger module, and so on. Some debuggers provide a facility to support this kind of process grouping, such as hierarchical process groups in Bugnet. To debug such grouped processes, researchers have proposed two very familiar methodologies: *top-down debugging* and *bottom-up debugging*. In top-down debugging, the behaviour of the entire cluster is considered first and erroneous modules are identified. Then, debugging focuses on the erroneous modules and more detailed behaviour within them is considered. The process is repeated until the bug is located. In bottom-up debugging, the process is reversed. In the first step, each process is debugged separately in an artificial environment. Then, several processes are merged together and debugging concentrates on the interactions among the processes. Again, the process is repeated until the whole cluster is formed and debugged.

Basically, these two methodologies attempt to 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 fully developed system, as all processes are implemented. Bottom-up debugging is more appropriate for a newly-developed system since we can test and debug the system gradually from processes 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 message flows between processes instead of data flow between 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 dynamically. Therefore, we must not only group the processes into functional modules, but also restrict message interactions among these modules

in order to use the methodologies effectively.

### 5.3 Two-Phase Debugging

One basic idea of debugging is to monitor the events of a cluster, and hope that the trace will exhibit an abnormality which may help in localizing the fault within a relatively small part of the cluster. Two-phase debugging, introduced by Garcia-Molina [9], applies this idea in distributed debugging. The debugging process goes through two phases:

1. *Phase One.* The software continues to execute until an error is observed. During execution, the system saves a monitoring trace for truly significant 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 reconstructed and execution is replayed. This replay 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.

Creating the artificial environment for the second phase may be a difficult and time-consuming task. Also, this methodology encounters problems similar to those of monitoring. Garcia-Molina 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, with the newest data written over the oldest data. However, it is not always easy to determine a “reasonable size.”

## 6 Three General Approaches for Distributed Debugging

A general approach provides a high-level abstract model for tackling the debugging process and gives guidelines on how to manipulate debugging information. We identify three general approaches in the literature: the database approach, which emphasizes information manipulation; the behavioural approach, which emphasizes cluster behaviour manipulation; and the AI approach, which considers automation of the debugging process, using artificial intelligence techniques. Not all debuggers incorporate a particular, well-defined approach. Some simply provide sets of basic debugging tools to support the techniques described in Section 4.

## 6.1 The Database Approach

This approach views debugging as performing queries on a database that contains program information (*e.g.*, system specification, source code) as well as execution information. Initially, program information is stored in a database. Then, execution information is forwarded to the DDS as database updates. The programmer tries to understand the program behaviour by making database queries, the intent being that answers to the queries provide clues to the bug. Thus, in this approach, debugging is a matter of understanding the execution and specification of a distributed program.

The use of the database paradigm offers several advantages. First, the facilities of a typical (distributed) database system are available, including storage management, data retrieval and concurrent access control. Therefore, a debugger which adopts this approach simply monitors program execution and forwards execution 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, the programmer can access program and execution information in an integrated fashion for debugging.

In the following, we briefly discuss two examples of this approach: Snodgrass' monitor [10] and the Program Visualization System [11].

To capture execution information, Snodgrass' monitor uses a relational database with a query language, both extended to incorporate time. The system was actually implemented to monitor programs on a tightly-coupled multiprocessor system. Due to central manipulation of the execution information, the maximum number of processors is bounded to roughly fifty.

The Program Visualization System is implemented using the INGRES relational database. It represents a distributed program as a set of objects (*e.g.*, program blocks) and binary relations among these objects, and stores them in a database. A *view* of the program is a sequence of queries regarding the objects and their relations in the database. A sophisticated graphical interface is provided for view construction and display. Multiple, alternative views of a single distributed program can be constructed so that a programmer can consult different views to obtain various information for debugging. The program views are static relations, and do not change automatically as new information is added to the database. This feature has the advantage of obtaining a series of "snapshot" views of the program. However, it does require the programmer to rederive the views explicitly if he wishes to include new information.

