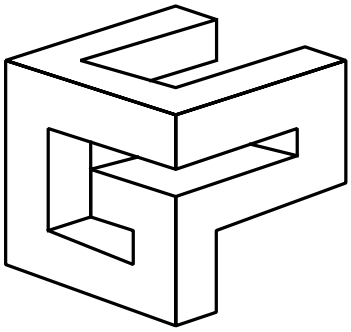
EVENT GRAPH ANALYSIS FOR DEBUGGING MASSIVELY PARALLEL PROGRAMS

DIETER KRANZLMÜLLER



GUP Linz
Dept. for Graphics and Parallel Processing
Joh. Kepler University Linz, Austria

EVENT GRAPH ANALYSIS FOR DEBUGGING MASSIVELY PARALLEL PROGRAMS

(Ereignisgraph-Analyse zur Fehlersuche
in massiv-parallelen Programmen)

DISSERTATION

zur Erlangung des akademischen Grades
Doktor der technischen Wissenschaften

von

DIPL.-ING. DIETER KRANZLMÜLLER

angefertigt am

Institut für Technische Informatik und Telematik
(Abteilung für Graphische und Parallele Datenverarbeitung)
der Technisch-Naturwissenschaftlichen Fakultät der
Johannes Kepler Universität Linz

eingereicht bei
o.Univ.Prof. Dr. Jens Volkert (Betreuer)
Prof. Dr. Bernard Tourancheau (2. Begutachter)
o.Univ.Prof. Dr. Franz Pichler (2. Prüfer)

Linz, September 2000

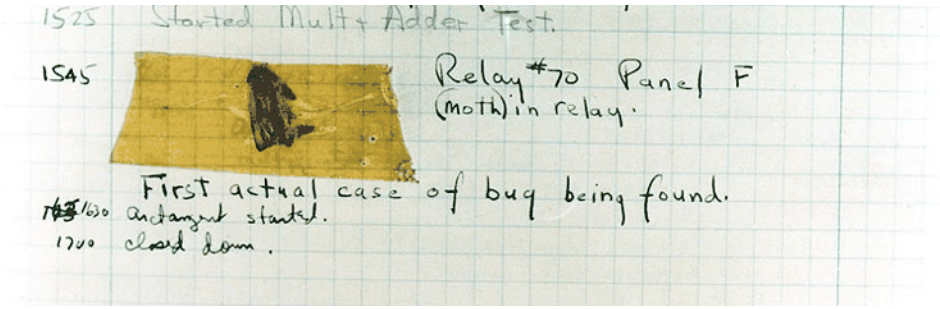
Author contact information:

Office:
GUP Linz, Joh. Kepler Universität Linz
Altenbergerstr. 69, A-4040 Linz, Austria
phone: +43 732 2468 - 9499
fax: +43 732 2468 - 9496
dk@gup.uni-linz.ac.at
http://www.gup.uni-linz.ac.at/~dk/thesis

Private:
Landgutstr. 11, A-4040 Linz, Austria
phone: +43 664 1318466
dieter@kranzlmueller.at
http://www.kranzlmueller.at/dieter/diss

Document version: 4.47
Printed: 11.Dec.2000,11:12

"First actual case of bug being found"
Grace M. Hopper (1906-1992)
Harvard University, 9 September 1945
Moth trapped between points at Relay #70, Panel F,
Mark II Aiken Relay Calculator



Naval Surface Warfare Center, Computer Museum at Dahlgreen, Virginia, USA
U.S. Naval Historical Center Photograph #: NH 96566-KN

Picture: http://www.history.navy.mil/photos/images/h96000/h96566k.jpg
Documentation: http://www.history.navy.mil/photos/pers-us/uspers-h/g-hoppr.htm

Danksagung

An dieser Stelle möchte ich mich bei all jenen bedanken, die zum Gelingen dieser Arbeit durch ihre fachliche und persönliche Unterstützung beigetragen haben.

Mein besonderer Dank gebührt meinem Betreuer Herrn Prof. Dr. Jens Volkert, der mir im Rahmen meiner wissenschaftlichen Arbeiten einerseits die größtmögliche Freiheit zur Entfaltung meiner Ideen, andererseits aber jede benötigte Hilfestellung gewährt hat.

Weiters danke ich Herrn Prof. Dr. Bernard Tourancheau für die Zweitbegutachtung meiner Arbeit, sowie Herrn Prof. Dr. Franz Pichler für seine Bereitschaft als Zweitprüfer beim Rigorosum zu fungieren.

Auch meinen Kollegen an der Abteilung für Graphische und Parallele Datenverarbeitung möchte ich hiermit für die anregenden Diskussion und ihre Hilfsbereitschaft danken, allen voran Herrn Dr. Rainer Koppler der mit mir während der letzten sechs Jahre ein Büro geteilt hat. Erwähnt seien auch die unzähligen Studenten, die mich durch ihr Engagement in Diplomarbeiten, Praktikas und meinen Lehrveranstaltungen motiviert und damit auf immer neue Ideen gebracht haben.

Von den Freunden möchte ich mich bei Dipl.-Ing. Edith Spiegl, Dipl.-Inform. Ralf Reussner, und Christian Schaubschläger bedanken, welche die Mühen des Korrekturlesens auf sich genommen und damit zur vorliegenden Arbeit tatkräftig beigetragen haben.

Schließlich gebührt wesentlicher Dank meiner Familie, hauptsächlich meiner Frau Judith, deren Verständnis und Geduld eine Grundlage für die Vervollständigung der Arbeit darstellten. Durch die Geburt meiner Tochter Miriam habe ich zusätzliche Energie und Kraft für den Abschluss des Studiums bekommen.

Abstract

Error detection and performance analysis of parallel programs is tedious and difficult, especially in massively-parallel environments. Many concurrently executing and communicating tasks result in highly complex applications, which generate huge amounts of state information for the debugging process. Managing these data requires abstraction and steering support from program analysis tools, which can only be achieved with graphical representations of the program execution.

A widely used model of parallel program behavior is the event graph, which can be displayed as a state-time diagram indicating the occurrence of events and their connections in a parallel system. Based on this abstract model, various activities of error detection can be performed, which qualifies the event graph as a suitable debugging interface. Among the debugging techniques are simple analysis features for detecting incorrect communication pairs as well as traditional methods like breakpointing and inspection.

Another important set of event graph analysis techniques covers nondeterministic parallel programs, which exhibit different program behavior in successive executions even if the same input data are provided. Consequences are the irreproducibility effect, the completeness problem, and the probe effect, which require special attention from users and corresponding tool support. The debugging strategy integrated in the event graph model permits to solve these difficulties with sophisticated event manipulation and artificial replay techniques, that allow to steer a program's execution at nondeterministic choices.

A major problem of massively parallel computing is that debugging data grow rapidly with the execution time of a program and the number of participating processes. As a consequence it may often be difficult or even impossible to analyze massively-parallel applications, and many tools are limited to small numbers of processes. With the event graph model, corresponding solutions are provided by automatization and abstraction. Firstly, every debugging approach should be capable of automatically performing as many debugging tasks as possible. This should relieve the users from repeated, time-consuming low-level activities. Secondly, higher levels of abstraction should reduce the complexity of the debugging data and improve the user's understanding of the program's behavior. Since both goals can be achieved with the event graph model, which therefore represents a significant improvement for users during debugging of high performance computing applications.

Kurzfassung

Fehlersuche und Leistungsanalyse in parallelen Programmen ist eine ermüdende und schwierige Aufgabe, besonders in massiv-parallelen Umgebungen. Solche hochkomplexen Anwendungen, in denen eine große Menge an Prozessen miteinander kommuniziert, erzeugen riesige Mengen an Zustandsinformationen für die Fehlersuche. Um diese Daten verwalten zu können werden Werkzeuge benötigt, die entsprechende Abstraktions- und Steuerungsmöglichkeiten basierend auf einer graphischen Darstellung der Programmausführung bereitstellen.

Ein entsprechendes Modell für paralleles Programmverhalten ist der Ereignisgraph, welcher das Eintreten von Ereignissen und deren Zusammenhänge beinhaltet und auf einfache Weise als Raum-Zeit Diagramm dargestellt werden kann. Basierend auf diesem Modell können verschiedenste Aktionen zur Fehlerentdeckung und -verfolgung durchgeführt werden, wodurch sich der Ereignisgraph auch als geeignete Schnittstelle zur Fehlersuche anbietet. Zu den entsprechenden Techniken zählen sowohl einfache Analysen zur Entdeckung von fehlerhaften Kommunikationspaaren, als auch traditionelle Methoden wie Haltepunkte und Inspektionen.

Eine zusätzliche, wichtige Menge an Ereignisgraph-Analysetechniken beschäftigt sich mit nichtdeterministischen parallelen Programmen, die in wiederholten Programmausführungen trotz gleicher Eingabedaten unterschiedliche Ergebnisse liefern können. Die daraus folgenden Konsequenzen sind das Reproduzierbarkeitsproblem, das Vollständigkeitsproblem und der Beobachtungseffekt. Diese erfordern spezielle Aufmerksamkeit von BenutzerInnen und entsprechende Werkzeugunterstützung. Mit einer auf den Ereignisgraph aufbauenden Fehlersuch-Strategie ist es möglich, diese Probleme zu lösen. Durch Ereignismanipulation und künstlicher Wiedergabe können nichtdeterministische Wahlmöglichkeiten während eines Programmlaufes gesteuert werden.

Ein weiteres Hauptproblem des massiv-parallelen Rechnens ist, dass die Menge der zur Fehlersuche benötigten Daten mit der Programmausführungszeit und der Anzahl der teilnehmenden Prozesse schnell ansteigt. Als Folge daraus ist es oft schwierig, wenn nicht gar unmöglich, eine massiv-parallele Anwendung zu analysieren. Viele Werkzeuge sind daher auf kleine Prozesszahlen beschränkt. Mit dem Ereignisgraph stehen entsprechende Lösungen durch Automatisierung und Abstraktion zur Verfügung. Erstens sollte ein Ansatz zur Fehlersuche so viele Fehlersuche-Aufgaben als möglich automatisch durchführen. Dadurch können BenutzerInnen von wiederholten, zeitaufwändigen, niederwertigen Aktivitäten befreit werden. Zweitens sollten höhere Ebenen der Abstraktion die Komplexität der Fehlersuch-Daten verringern und das Verständnis der BenutzerInnen für das Programmverhalten erhöhen. Da beide Ziele mit dem Ereignisgraph-Modell und den dazugehörigen Fehlersuchstrategien erreicht werden können, repräsentiert dieses eine signifikante Verbesserung für BenutzerInnen während der Fehlersuche in Hochleistungsanwendungen.

Table of Contents

Abstract **9**

Kurzfassung **11**

Table of Contents **13**

Symbols, Variables, and Constants **19**

List Of Figures **23**

List of Tables **27**

1. Introduction **29**

2. Area of Application: Parallel Computing **33**

2.1 The Need for High-Performance Computing 34
2.1.1 Grand Challenge Problems 34
2.1.2 Requirements on Computer Systems 36
2.1.3 Teraflop Computing 36
2.2 Supercomputing 37
2.2.1 History of Parallel Computing 37
2.2.2 What is a Supercomputer? 39
2.2.3 The TOP 500 Supercomputers 39
2.2.4 Moore's Law 41
2.2.5 Supercomputer Architectures 43
2.2.6 Massively Parallel Processing 45
2.3 Parallel Programming 46
2.3.1 Basic Approaches 46
2.3.2 Possibilities for Parallelization and Parallel Programming Models 47
2.3.3 Approaches to Implicit Parallel Programming 49
2.3.4 Explicit Parallel Programming in Practice 50
2.3.5 Message-Passing Programs 52
2.4 Summary: Programming the World's Most Powerful Machines 53

3. Problem Domain: Debugging Program Errors **55**

3.1 Software Life-Cycle Models 56

3.2 Testing and Tuning 56
3.2.1 Characteristics 56
3.2.2 Problems of Testing 57
3.2.3 Software Quality 59
3.2.4 The Testing Process 60
3.2.5 Debugging Basics 60
3.2.6 Definition and Classification of Errors 63
3.3 Notorious Errors 67
3.3.1 History of Debugging 67
3.3.2 Lethal Software Errors 68
3.3.3 Airplane Crashes 68
3.3.4 Governmental and commercial mishaps 69
3.3.5 Internet Worm/Hackers 70
3.3.6 The Millennium Bug 71
3.4 Errors in Highly-Complex Systems 72
3.5 An Example of Bug Potential 73
3.6 Summary: Testing and Debugging 76

4. Debugging Parallel Programs - Related Work 77

4.1 Traditional Testing and Debugging Cycles 78
4.1.1 Simple Checking 78
4.1.2 The Testing Cycle 78
4.1.3 The Debugging Cycle 80
4.2 Obstacles During Program Debugging 83
4.2.1 Problems of Sequential Debuggers 83
4.2.2 Fundamental Differences between Sequential and Parallel Debuggers 84
4.2.3 Errors from Process Communication and Synchronization 86
4.2.4 Nondeterministic Parallel Programs 87
4.2.5 Race Conditions 91
4.2.6 The Irreproducibility Effect 93
4.2.7 The Completeness Problem 96
4.2.8 The Probe Effect 99
4.3 Extended Parallel Testing and Debugging Approach 101
4.4 Requirements on a Parallel Debugging Tool 104
4.4.1 Architecture and Functionality of an Arbitrary Debugger 104
4.4.2 Criteria for Developing a Parallel Debugger 105
4.5 Parallel Program Analysis and Visualization 107
4.5.1 Collaborative and Standardization Efforts to Debugging 107
4.5.2 Models of Generic Program Analysis Tool 108
4.5.3 Monitoring Tools 110
4.5.4 Software Visualization 113
4.5.5 Visualization Tools for Program Analysis 115
4.5.6 Additional Levels of Abstraction 117
4.6 Debugging Activities at GUP Linz 120
4.7 Summary 123

5. Modelling Programs with the Event Graph 125

5.1 Inspecting Program States during Execution 126

5.1.1 Program State Changes 126
5.1.2 Objects of Interest 127

5.2 Events and Event Attributes 128

5.2.1 Basic Event Definition 128
5.2.2 Characteristics of Events 130

5.3 Event Ordering and Relations 131

5.3.1 The Observability Problem 131
5.3.2 Methods for Event Ordering 134
5.3.3 The “Happened Before” Relation 140
5.3.4 Event Time Zones 141

5.4 The Event Graph Model 143

5.4.1 Abstract Definition 143
5.4.2 Event Graph Visualization 145

5.5 Summary 148

6. Example: A Simple Event Graph Model 149

6.1 Overview of Event Modelling Approach 150

6.2 Selection of Events 150

6.2.1 Interesting Event Classes 150
6.2.2 Communication Events 151
6.2.3 Mapping Events to MPI Statements 152
6.2.4 Communication Event Data 154
6.2.5 Other Events of Interest 156

6.3 Instrumentation of the Code 157

6.3.1 Applying C-Preprocessor Macros 157
6.3.2 Possibilities for Program Instrumentation 158

6.4 Monitoring the Program’s Execution 160

6.4.1 Basic Operation 160
6.4.2 Monitor Operations 162

6.5 Constructing the Event Graph Model 163

6.5.1 Event Record Structure 163
6.5.2 Partial Event Ordering 165

6.6 Visualization of MPI Programs 168

6.7 Summary 170

7. Basic Debugging Activities with the Event Graph 171

7.1 The Debugging Strategy Based on the Event Graph 172

7.2 Automatic Event Graph Analysis for Error Detection 172

7.2.1 Program Failures 172
7.2.2 Computational Errors 173
7.2.3 Example: Communication Errors 174

7.3 Visualization of the Detected Behavioral Characteristics 178
7.3.1 Error Detection with the Space-Time Diagram 178
7.3.2 Performance Analysis with the Space-Time Diagram 179
7.4 Interactive Event Graph Inspection 181
7.4.1 Inspecting Communication Events 181
7.4.2 Variable Inspection Events 182
7.4.3 Array Inspection 182
7.4.4 Control Flow Analysis 185
7.5 Breakpointing in the Event Graph 185
7.5.1 Characteristics of Breakpoints for Parallel Program Debugging 185
7.5.2 Setting Breakpoints in Parallel Programs 187
7.5.3 Generating Cuts in the Event Graph 189
7.5.4 Additional Notes on Breakpoint Cuts 191
7.6 Summary 192

8. Analysis of Nondeterministic Behavior 195

8.1 Modeling Nondeterministic Behavior with the Event Graph 196
8.1.1 Nondeterministic Program Behavior 196
8.1.2 Responsible Events 197
8.1.3 Random Number Generation 198
8.1.4 Nondeterministic Receives 198
8.1.5 Additional Nondeterministic Events in Parallel Programs 199
8.1.6 Problems During Debugging 200
8.2 Detection of Nondeterministic Events 201
8.2.1 Evaluation of Responsible Events 201
8.2.2 Detecting Nondeterministic Receive Events 201
8.2.3 Some Remarks on Nonblocking Receives 204
8.2.4 Evaluation of Racing Messages 205
8.2.5 Simplifications for Evaluation of Race Condition Candidates 207
8.2.6 Visualization of Nondeterministic Behavior 210
8.3 Re-execution of Nondeterministic Programs 212
8.3.1 Equivalent Execution 212
8.3.2 Record Phase 213
8.3.3 Replay Phase 214
8.4 Debugging in the Presence of Nondeterminism 217
8.4.1 Evaluation of Race Condition Candidates 217
8.4.2 Event Manipulation 218
8.4.3 Cut Placement for Event Manipulation 219
8.4.4 Additional Notes on Exchange Cuts 222
8.4.5 Artificial Program Re-execution 225
8.5 Summary 226

9. Automatization of Nondeterminism Analysis 227

9.1 Problems of Manual Nondeterminism Analysis 228

9.1.1 Complete Nondeterminism Testing 228
9.1.2 Example Problem 1: All-To-One Communication 229
9.1.3 Example problem 2: Data Circulation 231
9.1.4 Example Problem 3: Distance Doubling 235

9.2 Automatic Nondeterminism Analysis Strategy 238

9.2.1 Principal Approach 238
9.2.2 Equivalence of Executions 238
9.2.3 Finger-printing of Executions 240
9.2.4 Possible Optimizations 243

9.3 Ideas for Test Reduction 244

9.3.1 Overview of Ideas 244
9.3.2 Integrating Initial User Feedback 246
9.3.3 Exploiting Control- and Data-Flow Characteristics 246
9.3.4 Estimating Message Arrival Order with Observed Timings 248
9.3.5 Estimating Message Arrival Order with Hardware Characteristics 250
9.3.6 Exploiting Observed Event Graph Patterns 254

9.4 Summary 255

10. Advanced Event Graph Techniques 257

10.1 Requirement and Ideas 258

10.1.1 Problem: Numbers of Processes and Execution Time 258
10.1.2 Principal Idea: Focus Only on Important Aspects 259

10.2 Classification of Event Graph Abstraction Techniques 261

10.3 Basic Techniques for Post-Mortem Abstraction 263

10.3.1 Abstraction Along Time-Axis 263
10.3.2 Abstraction along Process-Axis 265
10.3.3 Combined Post-Mortem Abstraction 267

10.4 Pattern Matching 268

10.4.1 Overview and Principal Idea 268
10.4.2 Specification of Patterns 268
10.4.3 Future Perspectives and Benefits 273

10.5 On-line Abstraction Technique 273

10.5.1 On-line Abstraction Along Time-Axis 273
10.5.2 On-Line Abstraction Along Process-Axis 275

10.6 Summary 277

11. Summary and Conclusions 279

12. References 283

People **309**

Program Analysis Tools **313**

Index **315**

Symbols, Variables, and Constants

$\oplus$	exclusive-or operation	p. 271
$\parallel$	concurrent relation	p. 141
$\rightarrow$	“happened before” relation	p. 140
a_t	instruction at time t	p. 64
$B(0)$	defect potential of software	p. 74
$B(k)$	defect potential after k test-and-fix iterations	p. 75
b_p	breakpoint on process p	p. 188
C	cut (= set of breakpoints)	p. 188
$\overset{C}{\rightarrow}$	concurrent order	p. 140
d	hypercube dimension	p. 231
D	failure density	p. 75
$\mathrm{DEX}\,(i, d)$	dimension exchange pattern along dimension d starting at event i	p. 271
$\mathrm{DEX}_{\mathrm{ev}}\,(i, d)$	set of events in dimension exchange pattern	p. 271
$\mathrm{DEX}_{\mathrm{rel}}\,(i, d)$	relations between events in dimension exchange pattern	p. 271
DO	debugger operations: breakpoints, trace, state-inspection, state-manipulation	p. 104
DS	debugging system	p. 104
E	set of events e	p. 143
e_p^i	i^{th} event on any process p	p. 131
$\mathrm{EX}\,(i, p, q)$	simple exchange pattern between process p and q starting at event i	p. 269
$\mathrm{EX}_{\mathrm{ev}}\,(i, p, q)$	set of events in simple exchange pattern	p. 269
$\mathrm{EX}_{\mathrm{rel}}\,(i, p, q)$	relations between events in simple exchange pattern	p. 269
$f(e_p^i)$	result at nondeterministic event e_p^i	p. 241
$f(x)$	arbitrary mathematical function over input x	p. 78
$f_0, f_1, \ldots, f_n$	subfunctions of f	p. 85
fp_p	finger-print of process p	p. 241
$\mathrm{FP}(x)$	finger-print of a nondeterministic execution	p. 240

FP	number of function points	p. 74
first(E,p)	function to determine first event in set E on process p	p. 209
$\mathrm{FUTURE}\left(e_p^i\right)$	time zone for a given event e_p^i	p. 142
G	event graph	p. 143
GEX (i, d)	global exchange pattern along dimension d starting at event i	p. 272
$\mathrm{GEX}_{\mathrm{ev}}$ (i, d)	set of events in global exchange pattern	p. 272
$\mathrm{GEX}_{\mathrm{rel}}$ (i, d)	relations between events in global exchange pattern	p. 272
k	required test-and-fix iterations	p. 76
last(E,p)	function to determine last event in set E on process p	p. 189
messageLength(e_p^i)	function to determine message length of communication event	p. 177
Mflop/s	million floating-point operations per second	p. 31
M	set of racing messages	p. 196
n	number of processes	p. 230
next()	method to determine next successive event on same process	p. 190
$\mathrm{N}_{\mathrm{max}}$	LINPACK: problem size	p. 40
numProcesses	number of processes (constant)	p. 191
$\mathrm{N}_{1/2}$	LINPACK: problem size for half of $\mathrm{R}_{\mathrm{max}}$	p. 40
p	process	p. 131
PD	program behavior data: arbitrary connections of control flow-data flow, and concurrency events	p. 104
$P(x)$	program P with valid input x	p. 78, p. 196
$\mathrm{PARALLEL}\left(e_p^i\right)$	time zone for a given event e_p^i	p. 142
$\mathrm{PAST}\left(e_p^i\right)$	time zone for a given event e_p^i	p. 142
process_id()	method to determine process identifier of current event	p. 190
R	learning rate during one test-and-fix step	p. 75
`receive`	blocking receive event	p. 151
$\mathrm{R}_{\mathrm{max}}$	LINPACK: maximal possible performance for $\mathrm{N}_{\mathrm{max}}$	p. 40
$\mathrm{R}_{\mathrm{peak}}$	LINPACK: theoretical peak performance	p. 40

`send`	nonblocking send event	p. 151
$\xrightarrow{S}$	sequential order of events	p. 131
S0, *S1*, ..., *Sn*	program states	p. 126
s_t	state of the program at time t	p. 64
$s_t(v)$	value of variable $v \in V$ at time t	p. 64
`test`	nonblocking receive event	p. 151
Tflop/s	10^{12} floating-point operations per second	p. 36
V	set containing all the variables of a program	p. 64
vector()	method to return vector clock of current event	p. 190
x	selected input	p. 78
X	set of possible (and valid) inputs x	p. 78
X_p	exchange sequence (for hypercube data circulation)	p. 232
$\langle x, y \rangle$	input/output pair	p. 78
y	computed output or result	p. 78
Y	set of possible outputs y	p. 97

List Of Figures

1. Introduction 29

2. Area of Application: Parallel Computing 33
2-1 Overlapped design space of clusters, MPPs, SMPs, and distributed computer systems [HwXu 98] 45

3. Problem Domain: Debugging Program Errors 55
3-1 State of a program at a given instruction 64
3-2 Conceptual overview of notions and terms 66

4. Debugging Parallel Programs - Related Work 77
4-1 Traditional testing and debugging cycle 79
4-2 Iterations of the debugging cycle 81
4-3 Comparison of sequential and parallel program influences [KrVo 98] 86
4-4 Testing and debugging cycle for nondeterministic parallel programs 103

5. Modelling Programs with the Event Graph 125
5-1 State changes in a program 126
5-2 Observability effect (compare [Fidg 93]) 132
5-3 Observability effect for multiple observers (compare [Fidg 93]) 133
5-4 Observability effect for incorrect perceived orders (compare [Fidg 93]) 134
5-5 Logical clock ordering with Lamport's approach [Fidg 93]) 136
5-6 Possible total order for Lamport's approach of Figure 5-5 137
5-7 (Partial) logical vector clock ordering [Fidg 93]) 138
5-8 Event time-zones 143
5-9 Example event graph [Kran 96a] 145
5-10 Time tunnel visualization [Reed 95] 146
5-11 Example event graph visualization of a program execution 147

6. Example: A Simple Event Graph Model 149
6-1 Monitoring functionality during program execution 161
6-2 Monitoring activities [Kran 96a] 163
6-3 Simplified event ordering mechanism 168
6-4 Visualization of events observed during program execution 169

7. Basic Debugging Activities with the Event Graph 171
7-1 Basic event graph analysis (communication errors) 179
7-2 Visualization of blocking times 180
7-3 Interactive event inspection (communication errors) 181
7-4 Variable inspection 183
7-5 Array inspection 184
7-6 Errors detected with array inspection 186

7-7 Setting a cut (breakpoints on all processes) 190
7-8 Setting a cut in the event graph display 192

8. Analysis of Nondeterministic Behavior 195
8-1 Nondeterministic receive events 203
8-2 Nondeterministic nonblocking receive events 204
8-3 Nondeterministic receive without effect 206
8-4 Wild card receives and racing messages 211
8-5 Distance doubling: incorrect nondeterministic executions 212
8-6 Testing and debugging in case of nondeterminism 218
8-7 Placement of exchange cut for event manipulation 220
8-8 Cut for exchange (above) and cut for breakpointing (below) 221
8-9 Placing cuts for event manipulation 223
8-10 Possible program executions (resulting from Figure 8-9) 224

9. Automatization of Nondeterminism Analysis 227
9-1 Four executions of a nondeterministic all-to-one communication program . 229
9-2 Three executions of a nondeterministic data circulation program 233
9-3 Executions of a nondeterministic distance doubling program 236
9-4 Tree of executions 237
9-5 Testing cycle for automatic nondeterminism analysis 239
9-6 Complete set of executions of distance doubling program 242
9-7 Control- and data-flow analysis and event graph [Kran 97b] 248
9-8 Probability of message arrival 249
9-9 Results on the nCUBE 2 multiprocessor [Kran 99a] 251
9-10 Cumulative results on the nCUBE 2 multiprocessor [Kran 99a] 252
9-11 Results on the Origin 2000 without process placement [Kran 99a] 253
9-12 Origin 2000 hypercube architecture [Kran 99a] 253
9-13 Results on the Origin 2000 with process placement 254

10. Advanced Event Graph Techniques 257
10-1 Complex event graph displays observed in massively parallel programs 259
10-2 Detecting loop deviations 264
10-3 Process grouping 266
10-4 Process extraction and hiding 266
10-5 Basic event graph patterns 269
10-6 Simple event patterns (round robin, tree) 270
10-7 Scalable patterns 271
10-8 More complex pattern 272
10-9 Detecting almost correct patterns 273
10-10 Debugging and checkpointing 275
10-11 Process isolation 275
10-12 Combined on-line abstraction 277

11. Summary and Conclusions 279

12. References 283

List of Tables

1. Introduction 29

2. Area of Application: Parallel Computing 33
2-1 Excerpt from the TOP 500 supercomputer list [Dong 99] 40
2-2 TOP 500 position N=500 [Dong 99] 42
2-3 TOP 500 position N=1 [Dong 99] 42
2-4 TOP 500 supercomputers: share of architectures over time [Dong 99] 45

3. Problem Domain: Debugging Program Errors 55
3-1 Reliability of commercial UNIX utilities 74

4. Debugging Parallel Programs - Related Work 77

5. Modelling Programs with the Event Graph 125
5-1 Event types and event data 129

6. Example: A Simple Event Graph Model 149
6-1 Example places for event occurrence 155
6-2 Example event record structure for MPI_Isend 163
6-3 Example data of simplified event ordering mechanism 166

7. Basic Debugging Activities with the Event Graph 171
7-1 Communication event data 176

8. Analysis of Nondeterministic Behavior 195
8-1 Receive parameters (standard case) 202
8-2 Receive parameters (non-overtaking case) 209
8-3 Example for record&replay mechanism 216

9. Automatization of Nondeterminism Analysis 227
9-1 Number of executions corresponding to number of processes 230
9-2 Experimental results for 2 iterations of distance doubling 237
9-3 Example finger-print file 242

10. Advanced Event Graph Techniques 257
10-1 Classification of abstraction mechanisms 263

11. Summary and Conclusions 279

12. References 283

1. Introduction

Mankind considers itself as the toolmaker in the Theory of Evolution and consequently the computer is often considered as the most superior tool developed by mankind[1]. Today computers are part of everyday life, and it would be difficult or probably impossible to continue living if computers would somehow be removed from our way of life.

Computers are accepted and ubiquitous[2]. They occur in more and more areas, occupying regions which we may not have thought about applying a "calculating machine" before. Obvious examples are the Internet in every household, personal digital assistants (PDAs), and various kinds of entertainment machines that rely heavily on high processing power. For instance, consider how much computing power is required to produce cinema films today.

Of course, there is also the backside of the medal, and some people believe that there are already too many computers around and many machines are applied for more activities than useful. An example is the Millennium Bug and the related Y2k-discussion (year 2000), where one could ask, why an elevator needs to handle dates and thus may even be affected by the Y2k-problem. More questions about usefulness are raised when people read about refrigerators with display screens and Internet-connection.

Yet, besides "everyday"-computing there is still the traditional field of computing, which is often called "Computational Science & Engineering". In this area, computer simulation is used as the third category of the scientific methodological tripod besides the traditional categories theory and experiment. With computer simulation, theories can be verified without performing expensive and probably impossible experiments. Furthermore, this tripod of science and engineering is often designated as the only stable methodological basis and instrumental laboratory to approach the many existing complex problems, which are crucial to the future of science, technology, and society.

At present, these problems are either unsolved or only insufficiently solved due to the limited existing computing power, and while it might be possible to solve some of them in the near future, it is also obvious that new problems will emerge with even higher demands to computer performance. This group of problems is called the "Grand Challenges", and some examples are climate research and weather forecast, chemical reactions and combustions, astrophysics and cosmology.

As a consequence, there is an ever increasing demand for computing power in this domain, which also drives the high-performance computing research in order to provide faster computers with more memory and disk space. An example is the USA's Department of Energy ASCI Program (Accelerated Strategic Computing Initiative), which seeks to provide Teraflop computing systems far beyond the current level of available performance to meet the needs of nuclear stockpile stewardship. One of the next steps of this program is the ASCI Option White supercomputer, which is being developed by IBM and the Lawrence Livermore National Laboratory to be installed in the year 2000.

This scalable parallel machine is basically a highly improved version of IBM's "Deep Blue", which defeated the world's leading human chess master in a publicized series of games.

1. This view about the computer as a "universal tool" has been discussed by several publications, for example by [Coy 92] where authors and editors relate the basics of computer science to philosophical, social, legal, political, and cultural statements.
2. In [MüSc 99] the authors describe the evolution from the traditional workstation computer to the ubiquitous computer in "everyday" life with examples from different areas of computer science, and offer their opinion about the possibilities and dangers of this development.

It is expected to achieve up to 10 Tflop/s (i.e., 10 trillion floating-point operations per second), will require 17,000 square feet of floor space and draws over 6.2 megawatts for power, cooling, and mechanical equipment - enough power to supply a small town full of air-conditioned homes.

Clearly, the users of this field of computing differ significantly from the users of everyday applications. Their goals are efficiency and feasibility. They know how to program their applications and they know how to target the complete computing system most efficiently at one particular problem. Furthermore, these computers are working days, weeks or even months on solving only one particular problem.

On the other hand, these large scale computers together with their applications introduce new problems, which are established mainly by their complexity, the number of processes, and the amount of data to be processed. Additionally, programming high-performance parallel computers is still rather difficult compared to sequential programming. Often, message-passing computing is compared with assembly language from the old days. However, to achieve a certain level of performance, this low-level programming is often required.

Many problems are observed during the parallel software engineering life-cycle, especially during the debugging and tuning phase. The goal of this phase is to detect and correct critical errors and performance bottlenecks in order to improve the reliability and efficiency of the code, and thus the overall quality of the application. This is achieved by analyzing program states during the execution and interpreting the contents of execution traces. Obviously, the amount of data describing a program state depends on the number of participating processes. Consequently, the size of the traces depends on the number of supervised program states and the overall execution time of the program. Therefore, parallel programs tend to generate huge amounts of trace data and program states from massively-parallel programs can easily exhaust the available memory and thus limit the debugging activities.

The traditional solution to this problem was to down-scale the application to small numbers of processes, smaller problem sizes, and thus shorter execution times. While this may be successful in many cases, it may also be a pitfall with serious consequences for other cases. Parallel debugging tools should provide software-engineers with efficient functionality to cope with the amount of data in debugging traces.

At present, the biggest problem of debugging tools for parallel programs is, that they usually combine several instances of a sequential debugger and integrate them under a common user interface. Thus, they still rely on sequential functionality, which may be insufficient in many cases. Besides that, many tools are base on textual representations which may be inappropriate in many cases to display and manage the inherent complexity of parallel programs. This loads a heavy burden onto the debugging user.

The work described in this thesis differs from existing approaches due to the fact, that debugging activities are based on the event graph model instead of the underlying source code. This parallel program model describes a program's execution by occurring state changes and their interactions on concurrently executing processes, which allows to cope equally with programs based on message-passing and the shared memory paradigm. In addition, this model allows to integrate sophisticated techniques for analyzing small-scale parallel applications as well as massively-parallel programs.

The event graph model also serves as the basis for two other important ideas - automatization and abstraction. With automatic error detection facilities, repeated, time-consuming debugging activities can be carried out by the debugging tool without user interaction, allowing the analysis of even large program traces. Users are relieved from tedious activities and

their focus is drawn onto the important results of this analysis, which should cover the most critical sections of the program.

Another attempt to improve the debugging process is automatic investigation of nondeterministic behavior. Nondeterminism in parallel programs can be observed, when successive executions of a single parallel program delivers different results, although the same input data has been provided. This can be attributed to different orders of memory accesses or variations in the message-delivery order and may often be the source of critical bugs and runtime failures.