## 6.2 The Behavioural Approach

In this approach, debugging is viewed as the process of comparing the execution behaviour to the expected behaviour as defined by the user in some formal specification. The debugger monitors execution events, deduces the actual behaviour from the events and compares it with the expected behaviour. If they are different, the debugger notifies the programmer of the discrepancy. The programmer attempts to locate the bug by studying the discrepancy.

Current research emphasizes the development of machine-based tools to aid a programmer in understanding program behaviour. This is achieved by providing for the creation of multiple viewpoints of a program and the ability to view the program at different levels of abstraction. The notion of *behavioural abstraction* [12] is one example. A set of primitive events constitutes the lowest level of cluster behaviour. This may provide a particular viewpoint of the program. Based on this, the mechanism allows the programmer to define alternative, high-level viewpoints. This idea is similar to that of multiple views in the Program Visualization System except that the behavioural abstraction is based on a behavioural description of a cluster, while the multiple views are based on program objects and their relations.

Usually, this approach provides a language facility for specifying events, behaviour and actions. It may include a *behavioural definition language* to define expected behaviour, based on primitive or user-defined events, and an *action language* to specify the operations (*e.g.*, halt the cluster, log the event) to be performed automatically after certain behaviour has been detected. For behavioural abstraction, the behavioural definition language can also specify alternative high-level viewpoints.

Using a behavioural definition language, we can impose a redundant, independent specification of the expected behaviour on a program. Based on this specification, the debugger checks if the actual behaviour deviates from the expected behaviour. In addition, the reporting mechanism may be invoked selectively, thus avoiding the need to analyze large volumes of data. Moreover, the use of behavioural definition languages 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, and the programmer must determine by himself where to place it. Also, several execution paths may lead to the same breakpoint, and not all of them are necessarily useful to the programmer. In contrast, in a behavioural definition language, it is easy to express an execution path, thus allowing very selective debugging. Another example is that of an assertion statement, which tests for state violations at single points in the execution history. Behavioural definition languages, on the other hand, are able to express the correct *sequences* of interactions.

In recent years, there have been a number of studies in distributed debugging using this approach. We describe the EBBA Toolset [12], the ECSP Debugger [13], and the Task Graph Language and

Token Lists [14].

The EBBA Toolset uses a language called EDL to specify some models of expected cluster behaviour in hierarchies of *event definitions*. Each event definition describes how an instance of the event might occur and what the attributes and constraints of the instance are. The debugger tries to match these models to the execution events. When the models fail to match, the debugger attempts to characterize the differences. However, when the models match the actual behaviour, they may only demonstrate that some aspects of the program behave as expected, since the models need not fully describe the cluster behaviour.

The ECSP Debugger has a notion of grouping processes in a hierarchy. A process  $P$  hides the interactions between processes  $P_1, \dots, P_n$  activated by a parallel statement of  $P$ . The basic events are defined as a set of *Event Specifications*. 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, 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 process levels and the strong connection between the debugger and the semantics of ECSP. However, the latter feature precludes porting the debugger to other languages. Also, the debugger has no notion of behavioural abstraction.

The Task Graph Language (TGL) is based on a general communication model in which IPC is through message exchanges. It is intended for expressing expected IPC patterns in a task (or process) graph. Using TGL, a programmer specifies expected IPC interactions in regular expressions. A compiler parses a TGL specification to generate a set of *token lists* which form 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 its peers. A run-time 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 has no notion of process or behavioural abstraction and has poor support for dynamic process creation.

### 6.3 The AI Approach

The idea here is to use artificial intelligence techniques to help in detecting bugs, suggesting possible causes of bugs, and even proposing corrections. One method is to use an expert system. The intent is to capture the debugging 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.

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 sequential program debugging at source level, such as the Fault Localization System and PROUST. Seviora provides a good general study of such knowledge-based debugging systems [15].