The goal of abstraction is to reduce the amount of debugging data presented to the users. With ideas like grouping and hiding of processes, extraction of failures, and pattern matching, the size of the event graph can be reduced without reducing the data in the traces. This allows to display the data without compromising the information contents of the traces. Another idea is the integration of checkpointing mechanisms, which store a snapshot of a parallel program's execution to disk, in order to allow restarting the program from that consistent state. For the debugging cycle this means, that the complete program execution can be divided into smaller time slices, where only some of the slices need to be analyzed.

This thesis is organized as follows. Firstly, we will introduce the target of our approach, which is the high performance computing domain, where supercomputers are trying to solve grand challenge applications. We will briefly describe the currently preferred architectures and how they are programmed. This leads to the general description of debugging parallel programs, and an overview of the current state of work in this area. Most important is the specification of requirements to a parallel debugger, which have not been met by any of the existing tools. We describe the modelling process for the event graph, and how an event graph is generated from the observation of a parallel program run. Based on this simple model, basic debugging activities will be described which have already been implemented in an existing debugging environment. These basic features are improved, leading to extensions required for debugging massively parallel programs, which have been described above as automatization and abstraction.

Notations and conventions

One of the problems of computer science is that often many definitions are used out of their original context with confusing notations and ambiguous meanings, while others are intentionally fuzzy to avoid critical arguing. To cope with this problem, we adapt the notations and conventions as defined by [HwXu 98]. In particular, readers should not be confused with the shorthand notations for basic units of time, byte and bit respectively. The basic unit of time is second, abbreviated as s, while the basic information units are byte and bit, where one byte is 8 bits. A frequently used workload unit is the number of floating-point operations, abbreviated as flop. A unit for computing speed is the number of floating-point operations per second (flop/s), with the prefix indicating the scale of the operation. Thus, we use 1 Mflop/s as the proper notation for the speed of executing 1 million floating-point operations per second. Consequently, we will avoid using 1 Mflops/s, 1 MFLOPS, or 1 Mflops, even though they are often used in the field of computing. Another remark concerns the logarithmic function, which will always be used with base 2 unless otherwise specified.

2. Area of Application: Parallel Computing

This chapter introduces the environment for the work described later in this thesis. The problem domain is part of parallel computing, and it is therefore important to describe the reasons for establishing parallel computing hardware, which can be attributed to the largest problems of science and engineering. This area depends on high-performance architectures in order to solve the memory-intensive problems within a certain amount of time and accuracy. We will try to illustrate the term supercomputer and will give an overview of the current situation. Afterwards, we will briefly introduce the software engineering process in order to direct the reader's attention to the field of parallel program debugging.

- Huge Problems: Grand Challenges
- High Performance Computing: Parallel Computing
- What is a supercomputer?
- How to program: SMP/MPP

2. Area of Application: Parallel Computing

2.1 The Need for High-Performance Computing

2.1.1 Grand Challenge Problems

Parallel computing is sometimes called the "key enabling technology" of modern computers that allows to fulfil the ever-increasing demand for higher and higher performance at lower production costs, which allows sustained productivity in real-life applications [Hwan 93]. Therefore, parallel computing is integrated wherever performance is one of the key issues, and parallelism appears in various forms, such as pipelining, vectorization, concurrency, data parallelism, time sharing, multitasking, multiprogramming, multithreading, and distributed computing at different processing levels. Yet, while parallelism is widely available in every computing platform, from powerful game consoles to mainframes in companies, some problems still continue to require more processing power than available.

At the foremost front of problems are the so-called "Grand Challenges" [Gran 89], [Gran 92], and [Wald 95], which can be defined as follows:

Definition 2-1: Grand challenge problems [WiAl 99]

> A grand challenge problem is one that cannot be solved in a reasonable amount of time with today's computers.

These fundamental problems generate increasingly complex data, require realistic simulations of the processes under study, and demand intricate visualizations of the results [Gran 99a]. These problems often require numerous large-scale calculations and have significant elements of multi-discipline and multi-geographic collaborations [Gran 98]. Their solutions will have a broad economic and scientific impact, not only on the computing industry but probably on the human life in general [Gran 99b]. The following list contains an overview of some problems in the Grand Challenges category as summarized in [Gran 96], which can only give an impression of today's situation, since yet unrecognized application might emerge soon, for example from the expanding multimedia field [Hoss 99].

- Applied Fluid Dynamics
- Meso- to Macro-Scale Environmental Modeling
- Ecosystem Simulations
- Biomedical Imaging and Biomechanics
- Molecular Biology
- Molecular Design and Process Optimization
- Cognition
- Fundamental Computational Sciences
- Grand-Challenge-Scale Applications

Another idea would be to identify the equations, that have to be solved in order to get results for the Grand Challenge problems. These so-called Grand Challenge equations have been collected by [Gran 99a]:

- Newton's Equations
- Schrödinger Equation (time dependent)
- Navier-Stokes Equation
- Poisson Equation
- Heat Equation
- Helmholtz Equation
- Discrete Fourier Transform
- Maxwell's Equations
- Partition Function
- Population Dynamics
- Combined 1st and 2nd Laws of Thermodynamics
- Radiosity
- Rational B-Spline

To be more concrete, we will now give two representative examples of Grand Challenge problems in more detail:

Example 2-1: Grand Challenge problem [WiAl 99]:
Numerical weather prediction (weather forecasting by computer)

The global atmosphere is modeled by dividing it into three-dimensional regions or cells, e.g. cells of size 1 mile x 1 mile x 1 mile which would lead to roughly 5 x 10^8 cells. The conditions in each cell (temperature, pressure, humidity, wind speed and direction, etc.) are computed at time intervals with rather complex mathematical equations and using various conditions from the previous time intervals. If we assume 200 floating-point operations for each calculation, one time step would require 10^{11} floating-point operations. In addition, the calculations for each cell have to be repeated iteratively in order to model the passage of time. Thus, predicting the weather over a time period of 10 days using 10-minute intervals would require 10^4 time steps and 10^{15} floating-point operations in total. Using a computer that operates at 100 Mflop/s (10^8 floating-point operations per second) would than lead to a computation time of 107 seconds or approximately 116 days to perform this calculation. On contrary, to perform this calculation within 10 minutes would require a computer operating at 1.7 Tflop/s (1.7 x 10^{12} floating-point operations per second). If you compare this with the current TOP 500 supercomputer list [Dong 99] as described in Section 2.2.3, only the world's fastest supercomputer, Intel's ASCI Red, achieves more computational speed for the LINPACK benchmark [Dong 94].

Example 2-2: Grand Challenge problem [WiAl 99]:
Astrophysical N-body simulation
(motion prediction of astronomical bodies in space)

Each astronomical body in space is attracted to each other body by gravitational forces. These are long-range forces that can be calculated with a simple formula (see [WiAl 99], page 126), and the movement of each body can be predicted by calculating the total force experienced by the body. If there are N bodies, there will be N-1 forces to calculate for each body, or approximately N^2 calculations in total. Thus, if we assume that a galaxy might contain 10^{11} stars, we would require 10^{22} calculations. Again, these are only the calculations for one time step, while scientists are usually interested in simulating a certain period of time, which means that these calculations will have to be performed iteratively. If we provide a powerful computer, that could perform each calculation within 1µs (10^{-6} seconds), it would still take approximately 10^9 years for one iteration of this N^2 algorithm.

2.1.2 Requirements on Computer Systems

These two examples indicate, that there exist problems, which can only be solved with very powerful architectures that may not even exist today. Some of them require systems of unheralded capability [Sieg 92]. Hence, besides the problems of feasibility, Grand Challenge applications also serve as the representative guides to the development of system software and hardware [Gran 98]. Furthermore, workstations, however powerful they are or will be in the near future, cannot replace the potential of parallel computers, which are basically built upon the technology of powerful microprocessor chips by tying them together via sophisticated broadband interconnection networks in order to support massive parallelism [Hoss 99].

The following three requirements manifest the need for parallel computing:

- Computational speed
- Amount of memory
- Accuracy of results

Clearly, computational speed is the most obvious characteristic required from a parallel computing system. Often, computations must be completed within a “reasonable” time period or have a specific deadline for the computations [WiAl 99]. For example, the predictions of a weather forecasting program that requires two days to compute next days weather is certainly useless.

A second requirement is the amount of memory consumed by this large and complex problems, which can easily exhaust conventional computing systems. However, by coupling large numbers of processors which have access to their local memory, the total amount of memory can be scaled up. Please note, that the amount of memory also affects the computational speed, because obviously the data to be processed has to be transported from memory to CPU and return. This defines the need for a sufficient degree of interconnection bandwidth.

The last requirement is the accuracy of the results, that have to be computed. Yet, this is strongly connected to the other requirements, simply because the higher the computational speed and the amount of memory available, the higher the degree of accuracy that can be achieved. On contrary, restrictions in the available computing power and/or the available memory certainly reduce the obtainable accuracy, and many actual problems cannot be solved with sufficient degree of accuracy by today computers.

2.1.3 Teraflop Computing

The above mentioned requirements have also been visible in the promises of computer manufacturers, who specified their goal for 1995 to be the magical “3 T’s” - 1 Tflop/s (10^{12} floating-point operations per second) in execution rate[1], 1 Tbyte of main memory, and 1 Tbyte/s interconnection bandwidth [Hoss 99]. This goal lead to the definition of key attributes for such a powerful computing system, which have been specified by Hossfeld as [Hoss 99]:

1. Please note, that there is often a discrepancy when specifying actual performance rates of arbitrary computer systems, which mostly stems from the difference between peak performance and sustained performance. While peak performance pictures the best performance results a computer system can gain under optimal conditions, sustained performance represents performance numbers that can be observed during operation.

- Multiple high-performance commodity priced compute nodes from regular commercial product lines (not special-purpose designs)
- Hierarchical memory systems (e.g. cache-only memory architectures and distributed shared memory systems with low-latency memory access)
- Very high performance storage and parallel I/O systems
- Scalable programming environments and operating systems
- A universal programming paradigm

Today, four years later than expected, the 1995 goal in execution rate was achieved, which can be attributed to the Accelerated Strategic Computing Initiative (ASCI) project [ASCI 99]. This $1-billion program launched in 1994 by the US Department of Energy (DOE) seeks to build Tflop/s computer systems within 10 years [Craw 96]. The participating government laboratories - Sandia National Laboratories, Lawrence Livermore National Laboratory, Los Alamos National Laboratory - collaborate in creating the leading-edge computations modeling and simulation capabilities required for maintaining safety, reliability, and performance of US nuclear stockpile and reducing the nuclear danger [Gust 98]. Besides nuclear weapons research, simulation of other areas like biotechnology, medical and pharmaceutical research, weather prediction, aircraft and automobile design, industrial process improvement, environment projection, etc. are targeted [HwXu 98]

The strategies adopted by the ASCI program have two notable features: accelerated development and balanced scalable design. Thus, ASCI aims not just at the peak speed of a particular computer system, but at a total system solution that can deliver sustained application performance at five orders of magnitude better than that in 1994. The scalable design approach defines the following steps [HwXu 98]:

- Focus on high-end platforms for scientific computing
- Use of commodity off-the-shelf (COTS) hardware and software components
- Use of massively parallel architectures

A step further than the ASCI program is targeted by the PetaFlops project [NASA 97]. This project aims at the long term goal to achieve a supercomputing speed of 1 Pflop/s (10^5 flop/s). Recent studies suggest, that such a system could eventually be built in 2007 with exotic technologies and unknown costs. However, the more realistic idea is to build a massively parallel processor with conventional technology and COTS components by year 2015 at a cost of US$ 1 billion [HwXu 98]. Such a system must effectively exploit massive parallelism in the range of thousands of processors to over a million of threads concurrently. In the following section, we will briefly summarize the historical steps that led to the current situation and have been performed to achieve these goals, before presenting an overview of the current state of supercomputer developments.

2.2 Supercomputing

2.2.1 History of Parallel Computing

The history of parallel computing is clearly part of the historical milestones in the development of computers. The earliest machine can be tracked back at least as early as to the Abacus used

in China 500 BC [Hwan 93]. Later, machines capable of performing arithmetic operations were developed in the seventeenth century with the mechanical calculator of Wilhelm Schickhard in Germany 1623, the mechanical counter of Blaise Pascal in France 1642, and the machine of Gottfried Leibniz again in Germany 1671 [Haye 88]. These machines remained academic curiosities until the nineteenth century, where fundamental advances were achieved by Charles Babbage with the Difference Engine (1823) and the Analytical Engine (1834), although both remained uncompleted [Haye 88].

No significant attempts to build general-purpose digital computers were made after Babbage's death until the 1930s, when independently Konrad Zuse in Germany and Howard Aiken in the United States developed their machines. Zuse started with first theoretical ideas in 1934, which lead to the "test models" of mechanical computers Z1 and Z2 in 1936. [Zuse 76]. The subsequent Z3 binary mechanical computer, accepted as the first fully operational general-purpose program-controlled computer, was completed in 1941 [Zuse 86]. On contrary, Aiken proposed a decimal mechanical machine in 1937, which finally lead to the Harvard Mark I computer that was completed by International Business Machines Corp. (IBM) in 1944 [Haye 88].

Since these mechanical computers suffered from limited computing speed as well as cumbersome and unreliable information transmission, electronic machines were proposed as a replacement. The first widely known general-purpose electronic computer was the ENIAC (Electronic Numerical Integrator and Calculator) built in the United States under the direction of John W. Mauchly and J. Presper Eckert in 1946 [Haye 88]. Associated with this project is also the Hungarian-born mathematician John von Neumann, who described the principle architecture of sequential computers.

Starting in 1945, electromechanical computers can be divided into five generations [Hwan 93]. The first generation (1945-1954) consisted of vacuum tubes and relay memories, CPUs driven by a program counter (PC) and an accumulator with fixed-point arithmetic, while the second generation (1955-1964) already integrated discrete transistors and core memories, as well as floating-point arithmetic, input/output processors, and multiplexed memory access. During the third generation (1964-74) integrated circuits, microprogramming, pipelining, caches and look-ahead processors already provided the technology and architecture to perform multiprogramming and time-sharing by the operating system, obviously a start for parallel computing.

However, a parallel computer was not a novel idea at that time [WiAl 99]. In fact, for computing terms it is already a rather old idea, since Gill already wrote about parallel computing in 1958 [Gill 58]. Similarly, Holland wrote about a "computer capable of executing an arbitrary number of subprograms simultaneously" in 1959 [Holl 59], while Conway described the design of a parallel computer and its programming in 1963 [Conw 63]. Around that time, Zuse also developed some ideas concerning parallel processing in his work on the "calculating cosmos" (sometimes also called "calculating space", in German: "Rechnender Raum", [Zuse 69]), which are nowadays mostly related to cellular automata and massively parallel computers. In 1960 Burroughs announced the D825, a four processor machine that was finally delivered two years later to serve in military command and control applications [LeRu 93]. Finally, in 1967 the 64-processor ILLIAC IV, which is often called "The First Supercomputer" [AlGo 94], was designed.

After this initial steps of parallel computing, further technological progress was achieved with developments to support parallelism in many ways. The fourth generation computer systems (1975-90) contained LSI/VLSI and semiconductor memory, multiprocessors, vector

supercomputers, and multicomputers. These systems provided multiprocessor operating systems, high-level programming languages, compilers and environments for parallel processing.

The history can be tracked until today with the fifth generation computers (1991-present) which consist of ULSI/VHSIC processors, memory, switches, high-density packaging, and scalable architectures. With these components, massively parallel processors and heterogeneous processing clusters can be built to target Grand Challenge applications. Flynn and Rudd wrote in 1996 [FlRu 96]: "... the continued drive for higher- and higher-performance systems... leads us to one simple conclusion: the future is parallel".

2.2.2 What is a Supercomputer?

Supercomputer is a term, that is often closely associated with parallel processing. A straight forward definition would identify it as one of the fastest computers available. Obviously, this definition is time dependent and last decade's supercomputers are less powerful than current personal computers [LeRu 93]. Today, all supercomputers employ parallelism in one way or another to attain some of their high performance [Quin 87], although it may sometimes be hidden on low levels of the hardware-architecture (e.g. pipelining in the processor).

Another definition of a supercomputer is given by [Morr 92]:

Definition 2-2: Supercomputer [Morr 92]

> Computer Technology: 1. Any of a category of extremely powerful, large-capacity mainframe computers that are capable of manipulating massive amounts of data in an extremely short time. 2. Any computer that is one of the largest, fastest, and most powerful available at a given time.

More characteristics are added by [HwXu 98], who place supercomputers at the top of the pyramid of computer classes, because their sales volume is very low due to their high price. In addition, a supercomputer integrates many resources into a single system, employs cutting-edge technology, and has the highest performance.

Therefore, we can identify the following characteristics for a supercomputer at any given period of time:

- Fastest available processing speed
- Largest possible memory size
- Biggest physical dimensions
- Greatest monetary cost

Please note, that these characteristics may be appropriate in most cases, but are also too vague to be exact. Thus, we need some kind of metric to establish a list of supercomputers, that must be based on one of the characteristics mentioned above. Since processing speed is obviously the most important identifier in many cases, we will now describe one approach to rank existing supercomputer architectures.

2.2.3 The TOP 500 Supercomputers

Based on their performance, the team around Jack J. Dongarra, Hans W. Meuer, and Erich Strohmeier are maintaining the TOP 500 list [Dong 99], which is published twice a year in June

at the Mannheim Supercomputer conference and in November at the Supercomputing conference series since 1993. The objective of this list is to provide manufacturers, users, and potential users with statistics on high-performance computers. The ranking of the machines is based on the LINPACK benchmark, which belongs to the group of micro benchmarks in the area of numerical computing (linear algebra) [HwXu 98]. A benchmark is a testing program that supposedly captures processing and data movement characteristics for a class of applications. The LINPACK benchmark was created and maintained by Dongarra at the University of Tennessee [Dong 94]. Besides its usage as a benchmark, it is still widely used for real computation work, since it provides functions to analyze and solve linear equations and linear least-squares problems for general, banded, symmetric indefinite, symmetric positive definite, triangular, and tridiagonal square matrices.

For the TOP 500, Dongarra et al used the version of the LINPACK benchmark, that allows the user to scale the size of the problem and to optimize the software in order to achieve the best performance for a given machine. Then, this number reflects "... the performance of a dedicated system for solving a dense system of linear equations." Due to the regularity of the problem the achieved performance is expected to be rather high and give a good correction of a machine's peak performance. By measuring the performance for different problem sizes, the maximal possible performance R_{max} for problem size N_{max} as well as the problem size $N_{1/2}$ for half of the performance R_{max} can be determined. These three numbers are included in the list together with the theoretical peak performance of each machine R_{peak}. Table 2-1 shows an excerpt from the June 1999 list with the data for the top 10 supercomputers. The units for R_{max} and R_{peak} are given in Mflop/s.

Table 2-1: Excerpt from the TOP 500 supercomputer list [Dong 99]

N	Manufacturer Computer	Installation Site Location/Year	# Proc.	R_{max} R_{peak}	N_{max} $N_{1/2}$
1	Intel ASCI Red	Sandia Natl. Labs Albuquerque USA/1999	9472	2121.3 3154	251904 66000
2	SGI ASCI Blue Mountain	Los Alamos Natl. Lab. Los Alamos USA/1998	6144	1608 3072	374400 138000
3	SGI T3E 1200	Government USA/1998	1084	891.5 1300.8	259200 26400
4	Hitachi SR 8000/128	University of Tokyo Tokyo Japan/1999	128	873.6 1024	120000 16000
5	SGI T3E 900	Government USA/1997	1324	690.90 1191.6	229248 26880
6	SGI Origin 2000/250	Los Alamos Natl. Lab./ACL Los Alamos USA/1999	2048	815.1 1024	134400 80640
7	SGI T3E 900	UK Meteorological Office Bracknell UK/1997	876	552.92 788.4	- -
8	IBM SP Silver	IBM Poughkeepsie USA/1998	1952	547 1296	244000 58000

Table 2-1: Excerpt from the TOP 500 supercomputer list [Dong 99]

N	Manufacturer Computer	Installation Site Location/Year	# Proc.	R_{max} R_{peak}	N_{max} $N_{1/2}$
9	SGI T3E 900	Naval Oceanographic Office Bay Saint Louis USA/1999	812	515.1 730.8	- -
10	SGI T3E 1200	UK Centre for Science Manchester UK/1998	612	509.9 734.4	- -

The most interesting entries in this table are the number of processors and of course the achieved performance. As can be seen, there are only two machines (N=1 and N=2) with more than 6000 processors, while all other machines are less than 2048 processors. Accordingly, only the first two machines achieve a performance above 2 and 1 Tflop/s respectively, while all other machines are still below this barrier.

In terms of architectures, the TOP 500 list contains only the following machines:

- One Intel: ASCI Red
- One Hitachi: SR8000
- One IBM: SP Silver
- One SGI: ASCI Blue Mountain
- One SGI: Origin 2000/250
- Five SGI: T3E 900/1200

Please note, that the SGI ASCI Blue Mountain machine is actually composed of 48 Origin 2000 machines with 128 nodes each. Thus, it would be possible to identify this machine as a superscale SGI Origin 2000. As can be seen, SGI (Silicon Graphics) is currently the company with the most supercomputer architectures, which can be attributed to the acquisition of Cray Research, one of the traditional supercomputer manufacturers.

Another interesting fact, that is only indirectly given is the achieved level of peak performance, which is 67% for the Intel ASCI Red machine, but only 52% for the SGI ASCI Blue Mountain. The best result compared to the peak performance is reached by the Hitachi SR8000/128 (N=4), which achieves 85% although it consists of the smallest number of processors (128). The worst level of utilization for the LINPACK benchmark is obtained with the IBM SP Silver (N=8), that achieves only 42% of its possible peak performance with a very high number of processors (1952).

2.2.4 Moore's Law

The list of Table 2-1 only gives an impression of the situation at present, while an overview over a few years confirms, that the term supercomputer depends on the developments of the computer architecture in general. For example, while in June 1993 the SNI VP200 with 0,4 Gflop/s was a supercomputer, today's last position in the supercomputer list is occupied by the SGI T3E900 with 24.73 Gflop/s on 38 processors (see Table 2-2). This speedup factor of 62 is even bigger than for the number one supercomputer, which offers a speedup of almost 36 for the same time period (see Table 2-3).

Table 2-2: TOP 500 position N=500 [Dong 99]

Date	Performance	Computer	Installation Site
06/93	0,4 Gflop/s	SNI VP200	Debis Munich, Germany
06/96	3,3 Gflop/s	Convex SPP 1000/XA-32	Uni Michigan, USA
06/99	24,73 Gflop/s	SGI T3E900	CIEMAT, Spain

Table 2-3: TOP 500 position N=1 [Dong 99]

Date	Performance	Computer	Installation Site
06/93	59,7 Gflop/s	TM CM-5/1024	Los Alamos, USA
06/96	220,4 Gflop/s	Hitachi SR2201/1024	Uni Tokyo, Japan
06/99	2.121,3 Gflop/s	Intel ASCI Red	Sandia, USA

The phenomenon of exponential improvement was already observed as early as 1979 by Intel cofounder Gordon Moore [Moor 96], and can be interpreted in three ways [HwXu 98]:

Every 18 to 24 months delivers

- the double number of transistors on a microchip for the same price of the chip.
- the double speed of the microprocessor for the same price of the chip.

while at the same time

- the price of the microchip drops about 48% assuming the same processor speed and on-chip memory capacity.

One of the consequences of Moore's law is, that predictions about "the future supercomputer" are difficult, because technology changes constantly and will be completely different within a few years [Simo 99]. Considering the performance of the TOP 500 list, Moore's law would have expected an increase in performance of about 8 to 16 for the 6 years between June 1993 and June 1999. This has clearly been outdone by reality, with a speedup factor of 36 for that period. For the future, the ASCI program expects to extend from 1 Gflop/s in 1994 to 100 Tflop/s in 2004, which calls for a performance increase of 10^5 within this 10 years [HwXu 98]. Yet, according to Moore's law, such a 10^5-fold performance increase with constant hardware costs would require 26 to 32 years, which implies that 100 Tflop/s could not be achieved until year 2025.

Please note, that benchmarking is only one popular method to characterize the performance of applications and systems. Other performance metrics may be more appropriate for other areas of computing. Some well-known benchmarks are SPEC, NAS, STAP, LMBENCH, STREAM, PARKBENCH, Splash, TPC (Transaction Processing Performance Council) [HwXu 98], and SKaMPI [Reus 98].

2.2.5 Supercomputer Architectures

After showing one of the most well-known approaches to identify the world's most powerful supercomputers and their current representatives, we will now analyze their hardware architecture and operation. Of course, there are different possibilities to characterize these machines and clearly, there are several different classification schemes [Volk 97]. One of the best-known schemes is the Flynn classification [Flyn 72][Flyn 96], that divides computers by identifying the number of instruction streams and the number of data streams. The four basic classes of Flynn are:

- SISD: Single Instruction stream, Single Data stream
- SIMD: Single Instruction stream, Multiple Data streams
- MISD: Multiple Instruction streams, Single Data stream
- MIMD: Multiple Instruction streams, Multiple Data streams

Another approach is the Erlanger Classification Scheme (ECS) developed by Händler [BoHä 83]. It describes parallel computer architectures based on the number of control and processing elements that are available for problem solving. The notation divides the architectures into pipelining and concurrency principles, as well as mixtures between these two categories [Quin 87].

These first two approaches to classify parallel systems are still widely used. Yet, today another way of distinguishing supercomputers seems to be equally appropriate. Kai Hwang and Zhiwai Xu identify five different classes of parallel architectures [HwXu 98][2]:

- PVP - Parallel Vector Processors
- SMP - Symmetric Multiprocessors
- MPP - Massively Parallel Processors
- DSM - Distributed Shared-Memory Machines
- COW - Clusters of Workstations

Most of today's parallel machines are built with commercially available, off-the-shelf commodity hardware and software components. The only exception are PVP machines, where many building blocks are custom-made. Such systems contain a small number of powerful custom-designed vector processors, each capable of at least 1 Gflop/s performance. Examples for PVPs are the Cray C-90, the Cray T-90, and the NEC SX-4.

In contrast to the PVPs, SMPs apply commodity microprocessors with on-chip and off-chip caches, and access a shared memory through a high-speed bus architecture. These systems are symmetric, because every processor has equal access to the shared memory, the I/O devices, and the operating system services. A problem of SMP architectures is scalability, which is limited mainly due to the use of a centralized shared memory and a bus or crossbar system inter-

2. Please note, that the definition of some acronyms changed over time, so that some of them may be misleading if approached with their original meaning. For example, PVP is often also called "Pseudo Vector Processing" (e.g. in the Hitachi SR 8000 [Hita 99]), while SMP used to be "Shared Memory Multiprocessing" in former days [Hwan 93]. Another example is MPP, which sometimes addresses the Goodyear Massively Parallel Processor [Will 90], a SIMD parallel architecture [AlGo 94], and sometimes the Cray/MPP series, which are now being called T3D and T3E respectively [Hwan 93].

connect, which are both difficult to scale once built. Examples for SMPs are the IBM R50, the SGI Power Challenge, and the DEC Alpha server 8400.

The last three groups of parallel architectures, MPPs, DSMs, and COWs, are distributed memory machines, which allow to overcome the limitations of scalability and to take advantage of higher parallelism available in some applications of scientific computing, engineering simulation, signal processing, and data warehousing.

The group of MPPs generally refers to very large-scale computer systems. They apply commodity microprocessors with distributed memory as processing nodes, which are interconnected with a high communication bandwidth and low latency network, that can be scaled up to hundreds or even thousands of processors. Programs on these machines consist of multiple processes, each with its private address space, that communicate via message-passing routines. Synchronization takes place through blocking message-passing operations instead of shared-variable synchronization operations.

The DSM machines provide special hardware and/or software extensions, that provide the application user with a single address space although the memory is physically distributed among different nodes. Examples for DSM machines are the Stanford DASH architecture and the Cray T3D.

The last group of distributed memory machines is based on the cluster concept as implemented in Digital's TruCluster, IBM's SP2, and the Berkeley NOW. In a certain sense, Clusters are the low-cost variation of MPPs, which makes them an interesting choice for many research institutions ([Buyy 99a] and [Buyy 99b]). One important characteristic of COWs is that each node is a complete workstation (probably minus peripherals like monitor, keyboard, etc.) which is connected to the other nodes through a low-cost commodity network (e.g. Ethernet, FDDI, ATM, or Myrinet [Pryl 99]). In contrast to MPPs, the network interface is only loosely coupled to the I/O bus of a node, and each node in a COW usually contains a local disk as well as a complete operating system, while MPPs may not provide a local disk at each node and only a microkernel.

However, it has to be noted, that the boundaries between these classes of supercomputer architectures are fuzzy, since it is often difficult to assign a selected hardware architecture to only one group. For example, the node of a COW may be an SMP itself, while DSM machines can also be implemented with software techniques on networks of workstations. Other examples are highly scalable clusters, which are often considered MPPs. Therefore, SMPs, MPPs, DSMs, and COWs are four overlapped architectural concepts as shown in Figure 2-1 [HwXu 98]. For a more detailed discussion about the differences and similarities between these architectures, please take a look at [HwXu 98] pages 31 to 36.

Another remark concerns the possibility extend the clustering idea into large-scale systems by connecting distributed supercomputer systems into so-called "Metacomputers" [Brun 99], "Metasystems" [Grim 98], "Computational Grids" [FoKe 99], or the "Information Power Grid" [LeKu 99]. A well-known example is the multi-institutional Globus project [FoKe 98], which tries to enable computational grids that provide pervasive, dependable, and consistent access to high-performance computational resources for distributed users and resources. Such systems are composed from loosely coupled hardware architectures in order to perform computation on one particular problem [FoKe 99]. Therefore, they are often addressed as the most-powerful computing platforms around.

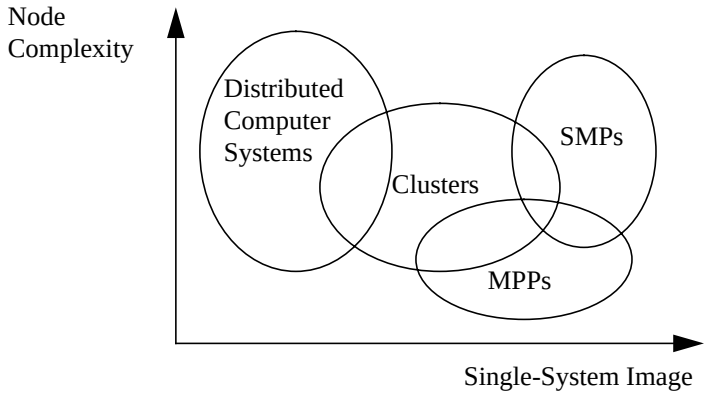


Figure 2-1. Overlapped design space of clusters, MPPs, SMPs, and distributed computer systems [HwXu 98]

2.2.6 Massively Parallel Processing

Analysis of the TOP 500 list according to the classification of [HwXu 98] shows, that the most important supercomputers are certainly MPPs, followed by SMPs. Yet, this was not always the case, as is indicated by Table 2-4. In fact, in June 1993 vector processor machines made up over two thirds of the most powerful machines worldwide. During the last 6 years the situation changed and PVPs vanished completely from the TOP 500 list, which is now determined by MPPs (and clusters of SMPs arranged as MPPs). This is also reflected by the top 10 entries, which are all considered massively parallel architectures.

Table 2-4: TOP 500 supercomputers: share of architectures over time [Dong 99]

Date	MPPs		SMPs		PVPs		Others	
06/93	156	31,2%			344	68,8%		
06/96	265	53,0%	101	20,2%	134	26,8%		
06/99	358	71,6%	118	23,6%			24	4,8%

An example is the current leader of the supercomputing list, the Intel ASCI Red machine installed at the USA's Sandia National Labs. This machine is a successor to the Intel Paragon MPP system and follows therefore the traditional distributed-memory MPP approach, with small nodes, a tightly coupled network interconnection, and a microkernel operating system (LWK - Light-weighted kernel) on the compute nodes. The compute nodes consist of two Pentium class processors with up to 256MB of memory each.

Other example MPP architectures in the top 10 are the SGI Cray T3E series, which shipped in 1995 as a successor to the Cray T3D. This machine is considered a distributed shared-memory multiprocessor, where the processing elements (6 to 2048 DEC Alpha 21164 microprocessors) are interconnected by a 3D bidirectional torus network for fast communication. The processing elements apply the UNICOS/mk operating system and can address between 64 MB and 2 GB memory space. The main difference between the T3E-900 and the T3E-1200 is a

faster processor clock (450 Mhz vs. 600 MHz Alpha processors), thus offering higher processing speeds for the machines (900 Mflop/s vs. 1200 Mflop/s) [HwXu 98].

Another approach to constructing MPP-class architectures is offered by the SGI Cray Origin 2000. Again, the memory is physically distributed among the processing nodes, but it is globally accessible with hardware support for cache coherency (while the T3E supports distributed shared memory without hardware support for cache coherency). Another difference are the processing nodes on the Origin 2000, which consist of 2 MIPS processors combined on one board. In addition, the interconnection network resembles a hypercube architecture. SGI calls the Origin 2000 approach scalable shared-memory multiprocessing (S^2MP) [LaLe 97].

However, the differences between the machines are blurry. For example, the Hitachi SR8000 Super Technical Server [Hita 99] is intended as a replacement system for former vector supercomputers. Yet, the traditional vector processing units have been replaced with pseudo vector processing facilities, that allow to fetch data from memory in a pipelined manner without stalling the succeeding instructions in order to fed the data effectively into the arithmetic units. Each of the 128 SR8000's nodes is formed from multiple RISC microprocessors, which operate on a single address space.

Another example is IBM's SP Silver at position 8 of the TOP 500 list. This machine is at present the most powerful SP (Scalable Powerparallel) architecture, which became famous by defeating World Champion Garry Kasparov in a chess match [HaGa 97]. While it is often considered an MPP, it is built upon a cluster architecture with a proprietary high performance switch as the communication network [HwXu 98].

This small discussions about only the top 10 supercomputers already revealed some of the problems when analyzing parallel machines. A clear classification is difficult if not impossible to achieve. In the next section we will try to identify the paradigms needed to program these architectures, and we will eventually see, that similar problems occur on the software side.

2.3 Parallel Programming

2.3.1 Basic Approaches

Building powerful machines as mentioned above is only one part of the story. Obviously, such supercomputers must be programmed and dedicated software must be developed to obtain the desired performance. Software development for any system is difficult and time consuming, but it is even more difficult when dealing with the details of parallelism [ElRe 93]. Quinn suggests the following three ways to design a parallel algorithm to solve a particular problem [Quin 87]:

- Detect and exploit any inherent parallelism in an existing sequential algorithm
- Invent a new parallel algorithm
- Adapt another parallel algorithm that solves a similar problem

While sequential programming is already widely accepted and many people have been practicing it for a long time, parallelism is sometimes viewed as a rare and exotic subarea of computing, interesting and intellectually challenging but of little relevance to the average programmer [Fost 94]. Furthermore, it is generally agreeable that parallel programming is in a sorry state, and we can identify the following reasons for its inferiority [HwXu 98]:

- Parallel programming involves all issues of sequential programming plus many more issues for managing the parallel complexity.
- Sequential programmers rely only on the von Neumann model, where a processor executes one sequence of instructions, while there exist many different parallel programming models [Fost 94].
- Software environments (compilers, debuggers, profilers) are much more advanced for sequential program development.
- Sequential programming is more mature, with a huge base of accumulated knowledge, including mistakes discovered and lessons learned in the past.

Foster reports about studies of trends in applications, computer architecture, and networking which indicate that this view is changing. He believes that "parallelism is becoming ubiquitous, and parallel programming is becoming central to the programming enterprise" [Fost 94]. This view is supported by Hwang and Xu, who describe recent advances in this area [HwXu 98]:

- Many parallel algorithms have already been developed or are currently being investigated.
- A small set of simple parallel algorithmic paradigms has emerged and is becoming accepted (e.g. phase parallelism, divide and conquer, pipelining, etc.).
- Native models are converging either towards the single-address space, shared-variable model for PVPs, SMPs, and DSMs, or towards the multiple-address space, message-passing model for MPPs and COWs.
- High-level models are converging toward three standard models, data parallel, message-passing, and shared-variable.

Before discussing these models, we will briefly identify the fundamental requirements for parallel software. Regardless of the architecture of the target parallel computer the challenge of parallel programming is to harmoniously coordinate several program segments while assuring correctness as well as high performance [Lewi 93]. Due to the fact that good performance can only be achieved if an algorithm fits the architecture [Quin 87], extensive interaction between algorithm designers and computer system architects is required [Sieg 92]. Thus, it is necessary to investigate parallel programming paradigms in connection with the intended hardware architectures.

2.3.2 Possibilities for Parallelization and Parallel Programming Models

In principle, every parallel program is simply a collection of concurrently executing processes, which may be connected to one another through either message-passing or access to shared data. If these processes operate independently of each other, the program is called "trivially parallel" [Lewi 93]. This can be desired if a high-performance parallel computer is viewed as a huge workstation with a single system image. Then, users may want to take advantage of the enhanced processing and storage capabilities to run multiple sequential jobs. This mode of operation is called "throughput processing" [HwXu 98] or the serial program parallel system (SPPS) model [Pfis 97].

Although the "trivially parallel"-model is certainly important for many different applications, it is less interesting in the scope of this work because the actual program is still sequen-

tial. More challenging are programs with processes that communicate and coordinate among themselves. For such programs, we distinguish the following characteristic items:

- Programming model
  - Data-parallel programming
  - Task-parallel programming
- Parallelization approach
  - Implicit parallelism
  - Explicit parallelism

The parallel programming model identifies, how a program synchronizes its parallel parts. In the data-parallel or Single-Program-Multiple-Data (SPMD) model, the execution of the code is driven by the program's data, which means that the same code is executed on different data domains [Hwan 93]. Therefore, the application's data are partitioned into separate sets, and identical operations are performed on all sets at the same time whenever possible. Obviously, the data-parallel approach is only meaningful, if large arrays of data are processed, e.g. with vectors and matrices [Lewi 93]. This limitation does not exist in the task-parallel (also called control-parallel) or Multiple-Program-Multiple-Data (MPMD) model, which supports more than one thread of control. A single program can perform different operations in the same time interval, and different programs may operate on different data within one application. Additionally, the order in which concurrent parts are executed is governed by program control rather than by the availability of data.

Both types of programs, SPMD and MPMD, can be executed on MIMD architectures [Flyn 72], because different instructions can be executed by different processes at the same time. However, SPMD programs may also be more restrictive, if they permit not only multiple processes executing the same code, but also require them to execute the same instruction at the same time. Therefore, such restrictive SPMD programs may also be executed on SIMD architectures [HwXu 98].

In addition to the parallelism model, the parallelization approach describes different methods to develop corresponding programs for parallel machines. We distinguish two different ways of expressing the programs parallelism in the source code, implicit and explicit parallelism [Bemm 92]. Implicit parallelism means that the program's parallelism is hidden from the programmer, who applies accustomed abstractions as available in sequential programming languages. The explicit parallelization approach is characterized by the active role of the programmer in exploiting the parallelism; the applied programming models offer explicit constructs for describing the concurrent execution as well as communication and synchronization between the processes.

Please note, that it is often difficult to draw a clear line between the implicit and the explicit parallelization approach. In fact, for some people implicit parallelism would mean to write purely sequential code and leave all the parallelization work to the compiler and the optimizer (see [Bode 95]). For them, even adding declarative statements to the code would mean to introduce some explicit means of parallelization. This is for example the case in HPF-dialects (see below), which offer declarations to improve the compiler's results during data partitioning. Yet, these declarations are added in comments, so that every HPF program is still applicable to sequential machines. For that reason, the usual interpretation of explicit parallelism is, when the control flow of the program is actually determined by dedicated statements (e.g. for sending and receiving messages).

Another more practically oriented approach to characterize parallel programming is provided by Foster [Fost 94] as well as by Hwang and Xu [HwXu 98]. While Foster concentrates on the explicit parallelization approaches, Hwang and Xu also include the implicit approach. In concrete, they identify the following four models that are used on todays existing parallel computers:

- Implicit
- Data-parallel
- Message-passing
- Shared variable

Although this list is somehow contrary to the classification described above and mixes some areas, it is well-established in the parallel computing community due to its relevance in practice. We will also stick to these items and describe them in more detail below.

2.3.3 Approaches to Implicit Parallel Programming

In the implicit model, parallelism is automatically exploited by the compiler and the run-time support system [HwXu 98]. The programmer does not have to explicitly specify parallelism using special language constructs, compiler directives, or library function calls. The underlying system is concealed from the user, which shields programmers from the increased complexity of parallelism and shifts the burden to the compiler writer [ElRe 93]. The most popular approach of implicit parallelism is automatic parallelization of sequential programs. Based on dependence analysis a parallelizing compiler or restructuring compiler performs a suite of program transformation techniques on a sequential code in order to derive the parallel program.

The problem of parallelizing compilers is that their effectiveness is often questioned. Several performance studies indicate, that the generated code derived from real sequential programs is often not very effective [BlEi 92][HwXu 98]. Thus, a common approach is to include some user knowledge into the parallelizing process by providing the compiler with more information. This can be done with compiler directives or based on interaction between the compiler and the programmer. Another idea is run-time parallelization, where some of the parallelism, that cannot be recognized during compile time, is revealed at runtime.

The implicit parallelism approach is an interesting research issue due to its many advantages. The huge repository of existing sequential software can be reused for parallel computers. Programmers familiar with sequential languages do not need to know about parallel programming or parallel architectures to exploit their parallelism. In addition, it is easier to understand the semantics of implicit programs than of explicit ones. The resulting programs are usually portable among different parallel computers.

However, there are also many people who believe, that explicit parallelism is needed and comparable results can never be reached by parallelizing compilers. Their belief stems partially from the achieved performance of current automatic compilers [HwXu 98], and partially by the theory of Bernstein, who showed that the general problem, whether two segments of a sequential program can be executed concurrently, is undecidable [ZiCh 90]:

Theorem 2-1: Bernstein’s theorem [Bern 66]

> In the general case it is undecidable whether two arbitrary operations in an imperative sequential program can be executed in parallel.

This theorem implies, that there is no automatic technique, neither compile time nor run time, that can fully exploit all potential parallelism in a sequential program. As a consequence to overcome this theoretical limitations, two solutions are suggested: Either apply explicit parallelism as discussed below or use a programming language which makes parallelism recognition easier and abolish the imperative style altogether. The latter is an important research issue, which includes besides others the following computational styles [HwXu 98]:

- Functional programming
- Logic programming
- Computing-by-learning
- Object-oriented programming

There are some advantages of these approaches, for example a purely functional program is coherent, meaning it is guaranteed to be determinate, deadlock-free, and compositional, and it is abstract, meaning there is no need to explicitly specify parallelism, communication, synchronization, scheduling, etc.

However, there are also advantages in the imperative style. Firstly, it is mature, time-proven and familiar to most programmers. Secondly, it is derived from the von Neumann model with commodity hardware and can therefore be implemented efficiently. Thirdly, the imperative style dominates programming of today's sequential and parallel computers. As a consequence, we will now focus on the explicit parallel programming models with imperative computational style.

2.3.4 Explicit Parallel Programming in Practice

In terms of explicit parallel programming models, many different approaches and languages have been proposed in the past, with different intentions and advantages. From a practical point of view especially three groups of languages became dominant over all others. These groups and some of their representatives are [HwXu 98]:

- HPF-like: Fortran 9x, HPF, CM C*
- Message-passing: PVM, MPI, SP2 MPL, nCUBE 2
- Shared-variable: X3H5, OpenMP, Cray Craft, SGI Power C

The HPF-like group (often called the data-parallel languages) assumes a single address space, removing the need for data allocation. However, many of these languages allow to apply data allocation with explicit directives to improve the memory accesses. In addition, HPF-like programs are single-threaded, which ensures that they are determinate and deadlock-free, and loosely synchronous, because there is no need for explicit synchronization.

For the HPF-like group, there are two influential standard languages: Fortran 9x and High Performance Fortran (HPF). Programs written in these languages are portable on a wide range of machines (MPPs, DSMs, SMPs, COWs, and PVPs). Furthermore, there exist machine-specific languages, such as C* for the CM-5 (Thinking Machines Connection Machine), which are not standardized and thus not portable.

In message-passing programs, users must explicitly allocate data and workload to processes. Such programs are multithreading because multiple threads of control exist that may possibly execute different code. Therefore, both control parallelism (MPMD) and data parallelism (SPMD) are supported. In order to reduce designing and coding complexity, many par-

allel message-passing programs are implemented as SPMD programs, where only a single code is executed on all processes. (Consider an MPP with over 1000 processes, where each of them would execute a different program!) Furthermore, every MPMD program can be formulated as an SPMD program.

These programs execute asynchronously, which requires special operations to synchronize processes for correct execution order (e.g. barrier, blocking communication). Due to separate address spaces of the processes, the data variables in one process are not visible to other processes. Explicit interaction operations, including data mapping, communication, synchronization, and aggregation must be resolved by the user. This introduces the "owner-compute" rule [HwXu 98], where the process that owns a piece of data performs the computations associated with it.

An advantage of the message-passing over the HPF-like languages is its flexibility. Yet, it still lacks support to manage a global data structure efficiently. In addition, message-passing programs usually exploit large-grain parallelism, which can be attributed to high communication latencies occurring during message transfer. For programming, there exist two widely used libraries, PVM and MPI, which will be discussed in more detail in Section 2.3.5. Both of them are implemented in virtually all types of parallel computers, thus ensuring a high level of portability for message-passing programs.

The last of the three explicit parallel models is the shared-variable model, often called shared-memory model. It is comparable to the HPF-like approach because it offers a single address space with global naming where all shared data reside. Consequently, there is no need for data allocation. Furthermore, it is similar to the message-passing model due to its multithreading and asynchronous behavior, which again requires explicit synchronization methods for maintaining correct execution order. Yet, communication can take place implicitly through variable writes and reads in the shared memory.

A widespread misconception is that the shared-variable model is generally and in any case better suited for fine-grain parallelism than the message passing model [HwXu 98]. This is not true because as a model it can be implemented on any parallel architecture, be it PVP, SMP, DSM, MPP, or COW. On contrary the only requirement for fine-grain parallelism is the existence of efficient communication and synchronization methods on the underlying platform. This includes, that a shared-variable program may incur higher interaction overhead and run more slowly than a message-passing program on a COW, an MPP, or even on an SMP.

The advantage of the shared-memory model is that it is assumed to be easier to program than message-passing programs [Karp 87]. Some people consider message-passing as the low-level approach to parallel programming, while shared-memory is on a higher-level. While this popular statement is not wrong, it is also not an established fact backed by scientific research. Fact is that there are many more existing applications for SMPs and PVPs, than there are for MPPs and COWs [HwXu 98]. Developing an efficient parallel program which is loosely synchronous and operates in a regular communication pattern does not necessarily favor the shared-variable approach, but may also be implemented with message-passing. The shared-variable model has an edge for irregular parallel applications, and wherever global pointer operations are required. Shared-variable programs do not require explicit partitioning and data allocation, although their absence may affect performance badly.

One big disadvantage of shared-memory programs is, that subtle synchronization mistakes may probably occur easily and more frequently than in message-passing programs (see Section 4.2.3) [Karp 87]. In addition, there is no widely accepted standard yet, that allows to port programs easily between architectures [HwXu 98]. Several solutions to this problem are in the

line, most notable the three shared-memory programming standards ANSI X3H5, OpenMP, and POSIX Threads.

The ANSI X3H5 standard has not gained wide acceptance, but has substantially influenced the design of several commercial shared-memory languages. In fact, no commercial shared-memory systems adhere to X3H5 completely, although some parts are supported [HwXu 98]. The X3H5 definition consists of a conceptual standard programming model and bindings for C, Fortran 77, and Fortran 90. Its main ideas are parallelism constructs, parallel blocks, parallel loops, implicit barriers, thread interaction, and synchronization.

Furthermore, it was the basis for the OpenMP standard [Thro 99], which is now being supported by a group of hardware and software vendors, such as DEC, Intel, IBM, Kuck&Associates, SGI, Portland Group, Numerical Algorithms Group, US DOE ASCI program, etc. [HwXu 98]. It is a set compiler directives, library routines, and environment variables that collectively provide a shared-memory API (Application Programming Interface) for different platforms and operating systems. The key features of OpenMP are compiler directives, called orphan directives, to enable parallel execution of major portions of the code with small code modifications. Besides that, OpenMP provides a set of run-time library routines with associated environment variables, which are used to control and query the parallel execution environment, to provide general-purpose lock functions, set execution mode, etc.

The POSIX Threads, also known as Pthreads, stands for the official IEEE POSIX 1003.1c-1995 thread standard, which was established by the IEEE standards committee. It offers function for thread management (create, exit, join,...), thread synchronization (lock, unlock, wait, post,...), and others. It is widely available as an influential standard for multithreading at the Unix operating system layer. An example are Solaris Threads, which offer many common features of Pthreads and are a mature commercial product.

2.3.5 Message-Passing Programs

After providing an overview of current parallel programming approaches, we will now briefly describe the message-passing standards MPI and PVM. We choose message passing because it is probably the most widely used parallel programming model today [Fost 94]. In addition, the work described in the later sections builds upon message-passing systems.

The group of existing message-passing systems is very large, for example nCUBE's Vertex, Thinking Machines CMMD, Chameleon, Intel's Nx, P4, PICL, PARMACS, and Zipcode. Most of them have been implemented explicitly for one dedicated hardware architecture or series. Thus, programs are not very portable. This was noted before and inspired Geist, Heath, Peyton, and Worley at Oak Ridge National Laboratory to implement PICL, the Portable Instrumented Communication Library [Geis 90][Worl 92]. A similar idea was followed by Butler and Lusk at Argonne National Laboratory with the p4-library (Portable Programs for Parallel Processors) [BuLu 94]. Both libraries offer function calls for programming parallel machines, with a focus on portability between different hardware architectures. The difference between PICL and p4 is, that the former defines a generic message-passing interface, while the latter implements various features for shared-memory and message-passing programming.

While the message-passing systems described above where intended mainly for real parallel architectures, the Parallel Virtual Machine PVM has been started in 1989 as a research project on heterogeneous distributed computing at the Oak Ridge National Laboratory [Geis 96]. Its initiators Geist and Sunderam [GeSu 92] have designed the notion of a "virtual machine", which is a set of heterogeneous hosts connected by a network that appears to the user logically as a single large parallel computer. The exchange of data between the parallel tasks was

accomplished using simple message-passing constructs. Two goals defined the original path of PVM [Sund 90]: Firstly, it should provide a simple interface that is easy to use and understand. Secondly, performance was not considered as important as portability, because communication across the Internet was slow anyway and some of the research goals were scalability, fault tolerance, and heterogeneity. The use of PVM grew rapidly worldwide up to its current status as a pseudo-standard.

In contrast to PVM, which has started and continues to evolve as a research project, the Message Passing Interface library MPI has been specified by a committee - the MPI Forum - of about 40 high performance computing experts from research, academics, and industry around 1993 [MPI 95]. Its intention was to define a standard message-passing API, that allows to write portable parallel applications [Snir 98]. The vendor participants of the MPI Forum intended to implement it for their particular parallel computer architecture. Since hardware vendors always try to provide the highest possible performance, this was also one of the main goals of the MPI design, and it is expected that MPI is faster than PVM on dedicated MPP architectures [Geis 96]. Furthermore, MPI provides the ability to specify a logical communication topology and a richer set of point-to-point and collective communication operations, which may be important, if an algorithm is executed on a machine with a special communication feature. Yet, even though MPI's functionality is very rich (Version 1.1 contains around 120 routines), it has been noted before, that successful programs could be written with only six functions [Fost 94][Grop 94].

There exist several free implementations of the MPI standard, including MPICH from the Argonne National Laboratories and the Mississippi State University, LAM from the Ohio Supercomputing Center, CHIMP from the Edinburgh Parallel Computing Center, and UNIFY, a subset of MPI within a PVM environment, from Mississippi State University. In addition, there are specialized versions for high-performance networks like Myrinet [Pryl 99] and vendor implementations for almost every multiprocessor architecture currently available. Similar to PVM all the function calls are available for both C and Fortran [WiAl 99].

Each of these message-passing libraries has its unique strength and programmers often wonder, if they should stick to the existing de facto standard, PVM, or whether they should shift their codes to the MPI standard. Yet, this question cannot be answered without further examination of the application, its functional requirements, and the running environment.

2.4 Summary: Programming the World's Most Powerful Machines

After the brief discussion of existing high-performance machines and some explanations about how to program the wide variety of parallel architectures, we will now summarize the current situation. In Section 2.2.6 we have shown, that the most powerful computers today are Massively Parallel Processors with up to several thousands of processors. Their share in the TOP 500 list is currently more than twice that of the second contenders, the Symmetric Multiprocessors. Yet, outlooks about future supercomputer architectures predict an increased amount in clustered SMPs ([Simo 99] and [Geof 99]), which means that they must be considered as potential targets.

Considering the architecture, MPPs are almost always distribute memory machines, which suggest that their main model of programming is the message-passing paradigm. On contrary, SMPs support both, the message-passing and the shared-variable approach, and any message-passing algorithm can be executed on a shared-memory architecture with no additional run-

ning time [AlGo 94]. Yet, if we take the number of applications into account, message-passing will still cover the majority of parallel machines today.

The remaining question is, if MPI or PVM is more suitable for the most powerful machines today. Clearly, both message-passing libraries may be suitable for users in this area. However, in most cases MPI will be given the preference, simply because of its efficient implementation and its inherent goal for high performance. We should not forget, that the target problems discussed in Section 2.1.2 are Grand Challenges, which require the highest possible performance from a given computing system.

As a summary, we can identify the key attributes of the most-powerful systems suggested by the survey of the current TOP 500 list as follows:

- Architecture: Massively Parallel Processor (MPP)
- Programming Paradigm: Message Passing
- Message-Passing Library: MPI standard (Message-Passing Interface)

Such machines are the basic background of the research described in the following chapters. However, the actual situation also points to a change for either SMP or COW systems. Therefore, we tried to operate with an open mind onto these other representative supercomputer architectures and all our suggestions may also be used for other machines with minor modifications, e.g. for SMPs with shared-variable programming under OpenMP.

3. Problem Domain: Debugging Program Errors

In this chapter, we will review the software engineering process with a special focus on the areas error detection and performance analysis. This will lead to a discussion about the debugging task itself, with its goals and main problems. Furthermore, we will study this history of bugs and the origin of bug-hunting, and describe some of the most notorious errors in computing history. The results of this digression will allow us to summarize the importance of the debugging process for the quality of the resulting program and its problems for parallel computing.

- Software Engineering/Debugging/Performance Analysis
- Debugging: Goals/Problems
- Bug Hunting History
- Overview of most notorious Bugs
- Importance of Debugging

3. Problem Domain: Debugging Program Errors

3.1 Software Life-Cycle Models

The professional development process of software systems is called software engineering and covers the application and usage of methods and tools by teams of software developers in order to create software systems of different size [FlZü 97]. Main aspects are the management and coordination of such projects, its components and their relations, as well as sufficient understanding of the software development process. This knowledge must consist of life-cycle models, working conventions, and quality assurance capabilities. As a consequence, software is often considered to include not only computer programs, but also the associated documentation required to develop, operate, and maintain the programs [Boeh 76].

Software life-cycle models define the order between different activities and their relations in the software development process. Three life-cycle models can be distinguished [FlZü 97]: the linear phases model or waterfall model, the spiral model, and the cyclic models. The most basic approach is represented by the phases model [Boeh 76][Enge 88], which often contains the following activities:

- Problem analysis and definition
- Design and specification
- Implementation and documentation
- Testing and tuning
- Installation and maintenance

Although the phases model exists in different variations, it always consists of similar tasks that are carried out in chronological sequence. Intermediate result (milestones) between the tasks determine the status of the project and may require and thus initiate reiteration of earlier phases. Therefore, the intermediate results at the end of each phase must be thoroughly understood, and preferably documented, before continuing with subsequent phases. If this isn't done, critical and costly errors may be detected too late, usually during testing, but frequently even not before operation. Correcting these errors later involves much greater expenses and may delay project schedules due to the required reworking (and retesting) of design, code, documentation, training material, etc. [Boeh 83].

While the phases model is rather simple, its main problem is flexibility which is important since initial specifications and goals may be altered. Therefore, its importance is often questioned and other alternatives are proposed for the handling of projects [FlZü 97]. However, it is sufficient for our purposes.

3.2 Testing and Tuning

3.2.1 Characteristics

In the scope of this work, we are interested mainly in the "testing and tuning" phase, which consists of two main subtasks, the testing of functionality and the evaluation of achieved per-

formance [Enge 88]. The former is used to examine the input/output behavior of a program corresponding to its specification. The latter is applied, whenever a functionally correct program has been achieved, and its runtime performance and efficiency are investigated. In the following, we will focus mainly on the testing phase. We will also show, that testing and tuning are closely related and that their activities are often connected.

The testing phase is usually initiated, whenever a module or a complete program has been implemented and its correct behavior must be reviewed. For that reason, testing is closely related to the two activities verification and validation. While the former is used to proof the target's correctness according to its behavioral specifications, the latter is applied to check to target's correct execution for a limited sets of inputs.

Verification means that the characteristics of modules or programs are thoroughly checked according to their behavioral specifications. A program is defined to be correct, whenever it complies fully to the specifications, which means that for all given inputs it reacts with correct outputs. We distinguish partial and total correctness of a program, where the former means that it terminates for a given input and delivers the correct results. A program is totally correct, if it terminates and it is partially correct for all inputs [Enge 88].

In contrast to verification, the process of validation is applied to check, if the system or one particular module corresponds to the given requirements with correct results according. To achieve this, the program is either executed or simulated. Two basic approaches can be distinguished: the validation of the complete system (according to the user's requirements) and the validation of the functional requirements. Testing may also be described as the iterative application of validation.

The testing phase can be divided into the module test, the integration test, the installation test, and the acceptance test. For each of them, we distinguish two different approaches, black-box testing and white-box testing [Paul 99]. During black-box testing, the program is only verified according to its specification. Often, such tests are applied, when the input space and output space of a program are finite. In that case, a set of black-box tests can be constructed, that can show conclusively that the program is correct and its behavior matches its specification [Boeh 76]. White-box testing also considers the internal structure of a program, in order to cover certain instructions or execution paths. White-box tests are especially important for the respective developer during component testing [FlZü 97].

3.2.2 Problems of Testing

A principle problem of testing is that in the general case and for a program of arbitrary size and complexity, it can always only demonstrate the existence of errors, but it can never prove the absence of errors [Paul 99]. Furthermore, exhaustive testing is usually impossible due to the amount of possible inputs a program states in most software systems [FlZü 97]. Yet, since every test suite feeds a program under consideration only with selected inputs out of the often enormous space of all possibilities, certain bugs in the code may never be evidenced [WaBl 97]. This means, that the total elimination of bugs is an unrealistic goal [Lieb 97].

Usually, programs are not correct at the first place. In contrast, it is even highly probable, that any developed program contains some errors [Enge 88]. The reason for this is partially the complexity of the program development process itself, partially the difficulty to define solutions and realization as software, but also the high precision that is required from software engineering in general.

Another main problem of testing is, that it is surprisingly often not considered until the code has been run the first time and is found not to work [Boeh 76]. Statistics show that many errors

have already been made before coding begins, and the later the errors are detected, the higher are the associated costs to rework the system in order to function correctly. Boehm states that the most important message about developing software is to get the errors out early [Boeh 83]. In addition, the testing phase is still a highly tedious task, that is mostly carried out manually, which is therefore error-prone in itself. A related question is the amount of required testing, which is more often defined by the amount of available budget or the given schedule.

And finally, it is very difficult to determine the sufficient amount of testing. It is clear, that exhaustive testing is impossible in both principle and testing. In principle, there are two possibilities to decide about test completion: statistical, when error statistics attest a sufficient degree of reliability, and systematic, which tries to determine methods for identifying the functional properties and classes of possible errors in programs [KrWi 98]. A very common approach is to fix every detected bug and perform testing until the project management orders a product release. Bach tries to articulate and perhaps quantify the amount of testing [Bach 98], that is sufficient for different projects depending on the circumstances and may be useful for everyday projects as well as mission- or life-critical projects.

Definition 3-1: Good Enough testing [Bach 98]

> Good Enough testing is the process of developing a sufficient assessment of quality, at a reasonable cost, to enable wise and timely decisions to be made concerning the product.

This definition identifies several characteristics of the professional software engineering task related to the quality of a product and its evaluation. The following questions are of major importance:

- How accurate and complete is the product's quality assessment?
- What level of testing is possible within the project costs constraints?
- How well does the assessment serve the project and the business in terms of potential product risks?
- How to time all these activities?
- When has enough testing been performed?

Similarly, Dubois argues that the verification of a program is often based on a series of problems with known results [Dubo 99]. Since it must be prepared to solve a general problem, its results can only be taken as an approximate answer and it requires good judgement to decide about the program's correctness and reliability. These differences between the actual and the desired answers arise from numerical noise, chosen approximations, inaccurate models of physical properties, or actual coding errors [Dubo 99]. Therefore, it is often difficult to judge whether an answer is a bug or a modeling problem. Many bugs are indistinguishable from errors in modeling or deficiencies in the applied numerical techniques. At worst, a bug may cause a model's revision or a complete rejection of an approach. At best, a bug can only be detected through considerably high efforts of testing[1].

1. Please note, that the term bug is used here to express some kind of erroneous behavior. A more detailed definition of this term will be given in Section 3.2.6.

3.2.3 Software Quality

As defined above, the goal of testing is correctness or reliability, respectively. Software reliability is defined as the probability of future satisfactory operation of software [Boeh 83]. Software development using formal methods faces two problems in achieving reliability [Boyl 99]:

- Obtaining a formal specification of an informally described problem.
- Based on this formal specification, selection of a mathematical verification method to prove the software's correctness with effective, affordable methods

The quality of software is defined by its characteristics and features offered during application and development to achieve the demanded requirements [FlZü 97]. Most frequently, these quality criteria are correctness, reliability and portability [Enge 88]. The correctness of a program can be defined as follows:

Definition 3-2: Program correctness [Schn 91]

A program is correct, if it exactly meets its given specification.

Consequently, the correctness of a program is always expressed relative to its specification. A major problem in determining the correctness of a program is the difficulty of finding a complete and adequate specification, which is often described in abstract mathematical or physical statements [Dubo 99].

Any method applied to achieve a high level of quality is summarized under quality assurance techniques, which can be divided into constructive quality assurance and analytical quality assurance. Constructive quality assurance is based on the application of methods, tools, construction principles, formal procedures, and life-cycle models, and covers the area of application quality, product quality and process quality [FlZü 97].

Analytical quality assurance techniques verify, if predefined quality characteristics are fulfilled by an existing software component. This contains static and dynamic analysis. During static analysis, the object under consideration is not executed. Examples are the application of metrics, various informal methods like review and audit, but also the usage of analysis tools and formal verification methods. Dynamic analysis is performed by testing. In that case, the program is executed in order to detect errors [FlZü 97].

The best software is that which has been tested by thousands of users under thousands of different conditions over years. Then it is known as "stable". Yet, this does not mean, that the software is now flawless, free of bugs. It generally means that there are plenty of bugs in it, but the bugs are well identified and fairly well understood [Ster 92]. There is simply no way to assure that software is free of flaws. Though software is mathematical in nature, it cannot be "proven" like a mathematical theorem; software is more like language, with inherent ambiguities, with different definitions, different assumptions, different levels of meaning that can conflict.

Bach defines requirements as the set of ideas that collectively characterize the quality for a particular product, and testing as a process of developing an assessment of product quality [Bach 99]. There are at least four alleged truisms about testing a product against requirements, and each of these principles reflects some truth about the dynamics of testing:

- Without stated requirements, no testing is possible.
- A software product must satisfy its stated requirements.
- All test cases should be traceable to one or more stated requirements and vice versa.
- Requirements must be stated in testable terms.

3.2.4 The Testing Process

Boehm defines the main types of automated aids for the steps of testing between the point the code has been written and the point it is pronounced acceptable for being used, as follows [Boeh 76]:

- Static code analysis
- Test case preparation
- Test monitoring and output checking
- Fault isolation, debugging
- Retesting (once a presumed fix has been made)
- Integration of routines into (larger) systems
- Stopping

Static code analysis starts with the usual compiler diagnostics and detailed data-type checking. In addition control flow and reachability analysis are applied with structural analysis programs and others. As an extension to structural analysis tools, test cases preparation aids assist the engineers in choosing (input) data values to make the program execute along a desired path. Yet, these tools are only sufficient to generate a limited amount of inputs, for which the outputs must still be calculated manually or with other means of computer-support.

For this work, the more important parts are test monitoring and output checking, as well as fault isolation and debugging. In order to perform a maximum of testing automatically, techniques have been developed for various kinds of dynamic data-type checking and assertion checking, and for timing and performance analysis. In addition, test post-processing tools aid the users with output comparators and exception report capabilities, as well as test-oriented data reduction and report generation packages. For the area of fault isolation and debugging, we may distinguish well-established methods like core dumps, traces, snapshots, and breakpointing, from highly interactive capabilities like replay or backtracking.

After fixing the code, retesting is used to compare previous test cycles with the results of the corrected code. Again comparators to check the differences in the code, inputs, and outputs between the original and the modified program are applied. These test management tools are further extended for integrating the routines into bigger systems, where additional activities like interface consistency checking have to be performed. Finally, the halting problem of testing as defined above with Good Enough testing may also be supported by tools, which means that tools offer a means of quantifying the amount of testing that has been performed.

3.2.5 Debugging Basics

Probably the most important part of testing is debugging or error location and correction. It is an essential part of the software life-cycle, because a program is obviously only useful if it executes correctly and is sufficiently reliable [WiAl 99]. In standard software engineering terms, this activity can be defined as follows:

Definition 3-3: Debugging [McHe 89]

> Debugging is the process of locating, analyzing, and correcting suspected errors.

Today, a lot of a programmer's development time is spent on debugging [Agra 91]. Therefore, it is often ranked as one of the most significant bottlenecks while at the same time the weakest link in the software life-cycle [Stit 92]. The debugging phase is applied after runtime failures or incorrect results have been observed during testing. In that case, the errors and their origin must be detected and corrected [Enge 88]. Whenever bugs arise, programmers should be able to locate and correct them with a minimum amount of effort, leaving more time and energy available for progressive work [Stit 92].

While the importance of testing and debugging is highly accepted in the software engineering domain, there is still a surprising lack of books that thoroughly discuss software debugging at a practical level [Stit 92]. Of course, there are books describing the features of one or more debugging tools from a user's view, but these do not provide a general look onto the problem of solving program bugs. From the scientific point of view, debugging is still studied very little, compared, for example, to compilers [Rose 96]. In addition, debugging is still absent from current-day computer science curriculums, which leaves students and programmers without much guidance.

In [Lieb 97], Lieberman defines debugging as "... the dirty little secret of computer science". He argues that the computer science community as a whole has largely ignored the debugging problem. Today's commercial programming environments provide debugging tools that are little better than the tools that came with programming environments 30 years ago. The most difficult problem is to figure out exactly what went wrong. This results from the fact, that programs are complex artifacts, and consequently, debugging often struggles against this complexity [Lieb 97]. As a consequence, debugging is still a matter of trial and error.

Bug-hunting is often described as detective work. Stitt defines the phases of the debugging process to be arranged around the "research, gain knowledge and deduce phase". Furthermore, he identifies five major tactics of debugging, which may be used individually or in combination with others[Stit 92]:

- Preliminary investigations: is it really a bug?
- Static debugging: verification of requirements and design as well as source code analysis
- Run-time observation: testing, experimenting and analysis of program results
- Source code manipulation: instrumentation and partial execution
- Dynamic debugging: breakpointing, single-stepping, monitoring and testing

A tool that supports the user during this process is called a debugger. Two possible definitions are given below:

Definition 3-4: Debugging tool [Stit 92]

> A debugging tool is anything that provides useful knowledge about a program's execution and its occurring program state changes. Different forms of evidence are given by software debuggers, source language instrumentation, instrumentation drivers, interrupt interception drivers, and hardware monitoring aids.

Definition 3-5: Debugger [Rose 96]

> A debugger is a tool to help track down, isolate, and remove bugs from software programs. In truth, debuggers are tools to illuminate the dynamic nature of a program in order to understand it as well as find and fix defects.

Rosenberg also compares debuggers to other tools like the magnifying glass, the microscope, the logic analyzer, the profiler, and the browser with which a program can be examined. In principle, a debugger is something like a sophisticated software analyzer.

Such debugging tools are as necessary for the incremental evolution of software as they are for finding errors [Lieb 97]. The basic capabilities of a debugger are:

- to follow the executable program step by step (tracing)
- to stop and continue the programs execution at breakpoints (breakpointing)
- to display and manipulate the contents of variables, registers, and memory locations (inspection)

In addition to these basic features, debuggers may also include support for more advanced or specialized activities. An example advanced technique often anticipated by users is program slicing, a technique which extracts only those lines of code that are related to the current track of investigations (see [Weis 82] and [Weis 84]). Other examples are debugging capabilities tailored to one particular set of errors, like memory leaks or incorrect pointer references.

Probably the most important activity of debugging is breakpointing. It is based on breakpoints, which can be defined as follows:

Definition 3-6: Breakpoints [Stit 92]

> A breakpoint is a controlled way to force a program to stop its execution. Breakpointing may occur on software interrupt calls, on calls to a program subfunction or on selected points within the program.

Single-stepping is a special kind of breakpointing, which may occur on either machine instruction and assembler level or on source lines in a higher-level language. Other convenient functions may be based on breakpoints, like watchpoints, “run-to-here”-features, and conditional program control [Rose 96].