An example closer to our interests is the Message Trace Analyzer (MTA) developed by Gupta and Seviora [16]. MTA is a knowledge-based system designed for debugging a single-processor concurrent system in which all activities are encapsulated in processes and their interactions are only via message exchanges. Given a correctly integrated trace of message events from an execution, it consults a knowledge base of rules to detect any illegal message sequence and its cause. MTA works on the assumption that a trace would exhibit an abnormality which would help to localize the bug, and it only localizes a failure rather than the bug itself. Also, it assumes that failures are known at the system level. This limitation makes MTA unable to handle unspecified failures. However, this work demonstrates the potential of AI techniques for debugging concurrent programs.

## 7 Concluding Remarks

We have identified three major approaches: the database approach, the behavioural approach and the AI approach. 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 it to the programmer to locate the bug, by formulating the queries and by analyzing the answers. The behavioural approach is able to detect errors by checking discrepancies between actual and expected behaviour. It is a step beyond the database approach, towards provision of automatic error detection based on behaviour description. The event sequence and the discrepancy also give additional clues for bug location. The AI approach is very attractive as it has the potential to release the programmer from much drudgery. However, current research in AI on diagnosis is still at an early stage. More importantly, heuristics cannot replace reasoning. In our view, further research on debugging reasoning is required before we can consider this approach seriously.

None of the three approaches suggests any way to locate and repair a bug. Perhaps, bug locating and repairing should not be considered solely in the realm of debugging. The integration of program development tools is probably the better strategy, since we can use more program-related information for debugging analysis. In our opinion, the behavioural approach is currently the most effective and promising one. It also appears the one most suited to the use of formal specification

techniques.

Although many issues in sequential debugging are relevant to distributed debugging, the latter has some unique problems. These include lack of global knowledge, and management of a large system state space with multiple foci of control. Even though various algorithms, mechanisms and debuggers have been developed in the last few years, there is still room for improvement and a number of open issues remain.

The first area is DD models. Various approaches, methodologies and basic techniques need to be explored to tackle the technical difficulties in distributed debugging. In recent years, although there have been many studies on particular aspects of DDS models, few general principles for designing and evaluating DD models have appeared.

The second area concerns the interaction between a DDS and its debugging environment. This area is particularly important for the the human domain. Research here includes the role of people in distributed debugging, human debugging behaviour, human-oriented representations of debugging objects, effective communication between human and DDS, and human factors related to the understanding of parallel dynamic execution. While there are some interesting studies on the psychological aspects of sequential debugging, their extension to distributed debugging is an unexplored subject. On the other hand, the Execution interface has been recognized as difficult to study. The issues are basically time delay, synchronization, size of state space and the lack of a global viewpoint. 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, 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 are still no general principles to guide the development of such languages.

In recent years, sophisticated workstations with high-resolution color graphics displays have been used extensively in developing distributed debuggers. With such a graphical debugger, researchers usually claim that a programmer can easily observe cluster behaviour and detect bugs. However, little empirical evidence exists to support this claim. Moran and Feldman [17] 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 (textual or graphical) has no statistically significant impact on debugging time. One possible reason for this is that the bugs and the programs were so simple that the subjects in the experiment did not fully utilize the graphical debugger. Further experiments are necessary to determine if a graphical debugger is better than a textual debugger for hard bugs in a complex, distributed program.

In general, it is useful to study the characteristics of the debugging 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, we can make assumptions about the execution behaviour and also design the execution interface effectively. Also, the study of the interfaces may give interesting results. If system-independent elements are isolated from all the interfaces, we may be able to construct a portable DDS. Portability is a very attractive feature as it allows us to transfer a DDS from one implementation to another implementation with little effort, and to maintain a well-developed DDS despite changes to the environment. Moreover, the interface design has a significant impact on the DDS. For example, a poor design of the human interface clearly discourages a programmer from using it and hence degrades its usefulness. A design of the execution interface which requires a lot of remote accesses definitely reduces the performance level of the DDS.