Please note, that breakpointing is a difficult technique and implementing an accurate breakpointing mechanism is a delicate challenge. In principle, a breakpoint is a special instruction inserted into the executable by the debugger, which causes the target to halt its execution immediately and to call a matching service routine (the breakpoint exception handler) of the operating system [Rose 96]. These special “trap”-instructions may be either general interrupt instructions with corresponding return values (e.g. on Intel’s x86 architecture) or dedicated breakpoint instruction (e.g. on the MIPS RISC architectures or the Digital Alpha processors). Dealing with breakpoints always means handling of exceptions, which is a big obstacle during processor design. Some delicate examples are overlapping of instructions and branch prediction on pipelined-architectures. In these cases, an instruction is executed piece by piece and is not completed for several clock cycles. Thus, exceptions may require to abort instructions in the pipeline before they complete, or predicted branches may have to be discarded altogether. For a more detailed discussion of these critical issues, please refer to the corresponding litera-

ture, e.g. [HePa 96]. Breakpoints in parallel and distributed systems will be addressed in Section 7.5.

3.2.6 Definition and Classification of Errors

Diagnosing and troubleshooting a software failure is a difficult activity. It starts with the problem, that most of the terms and notions in this area have a common sense appeal and lack precision [Brit 98]. Thus, it is important to try to exactly distinguish the following words:

- Error
- Failure
- Fault
- Bug

We will provide corresponding definitions in the following paragraphs, which have been derived from existing literature, in this case especially the opinions given in [Brit 98], [Choi 91], [Enge 88], [KrWi 98], [McHe 89]. Probably the most precisely and formally defined approach is offered by Grabner's view of errors and associated program states [Grab 99], which is also included in this section.

Remark

Although the definitions of Grabner are preferred in most cases, we do not need this high precision throughout this work. Therefore, we will stick to the common usage of the term "error", which covers all different kinds of errors, bugs, failures, or faults. This view is sufficiently accurate and precise for our needs.

A very informal definition of the most basic term "error" corresponding to computer science is as follows:

Definition 3-7: Error (informal definition) [Enge 88]

An error is defined as the computation (calculation and outputting) of one or more incorrect results by a computer.

This definition of an error is the most commonly used, and the term "error" has also been used in Definition 3-3 for debugging. The noted lack of precision is established by two issues: On the one hand, there needs to be a distinction between the observed erroneous behavior and the original error itself. On the other hand, the term "results" identifies not only the output of a program, but also incorrect states during execution.

Grabner discusses these problems in more detail by focusing on the states of application programs, which consist of simple and compounded instructions [Grab 99]. For each of these instructions there is a well-defined initial state - before the instruction is executed - and a terminal state - after the instruction has been executed. A compounded instruction may consist of several intermediate states. During debugging, the user can investigate these states and apply corresponding activities. Grabner defines the state of a program during execution as follows:

Definition 3-8: State of program during execution [Grab 99]

> For a given view onto a program, let t represent the instant of time at the terminal state of instruction a_t, and let $s_t(v)$ be the value of variable $v \in V$ at time t.
>
> Then, the state s_t of the program at time t is the set of the values of its variables $s_t(v)$, so that: $s_t = \bigcup_{v \in V} s_t(v)$.

In this definition, a_t is an instruction, whose operation assigns a new value to variable $v \in V$. Although the set V contains all the variables of a program, the user's main interests are those variables, that are modified by instruction a_t [Grab 99]. Figure 3-1 describes the relation between the instruction and its variables in a small diagram. In [Pine 99], the author introduces another distinction by classifying those variables as "noncurrent", whose actual values as computed by instruction a_t differ from the values expected by the user.

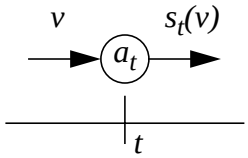


Figure 3-1. State of a program at a given instruction

An important issue of Definition 3-8 is the view onto the program. This view is usually defined by the user by selecting a corresponding granularity for the analysis activities. The most coarse granularity is certainly the program itself. In this case, the initial state is defined by the initial state of the variables, while the final state represents the results of the program. In most cases, the user selects a finer level of granularity. Typical examples for debugging are the instructions of the program in its high-level programming language or in assembler language [Grab 99]. In Section 5.1.1, we will review and adapt this definition of a program's state for our event graph.

Based on the definition of a program state, an error can formally be defined as follows:

Definition 3-9: Error [Grab 99]

> An error in a program is observed through an incorrect program state s_t, which means that one or more variables $v \in V$ contain a wrong value at time t.

In terms of errors, two kinds can generally be distinguished: hardware errors and software errors. The former result from erroneous functions of electronic elements, which have been initiated e.g. by dust, heat, variations in electrical currents, and others, or have been introduced through corrupted connections and cables or by other faulty components. The appearance of hardware errors leads to interruptions of service, which often result in complete breakdowns or system shutdowns. An improvement to this situation is offered by fault-tolerant systems, which increase the systems hardware reliability by incorporating replicated and redundant system components. In the scope of this work, we will not consider hardware errors any further, but instead concentrate on the second kind of errors, that manifest themselves in software.

Software errors (often called logical errors) are based on an incorrect sequence of statements, which can further be attributed to errors in the implementation or in the basic algorithm. These kinds of errors are related to program behavior, that is unexpected, unintended, or otherwise wrong due to some incorrect activity of a human or the environment [KrWi 98].

Additionally, Britcher believes, that a fault is rarely the work of a single operation. More often than not is it a logical sequence that runs to dozens of lines of code and crosses modules [Brit 98]. In this context, Grabner distinguishes between the following kinds of errors [Grab 99]:

- Errors of origin
- Subsequent errors

An error of origin is one, that is observed at the program state where it occurs initially. Therefore, such an error manifests the first occurrence of an incorrect operation. This is also the place, where the correction should be applied. However, since errors are seldom detected immediately, users are usually confronted with subsequent errors first. Such subsequent errors exist always only as a consequence of an error of origin. They reveal an incorrect program state, that results from both, their operation and incorrect input variables. This relation between several states of a program is also expressed in the term fault, which can be defined as follows:

Definition 3-10: Fault [Brit 98], [McHe 89]

> A fault is a logical sequence of statements (or rather state changes) or an accidental condition that produces a result other than that which the programmer intended or causes a program to fail to perform its required function. Therefore, a fault gives rise to errors and inversely, an error is the result of a fault during the creation of the program.

Probably the most common faults are typos in the source code text, that have not been detected by the compiler, or some other incorrect statements which do not comply with the intended semantics of a program.

The worst kind of fault is a failure, which can be defined as follows:

Definition 3-11: Failure [KrWi 98]

> A failure is the inability of a system (or one of its components) to perform a given function within given limits.

Additionally, a failure is often defined as a loss of some service to the user [Brit 98]. Therefore, it represents one special kind of manifestation of erroneous program behavior [Choi 91], because a program fails to produce expected results and may even terminate abnormally.

The missing term in this context is “bug”, which is often used as an equivalent replacement for error. However, in practice most people identify some additional characteristics for a bug as specified in the following definition:

Definition 3-12: Bug [KrWi 98]

> A bug is the commonly used term for an error with originally unknown location and reason. A bug can also be attributed to a poorly prepared and executed test, as well as a bad program description that may lead users to mistake correct behavior for bugs.

The notions and terms described above can also be described with a conceptual overview as presented in Figure 3-2. A special group are the false errors, which occur at state changes, that are incorrectly assumed to be wrong.

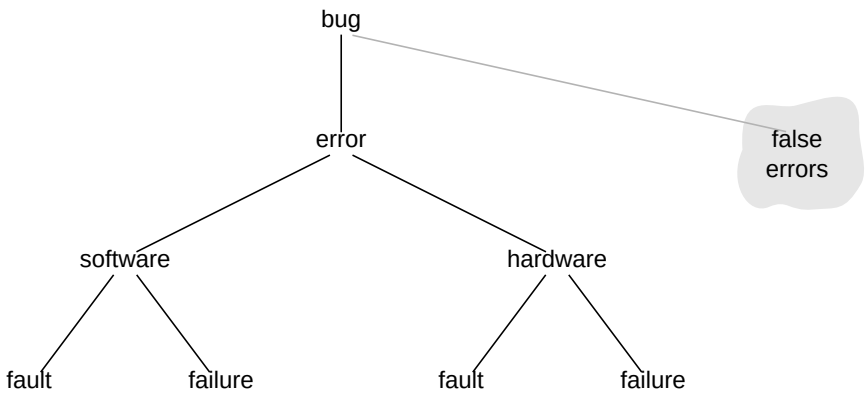


Figure 3-2. Conceptual overview of notions and terms

During debugging, we want to pinpoint the very origin of an error, i.e. the first incorrect operation, which is often a difficult task. For that reason, we may identify several more characteristics of bugs.

In principal, if the software's design is logically and mathematically correct, it should be so into infinity (with exception of certain types of calendar calculations - see Section 3.3.6, "The Millennium Bug"). Consequently, if a software error arises it may only originate from the following set of possibilities [Stit 92]:

- Included in the original design or coding.
- Introduced through program modification.
- Introduced as a side effect of another bug fix.

Additionally, there are those errors related to failures of the hardware on which the program is executed, and to the corruption of the media on which the program is stored. Software components differ fundamentally from hardware components, because they don't suffer from aging or transient failures. Instead, when software components fail, the reasons are [Boyl 99]:

- Inaccuracies of the problem specification, which results in programs that behave differently than intended by the specifier.
- Errors introduced during software development, which results in programs that behave differently than intended by the programmer.
- Errors introduced during compilation, which results in programs that behave differently than intended by the compiler.

The so-called group of “occasional errors” may reside undetected in a program for very long time, even several years. They manifest themselves sporadically only on certain rare combinations of input and system state [BlWa 96]. These malicious errors may only surface, when other factors of the system environment change (e.g. marginal stack allocation, conflicting use of resources between system components, timing failures, and other results from heavy system load) [Stit 92].

Stitt offers some more characteristics to establish a classification of errors with the following three classes [Stit 92]:

- Consistency: errors may occur either with consistency or with a frustrating lack of consistency
- Intermittence: both, consistent and variable type of failures may be intermittent
- Fuzzy cases: is the problem really a result of a errors?

These different characteristic are very useful during debugging. Only if users are aware about all the different possibilities, they are able to react to incorrect program behavior and perform corresponding error detection activities. To underline the importance of these tasks, the following section describes actual cases of erroneous behavior observed in computer systems.

3.3 Notorious Errors

The reasons for testing have been defined above. Obviously, errors in software are unavoidable and very likely to take place in general. The importance and implications of errors may still be underestimated, and often the financial consequences are overlooked. Porter reports about a survey by the Standish Group, which examined 175.000 software development projects in 1995 [Port 98]. The results of this research revealed, that 31% of these projects were cancelled before completion with a financial loss of US$81 billion. Of the remaining projects, more than 52% suffer significant cost overruns, with an average of 189% more than estimated, resulting in a US$59 billion loss. Additionally, there is the possibility of other consequences for humanity, sometimes even leading to personal harm. The following section contains an overview of some of the most critical errors. The first example of a bug represents the very origin of the term “debugging”.

3.3.1 History of Debugging

The story is reported to have taken place on September 9, 1945 at the Computation Laboratory of Harvard University [Stit 92]. Grace M. Hopper, assigned to the Bureau of Ordinance Computation, was working on Mark II, the successor of the Mark I computer engine, when suddenly the machine stopped for no apparent reason. Upon inspection, Hopper and her team detected a moth, which had flown into a relay from an open window; the moth had been pulverized by the relay and consequently had caused the device to fail. Her entry in the log book read: “First actual case of bug being found”. Afterwards, the term “bug” was popularized to signify any system malfunction. Yet, it has been noted that the term itself had been used as far back as the end of the last century, when it was applied to electrical equipment.

More recently in 1995, the twentieth anniversary special issue of the Byte computer magazine devoted one chapter to a list of the 20 most notorious bugs [Need 95]. Some of these errors as well as a few other important examples are described in the following.

- Lethal errors
- Airplane crashes
- Governmental and commercial mishaps
- Internet worm/hackers
- Millennium bug

3.3.2 Lethal Software Errors

Probably the most critical errors occurred in situations, where human beings have been harmed or eventually killed. The best-known example in that domain is the story about the Therac-25 linear accelerator machines between 1985 and 1987. The tragedy happened at the East Texas Cancer Center in Tyler, where at least four people received lethal doses of radiation from Therac-25 machines during their medical treatment for cancer [Need 95]. There were several errors, among them the failure of the programmer to detect a race condition[2] (i.e., a miscoordination between concurrent tasks) [Joch 95].

Another group of errors, that is seldom published for the public's knowledge but rather more often hidden by governmental agencies, is related to software systems in warfare equipment. Since errors are unavoidable, even these system must be affected from time to time. One example is a critical bug observed on American Patriot missiles during the Persian Gulf War. It is reported that one of these missiles erroneously killed 28 American soldiers in their barracks in Dahran, Saudia Arabia, due to a software glitch in the system [Need 95].

3.3.3 Airplane Crashes

A topic related to software errors that always gets the media's attention is airplane crashes. Although travelling with commercial aircrafts is often declared to be the safest possibility for any kind of journey (accident rates are given with one accident per 1 million flights), an accident with an aircraft is much more media-important than most other means of public and private transportation. A lot of investigations in this area has been performed by Ladkin [Ladk 99], who studies the reasons for computer-related incidents with commercial aircrafts. He states, that until now programming errors have never been mentioned as the main reason for any critical accident. In fact, most of the breakdowns, incidents, and accidents do not happen due to one single source in one independent component, but always due to the cooperation of several factors.

An example is the crash-landing of one of Lufthansa's Airbus A320 in September 1993 in Warsaw, which killed two persons. The A320 is the first commercial aircraft, which connects pilot's joystick (in contrast to other planes that have a control lever) and rudder only via computers, in order to save weight, increase the reliability of the system, and to block human mistakes. The problem in Warsaw appeared, when the plane was preparing to land and the pilots were waiting for a direction change of the wind that never happened. As a consequence, the landing took place with the wind instead of against it. In addition, the pilots had increased the

2. Race conditions are some of the most critical errors in parallel programs. They are extensively discussed in Section 4.2.4.

planes speed in order to fight the high winds. This resulted in a far to high landing speed, and due to additional up-wind, the sensors at the planes wheels didn't signal that the machine has landed, which in turn didn't activate the brakes to stop the plane before leaving the runway and hitting a hill.

Other problems with planes and computers are known, e.g. the landing of one of Dragonair's A320 in June 1994 in Hong-Kong. In that case, a blast of wind prevented a change of the flaps, while the computer believed that the flaps were in the correct position, which induced heavy problems during landing. Other computer problems lead to a big catastrophe with 160 victims, when a Boeing B757 of American Airlines hit a hill in Colombia. In that case, the flight management computer (FMC) was operating with an erroneous navigational data base and two corrupt navigational transmitters, which resulted in wrong positional informations for the flight crew.

Besides the planes, an airport itself may be hidden by malfunctioning system. For example, in 1994 bugs in the computerized baggage-handling system delayed the opening of the new Denver airport. Automated baggage carts drove into walls or deposited bags at the wrong airport destination. The financial consequences were some $80 million to fix the system, before finally the airport opened in February 1995 with a manual baggage-handling system [Need 95].

Another well-known example besides commercial airplane travels is the termination of the Ariane-5 rocket on June 4th, 1996 [Ladk 98]. Due to a conversion problem from 64-bit floating point numbers to 16-bit integers, an exception was called that was never taken. As a result, both computers on board the rocket shut the system down, leading to navigational errors of the rocket only 40 seconds after its start. In order to avoid further damage, the self-construction mechanism was initiated and destroyed the rocket. Similarly, in 1993 a software error caused the thruster rockets of satellite Clementine to fire continually, which consumed all its fuel before completing its asteroid-rendezvous mission [Need 95].

3.3.4 Governmental and commercial mishaps

As we have seen above, computer programs often don't work as they should, and too much buggy software reaches the end user [Lieb 97]. An important area of computer application is the governmental and commercial domain of everyday computing. Errors in these system often affect many people, but fortunately, their consequences are not lethal. An example happened in 1985, when a tax computer delivered warning notices about withheld taxes to 27.000 companies, which had already paid their fees [Need 95]. Another story is reported for 1989 in Paris, where 41.000 surprised traffic offenders received letters charging crimes like murder, drug trafficking, extortion, and prostitution instead of their traffic violations [Need 95].

One example for a big software error in a commercial company happened in 1988, when an automated Black & Decker distribution center in Northampton, England, restored corrupted backup data over all the main systems data [Need 95]. The destroyed data could only be corrected by inputting the complete inventory of the depot manually. While this was already an expensive consequence of a software error, failures in computer systems of banks may even be more costly. In 1989 a bug in the system of a British bank transferred an extra 2 million pound to customers within one hour, because payment orders were permitted to be issued twice [Need 95]. The problem in this case was, that the bank had to trust their customers in returning the money, because of missing transaction protocols.

Yet, it has to be mentioned, that these official cases of software bugs are probably only a minority of the estimated number of unknown cases. The reason is that companies and espe-

cially banks rely on establishing a basis of trust with their customers, which may certainly be harmed if such errors would be published. Thus, it must be accepted, that many more bugs happen than are reported in the media.

3.3.5 Internet Worm/Hackers

In contrast to the governmental and commercial mishaps reported above, increased media attention is focused on reports about the scene of hackers and phreaks. The high potential of devastating damages is a consequence from the massive expansion of the networking infrastructure and at the same time from the vulnerability of networked computers. The connection between software errors and hackers is the fact, that the former is usually misused as a starting point for illegal intrusions into computer systems.

The Internet Worm of November 2-3, 1988, created by Cornell grad student Robert T. Morris, was to be the largest and best-publicized computer-intrusion scandal to date [Ster 92]. Morris said, that his ingenious “worm” program was meant to explore the Internet harmlessly, but due to bad programming and a math error in the code, the worm replicated itself 14 times faster than intended and slipped out of control [Need 95]. Within a few hours, some six thousand Internet computers crashed, affecting systems for weeks before fully recovering from the damages wrought [Spaf 89].

A similar example is the system crash of AT&T’s long distance telephone network happened on January 15th, 1990 [Ster 92]. Its cause was an improvement or rather an attempted improvement in the System 7 software for AT&T’s 4ESS switching station, the “Generic 44E14 Central Office Switch Software”. This software has been extensively tested, and was considered very stable. Originally, the station with System 7 were programmed to switch over to a backup net in case of any problems. However, in mid-December 1989, a new high-velocity, high-security software patch was distributed to each of the 4ESS switches that would enable them to switch over even more quickly, making the total system much more secure. Yet, at 2:25 P.M. EST on Monday, January 15th, 1990, one of AT&T’s 3ESS toll switching systems in New York City had an actual, legitimate, minor problem (a missing break in a C switch statement) and went into fault recovery routines. This was the kickoff for a series of fallouts, and like in a chain reaction one machine after the other went down leading to an immense collapse of the US telephone system. The shut down lasted for 9 hours, leading to some 74 million uncompleted calls, which resulted in the most severe breakdown in the US telephone network ever [Need 95].

The story of this error was rather dubious, because initially hackers were made responsible for this failure and were blamed by the media and US law enforcement agencies. This is a usual way to cover tracks. Quite often, big companies do not blame themselves for software errors, or blame other people to be responsible for their failures instead. At the same time, some companies cannot effort to make system failures publicly known, simply because their relation with customers relies on trust, as in case of banks, which have surely been hit by hackers following software bugs in past, but would never confirm these incidents ever happened.

Similar events are documented from time to time, like one on July 1st, 1991, which disrupted the telephone service in Washington, D.C., Pittsburgh, Los Angeles, and San Francisco. Once again, a seemingly minor maintenance problem had crippled the digital System 7 and affected about 12 million people. This time, it was a single mistyped character, one tine typographical flaw in one single line of the 10 million lines of code.

3.3.6 The Millennium Bug

The most publicly interesting and therefore media-present bug is undoubtedly the millennium bug or Y2k (Year 2000) error. This error even lead to the creation of the International Y2K Cooperation Center under the auspices of the United Nations in order to minimize Y2K impact throughout the world (see http://www.iy2kcc.org). Besides that it has been in the media for quite a long time, and many self-proclaimed prophets predicted the end of the world due to computer failures. The whole problem is based on the fact, that some legacy mainframe computer programs were hard-coded to treat the years by their last two digits. Consequently they cannot handle 4-digit year dates, and dates from the year 2000 may possibly be interpreted as 1900 [Need 95]. Thus, unique problems and bottlenecks were expected and weaknesses of our infrastructure should have been exposed. Worst-case scenarios included examples like disruption of public transportation, banking and finance, hospitals, telephone and mail services, and various kinds of governmental services, e.g. social security, tax offices, and worst of all military installations [YoYo 99]. Besides these immediate failures, many worst-case scenarios were derived for the long-term range, like breakdown of international economy, a plague of lawsuits filed by shareholders, the families of deceased patients, and swarms of other people harmed by Y2k failures [Hyat 99].

Most consequences of the millennium bug have been predicted for the turn of the year, when the clock struck midnight on January 1st, 2000. However, at the time of writing only few and so far harmless errors have been observed. Some examples are given in the following list (Sources: CNN, Reuters, CNET, ZDNet, Der Standard, ORF, Heise Online, IY2KCC, USENET comp.risks and RISKS Forum):

- The radiation controllers in two Japanese nuclear power-plants failed to operate correctly, requiring to manually input the correct data.
- Seven US nuclear power-plants reported minor problems, although none of them affected safety systems.
- A US spy satellite and a French military satellite encountered minor glitches, but were reportedly never out of control.
- The atomic weapon laboratory in Oak Ridge, Tennessee, which is also the biggest US uranium depot, faced allegedly harmless date problems
- Egypt's national news wire service briefly stopped filing.
- In three Swedish hospitals cardiac control systems shut down operation.
- A Stockholm bank refused electronic access to customer's accounts.
- A bank in Cologne incorrectly transferred several millions to some customers.
- Several hundred slot machines at the Delaware horse track shut down operation.
- A CyberCash credit-card verification caused duplicate transactions.
- Some jail sentences in Italy were increased by a century due to Y2k-bugs.
- The South-Korean Millennium Baby was registered with an age of 100 years.

These news items are also confirmed by official authorities. The US government (President Clinton's Y2k-representative John Koskinen) announced, that only minor incidents have been observed, which have been solved immediately. A similar statement was issued by Bruce McConnel, the Year 2000 representative of the United Nations, who stated, that globally no major disturbances have been perceived. Yet, the French magazine "01-Informatique" reported, that more than 15 percent of the French industry observed glitches due to the Y2k-

bug, especially in the areas accounting and finance, but also in personnel, payment, and production.

However, all in all it seems, that the preparations for the Y2k-bug have paid off. Considering the amount of money, this was not a cheap effort. For example, the US government estimates, that governments and companies worldwide spent US$200 billion to prevent a Year 2000 catastrophe. These numbers are even exceeded by the International Data Corporation (IDC), who reckoned up to US$280 billion, while the Gartner Group evaluated even US$300 to US$600 billion in expenses. Of course, these amounts and the actually observed effects of the Y2k bug lead to some critical voices concerning waste of money. There are some critical issues, that have to be mentioned in this context. Firstly, it remains to be seen if the upcoming months still do not reveal any major Y2k failures. Secondly, it will forever be unknown, what would have happened if these investments in the information technology sector would not have been placed in advance. Thirdly, there is always the problem of concealment, and some (probably critical) Y2k-issues will never be revealed in public.

Related to the Y2k bug there are other notorious dates, like February 29th, 2000, which is an intercalary day that may be missed by some systems. Another previous example is the "Nines Problem", which was predicted for September 9th, 1999. Concerns over this date arose from a programming convention that used four nines in a row - 9999 - to tell computers to stop processing data or to prepare for maintenance. Some thought, that the 9/9/99 would be a precursor to the millennium bug, although the latter is expected to be much more critical. Yet, September 9th, 1999 has passed without any major incidents or disruption of computer systems in Asia, Europe, and the US. Some reasons for this unexpected non-appearance are, that the date would have to be stored in BCD or ASCII-format and would equal "090999" or "990909" as usual.

3.4 Errors in Highly-Complex Systems

While these examples of bugs have shown, that there is a great impact of software errors on mankind, there may be even more upcoming difficulties.

A system, that was cancelled before it could cause any dramatic errors and failures, was the proposed US Strategic Defense Initiative (SDI) in 1983. It was intended to defend the US against nuclear missile attack with computer-aimed weapons. Estimations predicted it to require up to 100 millions lines of code, a complexity that hasn't been reached by existing systems until now. Yet, testing was impossible, since it would have required to fire missiles onto US targets, which meant, that the system was expected to work bug-free the first time it was ever used.

However, testing and debugging has already to face problems of this magnitude. Due to the ever increasing number of elements in computer systems, the probability of failures increases steadily, even though the reliability of the system components is improved [Feyn 96]. For example, the scale of massively parallel systems will also mean drastically reduced system stability and reliability [Gust 98].

Example 3-1: Reduced system stability and reliability (compare with [Hoss 99])

> Consider a parallel system that consists of 100 SMP clusters with 128 processing elements each, making 12800 processing elements in total. If we assume the mean time between interrupt (MTBI) for hardware failures of one such cluster is 500 hours, than we would obtain an overall MTBI of 5

> hours for the total system. If we assume that a processor-memory module yields a mean time between failure (MTBF) of 3 years, than the MTBF of the whole 12800 modules would be approximately 2 hours. Furthermore, if we assume a system being capable of 1 Tflop/s (10^{12} operations per second) operational speed, only 10^{16} operations could be performed between two failures, which is not really much measured at the criteria of Grand Challenge applications.

This means, that parallel systems offer a high degree of possible errors. The situation is getting worse when programming and other software bugs are taken into account [Hoss 99]. Dealing with Computational Science & Engineering (CSE) problems of Grand Challenge scale, one has to face the fundamental responsibility to verify the results of numerical computer simulations, since these will be used as replacements of experimental explorations [Hoss 99]. During testing, the expected results of a program are often compared with the output of an older, slower program, which is believed to be more reliable [WaBl 97]. Yet, there may not exist a program for problems of that scale.

The Advanced Strategic Computing Initiative aims to replace physical nuclear-weapons testing with computer simulations. This focuses much-needed attention on an issue common to all Grand Challenge computation programs [Gust 98]: How do you get confidence - opposed to mere guidance or suggestion - out of a computer simulation? Making the simulation of some physical phenomena rigorous and immune to the usual discretization and rounding errors is computationally expensive and practiced by a vanishing, small fraction of the high performance computing community. Some of ASCI's proposed computing power should be placed not into brute-force increases in particle counts, mesh densities, or finer time steps, but into methods that increase confidence in the answers it produces. This will place the debate over ASCI's validity on scientific instead of political grounds [Gust 98].

3.5 An Example of Bug Potential

In order to understand the implications of bugs for large scale highly-complex computer systems, we will now describe a well-known example of a widely available operating system, Microsoft's Windows. Of course, the discussion below may be biased toward's one operating system or another, but it is sufficient for our intention. In addition, this software product is the most widely used operating system currently available, and almost everybody in the modern world is confronted with it from time to time.

The operating system of Redmond's software manufacturer Microsoft has always been the target of error reports, which are very often presented in everyday's media. Early reports are already presented in 1991, when the "unrecoverable application error" of Windows 3.0 and later the "general protection fault" of version 3.1 became a household phrase after the introduction of Microsoft's new operating system [Need 95]. Another example is an error in the calculator of MS Windows, which did not display correct answers in certain cases. The interesting thing about this errors is, that it was admitted by Microsoft only after the detection of the Pentium Bug. This glitch in Intel's flagship processor architecture, which is the main hardware platform for Windows, is one of the most widely reported bug in history [Need 95]. Its results were incorrect floating-point divisions on Intel's chip, that were estimated to have an occurrence frequency from days to millennia [BlWa 96]. Although this bug happens only occasionally, it still is embarrassing and led to a flurry of controversy and an unexpected public-relations debacle for Intel. And yet, a stunned Intel management pointed out, that occasional bugs are the rule rather than the exception in released microprocessors [BlWa 96].

Of course, there are comparable examples from other software vendors. Even compilers contain lots of bugs that may seriously affect the program development process. Evidence is given in [Boyl 99], which describe experiences of programmers with bugs, that would only go away after compiler patching or upgrading. Furthermore, projects like the GNU C compiler, a de facto standard on UNIX machines, have evolved over a long history of problems and bugs. Sun Microsystems' Java Development Kit, JDK 1.1, was originally shipped with bug fixes planned for future versions. All these problems introduced the construction of verified compilers, which are compilers that have been proven to correctly compile any correct program into assembly language.

Another example is the failure rate of utilities on commercial versions of Unix, as indicated by Table 3-1 [Lewi 99]. The operating systems reliability is used as a metric to compare different software products, in concrete the reliability of a large collection of basic UNIX utility programs, X-Window applications and servers, and network services. The study was conduced by Miller et al in 1990 [Mill 90b] and repeated in 1995 [Mill 95b]. The applied testing methods were largely automatic and subjected these programs to random input streams. For this test, the results delivered the failure rate of these tools in terms of crashes and hangs (infinite loops).

Table 3-1: Reliability of commercial UNIX utilities

	lines of code	failure rate
Commercial UNIX	10 million	15-43%
Linux	1,5 million	9%
GNU utilities		7%

Neglecting the commercial and public consequences of this comparison as well as its accuracy, we can identify the defect rate as a possible estimation of software quality. Yet, the defect rate as a single-point estimate is less important as the defect density, which can be derived from the failure rate and the lines of code. In that case, the defect density of the Linux utilities is still higher than that of other commercially available UNIX utilities. This would lead to the conclusion, that commercial UNIX systems are more reliable than the Linux counterpart by a factor of 30 to 400 percent. Again, the validity of this comparison may be questioned and still, defect density is only a single-point estimate of software reliability.

A well-known story is based on Microsoft's Windows 98, which has been declared to have fixed 5.000 bugs that plagued Windows 95. In response, many journalists reacted by indicating that an operating system that contained 5.000 well-known bugs has been shipped in the estimated number of 50 million copies of Microsoft's product [Lewi 98]. Lewis asked, if these problems of Windows 95 indicate, that the problem of software quality is unsolvable. His estimations are based on an approach by Jones, who defined the defect potential of software as follows [Jone 96]:

Definition 3-13: Defect potential [Jone 96]

> The defect potential $B(0) = \mathrm{FP}^{1.25}$, where FP is the number of function points in the new software product. For enhanced software, an exponent of 1.27 is used instead of 1.25.

To determine the number of function points of a particular program, Jones recommends to divide the lines-of-code (LOC) by 100. Thus, for Windows 95, which has reportedly 15 million lines of code, we estimate FP = 150.000 function points. This yields a defect rate of potentially 2.95 million bugs in Windows 95! Please note, that bug potential is not the same as number of bugs, and the inaccuracy of the term “bug” itself (see Section 3.2.6), which is often used ambiguously. Not every bug is always a “failure” or “fault”. Thus, 5000 bugs are obviously not 5000 critical failures, but many may be occurrences of rarely used commands exceeding their response time marginally.

Another number often used for comparing the reliability of software is failure density, which is defined as follows [Brit 98]:

Definition 3-14: Failure density [Brit 98]

The failure density $D = \frac{\text{failures} \times 1000}{\text{LOC}}$ is used as a measurement of failures per thousand lines of code.

For the data above, we may thus calculate a failure density of 0.33 per thousand lines of code, which is comparable to other products of that size according to Britcher [Brit 98]. Please be aware, that these numbers should not be taken too seriously, because all the data is based on estimations. Additionally, the number of faults in any large system is virtually unknowable, and many of these faults may never cause failures. Some faults go undetected simply because that part of the system is never executed. Other faults may influence or even nullify other effects. In addition, a system could have been developed fault tolerant in the first place and may be able to react to the program’s behavior [Brit 98].

This leads us to the learning curve, a method to describe the improvement of software between subsequent testing iterations. Obviously, the bug potential as defined above is not the same as the number of bugs in the code. It may only serve as a way to estimate the amount of required testing. In this case, we introduce the defect-removal rule of Jones, which is defined as follows [Jone 96]:

Definition 3-15: Defect removal rule [Jone 96]

The defect potential $B(k) = B(0) \times (1-R)^k$, where R is the learning rate during one test-and-fix step, while k is the number of applied test-and-fix iterations.

If we assume a learning rate of 30% for each inspection or test step, which means that 30% of the errors are detected and removed, we would achieve a remaining defect rate of 5000 after 18 test iterations. Since Jones recommends 6 to 12 iterations, Microsoft’s software practice appears to exceed the industry norm. In order to evaluate the number of test iterations required to remove the defect potential from 2.95 million to 1, we may calculate the number of required test-and-fix iterations as follows [Lewi 98]:

Definition 3-16: Required test-and-fix iterations

$$k = \frac{-\log\left(\frac{B(k)}{B(0)}\right)}{\log(1-R)}$$

This means, that a total number of 42 steps would be required to reduce the defect potential from 2.95 million to one [Lewi 98]. It is clear, that Microsoft Windows is only one example. The estimations described above are similar for other projects, and the potential consequences of malfunctioning computer systems increase with the integration of computers in everyday life operation. The systems themselves grow more and more complex, which drives the need for new assurances of reliability [BlWa 96].

3.6 Summary: Testing and Debugging

This chapter described the testing and debugging phases of the software life-cycle, which are applied to ensure the quality of software products. Yet, even the best testing and debugging approaches can only target at stable software, which is not completely error-free, simply because not every bug may be detected. This is also explained with some of the most notorious errors, which indicate that errors in software packages are ubiquitous and may happen anytime and anywhere.

We have also shown that bugs exist on different levels of severity, indicating that some bugs are more critical than others. In practice this means, that debuggers should first help to eliminate the most critical bugs and then to solve minor problems, although this has been proven difficult in many cases.

Another critical problem is the nature of some programs itself. Due to their complexity, some programs require more testing than can ever be applied. Some applications even offer solutions for problems (e.g. on the Grand Challenge scale), which cannot be verified in reality. How can we be sure that the results are correct? How can we trust the computation of large scale problems?

4. Debugging Parallel Programs - Related Work

After describing the area of application - massively parallel computing - and the necessity to support users during the tedious error detection phase, we will now describe the testing and debugging cycles for sequential and parallel programs in more detail. Starting with a traditional approach of testing, we will introduce the problems and difficulties that programmers face when analyzing arbitrary parallel programs. This leads us to an overview about debugging obstacles due to nondeterministic program behavior and the necessary extensions of the traditional cyclic approach. Throughout this chapter, we will describe the relevant theoretical background as well as some of the important solutions and tool implementations related to this work.

- Traditional sequential testing and debugging cycles
- Obstacles during parallel program debugging
- Extended parallel testing and debugging approach
- Requirements on a parallel debugging tool
- Parallel program analysis and visualization
- Comparison with existing related work

4. Debugging Parallel Programs - Related Work

4.1 Traditional Testing and Debugging Cycles

4.1.1 Simple Checking

The testing and debugging phase of the software life-cycle is applied, whenever a stage of the implementation is reached that allows execution of the target, which is either the complete application or an independent module. This means, that all syntactical errors have been removed and the source code was compiled successfully.

Testing and debugging always operates in two cycles, during which the program is executed again and again until a sufficient degree of reliability has been obtained (see Section 3.2.2, “Problems of Testing” for a detailed discussion). The testing cycle is used to detect incorrect computations, while the debugging cycle allows the user to gain more information about program states and intermediate results during execution. The idea is to identify, locate, and correct program bugs based on comparing achieved results with expected results [WaBl 97]. In this context, Blum and Wasserman formally define the term “simple checking” as follows:

Definition 4-1: Simple checking [BlWa 96]

> Let f be an arbitrary mathematical function. Then consider the following task: given an input x and an output y, determine whether or not $y = f(x)$; that is, determine whether or not the input/output pair $\langle x, y \rangle$ represents a correct computation of f. This task is called *simple checking*.

In this case, the function $f(x)$ describes the behavior of some program $P(x)$ as intended by the user. Therefore it serves as a specification of intended program behavior, which is required for evaluating the correctness of the program as defined in Section 3.2.3. To recapitulate, a program is always correct relative to its specification. This correctness is verified during testing cycles, which apply simple checking iteratively for a set of inputs $X = \{x_0, x_1, x_2, \ldots\}$.

4.1.2 The Testing Cycle

The traditional approach to testing and debugging is sketched in Figure 4-1 (compare with [KrWi 98]). It consists of two cycles, each represented by a flow-diagram: the left flowchart represents the testing cycle, while the right diagram shows the debugging cycle. The testing phase starts with selection of appropriate input data. Therefore, the user chooses an input x from the set of valid inputs X. This input is provided for a program execution, which should compute the corresponding set of results y. Due to the fact that exhaustive testing would require an almost infinite number of inputs, the end-marker of the testing cycle has been omitted deliberately.

After the execution of the program terminated, the program’s execution is checked whether an error occurred or not. As specified in Definition 3-7 and 3-11, an error can either be a program failure or incorrect results. Thus, these two characteristics have to be inspected. Immediately after program termination, two distinct incidents may be observed. Either the program’s execution failed before the program arrived at its intended destination, or it terminated cor-

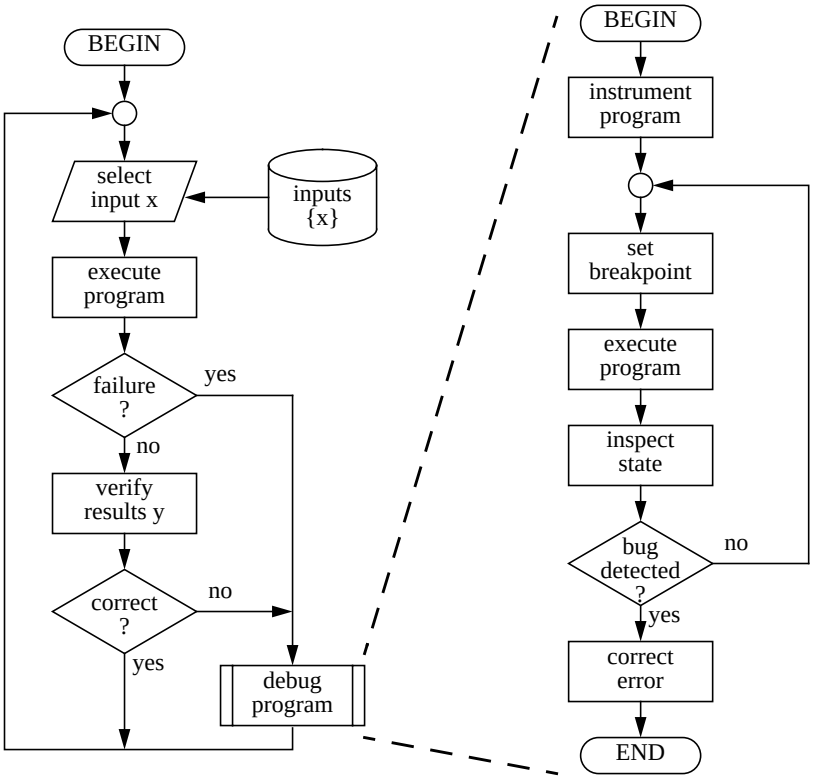


Figure 4-1. Traditional testing and debugging cycle

rectly. Obviously, a failure is always an indication of incorrect behavior, no matter if it produced incomplete results or no output at all. Consequently, the debugging cycle can be initiated immediately afterwards.

In case the program terminated correctly, the situation is more complicated because the program produced some amount of output, which may even be appropriate at first sight. The problem is that the validity and correctness of these results have to be verified. This may be difficult in many cases, because correct results have to be obtained somehow. For many functions, it is easier to check a completed computation than it is to carry out the computation itself. For example, it is difficult to factor an integer, but, given a, b, it is trivial to determine whether or not b is a factor of a. Various methods have been developed for checking a broad variety of functions asymptotically faster than they can be computed [BlWa 96].

An interesting approach in this domain is offered by the testing and debugging tool Guard [AbSo 95], which has been developed at the Griffith University, Australia. Guard emphasizes the fact that software is developed with evolutionary engineering techniques. This means, that the latest incarnation of an arbitrary application is always somehow related to its predecessors, a fact, that is also integral to source code control systems like SCCS and RCS, as well as program dependency tools like Make. For testing and debugging this means, that Guard enhances the traditional approach by automating the comparison of data structures between two running programs. With this technique, different iterations of the testing cycle can be compared automatically and Guard will deliver the variations between these executions. Furthermore, if a predecessor version of the target program is known to operate correctly, incorrect values of an arbitrary follow-up version can be detected easily.

A comparable approach, albeit based on statistics, is described with statistical testing in [Walt 95]. By introducing a software usage model, the authors claim to detect errors that occur

frequently in operational use of the software. Abrams et al have followed another comparable idea in the CHITRA trace analysis tool for performance analysis [Abra 96]. Their idea is to use statistical methods to characterize parallel program executions, and to view the obtained executions always only as samples from the set of all possible program execution, thus applying statistical methods for program analysis [Abra 95].

4.1.3 The Debugging Cycle

As mentioned above, the connection between the two cycles is established, whenever testing detects an anomaly. This may either be a program failure or incorrect results. If the observed results are incorrect or a runtime failure occurs, the program is subject to debugging.

A main difference between the testing cycle and the debugging cycle is, that testing operates on a set of individually different inputs, while debugging performs analysis of program execution based on one particular selected input. The input under consideration is obviously the input *x*, that is responsible for the anomaly and initiated the debugging cycle in the first place. Thus, the debugging cycle is always carried out with a particular set of input data selected during testing. Please note, that in large projects the testing and debugging cycle is often carried out by distinct teams or team members. The interface between these two collaborators is then the description of the anomaly and its corresponding input data.

The debugging cycle is presented in the right flow diagram of Figure 4-1. It starts with instrumenting the target program. This means, that debugging code is added to the source code which observes the program's execution. Afterwards the program is analyzed iteratively to detect the lines of faulty code. Although it depends on the applied tool, there are usually three activities taking place (compare Definition 3-6):

- Setting a breakpoint[1]
- Executing the program until the breakpoint
- Analyzing the process state at the breakpoint

The connections between these three activities are described with Figure 4-2. The first timeline shows one iteration of the testing cycle from Figure 4-1. Given some arbitrary input, the program is executed until it arrives at an observation point. At this point, an error is observed, which means that either a program failure occurs or incorrect results are detected. Obviously, the reason for this incorrect behavior must be placed at the observation point or somewhere before the observation point. Thus, the problem of the debugger is to deduce the conditions under which the program produced the incorrect output [Agra 91]. This characterizes debugging as a "backward" process. In practice, this activity is called "flowback" analysis [Balz 69] and there are basically two solutions to this problem:

- Reverse execution
- Backtracking with re-execution

1. As mentioned in Section 3.2.5 "Debugging Basics", setting a breakpoint is not a trivial activity, neither in the parallel case nor in the sequential case. It would therefore be highly appreciated, if debugging tools support the selection of an appropriate breakpoint position. We will discuss this problem related to our solution in more detail in Section 7.5.

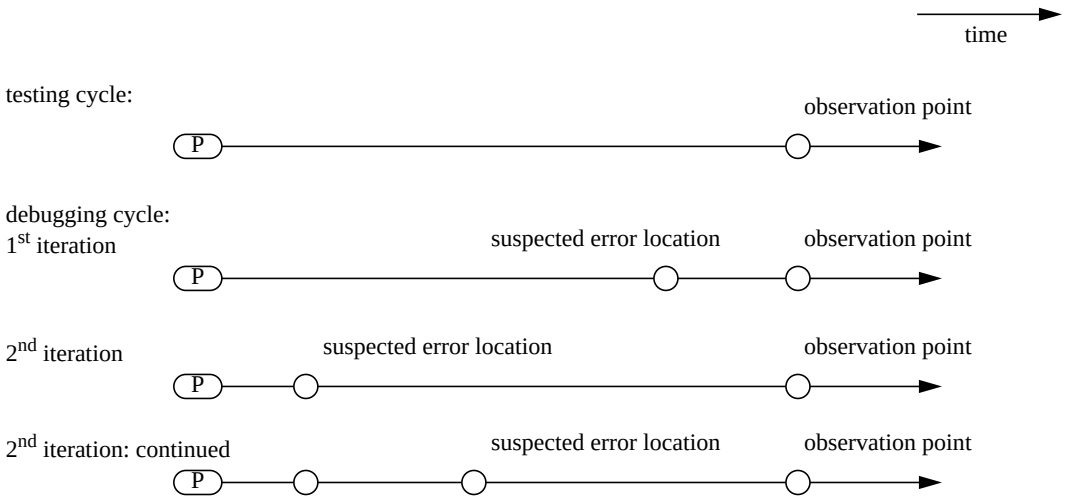


Figure 4-2. Iterations of the debugging cycle

The best solution would be to execute the program backwards one step at a time, starting at the observation point up to the point of the bug's origin [Choi 91]. By comparing the states of the program at each of these steps, the error would be localized between the point where the state of the program was what was expected and the first point where the state was not what was expected [Paul 99]. However, while this optimal solution could certainly improve the situation of many debugging users, it is not feasible in reality. The reason is that it is practically difficult and most times impossible to execute a program reverse in time, because many statements in a program and therefore the computing processes are irreversible[2]. One obvious example is the manipulation of file pointers and I/O descriptors [Feld 88].

As a consequence, debugging is always based on backtracking by re-execution, where the program's statements are executed using their original order. The goal is the same as for reverse execution, to track the processes' history backwards in time from the manifestation of the error (the observation point) to the point at which the bug initially caused the erroneous behavior [Choi 91].

In order to investigate the history of the process, the user is provided with the illusion of reverse execution [PaLi 88]. To achieve this, a breakpoint is positioned somewhere in the code before the observation point as visualized with the second time-line diagram of Figure 4-2. Since it is a priori unknown, if the breakpoint is really adequate, it can only be positioned at a suspected error location. Afterwards the program is executed until this breakpoint is reached. At this suspected error location the process state can be analyzed and information about the program's behavior can be gained. If this information is sufficient, and the bug has been detected, the debugging cycle is stopped and the faulty lines of code can be corrected. Otherwise, the error may either be located before or after the selected breakpoint location, which defines the new suspected error location. If the current breakpoint is placed after the new suspected error location, the program has to be restarted and executed again up to the new break-

2. An obvious possibility to revert every statement of a program would be to store the complete state of the processes at each state change. However, this would require huge amounts of memory and is therefore not feasible for average sized applications. A solution that comes close to this idea is described below with the approach of Balzer in EXDAMS, which must be connected to its invention in 1969, when programs were indeed smaller and consumed less memory than today.

point. On contrary, if the current breakpoint is before the new suspected error location, it is usually easy to continue execution from the current position.

Some related work in this context are the prototype implementations of IGOR [Feld 88], Spyder [Agra 91], and PPD [Choi 91]. IGOR is a tool that supports reversible execution with a mechanism available on arbitrary virtual memory systems [Feld 88]. The basic idea is to peek at the virtual memory page tables and trace the reasons why these pages have been dirtied since the last checkpoint. A different approach is taken by Spyder, a debugging tool for monoprocessor C-programs which offers dynamic program slicing [Weis 84] for execution backtracking [Agra 91]. Initially, an arbitrary variable and a corresponding program location is selected, for which Spyder automatically determines the statements that affect its value up to that position in the program. After executing the program with different test cases, Spyder helps to decide which statements in this dynamic slice to examine first. And finally, it allows to restore the program state at any location by backtracking the program execution to that location, without requiring to re-execute the program from the beginning.

A prototype debugging system dedicated to parallel programs of the Sequent Symmetry shared-memory multiprocessor is the Parallel Program Debugger PPD [Choi 91]. It provides information on the data and control flow of a program's execution, while trying to keep both execution-time and debug-time overhead low. This is achieved by combining incremental tracing and program re-execution. With incremental tracing, the amount of tracing is selected according to the needs of the debugging user. Initially, only a small amount of trace data is generated during program execution. Afterwards, this initial data is supplemented during follow-up debugging cycles by detailed information, which is generated by tracing only selected parts of the program that are re-executed on demand.

The biggest problems of these approaches are the costs associated with restarting the program's execution, which is especially unacceptable for larger programs [PaLi 88]. As a consequence, a variety of solutions is proposed to restart the program from intermediate code positions. In 1969, Balzer introduced the "history tape" within the EXtendable Debugging and Monitoring System [Balz 69]. This is a observation capability that collects and stores all the data about a program's execution in order to format and present it to the user during debugging. Based on these data, flowback analysis can be performed by constructing an inverted tree, that describes how information flowed through the program to produce the value at the bottom node of the tree. In addition, a function for debug-time history-playback allows to retrieve information from the history tape, at any time of the program's execution.

The problem of the history tape is, that it requires enormous amounts of memory and is practically useless for present-day applications. In order to decrease these memory requirements while at the same time omitting the need to restart the program's execution at the beginning is to store required execution data periodically during the program's execution [PaLi 88]. This is called "checkpointing" and provides the possibility to restart the program at all the places, where a snapshot of the program has been saved [Feld 88]. While this approach is especially useful for debugging, it has also been used simply to ensure efficient usage of expensive computing environment, whenever long-lasting programs are executed and require a certain degree of fault tolerance. In contrast to pure debugging, the main goal of checkpointing is to resume the execution, whenever a program's execution fails due to the loss of some external service [Plan 95].

The traditional approach to testing and debugging as described above provides a brief overview of these activities and how they are usually carried out for sequential programs. Several testing and debugging tools have been developed that support users in that area. In addition, programmers tried to apply the same kind of techniques to parallel programs, and expected

similar results. Yet, there are some fundamental difficulties that have to be solved for parallel program debugging, which cannot be mastered with this traditional debugging cycle. These problems will be discussed in the next section, before proposing a revised debugging approach that is also sufficient for testing and debugging of parallel programs.

4.2 Obstacles During Program Debugging

4.2.1 Problems of Sequential Debuggers

In principle, a parallel program is just a collection of several sequential processes, that execute concurrently and communicate in some way with each other. Consequently, the principle problems of sequential debugging are also evident for any parallel debugging tool. Rosenberg defines four key issues for designing and developing a debugging tool [Rose 96]:

- The Heisenberg Principle: a debugger should intrude the debuggee only in a minimal way.
- Truthful debugging: at all costs, the debugger must be truthful so that the programmer may always trust it.
- Program context information: the most important issue is the presentation of content information, to know where one is.
- Debugging trails system development: any debugger that may be applied is almost always behind the technology one is using.

The Heisenberg (Uncertainty) Principle describes the intrusiveness of the probes inserted into any system being measured or tested [LePa 85]. Originating from quantum mechanics, it states that one can never know everything about a quantum state [Feyn 96]. Applied to software, another name for this behavior is the “Probe Effect” [Gait 86], while errors established by these effect are often called “Heisenbugs” [LePa 85][Rons 00]. Obviously, the intrusion should be kept a minimum in order to guarantee analysis data as correct as possible. In particular, the act of debugging an application should not change the behavior of the application. Otherwise, the usefulness of the debugger falls into question [Rose 96].

In software debugging, a debugger violates the pure Heisenberg Principle in many ways, simply because it is in memory and controlled by the same operating system as the application being observed. This can affect the application in many ways, especially if the program is parallel and nondeterministic. For that reason, we will describe the Probe Effect in more detail in Section 4.2.8.

Bugs that are affected by this sort of thing are rare but extremely challenging and time-consuming because they are now so elusive. For sequential programs, bugs of this sort may also disappear when print statements are added to the source code of the debugged application, because the addition of the print statement may shift objects around in memory just enough to move the elusive bug somewhere else or even masks its existence. Another example is the disappearance of bugs when run under the auspices of a debugger, which leaves developers with no effective way to proceed [Rose 96].

The most simplest solution for the Probe Effect would be to constantly perform program observation. In this case, the observation data describes the program’s behavior accurately, but can be thrown away if not needed. This approach has been proposed many times, but isn’t rel-

evant in practice. Its main disadvantage is that observation activities need to be pinpointed well in advance, which makes this solution to rigid for practical usage. In addition, it requires some mechanism for managing observation data, which would be especially crucial on multitasking, multiuser systems.

Another critical issue of debugging is to always provide truthful information during debugging, which means, that the debugger must never mislead the user because the user is frequently testing out theories of how the observed failure may be cause [Rose 96]. Obviously, any misinformation will direct the user in a potentially wrong direction, may cause a general lack of trust into the debugging tool and will dramatically hamper effective progress simply by devastating the user. Some big problems of debuggers are observed with optimized code, simply because the optimizing compiler changes the code radically and may prevent accurate information about how the optimized code maps back to the original source code. Another problem is observed, when variables are inspected which are currently being transferred between their actual memory location and the processors registers. Furthermore, compilers may eliminate unused variables from the code, which may be difficult to be recognized by the debugger [Rose 96].

The most important issue for the debugging user is program context information, which may include several different types of information about source code, stack back-trace, variable values, thread information, and others. Therefore, a debugging tool must always relate the manifestation of the bug with the original lines of source code [Rose 96]. In principle, programs can only be debugged, if the connection between their behavior and the original source code can be established [Baec 97]. Yet, these lines may not contain the actual cause of the bug, because many bugs occur in a place much earlier to the manifested effects. Thus, the next important issue is to backtrack the code to the original place of the bug, the root-cause.

And finally, one of the biggest problems of debuggers is, that system developments occur long before any corresponding strong debugging support for the new systems developments is available. System vendors react to the mass market, which in turn requires certain technologies. Of course, debugging is only one of these technologies, but only with sufficient support for debugging, reliable systems may be developed. Therefore, Rosenberg suggests, that debugger developers need to push the systems vendors to provide the necessary infrastructure in order to enable support of the latest debugging technologies, while at the same time application developers need to push for debuggers in order to debug their complicated applications on the new architectures with its next great features [Rose 96].

4.2.2 Fundamental Differences between Sequential and Parallel Debuggers

The description of the debugging and testing phase is generally valid for the sequential and the parallel case. Similar to parallel programming, where many approaches start by extending sequential codes, many parallel debuggers are simply combinations of existing sequential tools. However, the nature of parallel programs requires some fundamental extensions from parallel debuggers. As a consequence, some existing solutions may be insufficient for some problems of parallel error detection.

In principle, there are mainly three reasons why parallel debugging differs from its sequential counterpart:

- Increased complexity
- Amount of debugging data
- Additional anomalous effects

One reason is the increased complexity of parallel programs compared to their sequential counterparts. While most sequential debuggers perform symbolic debugging[3] on source code level, this may be hard in the parallel case.

For example, consider opening a source code window for each participating process, and perform debugging a program with more than a handful of processes. It has been noted before that text is inappropriate in many cases to display and manage the inherent complexity of parallel programs [Panc 96a]. Therefore, some means of abstraction seems useful to establish a layer above the source code level. However, it has also been noted above, that some connection to the original lines of code is required in order to detect the corresponding location of faulty statements.

Another reason is the possible huge amount of storage required for debugging data, that has to be provided and analyzed. Since debugging is based on the program's results and the states of the processes during execution, analyzing these data from massively parallel programs may impose a heavy burden onto developers. Often immense and uncontrollable amounts of data are obtained [Ober 98], and users are often overwhelmed with potentially irrelevant data [Reed 97]. The result is the so-called "maze-effect" [Damo 94], which means that observers are completely lost. Although this is also possible in the sequential case, it is much more likely in parallel machines, simply because of the increased amount of debugging data. Therefore, a debugging tool must allow to concentrate on the essential details, instead of flooding the user with all the possible data [Netz 96].

In addition there are other problems which exist on smaller program scales, too. An example is the set of errors in parallel programs, which consists of multiple sequential bugs and errors established by synchronization and communication of concurrently executing tasks. While the multiplication of sequential bugs may already represent a big obstacle, the existence of anomalous effects due to concurrency makes parallel debugging even worse. The reason for these effects in parallel programs stems from the increased influence of the underlying parallel hardware system. Consider the debugging cycle for a simple program, consisting of the subfunctions f_0, f_1,... f_n as sketched in the left diagram of Figure 4-3. Applying the simple checking method, we would use repeated executions in order to determine the correct intermediate states between these subfunctions. If the sequence of subfunctions f_0, f_1, and f_2 describes the desired computation, the resulting output y is determined by the provided input x and the interaction with the system environment. If the input is the same and system-interactions can be reproduced, each iteration of the debugging cycle would yield the same output, and intermediate results can be analyzed with breakpoints and single-stepping during iterative re-executions.

In parallel programs the situation is more difficult, because the subfunctions are executed on concurrent processes and all process-interactions are influenced by the computer system. Consider the parallel program of Figure 4-3, consisting of subfunctions f_0, f_1, and f_2 on process 0, as well as g_0, g_1, and g_2 on process 1. Again the output is determined by the computation of these subfunctions, the input x, and all system interactions. Since communication and synchronization are carried out by the system, even small changes in the system state may trigger different behavior [Netz 96]. Some of these system characteristics are (compare with [McHe 89] and [Kran 97a]):

3. "Symbolic Debugging" in this context means, that the debugger operates on the original source code by relating the machine instructions to the actual program statements with the help of compiler generated symbol tables.

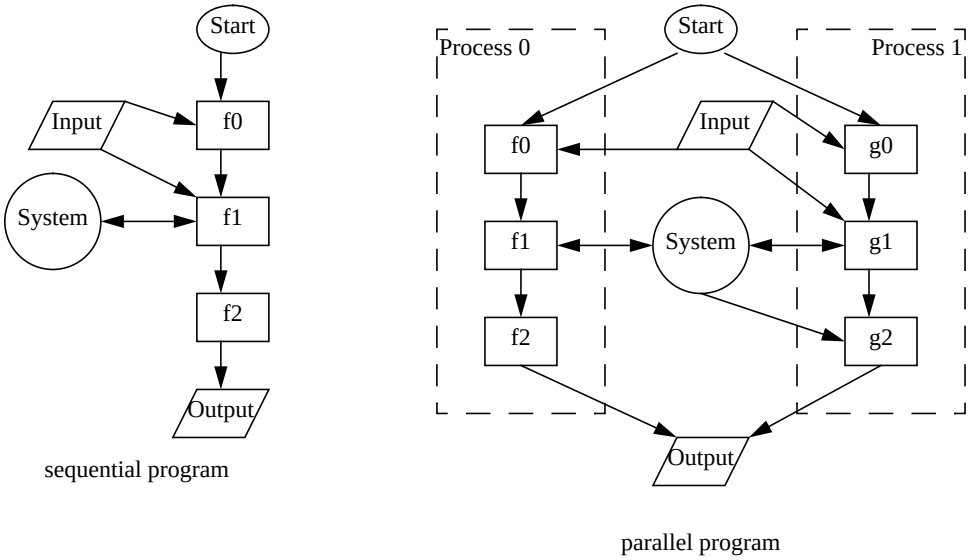


Figure 4-3. Comparison of sequential and parallel program influences [KrVo 98]

- Processor speed and load imbalance
- Scheduling decisions of the processor and the operating system
- Cache contents, cache conflicts, and memory access patterns
- Network throughput and network conflicts (latencies, contention)
- Nondeterminism on the interconnection network

Unfortunately several of these environmental parameters cannot be controlled by the user, neither during subsequent executions of the program in the testing cycle, nor during the debugging cycle with the debugging tool. Therefore, these small variations in the system environment may vigorously influence the execution path of a program and introduce lots of difficulties for program testers and debuggers.

4.2.3 Errors from Process Communication and Synchronization

The influences described above are experienced whenever concurrent processes interact, because there are always some kind of system services involved. Therefore, the necessity of parallel programs to exchange data or to synchronize the communication partners introduces effects, that do not exist in sequential programs and may therefore be ignored during corresponding debugging cycles. Yet, for parallel debuggers additional functionality for handling these communication effects is a basic requirement.

Since the main reasons for these obstacles are communication and synchronization operations, the amount of observable effects is determined by how this process interaction is carried out during the program's execution. As described in Section 2.3.4, the two contrary paradigms are shared variable and message-passing. Due to the fact, that processes in shared variable programs are tightly coupled, interprocess communication is usually less expensive than in message-passing programs. The latter suffer high communication costs due to communication latency and achievable bandwidth. For that reason, the amount of interprocess communication

and therefore the probability to encounter critical effects is usually higher in shared variable programs.

Although there are differences between these two paradigms in terms of occurrence frequency, the types of possible effects are similar in both cases. Some of the most tedious difficulties are:

- Deadlocks and livelocks
- Stampede and bystander effects
- Irreproducibility effects
- Completeness problems
- Probe effects

Deadlocks and livelocks are well-known effects arising from concurrently executing and communicating or synchronizing processes (e.g. [GlSh 80], and [Nata 86]). The former describes the situation, where each of the participating processes is mutually waiting or blocked until one of the others does something [MaJa 97]. The latter characterizes a group of processes that is continuously changing their state in response to changes of other processes without doing any useful work [Buth 99]. In both cases, the program's progress is suspended until something resolves the conflict. As a side effect, the state where each of the processes persists, is usually a good starting point for debugging, and errors of this type can usually be corrected easily [SnHo 88]. Several tool exists for deadlock and livelock analysis. A recent example is the Deadlock Checker [MaJa 97], which uses various innovative techniques to prove deadlock and livelock-freedom for communicating sequential processes. Other solutions are described in [Chan 83], [Germ 84], and [Sing 89].

In contrast, stampede and bystander effects are much more difficult to debug. The stampede effect [SnHo 88] may emerge after an error has already been encountered on one process. However, since it is rarely possible to stop all the other processes immediately, they may continue to execute and stampede over the evidence of this original error. As a consequence, only limited useful information is left for the subsequent debugging process. In case of the bystander effect [SnHo 88], the data space of one process is corrupted by another one. Obviously, the corrupted process is the first candidate for debugging, when in fact a different process originally caused the damage.

The effects described so far exist in any system composed from concurrently executing processes. In contrast, the irreproducibility effect and the completeness problem may only emerge in nondeterministic parallel programs, while the consequences of the probe effect are much more critical in this type of programs. Due to this fact, understanding, writing, and debugging nondeterministic programs is accepted to be one of the most difficult activities in parallel software development [McHe 89]. Consequently, a debugging tool should be able to cope with this group of effects, if it is intended to provide sufficient support during program analysis and error detection.

4.2.4 Nondeterministic Parallel Programs

The origin of determinism and nondeterminism can be traced to philosophy and ethics [Broc 96]. The term determinism defines the causal certainty, that every event in human life is predefined by the laws of nature and its preceding events. In contrast, nondeterminism describes the doctrine, that our lives are only partially determined by the laws of nature, and some parts cannot be predicted by the principle of causality. Therefore, nondeterminism is

much less restrictive than determinism and offers a higher degree of freedom. In reality, especially microphysics, many natural events have been observed (e.g. quantum theory), which contradict the deterministic school.

In computer science, these terms are especially important in the theoretical domain, where computational complexity theory and algorithm analysis are performed. An example is the classification of problems based on how hard they are to be solved. The class P describes problems that can be solved in polynomial time, while the class NP contains all nondeterministic polynomial time problems. The latter contains the group of NP-complete problems, for which all known solutions are essentially of the type “try all possibilities” (compare “The Completeness Problem”, Section 4.2.7), which can be impossible in some cases [Aho 83].

Besides this classification, the terms determinism and nondeterminism are used to characterize the relations between process states in a program. (Program states related to debugging have been defined in Definition 3-8 on page 64.) The interesting question is, if a unequivocal well-defined order of program states exists or not. This is also very important for debugging, since it operates by analyzing the programs states during execution. A possible definition of determinism is as follows:

Definition 4-2: Determinism [Enge 88]

> A program executed by a machine is called deterministic, if there exists exactly and only one follow-up state after each programming statement. This means, that the follow-up statement of any statement is always uniquely defined. In addition, every deterministic program must terminate[4].

Obviously, a program with deterministic behavior offers some desirable qualities, and developers believe, that consistent and reliable software products must be deterministic [Stei 98]. Yet, it also restricts the programmer’s and their code too much for many applications. This is also expressed in some research papers, which believe that the importance of nondeterminism is steadily increasing, especially in the domain of parallel and distributed programming. An important goal is to obtain a better level of performance. If the usage of nondeterministic communications is increased, the degree of parallelism can be raised. As a consequence, potential parallelism in parallel or distributed computing systems is better utilized than in deterministic programs.

A simple case is a FIFO queue (First In - First Out), where data is processed based on their order of arrival [NeMi 92b]. An actual example is offered by parallel ray tracing algorithms (e.g. [CaSc 89], and [JeRe 92]), which compute the final results by dividing the scene onto concurrent processes. The load is distributed between the processes by a master-worker scheme, that handles requests on a first come first served basis. Due to the nature of these algorithms even the same scene may be computed by different processes with a different order of the traced rays.

Although this is obviously the most important reason for using nondeterministic communication features, there are other substantial reasons, for instance when modelling reality. If the computing system simulates a natural phenomenon with nondeterministic characteristics, the simulation program must reproduce the same behavior. Please note, that this is usually implemented with random number generators, but it is also possible to achieve similar results with

4. Please note, that the opposite is not true, and terminating programs may either be deterministic or nondeterministic.

nondeterministic communication statements. Typical applications in this domain are chaotic, physical computations, which may be inherent nondeterministic. Examples are crashtest simulations for car manufacturers (e.g. PAM-CRASH [Lons 94] and [Lons 95]), where different results may be observed although the same kind of impact is used as an input. This represents reality, because the deformation of objects is determined by the applied forces and variations in the environment that are to inconceivable to be modelled. Thus, there is a certain degree of chance involved.

Another reason for using nondeterminism, which is probably more error-prone than the reasons described above, is "programmer's laziness". Often, determining the correct order of nondeterministic events is much more difficult than choosing one of the possible events. Thus, the program is implemented to react to the most likely program states in a way that is assumed to be correct. Yet, this method contains a major pitfall, because introducing nondeterminism often introduces unexpected behavior under certain conditions, which results in hazardous, occasional bugs (see 3.2.6). Therefore, this kind of using nondeterminism should clearly be avoided as much as possible.

Connected to programmer's laziness is our last example of nondeterminism in software, which may be labeled "accidental nondeterminism" or unintended nondeterminism (see [DaFr 94], [Kran 96d], and [KrWi 98]). While all the reasons above apply nondeterminism intentionally, some programs contain nondeterminism without the user's knowledge. In this case, unintentional nondeterminism is creeping into the code due to misplaced or misused synchronization or communication functions. This is very dangerous, because users are unaware about the potential havoc and may therefore miss to check for this possibility.

As a next step, we will now define nondeterminism based on the definition of determinism above, because obviously any program that is not deterministic is nondeterministic. At this point, we should remark that nondeterminism is sometimes called nondeterminacy (e.g. [EmPa 88], [Emra 92], and [DaFr 93]) or even indeterminacy (e.g. [LeMe 87]), but all these terms describe the same characteristics. We will stick to nondeterminism, since it is the most widely used of all these alternatives.

Definition 4-3: Nondeterminism

> A program is nondeterministic, if - for a given input - there may be situations where an arbitrary programming statement is succeeded by one of two or more follow-up states. This freedom of choice may be determined by pure chance or unawareness of the complete state of the execution environment.

This definition shows that the key problem of nondeterminism is the absence of sufficient knowledge about the actual, complete state and its follow-up states in the computing environment. This leads us back to Figure 4-3, which described the differences between sequential and parallel programs, and the influence of the computing system, respectively. As mentioned before, it is difficult if not impossible to describe the complete state of a parallel computing system at any time during a program's execution.

Based on Definition 4-3, we will now try to characterize a nondeterministic program with the following definition:

Definition 4-4: Nondeterministic programs

> Nondeterministic programs do not fully specify all possible execution sequences, but allow a degree of freedom in selecting subsequent program states. Consequently, a program is nondeterministic, if successive executions of the same program may yield different results although the same input is provided [NeMi 92b].

There are two remarks to this view about nondeterministic programs. Firstly, the input must be valid according to the program's specifications. This means, that only input data that may actually be processed by the application may be considered. The reason is, that illegal input may affect the program's execution in an undefined, unexpected way and may thus also be responsible for different results during successive program executions.

Secondly, the appearance and composition of the program's results have to be discussed in more detail. The question is, what are the actual results of a nondeterministic program? Of course, most important are the final results of the computation, which (hopefully) represent the purpose of the application and intent to provide a solution for the user's primary goal. Yet, in the sense of testing and tuning, it may also be interesting to determine those factors besides the input, that influenced the results of the program. Therefore we include the following items in the term "results" of Definition 4-4:

- Final results
- Intermediate results
- Interaction patterns between concurrent processes

All these items are influenced by the nondeterminism of the program and may thus vary in subsequent executions although the same valid input is provided. This distinction is necessary if the correctness of the program is investigated. The reason is, that it may be possible to obtain correct final results with different intermediate results for any nondeterministic program. In addition, we may observe the same final results from a parallel program, whose execution varies in the interaction pattern of its processes. As a consequence, the possible nondeterministic behavior of the program is often concealed for the user, which means a high potential for malicious occasional errors. Such programs may even operate correctly for long periods of time, until suddenly incorrect results are revealed.

Please note, that most research papers in this domain (e.g. [SnHo 88], [McHe 89], [NeMi 92b]) consider only the results of the program as being important. Only recently, Ronsse et al distinguish between external nondeterminism and internal nondeterminism [Rons 00]. External nondeterminism means that an application returns different results for repeated executions with the same input data, while internal nondeterminism means that repeated executions with the same input yield the same results, but the internal execution path is different.

In addition, [DaFr 94] distinguish intended and unintended nondeterminism, and propose to leave intended nondeterminism without investigation. Unfortunately, this leaves the problem, that even intended nondeterminism may reveal program errors. Similar ideas are described in [NeDa 94] and [Netz 96], who characterize feasible nondeterministic events as well as artifacts occurring due to nondeterminism, and propose to detect the set of artifacts and simply ignore them. Although these viewpoints may contain a critical pitfall as described above, we found only Oberhuber [Ober 98] to cover variations in the communication/synchronization pattern of concurrently executing processes.