The third area for future research is implementation issues of a DDS, including provision of a clock synchronization facility, consistent breakpoints, handling of monitoring data, information filtering and behavioural abstraction. Also, the general architecture of DDSs needs more investigation. 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, and coordination of DDS modules in a distributed environment.

As we have said, there is no lack of problems in distributed debugging. However, the growing use of distributed systems makes the need for an elegant distributed debugging system obvious and urgent.

## References

- [1] L. Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System", *CACM*, Vol. 21, No. 7, July 1978, pp. 558–565.
- [2] R. Cooper, "Pilgrim: A Debugger for Distributed Systems", *Proc. of the 7th Int. Conf. on Distributed Computing Systems*, Berlin, West Germany, September 1987, pp. 458–465.
- [3] K.M. Chandy and L. Lamport, "Distributed Snapshots: Determining Global States of Distributed Systems", *ACM Trans. on Computer Systems*, Vol. 3, No. 1, February 1985, pp. 63–75.
- [4] B.P. Miller and J.D. Choi, "Breakpoints and Haltings in Distributed Programs", *Proc. of the 8th Int. Conf. on Distributed Computing Systems*, San Jose, California, June 1988, pp. 316–323.
- [5] T.J. LeBlanc and J.M. Mellor-Crummey, "Debugging Parallel Programs with Instant Replay", *IEEE Trans. on Computers*, Vol. C-36, No. 4, April 1987, pp. 471–482.

- [6] S.H. Jones, R.H. Barkan, and L.D. Wittie, "Bugnet: A Real Time Distributed Debugging System", *Proc. of the 6th Symp. on Reliability in Distributed Software and Database Systems*, Williamsburg, VA, March 1987, pp. 56-65.
- [7] G.J. Myers, *The Art of Software Testing*, John Wiley & Sons, 1979.
- [8] M. Weiser, "Programmers Use Slices When Debugging", *Communications of the ACM*, Vol. 25, No. 7, July 1982, pp. 446-452.
- [9] H. Garcia-Molina, F. Germano Jr., and W.H. Kohler, "Debugging a Distributed Computing System", *IEEE Trans. on Software Engineering*, Vol. SE-10, No. 2, March 1984, pp. 210-219.
- [10] R. Snodgrass, "Monitoring in a Software Development Environment: A Relational Approach", *Proceedings of the SIGSOFT/SIGPLAN Software Engineering Symp. on Practical Software Development Environments*, SIGPLAN Notices, Vol. 19, No. 5, May 1984, pp. 124-131.
- [11] K. Schwan and J. Matthews, "Graphical Views of Parallel Programs", *ACM SIGSOFT Software Engineering Notes*, Vol. 11, No. 3, July 1986, pp. 51-64.
- [12] P. Bates, "Distributed Debugging Tools for Heterogeneous Distributed Systems", *Proc. of the 8th Int. Conf. on Distributed Computing Systems*, San Jose, California, June 1988, pp. 308-315.
- [13] F. Baiardi, N.D. Francesco, and G. Vaglini, "Development of a Debugger for a Concurrent Language", *IEEE Trans. on Software Engineering*, Vol. SE-12, No. 4, April 1986, pp. 547-553.
- [14] J. Livesey and E. Manning, "Protection and Synchronisation in a Message-Switched System", *Computer Networks*, Elsevier Science Publishers, Vol. 7, No. 4, August 1983, pp. 253-267.
- [15] R.E. Seviora, "Knowledge-Based Program Debugging Systems", *IEEE Software*, Vol. 4, No. 3, May 1987, pp. 20-32.
- [16] N.K. Gupta and R.E. Seviora, "An Expert System Approach To Real Time System Debugging", *The First Conf. on Artificial Intelligence Applications*, Los Alamitos, California, CS Press, December 1984, pp. 336-343.
- [17] M. Moran and M.B. Feldman "Toward Graphical Animated Debugging of Concurrent Programs in Ada", *Proc. of Int. Symp. on New Directions in Computing*, Trondheim, Norway, August 1985, pp. 344-351.