4.2.5 Race Conditions

As described above, there are different sources for nondeterminism to occur. In principle, the basic requirement is to allow two or more activities to progress in parallel [McHe 89]. More detailed analysis allows to identify the following main reasons [Ober 98]:

- External sources:
  - Interrupt service routines for handling requests from I/O peripherals
- Internal sources (defined by the program itself):
  - Return values from system calls like random number generators [NeMi 92b]
  - (Internal) communication or synchronization functions

The main distinction between external sources and internal sources identifies those items that can be controlled by the program and those that are completely independent from the user's code. In principle, external sources should be handled completely by the operating system, and should not interfere with the program in any case. Although these reasons may introduce very problematic situations, we may assume that their exceptions are covered by the operating system. Thus, we will not discuss them in more detail. (Please remember, that operating systems are also software and many critical errors are included in them, too - see Section 3.5)

In addition, external sources like interrupt service routines and return values from the random number generator, do exist in sequential as well as in parallel programs. Thus, a parallel debugger must handle each of these cases, but there are solutions for covering internal sources in sequential tools. (In Section 8.1.3 we will briefly discuss the integration of functionality for random number generators in our approach).

The remaining item - (internal) communication and synchronization functions - only exists in parallel programs, which contain concurrently executing and interacting processes. The question is, whether or not some kind of ordering is enforced onto the process communication and synchronization. If the interaction between the processes of one program is strictly ordered for all possible inputs, the interaction pattern will be the same for each execution. In that case, communication and synchronization does not introduce nondeterminism in the program's behavior. On contrary, if the parallel environment permits different orders of process interaction, the results of the program may possibly be influenced by the order that occurs during program execution. Such a place in the code, that allows different orders during program execution, is called a "race condition". A possible formal definition to race conditions is offered by Helmbold and McDowell:

Definition 4-5: Race conditions [HeMc 96]

> Fix an input to the program. If two (or more) conflicting events are unordered (with respect to the same input) then there is a race between the two events on the input.

In order to discuss this term in more detail, we have to take a closer look at the kind of interaction, that takes place during program execution. Again we have to distinguish shared variable and message-passing as described in Section 2.3.4.

If communication and synchronization are applied with the shared variable paradigm, Definition 4-5 describes two or more independent events accessing the same shared resource in a conflicting way. Furthermore, at least one of them modifies the resource and their execution

order depends on the scheduling of the corresponding processes. Thus, for shared-variable programs we may refine Definition 4-5 as follows:

Definition 4-6: Shared-variable race conditions

> Whenever two or more processes are allowed simultaneous access to a shared variable, a race condition may arise in which the value of the shared variable can become indeterminate. Thus, a race condition is the process that leads to an indeterminate value [Lewi 93].

These kinds of races are called “data races”, and have been discussed extensively in literature. Some of the work on shared variables includes [AlPa 87], [MiCh 88], [DiSc 90], [Helm 90], [Hood 90], [ChMi 91], [NeMi 91], [Adve 91], [Emra 92], and [JuMc 00]. Recently, [RiLa 98] described how to apply protocol-based race detection to DSM (Distributed Shared-Memory) architectures. Furthermore, races in shared memory programs can be identified on different levels of synchronization, from mutual exclusion to weaker synchronization such as Post/Wait. Of course, the more low level the synchronization, they more difficult is it to detect the races [RiLa 98].

In the scope of their work, Helmbold and McDowell also identified four disjoint kinds of races depending on the ordering relations that may be observed during execution: concurrent races, general races, unordered races, and omission races. For a detailed discussion about these shared variable races, please take a look at [HeMc 96]. In addition, Netzer and Miller [NeMi 92a] recognized three different types of data races: actual, apparent, and feasible. Actual data races occur when two conflicting events execute concurrently in a given parallel execution. Apparent data races capture the notion that, although a pair of conflicting events might not have executed concurrently they could have, given a program’s synchronization. Thus, apparent races may contain data races that are prevented from occurring by the program’s semantics. On contrary, feasible races are permitted by both a program’s synchronization and its control and data dependences.

Similarly to the shared variable paradigm, race conditions may also occur in programs based on the message-passing model, although there exists no shared memory. In this case, process interaction takes place via messages, and race conditions emerge only at communication and synchronization statements. The resulting race conditions are expressed as racing messages.

Definition 4-7: Message-passing race conditions - message races [Netz 96]

> A race condition in message-passing programs occurs, when a receive operation is pending, while two or more messages may be simultaneously in transit to arrive at this particular receive operation. Therefore, messages race with each other when their order of arrival at a process is not guaranteed and may be affected nondeterministically (by system characteristics as defined in 4.2.2).

In contrast to data races, only few research papers exist for message races. In addition, the results obtained for shared variable programs cannot be applied directly to message-passing, although some ideas of the former may have inspired the latter [Netz 96]. Since we are mainly interested in message-passing programs, we will describe some of the related work in more detail.

One well-known example is the approach by Netzer and Miller [NeMi 92b], who characterized racing messages and offered an on-the-fly detection algorithm (compare also with [AlVe 93]). Yet, their technique only indirectly detects the existence of racing messages, but does not uncover all racing messages in the program. On contrary, the approach by Damodaran-Kamal and Francioni actually reports message races on-the-fly using a technique called controlled execution for testing and debugging [DaFr 94]. They use receive semantics in one process at a time and try to distinguish between intended and unintended races. Intended races are those that have been deliberately introduced by a programmer, while unintended races occur without prior knowledge and usually represent the actual manifestation of a bug.

Based on these preliminary approaches, Netzer and Damodaran-Kamal joined forces and combined their ideas on message race detection. Their first combined results about message races have been published in 1994 [NeDa 94], which have been revised together with Brennan in 1996 [Netz 96]. One of their addressed problems is diagnostic accuracy, because they believe that overwhelming the programmer with too much information should be avoided (see "maze effect", Section 4.2.2). This opinion is derived from the fact, that message passing programs can exhibit thousands of races, and sifting through race reports this large is too difficult. As a consequence, they distinguish artifact and nonartifact races.

The critical races are the nonartifact races, which represent those places in the code, where unintended nondeterminism is actually being introduced. On contrary, artifact races are only side effects of nonartifact races or nondeterminism, respectively. Thus, a nonartifact race can cause many subsequent artifact races, and all the bug hunting support of a debugging tool must lead to the nonartifact races. As soon as a nonartifact race is fixed, the artifacts will disappear [Netz 96]. This also means, that there is no need to investigate artifacts, and they should be ignored during debugging. Instead, only races that can be guaranteed nonartifact should be reported [Netz 96].

While all the above approaches to message race detection apply runtime techniques, the approach of Tai presents post-mortem algorithms to detect all message races for certain message-passing implementations, such as fully asynchronous, synchronous, FIFO, and causal [Tai 95]. In contrast to Netzer and Damodaran-Kamal, his goal is to find all the races in a program, whether or not they are artifacts.

For Cypher and Leu [CyLe 93], race condition detection is just a side effect of modelling the properties of message passing programs. Their primary intention was not to present a race detection algorithm, but to discuss different types of message-passing semantics and prove results about each.

Similarly, Oberhuber [Ober 98] modelled the properties of message-passing programs, but his focus was again on nondeterministic behavior and thus on racing messages. His main contribution is a set of formalisms that have not been described in such detail before. He provides detailed analysis of the reasons for racing messages and an approach to control the execution of such nondeterministic programs[Ober 95].

4.2.6 The Irreproducibility Effect

After introducing the characteristics of nondeterministic programs, we will now return to look at the remaining communication and synchronization errors: irreproducibility effects, completeness problems, and probe effects.

The irreproducibility effect [SnHo 88], sometimes called non-repeatability [NeMi 92b], is the most obvious problem occurring when testing and debugging nondeterministic programs. Since these programs may yield different results (again, final results, intermediate results, or

interaction patterns, see Section 4.2.4) in successive program runs although the same input is provided, it cannot be guaranteed that subsequent executions of the program exhibit the same behavior. Yet, this is a basic requirement for the debugging cycle.

The consequence of the irreproducibility effect are twofold. Firstly, program executions observed during testing may not be reproducible during debugging. Thus, even if an error is detected during the testing cycle, it may be difficult or even impossible to reveal the original cause during the debugging cycle. Secondly, the error observed during testing may be replaced by a completely different error during (cyclic) debugging. Although it is certainly important to remove both errors, recognizing the dissimilarity between the faulty behavior (and also between the original sources) may be confusing. Therefore, a debugging tool for nondeterministic parallel programs must offer a way to reproduce arbitrary test cases.

A solution for these problems is provided by “record&replay” mechanisms, that provides any number of equivalent executions based on some previously observed program execution. According to Leu et al, an equivalent execution can be defined as follows [Leu 90]:

Definition 4-8: Equivalent execution [Leu 90]

> Two executions of a process *p* are considered to be equivalent, if the process *p* receives the same information from the other processes at the same instants. The instant of an arbitrary event is defined by the interval in which only this event takes place. This means that two executions of a parallel program will be considered to be equivalent if the execution of each of its processes is equivalent

In order to achieve such an equivalent execution, all the elements that influence a process have to be determined. These are the input data provided by the user and the items identified as reasons for race conditions in Section 4.2.5. Since equivalent input is a basic requirement for the debugging cycle, any method offering equivalent execution must only reproduce the return values from system calls and equivalent communication and synchronization functions.

The solution of record&replay methods is based on a two-step approach. During an initial record phase, the order of critical elements is stored in tracefiles, so that the tracefiles contain sufficient data to describe all the choices taken at nondeterministic events in the code. Afterwards, if the program has to be re-executed and an execution equivalent to the initial record phase is needed, these tracedata are used as a constraint for the program. This is performed by the replay tool, that forces the target program to take the same choice at any nondeterministic point as has been taken during the record phase. If the order of all elements is the same for every program run, their behavior is equivalent and the same errors and results are observed. Thus, record&replay methods allow nondeterministic programs to be tested and optionally re-executed according to the users needs.

The history of record&replay techniques - sometimes called deterministic execution [CaTa 91], trace-driven simulation [Fren 95], or Instant Replay [LeMe 87] - dates back to an approach by Curtis and Wittie [CuWi 82], who proposed a technique to record the contents of each message as it is received by the corresponding process. Similar approaches are described in [Smit 84] and [LeRo 85] with the advantage that selected processes could be replayed in isolation. Yet, the biggest drawback of these approaches was the requirement of significant monitor and storage overhead. Thus, this data-driven approach to record&replay techniques was dropped for control-driven approaches. The latter only preserve the order of occurring operations, and generate the transferred data again during each replay execution.

Probably the first control-driven approach of record&replay was Instant Replay by LeBlanc and Mellor-Crummey [LeMe 87]. Their idea was to trace the relative order of significant events (usually communication events occurring at send and receive function calls) instead of the data associated with these events. Since the contents of all process interactions do not need to be stored, this approach requires less time and space to save the information needed for replay. In addition, Instant Replay does not depend on a particular form of interprocess communication. Indeed it was originally intended to be applied for debugging parallel programs on tightly coupled multiprocessers, where processes communicate via shared variables and more frequently than in message-passing systems [LeMe 87].

Several other record&replay methods have been implemented as an extension of Instant Replay. In fact, most existing prototype tools that allow analysis of nondeterministic parallel programs are based on this approach. Some example implementations of record&replay tools are IVD [Mack 93], Panorama [MaBe 93], PDT [Clem 95b], Buster [Xion 96], and DDB [SiRa 97].

In [Mack 93], a program replay mechanism for the message passing library PVM is described, that has been developed and integrated in the Interactive Visualization Debugger IVD [Hao 93]. Another integrated parallel debugger for PVM programs that includes record&replay capabilities is Buster, which is described in [Xion 96]. On contrary, the Panorama debugger has been implemented for the Intel iPSC/860 and the nCUBE architectures [MaBe 93], and supports both on-line and post-mortem debugging. In this case, on-line means that the debugger is running without any constraints, while during post-mortem re-execution the program's behavior is controlled by a simple record&replay facility.

The Parallel Debugging Tool PDT is announced as an interactive, source-level debugger for distributed-memory parallel programs within the Annai programming environment [Clem 95a]. It offers debugging support for high-level data-parallel programs based on High Performance Fortran (HPF) as well as message-passing programs based on the MPI standard [Clem 95b]. For MPI programs, PDT also includes a record&replay mechanism, which represents an implementation of the approach described in [NeMi 92b] (see below) with an extension for asynchronous probe primitives. The main problem of this approach is its high amount of monitor overhead, which is generated by maintaining extensive timing information during the record phase.

Another implementation of Instant Replay is included in the Distributed DeBugger DDB [SiRa 97], which targets at the multihreaded Mach distributed operating system developed at Carnegie Mellon University. One of the key goals of Mach is to support a distributed system on top of heterogeneous hardware platforms. This is only one of the characteristics addressed by DDB, which supports the minimal requirements for debugging distributed programs as identified by its authors. In their paper [SiRa 97], they authors also cover 29 other distributed debuggers and classifies them according to the following criteria: What kind of replay is provided? What types of breakpoints are supported? What effects can be controlled with breakpoints? Which event types are observed by the system?

Of special interest is the proposal of Netzer and Miller, who developed an optimal tracing technique which tries to minimize the amount of necessary trace data, both for message passing systems [NeMi 92b] and shared variable programs [Netz 93]. Their idea is based on the fact, that only those events affecting the race conditions do have to be traced. By making such run-time decisions, only a fraction of the total number of events (as required with Instant Replay) have to be stored in the tracefiles.

Although this on-the-fly trace reduction approach of Netzer and Miller is optimal in the common case and offers a good performance in the worst case, it could be further improved for

space and time efficiency using certain characteristics of the underlying system. Such improvements for time efficiency have been described in [FaCh 95] and [FaCh 96], while improvements for space efficiency can be found in [LeSc 92] and [LeSc 93]. Several combined efforts for shared variable programs have been proposed by Audenaert, Levrouw, and Ronsse in [AuLe 95], [Levr 94], [RoLe 95], and [RoLe 96]. Additionally, they show how to apply very simple compression algorithms to traces in order to further minimize tracefile size [Rons 95]. In cooperation with Ronsse we developed a technique for message-passing parallel programs that produces even smaller traces and is less intrusive than the others described above (for details, please refer to [RoKr 98]). In Section 9.2.4, we will show that this approach can be further improved by exploiting certain characteristics of most existing message passing libraries. An approach that has been influenced by this work and is therefore comparable to ours is MPL*, which described by Chassin et al in [Chas 99].

From the approaches that are not based on Instant Replay, some worth-mentioning are Deterministic Execution [TaCa 91], Demand-Driven Replay [MiTa 95] and On-the-Fly Replay [Gers 94]. In Deterministic Execution, a sequence of synchronization constructs is generated based on an initial execution of the program. By inserting these synchronization constructs into the replayed execution, the irreproducibility effect can be eliminated [CaTa 91]. The trace of the initial execution is also required for Demand-Driven Replay, which replays a program between breakpoints by allowing only those interprocess activities, that are needed to reach the next breakpoint [MiTa 95]. The only technique that does not require a dedicated record phase is On-the-Fly Replay, because record and replay phase are executed simultaneously. The replay phase is steered by the record phase by exchanging ordering information between both executions [Gers 94].

4.2.7 The Completeness Problem

A related crux observed during testing nondeterministic parallel programs is the completeness problem, which is a kind of observability problem[5]. In this context, observability characterizes the amount of information that must be retrieved from a program code or specification in order to correctly specify, select, and evaluate test cases [KrWi 98]. Yet, obtaining all possible executions of nondeterministic parallel programs is difficult or even impossible. However, complete testing of such programs requires to consider every possible execution. In principle, the following items have to be addressed:

- Several valid results may exist for one set of inputs.
- Critical errors may occur only sporadically.

These problems may occur in nondeterministic programs, because different results may be computed for one set of inputs. Furthermore, different execution paths of the program may still compute the same results. Thus, the observed results may not necessarily represent the only valid executions of the program [DaFr 94]. For example, in programs modeling nondeterministic scenarios from reality, several valid executions may exist and may be of interest to the

5. Please note, that the term “observability problem” is used for different phenomena. In [KrWi 98] a test case is observable if it represents a unique execution history, while [Fidg 93] describes the observability problem in the context of distributed clocks and clock synchronization, where it is often difficult to determine a consistent global state of a parallel program. The latter represents the more conventional point of view.

user. In addition, some errors may be hidden in executions that may seldom or never occur during testing. Yet, since environmental parameters may affect the program unpredictably, every possible execution of a program must theoretically be considered during testing.

Formally, this means that the testing approach of Section 4.1.2 has to be extended for nondeterministic parallel programs. Again, we apply the simple checking approach of Blum and Wasserman [BlWa 96] (see Definition 4-1). After choosing an arbitrary mathematical function $f(x)$ that describes the behavior of some program $P(x)$, and a valid input x, we have to determine whether or not $y = f(x)$ in order to confirm if the input/output pair $\langle x, y \rangle$ represents a correct computation of f. The problem is, that there may be more than one correct output y for each set of inputs $X = \{x_0, x_1, x_2, \ldots\}$. Thus, for satisfying the test completion criterion with nondeterministic programs it is necessary to compute each output y from a set of possible outputs $Y = \{y_0, y_1, y_2, \ldots\}$ for each input of the set of valid inputs in X. As a consequence, some means of steering the choices at nondeterministic events have to be provided. The goal is to obtain alternate versions of particular test cases during replay, based on some previously recorded program execution.

Although this problem is well-known in the parallel debugging community, there are only few solutions that address this completeness problem. In fact, most of these solutions try to overcome this problem by eliminating or controlling the nondeterminism occurring due to communicating processes. A simple approach would be serialization of the code, which means that no process is allowed to proceed concurrently to any other existing process. If the processes are executed in a pre-defined sequential order, race conditions simply cannot occur.

One example of this approach is “controlled execution” [TaCa 96], which also offers a method to perform automatic testing. Based on a technique for describing the desired communication behavior, the replay tool forces the program to execute the communication as specified. Therefore, the order of all communication events is predefined and races are not permitted. In addition, selected events of the communication description can be varied, allowing to perform restricted permutation testing. An extended approach to this technique is offered by Oberhuber [Ober 98], who proposes to specify not only input data for testing, but also the sequences of interprocess communication [Ober 95]. In this approach control patterns are applied for establishing the order between the communication events. Such control patterns are dynamic rules to define the order of process interaction. They can be grouped hierarchically to express more and more complex patterns, which should then sufficiently describe the expected communication structure of a parallel program. This eliminates nondeterminism and guarantees reproducibility, if the predefined behavior is enforced by the debugging tool during execution and re-execution [Ober 95].

There are several problems to all these proposals. Firstly, the user is confronted with an additional means of specifying the program’s communication behavior. It may be difficult to demand this from the average user, who already described the program’s behavior by specifying the communication statements in the original source code. Not only does it require to learn a new description language, but also a correct relation between the program code and the communication specification. Secondly, the correctness and completeness of the communication specification must be guaranteed in order to mask all possible executions for all possible inputs. This may be possible in some cases, but is certainly limited in case of highly complex and large programs, which may exhibit totally unexpected behavior. Thirdly, these solutions do not exhaustively test the program’s execution, but instead pretend to test the most feasible cases. Therefore, the user may be mislead and trust in a program, that is actually not completely tested and may exhibit incorrect behavior whenever executed without this controlling mechanisms.

In contrast to the solutions described so far, our proposal addresses the completeness problem more directly. The first description of our idea is presented by Grabner and Volkert in [GrVo 93], who nicknamed this approach “event manipulation”. An improved version of their basic idea together with a corresponding replay mechanism can be found in [Kran 95a] and [GrVo 96]. Our basic idea was also extended in [Kran 97b], where we show that a combination of race condition detection with control and data flow analysis can provide vital information for selecting appropriate race condition candidates. Recently, the original manual approach has been automated as described in [KrVo 98]. This latest instance of automatic event manipulation will be described in more detailed in Chapter 9. Therefore, the following paragraph gives only a brief summary of our proposed approach.

The main idea of our approach can be formulated as follows: Firstly, an initial execution of the target program is observed during a record phase. The observed data is analyzed and encountered race conditions are identified. Based on these race conditions, the following question is explored: What would have happened, if the order of the communication events at some arbitrary race condition would be different from the observed order? The answer to this question for any selected race condition in the observed execution is derived with two steps:

- Event manipulation
- Artificial replay

Event manipulation means, that the observed order of communication events is changed. In the original approach, this change is applied graphically in a display that represents the program’s communication behavior. After the exchange has been performed, a replay of the program is initiated, that enforces the artificially generated new behavior. This re-execution has to perform two distinct tasks to solve the question stated above: Up to the point of the event manipulation, the program is replayed exactly as observed during the record phase. After the event manipulation has been performed, the program is executed without any constraints. This is necessary, because the future of the program after this exchange is unknown due to the nondeterminism and cannot be derived from the behavior stored during the record phase.

With this event manipulation and replay approach, we can easily investigate different program executions for one particular set of inputs. Furthermore, all possible permutations of the observed program execution can be derived, if all combinations of event orders at each of the racing receives are tested. For the original approach described in [GrVo 93] and [GrVo 96], this would mean that each possible event order has to be generated manually by the user. Since this could represent a major drawback for many parallel programs, we revised this manual approach to perform the event manipulation and artificial replay automatically [KrVo 98]. In this case, an initial execution is analyzed by a testing tool, that identifies all the race conditions in the program and generates the complete set of follow-up executions automatically without user interaction. Then, the resulting executions and their corresponding outputs represent all possible executions of the target program for the selected input.

The few comparable approaches for solving the completeness problem are described with the Hindsight prototype debugger in [Spie 93], with the Time Controlled Environment TCE in [Kraw 00], and with the macrostep approach in [Kacs 98]. Hindsight targets ordering errors occurring due to missing synchronization operations by exchanging the order in which synchronization operations are carried out [Spie 93]. The TCE (together with the Race Detection Testing Strategies RDS) tries to modify a program run during subsequent execution by introducing artificial delays or changes to the process allocation scheme [Kraw 00].

In contrast to these approaches, macrostep debugging as proposed by Kacsuk in [Kacs 98] can be used to perform complete testing. The main idea is, that the user describes the minimal relevant test sequence for a given problem, and the tool automatically derives each possible execution for this test sequence. This is achieved by exhaustively traversing the execution tree of a program, where the debugger takes a different choice at each race condition during subsequent re-executions of the program [Kacs 99]. The only limitation of macrostep debugging is, that it is based on the GRAPNEL visual programming language [Kacs 97]. Although this language uses PVM for low-level communication routines, the visual objects applied in GRAPNEL are much more restrictive than what would be possible with the underlying message passing library. While this is certainly an advantage for the inexperienced user, it also reduces the number of possibilities at race conditions vigorously.

In practice, the number of possible combinations would probably prohibit exhaustive testing [DaFr 94]. One important reason is, that these possible combinations and the corresponding replay phases may deliver some totally new executions with potentially new race conditions, which of course have to be treated by the testing program in the same way as the original execution. In certain cases it may even be imaginable, that the program delivers new executions endlessly, which prohibits to perform the complete tests of the program. An example is a program with a loop, that requires a specific ordering of events for loop termination.

Therefore, the debugging tool has to provide certain precautions in order to limit the testing cycle to the most probable executions (compare with Definition 3-1: Good Enough testing [Bach 98]). Some of our solutions to this problem are described in detail in Section 9.3. Another solution based on our basic event manipulation approach can be found in [KiCh 97], which is intended to detect occasional bugs (see Section 3.2.6) that occur only sporadically due to racing messages. Their idea is to re-execute the program while imposing a strictly different message ordering than observed. By maximizing the number of reordered pairs of message deliveries, errors that would otherwise occur only sporadically should materialize.

4.2.8 The Probe Effect

Another problem that occurs during debugging nondeterministic parallel programs is the Heisenberg (Uncertainty) Principle applied to software ([LePa 85], [Rose 96]), the so-called “Probe effect” ([McHe 89], [Ober 98]). The symptoms observed due to this effect are called “Heisenbugs” as described in [LePa 85] and [Rons 00]. This problem has already been described for the sequential case in Section 4.2.1, and is even more critical for parallel programs, especially in the nondeterministic case. In 1986, Gait formulated this problem as follows:

Definition 4-9: Probe effect [Gait 86]

> The probe effect is an alteration in the frequency of run-time computational errors observed when delays are introduced into concurrent programs.

This means that the behavior of a program is influenced when something slows down the execution of one or more processes. Obviously, adding a debugging tool represents a certain amount of delay, and that delay may vigorously affect the debuggee’s execution. The critical issue is the size of the observation functions which are added to the code during the instrumentation phase. This size determines the monitor overhead, which is on the one hand the time

spent in the instrumented code, and on the other hand the amount of memory needed by the observation functions.

Therefore, a primary goal of any debugging tool must be to generate very little overhead. Yet, even the smallest overhead consumes some amount of memory and there will always be an execution delay of the target program, which modifies the time of occurrence of events. Thus, the data that is generated by the observer describes events that occurred later in time, compared to an execution where debugging is turned off. An additional problem to the displacement of observed events is introduced, if the target program reveals nondeterministic behavior. In that case, the order of events at race conditions may be affected, leading to a completely different execution that would have been observed without the debugging tool. In fact, any attempt to debug a program may change that behavior so much, that the collected data will not be adequate [KrWi 98]. Furthermore, new errors may occur while others may be hidden in program branches not taken during the observed execution, which would otherwise have occurred if the program had been executed without the debugger. In more pathological cases the observed application may be prevented from exhibiting any sensible behavior at all.

These problems have been extensively addressed by Malony, Reed, and Wijshoff in [Malo 91a], [MaRe 91], and [Malo 92]. They distinguish between direct intrusion and indirect intrusion, the latter being very difficult to estimate and correct. In their model, direct intrusion is corrected post-mortem by estimating the amount of tracing and subtracting this overhead from the measured event data. This approach works well for programming models with deterministic communication (e.g. fork-join in [MaRe 91], and rendezvous in [Mail 95]). However, it is insufficient for nondeterministic behavior of asynchronous communication as offered by the standard message passing interface MPI, because the latter would require post-mortem changes in the order of traced events. As a consequence, Maillet introduces conservative approximation as described in [Mail 95], which applies only corrections that do not interfere with the traced event order. Another idea for nondeterministic programs was described by Leu and Schiper [LeSc 92], and later by Teodorescu and Chassin de Kergommeaux [TeCh 97]. Both ideas focus on tracing only the minimum amount of information required to allow re-execution of programs. Thus, this initial execution is assumed to be only slightly perturbed and is therefore considered as an approximation of a unperturbed execution [FaCh 96]. Afterwards the replayed executions are used to apply performance tracing and generate additional timing information.

Other contributions in this domain have been described in [Will 92], [SaMa 93], [Gann 94a], [Gann 94b] and [Jaki 95]. In contrast to all these correction algorithms, there also exists a group of monitor overhead prevention algorithms. One example is the “logical clock approach” of Cai and Turner, which uses a virtual time to reflect the real time execution of each process when running without monitoring [CaTu 94]. During execution, the interprocess communication is controlled by this virtual time rather than the real time in order to avoid the probe effect [TuCa 93]. A similar approach is described by Zhang et al (see [LiZh 95] and [Zhan 98]), who correct the monitor intrusion with artificial delays, so that the relative speed of all the processes remains the same. Similar ideas are proposed in [Wu 96] and [Wu 98], where different communication protocols are used to distinguish the monitoring activities from other computational tasks in order to allow automatic removal of monitor intrusion. In [HoMi 96], Hollingsworth an Miller describe a data collection cost system, that provides users with feedback about the impact of the data collection on the running program. In addition, their system allows to define the level of perturbation that affects the program’s execution, which is then controlled by the monitor.

The problems of all these previous approaches to correct the probe effect are, that either they do not treat the possibility of actual changes in event order, or they pretend that changes in event order did not occur in the first place. Yet, even the smallest overhead may lead to such program perturbations and different program behavior. Thus, a complete overhead removal strategy must deal with both types of program perturbation, changes in event timing and event ordering. In [Kran 99b] and [Scha 99], we describe a possible solution to this problem that includes necessary clock synchronization algorithms, calculation of corrected event occurrence timings, and automatic event manipulation and replay at places where changes in event ordering are detected.

One of the problems of our approach is the accuracy of the correction algorithms, which prohibits to identify the one and only execution that would occur without the debugging tool attached to the code. This problems may be attributed partially to the achievable accuracy of clock synchronization algorithms, partially to the measurements of communication and overhead. In our case, we implemented a simple solutions based on Lamport's condition that sends must happen before receives, as well as the more sophisticated synchronization method proposed by Maillet et al in [MaTr 95] and [Mail 96]. For the overhead measurements, we applied the SKaMPI benchmark developed by Reussner et al [Reus 98], which provided detailed timings for the complete MPI standard functionality [Kran 99c].

Another problem is that in many cases there exists not only one possible execution due to the nondeterminism, but several executions with almost equal probability. In the worst case, every possible execution path may occur in reality, which connects the probe effect to the irreproducibility problem and the difficulties of exhaustive testing described above. As a consequence, our correction algorithm derives the set of possible executions with the highest probability, where each of them may most likely occur if debugging is turned off. This issue will be discussed in combination with automatic nondeterminism analysis in Section 9.3.

4.3 Extended Parallel Testing and Debugging Approach

The problems briefly described in the previous section emphasize the fact, that parallel debugging differs from sequential debugging in many aspects, and parallel debugging tools need to offer additional functionality for handling the increased complexity, the high amount of debugging data and these anomalous effects.

A possible approach to parallel program debugging is suggested by Geist et al in [Geis 94a], who identify the following three steps:

(1) Try to run the program sequentially as a single process and perform sequential debugging.
(2) Execute the program with two to four processes on one single computer and verify the correctness of the message-passing structure determined by message tags and destinations.
(3) Execute the program with the same two to four processes on more than one computers to detect possible problems introduced by network delays related to synchronization and timing.

These propositions are often referred to in parallel programming literature (e.g. in [WiAl 99]), and are widely accepted as a possible way to perform parallel program debugging. In fact,

most existing debugging solutions apply down-scaling as an inherent functionality, because these debuggers are not prepared to cope with really large programs. In this case, down-scaling means that the program's size is reduced in terms of participating processes and problem size or executed numbers of iterations, respectively. Yet, while this suggested down-scaling of the program size may hopefully be sufficient in many cases, it may be inappropriate in other cases due to the reasons described before. In principle, some errors may occur only with many processes or on actual amounts of data, and it is therefore unavoidable that suitable testing and debugging must also consider the final production scale of the program and its associated data.

For that reason, we propose a different debugging strategy, which has been developed based on the flow-diagram of Figure 4-1. The basic idea was to integrate the functionality required to treat the obstacles described in Section 4.2. Firstly, it is necessary to obtain the order of events already during the testing cycle. Otherwise, the debugger would suffer from the irreproducibility effect and it would be impossible to guarantee debugging of the same execution that was responsible for the incorrect behavior. Therefore, some instrumentation has to be performed before testing is initiated.

In addition, the probe effect should be avoided as much as possible, which means that the amount of observation overhead should be as small as possible. The smaller this overhead, the lower the probability of program perturbations. This fact can be endorsed, because the functionality required for obtaining event ordering is far less than necessary for debugging, which means, that testing requires only reduced instrumentation. Additionally to adding instrumentation for the record phase to the testing cycle, some replay functionality has to be added to the debugging phase. Only with this replay functionality the same event order as obtained during testing can be enforced during debugging.

Another extension to the diagrams of Figure 4-1 concerns the completeness problem and its solution, event manipulation and artificial replay. The necessary changes have to be made in the testing cycle and before the debugging cycle. This time, another iterative cycle is added after the input has been selected. Then, each possible program execution for that selected input can be tested.

The resulting testing and debugging cycles for nondeterministic parallel programs are sketched in Figure 4-4. This revised strategy consists of the following three cycles:

- Input testing cycle
- Nondeterminism testing cycle
- Debugging cycle

The left-most flow-diagram represents the input testing cycle, which is used to iteratively test all possible inputs $X = \{x_0, x_1, x_2, \ldots\}$ for the program. As mentioned above, the only difference to the traditional testing cycle is the additional instrumentation phase that is needed for adding the observation code of the record mechanism. Again, the end-marker of the flow-diagram has been omitted deliberately, because of the possible high number of possible inputs that have to be tested, and the difficulties to determine the correct test completion criterion (see Section 4.2.7).

The central cycle of Figure 4-4 is needed to overcome the completeness problem. Its intention is to evaluate the program's behavior and to compute each output y from a set of possible outputs $Y = \{y_0, y_1, y_2, \ldots\}$ for the input x selected in the input testing cycle. For that reason, the program's execution is observed and the results of nondeterministic events (e.g. the order of incoming messages at race conditions) is recorded. Afterwards, the termination of the program and its results are inspected similar to the testing cycle of Figure 4-1. Again, the detec-

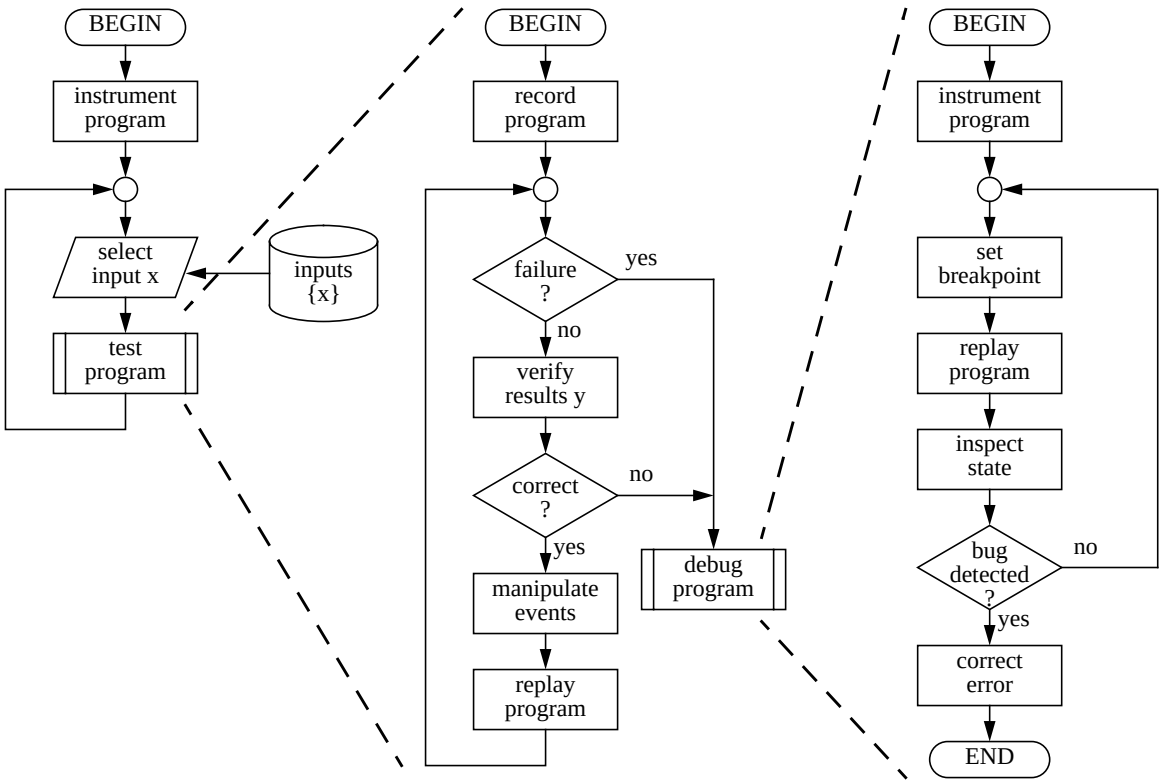


Figure 4-4. Testing and debugging cycle for nondeterministic parallel programs

tion of a program failure or incorrect results leads to the initiation of the debugging cycle. If no erroneous behavior is encountered, event manipulation and artificial replay are used to evaluate other program executions for the same selected input. This is iteratively repeated as long as further combinations of racing events can be derived. Since the number of necessary iterations is also unpredictable, the end-marker has been omitted again.

The remaining right-most cycle of Figure 4-4 is the debugging cycle, which almost resembles its equivalent from Figure 4-1. The main difference is, that this time the program is re-executed under the control of the replay mechanism, because of the irreproducibility effect. Furthermore, another instrumentation phase is needed that differs from the input testing cycle, because the debugger requires much more information than the nondeterminism testing cycle. Please note, that this increase of observation overhead does not introduce any perturbation of the event ordering, because the event order is determined by the record phase of the nondeterminism cycle and is therefore fixed during debugging. Thus, the only effects of the observation overhead are experienced for the event timings, which are delayed depending on the amount of instrumentation code. However, since we are mainly interested in the programs correctness, this increase in timing represents only a minor drawback and is much more important during performance analysis.

4.4 Requirements on a Parallel Debugging Tool

4.4.1 Architecture and Functionality of an Arbitrary Debugger

The debugging strategy described in the previous section is one possible approach offered for parallel programming. Indeed, many investigations and research projects have already targeted error detection in parallel programs, and some of the aspects described above have actually been implemented in existing tools and tool prototypes. Yet, a complete debugging tool should be valid for any scale of parallel program, from small-scale applications to massively parallel programs. Especially the latter have not been covered sufficiently, although it is accepted that some means of testing and debugging on the final scale of the applications are indispensable. In the following subsections, we will try to derive the requirements on a parallel debugging tool, that operates with the debugging strategy of Figure 4-4, and provides support for large-scale parallel programs.

Starting with the architecture, we will again take a glimpse on the sequential counterparts. In this context, Rosenberg defines the architecture of a debugger to consist of [Rose 96]:

- A (graphical) user interface with features for
  - Displaying source code
  - Inspecting (and manipulating) the processes stack, the hardware registers, and the applications variables.
  - Handling breakpoints
  - Controlling the execution of the target program
- A debugging kernel with functionality for
  - Managing the program's symbol table
  - Evaluating expressions
  - Controlling and executing processes
- An operating system for accessing
  - Operating system functions
  - The debuggee

A similar architecture is proposed by Bemmerl, who formally defines the key features of an arbitrary debugging tool as follows:

Definition 4-10: Functionality of a generic debugging system [Bemm 92]

Any tool for error detection, a so-called debugging system, $DS = (PD,DO)$ is defined by

- PD = arbitrary connections of control flow-data flow, and concurrency events with operators AND, OR, and a suitable ordering relation, as well as
- DO = breakpoints $\cup$ trace $\cup$ state-inspection $\cup$ state-manipulation

4.4.2 Criteria for Developing a Parallel Debugger

The functionality and architecture defined above is certainly a good starting point for the development of a parallel debugging tool. In addition, the following two items have to be investigated by developers of parallel debuggers:

- Heterogeneity and portability: supporting different programming models and hardware architectures (possibly in the same environment)
- Usability: handling the characteristics of parallel programs like
  - Increased complexity
  - Amount of debugging data
  - Additional anomalous effects

One of the most important criteria for any tool developer is a definition of target systems, which in our case are high performance supercomputers. Since there exist different architectures and possibilities of programming them (as described in Section 2.2 and Section 2.3), a wide variety of strategies and tools with different functionality have already been proposed, leading to some nice tool prototypes or even commercial products. Furthermore, every hardware vendor provides its own native support for debugging on their respective architectures, like for example nCUBE with the ndb Debugger, CONVEX with its CXdb, or Intel with the Paragon Debugger ipd on its iPSC/860 architectures.

This diversity is an issue of some on-going projects in this domain, which address the problems of portability and heterogeneity in parallel debuggers and performance analysis tools [Panc 95]. Corresponding solutions are usually based on the fact, that all vendor-specific tools provide similar functionality under a different user interface. Consequently, it seems useful to develop portable debuggers which could be used on different hardware platforms, thus dismissing the need to learn several debugging tools when switching between different platforms.

Some examples in this context are P2D2 [Hood 96], Panorama [MaBe 93], PDBG [Cunh 98b], PDT [Clem 95b], and DCBD [Wang 99], as well as the best-known commercial debugging tool Totalview [Dolp 98]. These tools offer debugging support on several multiprocessor architectures and networks of workstations. An important characteristic of almost all these systems is, that every approach applies several instances of an existing sequential debugger in order to perform the debugging task on the participating processes. Thus, their idea is to offer a graphical interface for managing the features of these “base debuggers” [MaBe 93], which are in most cases variants of the GNU debugger gdb [Stal 93]. In these cases, the parallel debuggers adopt a client-server architecture to provide a uniform interface for different platforms, communication libraries and programming models [ChHo 94].

Although this portability may be useful in many cases, it introduces some obstacles especially on large scale computing systems like massively parallel machines or heterogeneous clusters of supercomputers, the so-called metacomputers. In fact, one of the biggest problems experienced during debugging of supercomputer applications is connected to the amount of data that has to be analyzed [Frum 98]. Firstly, these programs tend to be long-lasting, from some hours to several days or even more. As a consequence, many state changes occur that have to be observed, transferred, stored, and processed by the debugging tool. In the worst case, debugging a supercomputer may require another supercomputer to perform the analysis task. Secondly, the execution time and the large number of participating processes leads to enormous interprocess relations, which cannot be comprehended by the user. Thirdly, a great

amount of debugging activities have to be performed equally for different processes and repeated iterations of the target application.

Thus, the basic requirements heterogeneity and usability have to be supported by methods for managing parallel programs in order to increase the lucidity and applicability. For that reason, we identified the following optional features:

- Abstraction
- Automatization

Abstraction means, that the amount of debugging data presented to the user is reduced to a suitable minimum. Besides the regular meaning of forming some kind of mental generalization, this also includes related techniques like transformation, reduction, extraction, isolation, and filtering. Error detection can only be applied successfully, if the user can comprehend the observed activities [Kime 95]. This means, that a debugging tool should offer capabilities allowing the programmer to focus the scope of attention to limited regions within the program [BeSt 94]. Furthermore, it may be equally important to permit different levels of abstraction in order to select the most appropriate level for the conducted debugging operations, so that only the data relevant to the user is presented [From 95c]. Sometimes it may be useful to see the execution of the program in a coarse graphical overview, while at other times it may be beneficial to display the program on the level of machine instructions. Thus, a most important feature is to provide connections for dynamically switching between different levels of abstraction. It's the nature of debugging, that one doesn't know just what to look for in advance, so it is certainly important to provide access to the vast amount of debugging data while at the same time offering functionality for managing these data.

Besides abstraction, a lot of debugging efforts can be supported by automatization if repeated debugging activities are performed without user interaction. Some examples are the automatic detection of program failures, abstraction of detectable, for the user probably invisible properties, and iterative testing for the completeness problem. For example, after reading the core file, users can be directed to the place in the code where the program exited. Or, race conditions and corresponding memory access patterns or racing messages can be evaluated automatically. Some sophisticated methods consider statistical testing [Walt 95] and automatic checking of dynamic properties using regular grammars or finite automata [Baba 95].

Of course, due to the nature of debugging and its strong relation to the user's knowledge, a complete automatization of the debugging process is certainly impossible. For instance, the correction of a detected anomaly must always be carried out by the user. Therefore, the goal of powerful debugging tools must be to perform as many debugging activities automatically or semiautomatically with the main directions given by the user.

Based on these requirements, we may now try to classify existing work and identify the drawbacks of previous approaches. Only a strategy or tool supporting these characteristics may then be applicable to full-scale applications, and may allow to perform debugging activities impossible with existing solutions. In addition, it may also improve the error detection task on smaller-scale application sizes, especially if it can be combined with other tools in this area.

4.5 Parallel Program Analysis and Visualization

This section tries to offer a brief overview of related work in the area of parallel program analysis and visualization. It starts with an overview of existing collaborative, standardization approaches for parallel tools and tool developers. Afterwards, a model of a generic program analysis tool will be introduced, which is used to classify existing tools in the domain error detection and performance tuning. Such surveys are interesting for potential users as well as tool developers. At the same time, it is probably impossible to capture all existing techniques and approaches, or even to distinguish between pure theoretical achievements, actual tool prototypes, and tools available for real end users. Some overview papers with similar content in this domain are in chronological order: [McHe 89], [PaNe 93], [Gu 94], [Hond 95], [TsYa 95], [Fran 96], and [Brow 99].

4.5.1 Collaborative and Standardization Efforts to Debugging

In the following sections, we describe a lot of different approaches and strategies for debugging parallel programs, and a variety of tools and software environments that have been developed with the goal to satisfy the user's need during error detection and performance tuning. As a starting point, we will briefly introduce some standardization efforts with the goal of leading to substantial improvements for both, tool developers and their consumers. The most important examples of concentrated efforts are

- the OMIS project,
- the Ptools consortium,
- and the HPD forum.

The first attempt, which will also be mentioned in Section 4.5.3, is the Online Monitoring Interface Specification OMIS[6] [LuWi 97]. This effort was initiated in 1996 by Ludwig and Wismüller at the LRR (Lehrstuhl für Rechnertechnik und Rechnerorganisaton) of the Technical University Munich together with Sunderam from Emory University, Atlanta, one of the creators of the Parallel Virtual Machine PVM. Since then, the authors of OMIS were able to dissipate it to a broad community, and several compliant tools have already matured at various research institutions around the world. Proposals for extensions to the standard, are coordinated by the OMIS group, while software products being developed in the framework of this project will be released under the GNU public license conditions to deliver a maximum profit for the user community.

In contrast to the OMIS standard, the Parallel Tools Consortium Ptools[7], established in 1993, seeks to increase the cooperation between representatives from federal, industrial, and academic sectors in order to address the factors that inhibit tool usage and tool usability on parallel computers [Panc 96b]. Their self-imposed mission is to take a leadership role in defining, developing, and promoting parallel tools that meet the specific requirements of users who develop scalable applications on a variety of platforms. This is attempted with two activities: On the one hand, Ptools organizes meetings for members of the above mentioned organizations in order to provide a forum, where tool developers interact directly with potential users in order to help formalizing and prioritizing user requirements for tool support. On the other

6. see OMIS Webpage at http://wwwbode.informatik.tu-muenchen.de/~omis/
7. see Ptools Webpage at http://www.ptools.org/

hand, Ptools promotes tool portability and consistency by sponsoring projects to develop general tools that can execute reliably across a broad range of parallel and clustered computing systems. This is supported by the Ptools infrastructure, that includes channels for acquiring input from the high-performance computing community, working groups involving tool users, researchers and developers, and mechanisms for formalizing and distributing tool components for adoption by the industry. Although Ptools is operated from the United States, its membership is open to all organizations and persons interested in parallel tools without any membership fees.

One of the activities sponsored by the Ptools consortium is the High Performance Debugging Forum HPDF[8], which was officially established in 1997. Its goal is the definition of a useful and appropriate set of standards relevant to debugging tools for high-performance computing [Brow 97]. This base level standard has to meet the sometimes conflicting constraints of being useful to users, realistically implementable by developers, and architecturally independent across multiple platforms. In order to meet the criterion for timeliness, the HPD forum restricted itself to two years, where the first year should be used to develop a standard that could then be implemented during the second year. And in fact, Version 1 of the HPD standard was released in 1998. However, while the standard was developing fast and is in some parts still being discussed, actual implementations of its functionality are still not available by now. However, there are notably some organizations working on it, and it will be interesting to see, whether their efforts can lead to substantial benefits for the group of debugging users.

Another interesting approach with some nice potential, although actually not termed as a standard, is collaborative debugging [DoMu 97]. Its idea is to create distributed debugging communities around the globe based on the World Wide Web technology. These groups of programmers will then describe and fix bugs in collaboration, with probably high potential benefits for the quality of the resulting software. A first solution is the Internet Software Visualization Laboratory (ISVL), which uses a client/server architecture to deliver visualizations of program behavior on any Java-enable Web browser [DoMu 97]. Another approach is offered by Kansas, a multiuser virtual world, which allows people to experiment with building programs and collaboratively hunting bugs [Smit 97].

4.5.2 Models of Generic Program Analysis Tool

After identifying standardization and collaborative efforts to program debugging, we will now characterize tools for debugging and performance analysis. The latter are included in this section due to the many similarities of tools in these two areas. The reason is, that both analysis activities rely on program state information and some data are equally useful for error detection and tuning (see Section 7.3 for more details and suggestions about this similarity). In fact, some existing tools actually try to fulfill the users' requirements for combined debugging and performance analysis tools [AgCh 96].

Any program analysis tool always consists of two principal items, an observation component and an analysis component, often referred to as monitoring and visualization component, respectively [Hond 95]. An even more detailed model is offered by Mohr et al ([Mohr 92], [Klar 95]), who identify six distinct layers of an arbitrary analysis tool, which are hierarchically established on top of the analysis target. These layers are [Klar 95]:

8. see HPDF Webpage at http://www.ptools.org/hpdf

- Monitor oriented layer: the monitor
- Analysis oriented layers:
  - Data access layer
  - Selection layer
  - Tool support layer
  - Tool layer
- Application oriented layers: application support

The bottom layer of this model represents the connection between the analysis tools and the target system. It denotes the observation component in [Hond 95], and is responsible for the data collection. Its main tasks are to record the dynamic course of state-changes that constitute the execution of the actually observed program. Obviously, there is a strong connection between the observer and the observed program, as well as the underlying system.

The next four layers above the monitor oriented layer are called the analysis oriented layers. The data access layer provides access functionality for obtaining the observation data from the monitor layer. In order to reduce this exhaustive data and relate it to corresponding analysis tools, the selection layer provides functionality for filtering and compounding the data. In the tool support layer, this data can be accessed by generic functions without any semantics and utility functions like statistics or other operations. The actual semantic oriented analysis tasks are provided by the tool layer, which offers a wide variety of activities: trace validation, statistical evaluation, visualization of execution behavior, animation, as well as processing the data for other tools on the next layer, like modelling tools, debugging and performance analysis tools, and tools for load balancing. This top-most layer of the hierarchy, the application oriented layer, constitutes the interface to the user, and is therefore the connecting link between the analysis tool and its client.

In [Hond 95], the four analysis oriented layers and the application oriented layers are combined in the analysis or visualization component. Its most important property is the way of displaying the program to the user. In general, we may distinguish textual representation (e.g. source code) or graphical representation (with many different ways to visualize the monitored data). As mentioned before, the advantages of the latter can only be acquired if a combination of both approaches is included through some kind of source code reference.

Tools can also be divided by whether analysis and visualization takes place simultaneously with the program execution or after it, which establishes the following two groups:

- On-line analysis tools
- Post-mortem analysis tools

In the on-line case, the data of the monitor is transported to the analysis component, while the target program is still executing. In the post-mortem case, the program is executed and analysis data is stored in tracefiles, which are investigated only after the program's execution terminated. Please note, while post-mortem allows to remove the analysis components (the five upper layers in Mohr's approach [Mohr 92]) from the executing program, it still requires to include at least the monitoring component, because this component can only retrieve state information when the program is actually running.

Both approaches have different advantages, with on-line being the primary choice for standard error detection tools. The reason is, that on-line analysis tools allow a much more flexible approach to program state inspection than post-mortem, because the user can determine what to inspect at each point during the execution. On the other hand, the post-mortem approach

allows some analysis techniques, that can not be performed with on-line tools, primarily because on-line tools can only rely on the past and present of a program's execution, while post-mortem approach may analyze the program from its beginning to its termination. An example, where post-mortem approaches are superior to on-line approaches is race condition detection, because on-line methods may not be able to identify the complete set of race conditions at any point during the execution.

4.5.3 Monitoring Tools

As mentioned above, every tool must contain at least a small monitoring component to observe the program's execution and gather state-information for the follow-up analysis phases. Therefore, most monitoring tools are an integral part of the analysis tool, especially in the on-line case, and cannot be examined in isolation. In fact, the monitor part is often invisible to the user, and therefore indistinguishable as a stand-alone component. Examples for this statement are most traditional sequential debuggers and the parallel debuggers, that are composed from sequential debuggers. For instance, when using the GNU debugger gdb [Stal 93], which is often used as the lowest level of parallel debuggers like in P2D2 [Hood 96], PDBG [Cunh 98b], and PDT [Clem 95b], the activities of the monitor are completely integrated in the debugging environment and cannot be evaluated in isolation. A similar model is offered by one of the few commercial parallel debuggers Totalview [Dolp 98].

However, there are also some other approaches with explicit monitoring components. Based on their monitoring method, these approaches can be divided into the following three classes (see [GrVo 93], [Klar 93], and [Grab 99]):

- Hardware monitoring
- Hybrid monitoring
- Software monitoring

Hardware monitors represent an extension of the target system on the hardware architecture level. Thus, their usage is generally restricted to vendors or hardware experts, and requires a lot of expertise and knowledge during installation and operation. The biggest problem of these monitors is, that even the most advanced tools are restricted to a few observation points and provide only low-level data as well as a low-level interface. In addition, such monitors offer little portability because they address only one particular hardware architecture. Yet, their advantage is the very little overhead that is produced during program observations, which denotes a lot of benefits in terms of the probe effect.

One of the few widely-known hardware monitors is ZM4 (german: "Zählmonitor 4") [Hofm 87], the fourth generation of hardware monitors developed at the University Erlangen-Nürnberg [Klar 95]. ZM4 tries to overcome the portability problem by offering a variety of interfaces for several target systems, from Transputers via the networks of personal computers to the MEMSY system (Modular Expandable Multiprocessor System), which is characterized by its extensive support for measurements [Hofm 93]. Additionally, the interpretation of ZM4's results is supported by the post-mortem software environment SIMPLE (Source related Integrated Multiprocessor and -computer Performance evaluation, visualization and modeLing Environment) [Daup 92], which tries to relate the observed data with the original source code in order to compensate for the low-level data obtained by the monitor.

Another kind of hardware monitors are so-called hardware counters, which are directly included in the microprocessor's hardware. These counters can be incremented whenever pre-

defined events occur, and thus may be used to obtain heuristics about process activities. At present, there are two efforts worth mentioning in this domain. One is PCL (Performance Counter Library) [BeZi 98] developed at the Research Center Jülich, the other is PerfAPI or PAPI (Performance Application Programming Interface) [Mucc 99] developed in the frame of the Parallel Tools Consortium (see Section 4.5.1). An example of using these hardware performance counters is described in [Rabe 99], where the PCL library was integrated on the SGI Cray T3E architecture to observe the frequency of calls to routines of the standard Message Passing Interface library.

The second monitoring technique, hybrid monitoring, is a mixture between software monitoring and hardware monitoring. Mohr characterized hybrid monitoring as follows [Mohr 93]: the recognition of events is done in software, but the recording and time stamping is done in hardware. It is still rather low-level in order to generate as little overhead as possible, while at the same time supports software probes and other functional characteristics from software monitors. Its main advantages compared to pure hardware monitors are, that it is much more flexible and portable, while at the same time providing an improved interface to the user.

An example hybrid monitor has been developed by Malony and Reed for the iPSC/2-Hypercube system [MaRe 90]. Their tool operates on a 5-bit parallel interface of the hardware architecture, which connects each of the processors to one common connector. With this approach, the 32 nodes hypercube is divided into two 16 nodes hypercubes, where one of them performs the computational task, while the other performs the observation [Klar 95].

A similar approach has been implemented in our EMU (Event Monitoring Utility) [Kran 96a], which offers a monitoring cube strategy for hypercube multiprocessor architectures (such as the nCUBE 2 multiprocessor and the SGI Origin 2000). Again, the available hypercube is divided into two smaller hypercubes with equal dimension, where one performs the computation while it is being observed by the other cubes. In principle, the functionality of the monitor is divided into two parts, with a very small part for accessing the process state on the computational nodes and a remaining larger part with the functionality for processing and storing these data on the monitoring nodes. In [Kran 96a] we describe the advantages of this hybrid monitoring approach compared to software monitoring approaches in terms of monitor overhead. Other hybrid monitoring approaches are described in [Gorl 91], [HaWy 90], [Lump 90], [Malo 89], and [Mari 90].

The last group of monitoring tools is probably the most widely used, the software monitoring approaches. As a consequence, there exists a wide range of different techniques for different hardware architectures, e.g. [Gris 91], [Joyc 87], [LeRo 85], [LePa 85], [Mail 95], [Mill 86], and [Snod 88]. Most of these tools are solely intended for one specific parallel system, prohibiting its usage on other operating systems and hardware architectures. Instead, it seems that every tool developer restarts with another implementation of a monitoring component, because each system has its own design goals and philosophy for solving one particular class of problems. Yet, this is a minor discrepancy which has been noted before, because the basic characteristics of all these approaches are comparable.

In [Mohr 93] Mohr tries to emphasize this fact and describes a possible way for a target-independent way of monitoring and analyzing parallel systems. The latest results of these ideas have been implemented in EARL, the Event Analysis and Recognition Language, which allows to construct new trace analysis tools by writing scripts in the EARL language [WoMo 98]. Similar goals are pursued by several standardization approaches for the monitoring layer of debugging and performance analysis tools. One of the earlier representatives in this category is PICL, the Portable Instrumented Communication Library developed at the Oak Ridge National Laboratory [Geis 90]. One of the central ideas of PICL was to offer the user a

communication library that is available on different parallel computer architectures and offers the same kind of communication operations. In addition, PICL allows to generate traces for post-mortem analysis, which report the calls to the PICL functions. Consequently, an analysis tool like Paragraph [HeEt 91a] can be used to evaluate the contents of the PICL traces and derive valuable information about the program's execution.

A comparable solution was developed by Reed at al within the Pablo performance tuning environment [Reed 93], which consists of several components for instrumenting, tracing, and analyzing parallel MPI programs. The trace file format of this toolset is the Pablo Self-Defining Data Format (SDDF) [Aydt 92], which is a data description language for specifying both data record structures and data record instances. In a certain sense it is a data meta-format, because it can describe general data records as opposed to a predefined set of records. Originally developed to link the Pablo instrumentation software to the other analysis components of pablo, like Autopilot [Ribl 98], Virtue [Reed 95], and svPablo [DeRo 98], it has proven to be even more general and extensible than first envisioned by its authors. By providing features like compactness, portability, generality, and extensibility, it is nowadays used by several other monitoring and analysis tools.

For the sake of completeness, we also require one remark concerning these three tracefile descriptions, EARL, PICL, and SDDF. Each of this formats is placed on ASCII representations, which offers a high degree of flexibility and readability. In fact, in most cases, users may even be able to understand the contents of the traces without any analysis tool. Yet, this feature includes also the drawback, that tracefile size may grow easily leading to enormous requirements of trace storage. For that reason, most comparable approaches try to establish binary representations, which allow to store a program's behavioral data more efficiently. In addition, there are approaches that try to apply compression mechanisms to traces as described in [Rons 95]. In [Frum 98] an interesting idea is described, which compresses the trace data based on information-theoretic techniques. Comparable statistical methods are followed in [Reed 97] and [Yan 98].

The problem of tracefile size is also less critical in relation to on-line monitoring tools, e.g. the Online Monitoring Interface Standard OMIS [Ludw 96] by Ludwig et al, which has already been described in Section 4.5.1 and is currently available as Version 2.0 [LuWi 97]. With this interface description, various types of run-time tools for parallel and distributed systems and the systems themselves can be interconnected. By using a monitoring system with a standardized interface, arbitrary target systems could be supported with the same powerful set of tools. Furthermore, tool developers can easily attach new tools to already existing implementations of OMIS compliant monitoring (OCM) systems on different target architectures, which in term can concurrently serve several compliant tools, thus offering a means for tool interoperability [Wism 98a].

Another interesting idea related to monitoring tools is the integration of record&replay mechanisms. Certainly, the record phase is performed by a monitoring system which retrieves all the necessary information to guarantee an equivalent execution during the follow-up replay phases. Of course, this monitor requires only the most basic features and needs not generate complete process state information for the analysis tasks. Instead, this process state information may be generated during the replay phases, when the amount of monitor overhead is harmless. Therefore, it seems useful to include the replay functionality directly in the monitoring code, so that for each record function a corresponding replay function is provided.

In order to evaluate the benefits of this approach, we have reworked our original approach to monitoring - EMU (Event Monitoring Utility) - and its corresponding replay tool PARASIT (PARAllel SImulation Tool) into the NOndeterministic Program Evaluator NOPE. The advan-

tages of this approach are manifold: Firstly, it improves the usability, because only one interface is provided and both the record and the replay component are steered with similar commands. Secondly, the code is more compact and additional instrumentation functions can be added for the record and replay phases simultaneously. Thirdly, and probably most important, it is very easy to switch between the monitor function and the replay function, which is especially important when event manipulation has been carried out.

In addition, we have studied the documentation of OMIS, and we believe that a similar approach could be applied by simply using the intrinsic steering possibilities of such a monitoring tool. The resulting OMIS compliant record&replay tool could profit from all the advantages of every OMIS compliant tool, while at the same time provides the same features as our own record&replay tool NOPE.

4.5.4 Software Visualization

After this brief discussion of the monitoring component, we will now present an overview of existing program analysis techniques. The simplest method of program state analysis is the print statement, that is added to the code. In fact, it is an accepted reality that print statements are the most widely used debugging tool for software developers [Lieb 97]. The reasons for this situation are diverse, and may be attributed to the simplicity of adding print statements to code compared to the learning process for any debugging tool, as well as the inadequacy of many error detection and performance tuning strategies and their corresponding user interfaces. This dilemma and the reasons for refusal of users to accept the products of tool developers are an focal point for the several studies of Francioni and Pancake et al as described in [Fran 96], [Panc 94], [Panc 95] and [Panc 99]. The suggested solution is a more user-oriented approach in designing the human computer interface of upcoming tools in this area.

A similar point of criticism is targeted at the standard debugging abstraction, the source code. In fact, most of the current error detection approaches confront the user with the original lines of code, and offer additional functionality for steering the program through this code and inspecting and manipulating the states of the processes based on the original code. However, as we have briefly discussed in Chapter 2, parallel computing is an area of high complexity, where programs consist of multiple concurrently executing and interacting tasks which possibly handle huge amounts of data over long periods of time. Yet, it has been noted before, that textual representations are inadequate to express the complexity established in such programs [Panc 96a]. Therefore it seems useful to investigate techniques for debugging and performance tuning in the area of software visualization [KrSt 93].

Visualization is widely accepted as a powerful technique to manage various stages of the software life cycle, especially to improve specification, design and development, programming, and program analysis [ZhMa 94]. Usually, two different fields of visualization are distinguished, visual programming and program visualization, which are introduced with the following two definitions:

Definition 4-11: Visual programming [Pric 98]

Visual programming is the use of “visual” techniques to specify a program.

Definition 4-12: Software visualization [Baec 97]

> Software visualization is the systematic and imaginative use of the technology of interactive computer graphics and the disciplines of graphic design, typography, color, cinematography, animation, and sound design to enhance the comprehension of algorithms and computer programs.

The main difference between these two areas is their goal. While visual programming seeks to make program specification easier by using a graphical (or "visual") notation, software visualization seeks to improve the understanding of programs and algorithms using various techniques. Yet, there are also some similarities which arise from the visual techniques being applied. For example, many visual programming tools can be cited that include both, a mechanism for specifying the program and a mechanism for investigating the actual behavior of the program (e.g. HeNCE [Begu 93b]). Often, different kinds of representations are used for these two tasks, but there exists some notable exceptions which offer the same visual representation during program construction as well as during error detection and tuning (e.g. Visper [Stan 98]).

The similarity between both fields, software visualization and visual programming, is also recognizable from their main goal. In both cases visualization is used in the sense of forming a mental image of something that can be aided by graphical, auditory, and other sensory modalities. These means are utilized to aid the reasoning and understanding of programs and their executions, respectively [Zhan 99]. This is underlined by Miller, who believes that visualization should guide, not rationalize [Mill 93]. In this context, to guide means that the visualization supports the user in discovering things, that were unknown before. In contrast, to rationalize means that things are presented, that are already well-known.

Since we are mainly interested in software visualization, we will take a closer look at the classification of Price, Baecker, and Small [Pric 98]. They distinguish the following different types of techniques, which are derived from the distinction between programs and algorithms, static and dynamic representation, and code and data values [Pric 98]:

- Program visualization
  - Static code visualization
  - Static data visualization
  - Data animation
  - Code animation
- Algorithm visualization
  - Static algorithm visualization
  - Algorithm animation

The main distinction is between programs and algorithms, where program visualization is used to display actual program code or data structures, while algorithm visualization operates on higher-level descriptions of software. Both kinds of visualization support either representations of code segments or data values, and are performed either statically, where the display contains a snapshot of the data, or dynamically, where the display is changed over a period of time and portrays data processing and the occurrence of state changes.

4.5.5 Visualization Tools for Program Analysis

According to this simple classification, a lot of different visualization tools using various techniques for program analysis have been produced. Much of this software is specialized or experimental, focusing only on certain applications and architectures. Only a minority of work has addressed the issues of reusability and configurability for analysis tools [MaWi 93]. In addition, there is a high degree of discrepancy for the acceptance of visualization when comparing performance analysis with testing and debugging. While the former already relies heavily on program visualization, the latter still concentrates on textual representations.

An interesting facet related to this domain is interactive auralization of code, which may be applied to inform programmers about key control and dataflow events [Baec 97]. Its idea is to simultaneously interact with the graphical interface, while sound is generated by the program's state changes. Although this may seem a little bit odd at first glimpse, there are actually some implementation for program analysis with auralization (e.g. [BrHe 91] and [FrJa 93]).

There are many different papers introducing a wide variety of taxonomies and classifications for program visualization and corresponding tools. In [Pric 92] and [Pric 93], a tree-based taxonomy is proposed that includes categories for input, output, and processing, along with some metrics. The classification of [RoCo 92] differentiates and categorizes the elements and methods of program visualization based on the mapping technique between the code and the graphical representations. Yet another approach is offered by [KrSt 93] which deals with the transformation of the abstract execution of a program into its graphical representation. Contrary to these classifications, which try do describe how to present program abstractions to the user, the paper [Ouds 96] deals with the question of what to present to the user in terms of program abstraction and data representations. In [BaEi 96], the authors address specifically the problem of visualizing large software systems, while in [ReRi 99] the suitability of visualization systems for program analysis in computational grids is discussed. Other overview papers are [MaRe 89], [PaUt 89], [Appe 90], [HeEt 91b], [Begu 93a], [Heat 95a], [Heat 95b], [Hyrs 95], and [Zhan 99]. In addition, several books covering this topic have been released, like [SiKo 90], [EaZh 96], and most recently [Stas 98].

Another remark in this context concerns complete testing and debugging environments as well as programming environments that include debugging as an integral part of their program development strategy. The usefulness of such a combined approach has been described in [ApMc 88]. In [Lour 97], Lourenco et al describe the combination of the distributed debugging engine DDBG [Cunh 98a], which is the predecessor to PDBG [Cunh 98b] that has already been mentioned above, with the structural testing tool STEPS [KrWi 96]. This combined approach allows to locate errors through symbolic source code analysis and controlled execution and inspection of the target program. A comparable solution is offered by the TOOL-SET, which aims to provide and integrated set of interactive and automatic tools for analysis of PVM programs on networks of workstation [Wism 97b].

Some examples for complete programming environments with integrated debugging capabilities are the Annai programming environment [Clem 95a], which has already been mentioned before with its PDT debugger, the Graphical Application Development Environment GRADE [Kacs 97], the EDPEPPS environment [Zeme 99], and the TRAPPER environment [AhBa 99]. The basic idea of GRADE is to support high-level graphical programming with the graphical language GRAPNEL for the PVM environment. Therefore, it offers modules for constructing, executing, debugging, monitoring, and visualizing message-passing programs, which can be accessed by the user within an integrated graphical user interface. As a debugging tool, GRADE integrates the distributed debugging engine DDBG [Cunh 98a], which is

the predecessor to PDBG [Cunh 98b] that has already been mentioned above. Furthermore, GRADE includes Tape/PVM and PROVE to support performance monitoring and visualization. The EDPEPPS environment is in many ways comparable to GRADE [Zeme 99]. Their key differences are, that GRADE applies its own pure graphical language, which does not model exactly all the semantics of PVM programs, while EDPEPPS combines graphical and textual representations. In addition to the tools above, the TRAPPER environment supports both, PVM and MPI, and runs on Windows NT [AhBa 99].

The same situation is applicable for the different types of applications and the tools incorporating one or several of them. One of the best known approaches for modelling parallel programs are Petri Nets, which are applied on different levels of program abstraction in several different characteristics ([Vaut 87], [Fers 92]). Another kind of graph is the dependence graph, which explicitly displays the control and data dependencies for the operations executed by a program ([DaKe 82], [Begu 93a]). Additional displays are the program graph (a static call graph), the Gantt chart (process state changes) and the critical path (responsible for program execution time). Furthermore there exist data access displays, histogram-bargraphs, xy-contour plots, memory access patterns, data-machine-processor views and a lot more [Jaki 95]. Some remarkable approaches are three-dimensional representations as in [ViSa 94] and parallel program analysis with Virtual Reality techniques [Reed 95].

A complete list of visualization techniques as well as its corresponding tools cannot possibly be covered in this work (and probably in no other paper as well). Thus, we concentrated on some representative examples instead of trying to achieve a higher level of completeness. Especially one technique is interesting, because it is used in many different tools under various aspects. This is the space-time diagram, which was introduced by Leslie Lamport to express the order of events in distributed systems [Lamp 78] and is therefore sometimes called Lamport diagram. Other synonyms for the same representation are Feynman display [Hond 95] or Hasse diagram [Klar 95]. However, the term Feynman display may be misleading, because it originated from quantum mechanics and to our knowledge, Feynman never used it referring to computer science (see [Feyn 96]). Similar representations with comparable semantics are also used in other domains, e.g. the message flow graphs as a means for performing deadlock analysis and other optimizations at compile time [LaLe 95].

Such a space-time diagram is used in many tools for parallel debugging and performance tuning ([McHe 89], [SiKo 90], [RoWr 93], [Brow 95]). One of the best-known examples is the already mentioned Paragraph ([HeEt 91a], [Heat 93]), which includes the space-time diagram and many other graphical displays. Following its success, many computer vendors announced a similar set of performance visualization tools, e.g. ParAide for Intel's Paragon, MPP Apprentice for Cray's T-3D, Prism for TMC's CM-5 [Alle 91], PV for IBM's SP2, and CXTrace for Convex's SPP1 [YaSa 96]. In ParaGraph the execution thread for each process is represented by a horizontal line, drawn from left to right, which changes color to indicate the state of the process (active, idle, blocked). Message operations are depicted by lines which connect the two communicating processes. The display reveals processors waiting for a blocking communication to complete.

Another tool that incorporates a space-time display is the AIMS toolkit ([Yan 95], [Yan 98]). AIMS is based on source-code-level instrumentation and enhances portability across platforms from different hardware vendors. One of the many graphical views in AIMS is the view kernel VK. It displays the dynamics of program execution using an animated view. This view presents the information about the execution of programming constructs. The constructs are source code statements, whose execution are displayed as graphical objects.

Similar to AIMS is XPVM [Geis 94b], a graphical user interface for analyzing PVM programs. The space-time view in XPVM shows computing time, communication overhead and waiting time. The animation within XPVM can be done on-line or post-mortem. A similar functionality compared to XPVM but for MPI programs is offered by the tools Upshot [HeLu 91] and its successor Jumpshot [Zaki 99]. In Falcon [GuEi 94] a variation of the space-time display animates thread creation and completion. When a thread is created a horizontal bar is added in the display. This bar is connected with the parent thread with a vertical line. The color of the horizontal bar represents the state of the thread. When a thread finishes its execution, the right end of the horizontal bar is connected to the parent thread with a vertical line.

The concurrency map [Ston 88a] is an alternative representation to the time-line diagrams, emphasizing potential concurrency at task level instead of interprocess dependencies. A variation of the basic diagram comes from the causality graph [Zern 92] or from the Animation Choreographer of the PARADE system [KrSt 94]. Other examples with similar views include Paradyn ([Mill 94] and [Mill 95a]) with the time histogram, PVaniM [Topo 95] with the Lamport-timeline view, Trapper [Born 95] with another state-time animation, and OrWell with the communication chart [WeKu 98]. Instead of displaying only discrete events, many representations choose to display process states as time-bars, where possibly different colors indicate different activities of the process, e.g. in VAMPIR [Nage 96] or Paje [ChSt 00a]. The latter is also noteworthy due to its capabilities of dealing with dynamically created threads of various lifetimes during program execution [ChSt 00b]. Other related tools may choose a more general form of the directed graph representations, e.g. the Computationally Oriented Display Environment CODE and the Heterogeneous Network Computing Environment Hence [Brow 95].

4.5.6 Additional Levels of Abstraction

From all the examples provided above it seems appropriate to use such a space-time diagram for displaying concurrently executing and interacting processes. Furthermore, we believe, that it is not only most appropriate for performance analysis, but it is also suitable for debugging large scale application. Therefore, we will introduce a formal model - the event graph - in the next chapter, that can easily be displayed as a space-time diagram and provides a useful program abstraction for error location and execution steering within a debugging tool.

Although space-time diagrams are obviously a good starting point for any kind of program analysis tool, it may be appropriate to combine it with representations on other levels of abstraction by offering a scalable approach. Especially the number of nodes and edges may easily be too demanding for conventional workstation computational resources [Kime 95]. An arbitrary, scalable visualization tool would then be adaptable to the user's needs in order to increase its efficient usage. One example is VISTOP, the VISualization TOol for Parallel systems, as part of the TOPSYS project [BeBr 93]. VISTOP offers scalability by providing selection mechanisms for "interesting" objects and "interesting" phases of the program. By limiting the display to the certain objects and key scenes of the program, the user's distraction due to uninteresting parts is decreased. A simple example of additional abstraction is the process grid feature of P2D2 [ChHo 94], which provides a graphical overview of all participating processes. A similar display is available in the Mantis debugger [LuCu 96] with the status window, where activities of the processes are displayed with colors and serve as an overview of program execution. Both of these approaches improve the program analysis task because they offer an overview of the processes state.

The benefits of abstraction mechanisms are also very useful in many other domains, e.g. for visual programming [Wirt 94], when graphs and icons on different levels of abstraction need to

be displayed with a suitable layout, or for performance tuning, when bottlenecks are extracted as a result of abstraction [HaMa 94]. In [Kess 96], a pattern-driven approach for automatic parallelization is described, that exploits frequently-occurring computations and programming concepts based on a programs source code.

The IPS-2 parallel program measurement system of Miller et al provides a hierarchical model of parallel and distributed programs with different views of performance data on multiple levels of abstraction [Mill 90a]. Similar strategies for performance analysis are proposed by Yan and Sarukkai [YaSa 96] and independently by Kohn and Williams [KoWi 93]. The idea of Yan and Sarukkai is to provide key strategies for possible time reduction, where the appropriate strategy is selected based an traversing through a hierarchical selection tree. The nodes of the tree identify different characteristics of the program, while the leaves indicate the actual reason for the problem. As soon as the actual reason is detected, an appropriate strategy is provided to solve that particular problem [YaSa 96]. The ATExpert system of Kohn and Williams is even more enhanced, because it (textually) explains the cause of performance problems in detail and proposes concrete actions to actually solve them [KoWi 93].

Similar ideas are provided in TraceView [Malo 91b], where the user can define views with an appropriate filtering of the data that are visualized. Another idea is to introduce different levels of event abstraction. Cai et al describe how to abstract events on lower levels of abstraction (low level events) into events on higher levels of abstraction (high level events), and display only high level events [Cai 93]. Bates distinguishes between primitive events and high-level events [Bate 95], where primitive events represent observed nondecomposed fundamental behavior, while the abstraction of primitive events into other, probably more representative events is achieved with high-level events. Thus, high-level events cannot be observed from the program's execution, but manifest only a way to abstract the observed primitive events into more useful abstractions on higher levels (compare also [From 95c]). This approach is not limited to parallel programs, but is also useful for other areas, like object oriented programs as described in [Nobl 95] and also in [Jerd 97]. Other useful approaches like center of interest [Furn 86] or the fish eye view [StBo 89] display the complete information without losing any global relations by concentrating on areas of major interest and displaying remote regions of the diagram in successively less detail. A fundamental idea of the fish eye view is to provide a balance of local detail and its global context.

This basic event abstraction mechanisms are certainly extensible. In EDL, the Event Definition Language introduced by Bates and Wileden [BaWi 83b], there exist two essential mechanisms for event abstraction: filtering and clustering. With filtering, all but a designated subset of events can be deleted from the original event stream. Clustering means, that one or more primitive events are gathered together into a higher level event, which requires the availability of a mechanism for constructing an abstracted view. This kind of abstraction can be performed automatically by the corresponding debugging tool as described in [Tayl 93], [KuBl 95], [Augu 95], [From 94], [From 95b], and [Reed 97].

The development of EDL has also lead to the high-level debugging approach EBBA, Event-Based Behavioral Abstraction [Bate 95] and the program behavior models of FORMAN [Augu 98]. Both models follow the idea, that the behavior observed in parallel programs may reveal useful patterns, which can be evaluated during program analysis. With EBBA, debugging is treated as the process of creating models of expected program behavior and comparing these models to the actual behavior exhibited by the program (see [Bate 88] and [Bate 95]). The main problem of EBBA is, that cyclic debugging is not supported, which requires to exhaustively describe all the interesting events before program execution [LeMe 87]. Similarly, in the FORMAN language users may specify event relations (e.g. partial order, hierar-

chy) as exhibited by patterns with assertions, that are generated during program execution and can be checked post-mortem [Augu 98]. One of these relations is the inclusion relation, which allows to consider program behavior at appropriate levels of granularity [Augu 99].

Among these tools that have been inspired by the work of Bates and Wileden is POET[9] [Mohr 93], which splits the description into two languages: the Trace Description Language TDL and the Filter Description language FDL. This approach allows to defined not only primitive and high-level events and their attributes, but also their format and representation. These ideas have also influenced the development of EARL (see Section 4.5.3), which offers a high level of flexibility due to its programmable abstraction mechanisms [WoMo 98]. Another interesting characteristic of EARL is, that it is intended to perform automatically as many analysis tasks as possible without user interaction [Gern 98].

A comparable interesting approach of event abstraction is described by Kunz et al in [Kunz 94], [KuBl 94], and [Kunz 95] with an overview of pattern matching by Seuren in [Seur 96]. Their idea is to display abstract events instead of complex communication patterns in the event tracer Poet[9] (see [Tayl 92] and [Kunz 97]), which uses a space-time diagram for displaying a communication graph similar to our event graph. In concrete, patterns (or rather so-called strings) detected in a program that represent logical units of work are replaced by abstract events for the display [Seur 96]. A corresponding grammar for pattern matching is offered by Fromentin et al in [From 95a] and an overview of pattern matching algorithms is presented by Fan and Su in [FaSu 93]. The technique proposed by Knuth et al [Knut 77] slides the pattern forwards in time to find a possible match, while Boyer and Moore [BoMo 77] perform pattern matching reverse in time. The pattern matching approach of Karp and Rabin [KaRa 87] perform pattern matching on hash tables, containing values resulting from the comparison of the patterns with the event traces. For more details please refer to the cited references.

Another interesting set of approaches is related to the program analysis tools Belvedere [HoCu 88], Ariadne [Cuny 93], and TAU [Shen 96]. The pattern-oriented parallel debugger Belvedere of Hough and Cuny facilitates the description, manipulation, and animation of logically structured patterns of process interactions for highly parallel programs [HoCu 88]. Its authors believe, that users must be able to determine logical communication patterns, that occur during program execution. Their conclusions are, that pattern-oriented approaches facilitate debugging of highly parallel programs, and that automatic, perspective animations are needed to expose logical patterns [HoCu 88]. Such perspective use should enable the user to create comprehensible visualizations by selectively ignoring some of the observed events and their dependencies [HoCu 90]. These ideas have been extended in the post mortem, event-based debugger Ariadne [Cuny 93], which matches user-specified models of intended program behavior against actual program behavior. The specification of the program's behavioral model is again based on Bates EBBA [Bate 88]. In [Shen 96], the authors describe how to combine the Ariadne approach with the TAU program analysis environment. In this case, the event-based modeling is applied to pC++, which is a data-parallel object-oriented programming language.

On contrary, there are also tools that allow to cope with the huge amounts of data being processed by parallel programs. For example, the CM debugger Prism [Alle 91] has strong data

9. Please note, that the name Poet is used for two similar tools. One is the Poet event tracer developed at the University of Waterloo, Canada [Kunz 97], and the other one is the Problem-Oriented Event Trace interface function library POET integrated in the event trace analysis environment SIMPLE of the University of Erlangen-Nürnberg, Germany [Mohr 93].

visualization capabilities for data parallel programs implemented in TM's proprietary CM Fortran. In addition, memory watching based on prohibitively slow single-stepping is supported, and the collective state (i.e. whether a process is running, stopped or fulfills a user-defined predicate) of process groups can be displayed. In its latest incarnation, the so-called Node Prism [Sist 96], many of Prism's original goals have been ported to support scalable debugging of massively-parallel message passing programs.

Another interesting idea concerned with managing large amounts of data is provided by the parallel debugging tool PDT within the Annai programming environment [Clem 95b]. Besides the source code browser and a program structure browser, PDT's graphical user interface include multi-dimensional and numerical distributed data displays for viewing entire arrays or array segments allocated across different processors in HPF programs. Such an overview can be useful in detecting illegal array elements or values, that may be responsible for program failures. The same feature has also been included within the SPiDER (formerly DeHiFo) debugging tool (see [Brez 99a] and [Brez 99b]) with the Graphical Data Distribution Tool GDDT [Kopp 97], again for HPF programs[10].

One of the problems of these approaches is, that the data of a parallel program are visualized without addressing the inherent application domain and its semantics. In many cases it would be interesting to display the data according to the user's understanding of the program. For example, the data of a crash test simulation can certainly be displayed with various kinds of representations, and an arbitrary 3-dimensional matrix display may cover some of the important features of the computed data, because the user cannot connect this visual representation with its original meaning. These problems are addressed by so-called application-specific visualization tools as described in [KrSt 93], [StKr 93] and [Cuny 97]. Both papers describe the experiences of their authors when building domain-specific visualization tools, as well as the benefits of users applying these tools to actual problems.

4.6 Debugging Activities at GUP Linz

In this section, we will briefly report about our own debugging efforts at the GUP Linz in order to shape the environment where this work has originated. At GUP Linz, debugging parallel programs is already a hot topic for almost ten years now. Starting with a vision on solving the problems of debugging nondeterministic message passing programs, many additional ideas have been developed and tools have been implemented for a lot of critical issues in this domain. In addition, some useful contributions in the scope of performance tuning have been addressed whenever these overlapping areas seemed to offer benefits for the user [Kran 94].

All these results have been included in the Monitoring And Debugging environment MAD, which combines a set of tools for program observation and analysis under a common graphical user interface [Kran 97a]. The original target platform was our 64 processor nCUBE 2 MIMD multiprocessor architecture, which is still running at the Computing Center of the Johannes Kepler University Linz. During the course of the years, the environment was ported to the

10. The reason, that such a useful array visualization component has only been developed for High Performance Fortran programs stems from the fact, that the specification of data distributions within HPF is an integral part of the parallelization process, and the possibilities in mapping the data onto the processes are very limited. On contrary, MPI imposes no restriction onto the data distributions, which permits the maximum flexibility for this task. Yet, it is also a big obstacle for debugging tools, as we have experienced when trying to implement a comparable feature for our own debugging environment.

Message Passing Interface standard MPI, which was utilized on the 24 processor HP CONVEX SPP Exemplar 1200 architecture and the 64 processor SGI Origin 2000, both also installed at our Computing Center. In addition, MAD can also be used on networks of workstations under the UNIX operating system, ranging from Intel personal computers with Linux to graphics workstations from HP, IBM, SGI, and Sun. On these machines, the graphical user interface utilizes the ET++ application framework, an object-oriented library of C++ classes for composing interfaces under X-Windows.

According to the models of generic program analysis tools, we classify the components of MAD into the monitoring modules and the program analysis modules. In terms of monitoring modules, we offer several possibilities. The initial development was called EMU, the Event Monitoring Utility, which generates traces for post-mortem analysis in our own proprietary tracefile format [Kran 96a]. One of the main ideas of EMU was to use it as a testbed for different monitoring techniques and approaches in order to study the effects of program observation onto the target program. In the scope of this work, we also implemented several strategies for monitor overhead correction, intended to remove the perturbation of both, event timing and event ordering. In addition to EMU, we also developed the trace-driven simulator PARASIT, the PARAllel SImulation Tool, which performs equivalent execution of nondeterministic parallel programs. When porting the environment to the standard MPI, these two efforts were merged into the NOndeterministic Program Evaluator NOPE, which thus represents a complete record&replay mechanism [KrVo 98]. An additional benefit of NOPE is the possibility to perform automatic event manipulation in order to overcome the completeness problem as described in Section 4.2.7. Besides these actual monitor implementations, we also developed several trace file conversion tools in order to use tracefiles generated by other monitoring tools with our own program analysis components.

The central program analysis component of MAD is ATEMPT, A Tool for Event ManiPulaTion, which displays a parallel programs execution as a space-time diagram [Kran 95b]. In this diagram, different graphical objects represent events like sending a message, receiving a message, inspecting a variable, an array or the message queue on a process, and many more. In addition, connected events like corresponding sends and receives are linked, which expresses the relation between events on concurrent processes. Besides visualizing the program's behavior, ATEMPT also performs some basic analysis tasks and highlights simple communication errors with different colors. Furthermore, ATEMPT is intended to be the main debugging interface, partly because it allows setting of breakpoints and inspecting of variables, partly because it provides connections to all the other elements in the MAD environment. Obviously the most important connection is established between ATEMPT and an integrated filebrowser, which allows to associate the graphical objects of the space-time diagram with the actual lines of code that are responsible for generating this event in the first place.

Other tools of the MAD environment which are connected to ATEMPT, are the Control- and Data-Flow Analyzer CDFA [Kran 96d] and the MUltiprocessor Class Hierarchy MUCH [Kran 96c]. CDFA performs control and data flow analysis of parallel programs and generates corresponding graphical views like a function call graph and a control flow diagram. Besides offering another representation of the program's behavior, these views can be used to perform program slicing, which means that only a subset of the original code is displayed, e.g. only statements that affect the value of a selected variable. Additionally, the results of CDFA can be used to improve race condition detection as shown in [Kran 97b]. An even lower level of program abstraction is provided with the visualization of MUCH, which is intended to model the hardware layers of selected multiprocessor architectures. Its idea was that user's benefit from understanding the connection between hardware and software, especially during performance

tuning, if some kind visual representation is provided. Therefore, the program's execution is displayed in MUCH with block diagrams well-known from hardware architecture books. Since both of these tools provide different levels of abstraction when displaying a programs behavior, they are quite useful when investigating problems, that cannot be derived from the space-time view.

A first conclusive overview about the theoretical background of our debugging efforts is described in Grabner's thesis [Grab 99]. His main contribution was to formally classify errors in sequential and parallel programs as well as their connections, and to distinguish root errors and follow-up errors. Based on this classification, a complete debugging strategy is derived, which is arranged around our record&replay mechanism. For the record phase, the three different monitoring models (hardware, software, hybrid) are described with special attention to the hybrid approach of our reference monitor EMU for the nCUBE 2 multiprocessor. Additionally, this initial approach is discussed regarding to its possible application on the HP CONVEX SPP Exemplar 1200 architecture. After monitoring, the observed data can be used to construct a simple communication graph, which represents a space-time view where the nodes are communication events like send, receive, and test.

The special focus of Grabner's thesis is on analysis of nondeterministic program behavior. Therefore, he describes a message race detection algorithm which determines race condition candidates by analyzing the attributes of events in the communication graph. The resulting racing messages can then be utilized by the debugging user to investigate possible reordering effects with the event manipulation functionality. This is achieved by manually selecting race condition candidates from the communication graph display and changing their order for subsequent program runs under control of a trace-driven simulator. In the frame of this approach, Grabner also describes some ideas for program slicing, which combines debugging with checkpointing and rollback recovery in order to perform flow back analysis. Important in this context is also the approach to place applicable cuts in parallel programs, which are breakpoints on several processes.

In this second work on the theoretical background of our approach, we have substantially extended the original approach of Grabner to cover parallel programs of arbitrary scale, from small-scale applications up to massively parallel programs. Although our original strategy is certainly valid on all these program scales, there needs to be some means of guiding the user through all the debugging data and managing the high complexity of parallel programs. This is achieved, by enhancing the simple communication graph to a more general version, called the event graph, which contains not only communication events but any event of interest to the user. Therefore, the event graph represents another level of abstraction, which can further be raised by identifying patterns observed during program execution. Based on the event graph model, automatic error detection activities are applied in order to remove as much tedious tasks from the user as possible. Yet, it has to be noted that this approach cannot be applied in isolation, but instead requires some kind of low level error detection techniques as described by Grabner.

In addition, we were able to improve some of the methods described in the original approach due to some practical experiences from the replay mechanism and slicing that were unavailable before. For example, the message race detection and the cut placement was optimized by exploiting certain characteristics of the Message Passing Interface standard MPI. Furthermore, in this work we distinguish between two different kinds of cuts, whether the breakpoint is set for event manipulation only or for sophisticated breakpointing activities. For our goal to overcome the completeness problem and at the same time to improve the testing cycle, we developed a method to perform event manipulation automatically within certain lim-

itations. In this frame, we also studied the possibilities of monitor overhead removal and developed a very efficient record&replay mechanism.

These achievements are described in more detail in the following chapters of this thesis. In addition, we will include screenshots or implementation details of our debugging tools whenever suitable to the contents of the theoretical explanations.

4.7 Summary

In this chapter, we have described the traditional approaches to testing and debugging of programs, consisting of the testing cycle and its corresponding debugging cycle, and why these solutions are insufficient in the parallel case. Therefore, we have identified the major obstacles which are encountered, whenever errors in parallel programs have to be detected. Besides difficulties due to the inherent complexity of parallel programs and the analysis data associated with the debugging tasks, a lot of critical effects are introduced by communication and synchronization. One of the most critical issues is nondeterminism, which leads to the malign irreproducibility effect and increases the completeness problem for testing. In addition, the probe effect causes severe problems due to the possible reordering of critical events.

This brief overview of the debugging problems lead to the formulation of required features for parallel error detection tools. Such tools always consist of two distinct components, a monitoring component and a visualization component, where the former obtains the necessary data by observing the program's execution, while the latter displays the analysis data for manual inspection by the user. During all these discussions we have included some of the more important references to representative tools in this domain. In addition, this description of the current situation in the area of parallel program debugging has shown, that some issues still remain to be solved in order to efficiently cover all the problems during parallel program debugging. Especially the domain of massively parallel programs has not been addressed sufficiently, although it requires dedicated functionality from a corresponding debugging tool. A solution to this problem will be proposed in the remaining chapters of this thesis, which concentrate on developing a set of debugging techniques, that emphasize automatization and abstraction during parallel program analysis.

5. Modelling Programs with the Event Graph

After introducing the basic principles of parallel program debugging and discussing the appropriateness of related work in this area, we will now evolve the so-called event graph model. For that reason we will recapitulate the main goal of debugging, namely investigating program states, and how to retrieve corresponding information from a program's execution. This leads us to the definition of events and a discussion about what constitutes an event and how its data is generated. Based on these notions about events we will derive the event graph model, which represents not only the set of event vertices but equally important their relations. Finally, our visualization approach to displaying event graphs is described.

- Obtaining program states from an executing code
- Definition of events and corresponding event attributes
- Event relations and ordering
- Event graph visualization

5. Modelling Programs with the Event Graph

5.1 Inspecting Program States during Execution

5.1.1 Program State Changes

In Section 3.2.5, “Debugging Basics” we define how debugging works in general. Now, we are taking a closer look at how debugging is actually performed in practice. The main idea of debugging is to analyze the behavior of a program. In this context, behavior identifies all the activities that have observable effects in an executing system [Bate 95]. To analyze this behavior, the states of a program and their relations as they are established during program execution are investigated. Based on the acquired information, appropriate actions are deduced either to continue the investigation process or, if an error and its source have been identified, initiate the corresponding correction activities.

This simplified description of an arbitrary debugging session also identifies the main source of information that has to be processed during error detection, which are the states of a program. In Definition 3-8 on page 64 we have formally defined the states of a program during execution as the set of the values of its variables [Grab 99]. In this section, we will recapitulate this original definition for use in the event graph model.

The state data of a program during execution are the main information for debugging. Based on these state data, the user tries to locate the place in the program, where an incorrect program state first occurred. This idea is sketched in Figure 5-1, which contains a diagram describing the state changes of a program. The selected program consists of 8 program states, the initial state *S0*, the final state *S7*, and the intermediate states *S1* to *S6*. The final state *S7* describes the expected behavior at the program’s termination point together with the expected results as defined by the program’s specification. Therefore, these state changes may express a valid program execution up to an expected termination point which delivers correct computation results.

In a second program run, the final state *S7’* does not resemble what the user expected. Thus, the user starts debugging and the debugging tool extracts state information from the program’s execution. As soon as this information is available, it can be inspected and analyzed, and the user’s goal is to detect that place in the code, where the program state changed from a correct state *Sx* to an incorrect state *Sy* (in the example *S4* to *S5’* instead of *S5*). As soon as this first occurrence of an incorrect state is detected, the user tries to identify the original reason for the program’s incorrect behavior, which hopefully pinpoints the programming error.

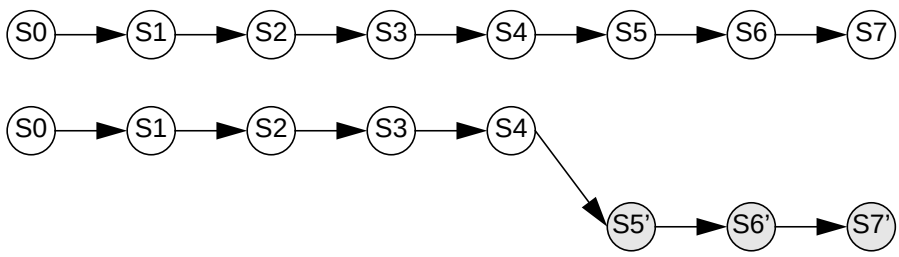


Figure 5-1. State changes in a program

This small example briefly explains, why the states of a program are most important for a debugger. Obviously, in the best case, a debugger would store all information about a program's execution and the debugging user could navigate through this data space backwards and forwards in time. This is exactly the idea that has been followed with Balzer's history tape (see Section 4.1.3). Yet, as mentioned before, obtaining all states of one program during execution is impossible, and storing these state data is even less feasible. One reason is certainly the number of possible states that may be observed. The other reason is the amount of data that has to be saved to allow execution analysis. This leads us to the following important questions in this context:

- What are the contents of a program state required for debugging?
- When and how to obtain the state information?

The answers to both questions depend on what the user seeks to analyze. Defining in advance which information is relevant and which can be neglected is in most cases impossible. Usually, this decision is induced by the debugging tool, whose developers bear the responsibility to designate the correct contents and amount of information that can be extracted, long before the erroneous program exists.

5.1.2 Objects of Interest

In theory, there are only a limited number of items that may be of interest. The contents of a computer system's memory is one of the most important items that describe a program's state. This contains not only the memory parts with the application data like variables and constants, but also address spaces containing buffers, the heap, stacks, and the original program itself. In addition, other elements have to be considered, which may influence and determine the state of the program at any time during execution. Some examples are the various system states like CPU state, I/O state, and others. These items may further be refined, for example in case of the CPU in register contents, cache contents, program counter, pipeline status, and many more.

These selected examples already express, how much data may be needed in the worst case. If one considers a program consisting of several hundreds of processes, the complete state space of a program at one selected point in time is already very exhaustive and cannot be processed by any parallel debugger. Yet, these data also describes only one particular state of the program. In reality, not only one state at one time must be analyzed, but several consecutive states as well as their relations. This leads to the second question in this context, which addresses the problems of when to obtain the state information and how this extraction of data is performed.

The answer to this question is certainly related to the execution time of the program. In principle, there are only two solutions, retrieving the data either at predefined time intervals or whenever certain actions are observed. The former is often called profiling and means, that the state of a program is inspected whenever a certain amount of time has passed. The results of profiling are so-called (execution) profiles, which report cumulative data about the program's execution. Some profiles also report units other than time (such as call counts) and/or report at a granularity other than per-routine, but the general idea is similar. One well-used example is the amount of time spent in each routine of a program, which is then used to find hot spots and tune the program's performance. While profiling is well-established in the area of performance tuning, its results are much less important for error detection.

The standard approach to debugging is based on the idea to obtain state data from a program, whenever a predefined action is taken. This method is often called tracing, recording, or logging, which represent more or less the same activity with minor differences, but may as well be summarized with the term "monitoring". Please note, that even profiling is often included in monitoring, simply because monitoring is such a general purpose characterization of this activity. Profiling and monitoring may even lead to similar results, if the granularity of the time interval and the events are selected accordingly.

Monitoring has already be introduced in Section 4.5.3. Its goal is to extract data whenever discrete, pre-selected events occur. This defines two basic characteristics. Firstly, monitoring is passive as long as none of the specified events occur. Only, if an event of interest happens, monitoring is activated and traces the data required for the subsequent analysis task. Secondly, monitoring takes place at discrete times during the execution, which determines one of the characteristics of events, namely their occurrence time. As we will see in the following sections, this time of occurrence is one of the important things for parallel program debugging.

5.2 Events and Event Attributes

5.2.1 Basic Event Definition

The next step to establish an event graph model is the concrete definition of events. In principle, an event is something that occurs during execution and thus, anything can be defined as an event. As a basic definition we utilize the notion provided by van Rick and Tourancheau in [RiTo 92]:

Definition 5-1: Event [RiTo 92]

> An event is defined as an action without duration that takes place at a specific point in time and changes the state of a process.

Thus, an event is characterized as an expression for a state change of a program. This emphasizes the statement above, that an event can actually be anything or, to be precise, can express any state change occurring during program execution. Please note, that the state change is described by giving the new state, or rather, when the new state initially occurred. Thus, an event cannot provide information about the previous state, although it identifies the transition from this previous state to the current state.

Furthermore, an event is a virtual item, because it does not require any time for its existence, but instead defines the instant when something happens. It does only exist to mark a certain activity over the course of time. This idea is also supported by Bates and Wileden with their "Behavioral Abstraction" approach [BaWi 83b]. They describe that the view of system in terms of events differs from the view of system states, because it emphasizes the activities that are going on [BaWi 83a].

This vague definition leaves a lot of details for discussion. Firstly, it is important to define those activities, that generate events during execution. Due to the freedom of the event definition, this set of activities is only limited by the user's requirements and imagination. For example, if the user is only interested in communication events, it may be useful to limit the debugging tool's focus to events generated by communication statements. In practice this

means, that whenever a communication statement is executed, a corresponding event describing this change in the programs state space is observed. This defines one of the basics of events, namely that they can be generated by statements in the program. Statements are only one example. Other possibilities include events like program start and termination, I/O activities, system interrupts, and of course execution failures. Table 5-1 offers some examples of events (compare with [KrWi 98]):

Table 5-1: Event types and event data

Event	Data
process start	input parameters,...
process termination	results, output parameters, variable dump,...
send message	source, destination, message type, parameters,...
receive message	source, destination, message type, parameters,...
failure	process number, failure number, failure data,...
user defined event/action	event name,...
random number generator	return value
open file	file name, access mode, buffer size,...
close file	file handle
read from file	file handle, record that has been read
write to file	file handle, record to be written
variable dump	variable type, value, number of elements,...
message queue inspection	message type, contents of message queue
system events	system related events
debugging events	artificial events required for debugging

In addition, we need to distinguish between different places, where events can be observed. For each statement, there are at least two places where an event may occur or the state of the process is changed respectively. On the one hand, each statement may generate an event upon its entry, e.g. when the corresponding statement is actually called by the running program. On the other hand, each statement may generate an event after its completion, e.g. when the statement returns after its operation. Many more places for events are imaginable. For example, in function calls to the communication library, events may be generated at the following places:

- When the communication function is called.
- When the data is available for transmission.
- When the communication port (or link) is available.
- When the communication has been completed.
- When the communication function returns.

These few examples already indicate, that giving a complete set of events is an impossible activity. In fact, the particular set of events that will be appropriate when searching for the causes of a given failure will vary, and no set whatsoever can be expected to be appropriate for all problems [BaWi 83b]. As mentioned above it completely depends on the user's intention and his target, what events should be selected during monitoring, and any set of events always restricts the user to this particular kinds of state changes. The most flexible system would be one that allows the user to specify events of interest and how and when the corresponding event data is generated.

Besides that, events are also restricted by the underlying system, since some types of events only exist on certain types of systems. Examples are events occurring on parallel machines. Obviously, we can distinguish between different parallel systems and thus, there are also types of events which may occur on one system and may not exist on others. Such a distinction can be made based on the chosen parallel programming technique (see Section 2.3.4). Certainly, programs based on the message-passing model exhibit other types of events than those based on the shared-variable paradigm. Since our main focus is on message passing systems, we will define events for sending and receiving messages, which will be called send-events and receive-events as a term for abstraction. Furthermore, we may distinguish different types of send and receive events, for example non-blocking receives, the so-called test-events.

5.2.2 Characteristics of Events

Besides the fact that one particular event type describes one corresponding state change in a program, there are several other characteristics that describe each event. Obviously, events may be classified, whenever similar actions generate the same events. In principle every event instance represent just one occurrence of the behavior represented by one event class [Bate 95]. For example, if we are talking about send events, we address the class of send operations that generate one particular kind of event instance. In addition, there may be minor differences between two send events, but both identify one particular kind of operation.

Besides this classification, events deliver data that describes the behavior that was responsible for their occurrence in the first place. These characteristic data are:

- Type of event
- Event attributes
- Scope and location
- Level of event generation

The type of each event is clearly a manifestation of the state change that was responsible for generating it in the first place. Obviously, this type also determines the other characteristics and their actual contents. Some events and their attributes are given in Table 5-1. Consider a send event, the corresponding event attributes contain all the data that are interesting during event state analysis and describe the actual message send operation. As indicated in Table 5-1, this could be the source and the destination process of a message, its message type, and additional parameters required for message transmission. Similar to the type of event, it is also desirable, that the actual attributes of an event are user-selectable. The reason is, that the user might be interested only in one parameter, for example, the contents of the message, and may neglect all other parameters during monitoring.

Other important characteristics of events are their scope and their location. Both, an event's scope and its location may be either global or local. The scope describes how an event affects a

program and is global if it affects the complete program, or local to one process if it only changes the state of this particular process. In addition, there may be events that affect only a subsets of processes like multicast message transmission or similar group operations. Please note, that the scope of the message is only of interest for the monitoring tool, because it may be possible to restrict state changes to smaller items than the complete program. It is also clear that the scope of an event always affects the global state of the program, simply because the global state represents the accumulation of all existing local states. On contrary, the event's location identifies where the state change has actually been observed. This may be a location on all or a subset of processes at the same time, but identifies in most cases one particular process. This viewpoint represents a difference to the work of Bates, who only addresses the location of a process, but neglects the possibility of different scopes [Bate 88]. Thus, in his work an event always affects the complete program, which may yield a higher complexity during program analysis.

The last characteristic is the level of event generation, which influences the observable event types and their corresponding event attributes. This level may be selected from a variety of possibilities, where some of them are the actual source code, the corresponding assembler code or machine code, or any other semantically interesting level of an arbitrary higher abstraction. If an event is generated on machine level, it obviously generates different data than one on the source code level. For example, if the user chooses to monitor the send event on the source code level, the event attributes usually resemble the parameters of the source code statement. However, the send on the source code level is also encoded as a series of statements on the assembler level, and a send event may also deliver the actual communication link that is applied for message routing, and other more hardware- and operating system-specific details. Other examples are imaginable, and again, the actual instantiating depends on the user's requirements.

5.3 Event Ordering and Relations

5.3.1 The Observability Problem

So far, the discussion about the program states and state changes was only focused on concrete states of processes. Equally important to the states of a target program during execution is the relation between these states. To deduce the program's behavior, the debugger must relate the occurring state changes by ordering them according to their causality. In other words, to locate a bug that is responsible for an error, the programmer must reason about the causal relationships between events in the program's execution [Choi 91].

Ordering events in a sequential program is trivial, because a correct causal order is established by adopting the event occurrence time. Events that happened later in time are causally dependent on events that happened before. Formally, this can be described as follows:

Definition 5-2: Sequential order of events

> The sequential order of events $\xrightarrow{S}$ defines, that the ith event on any process p, denoted as e_p^i, occurred before the $i+1^{th}$ event e_p^{i+1}.

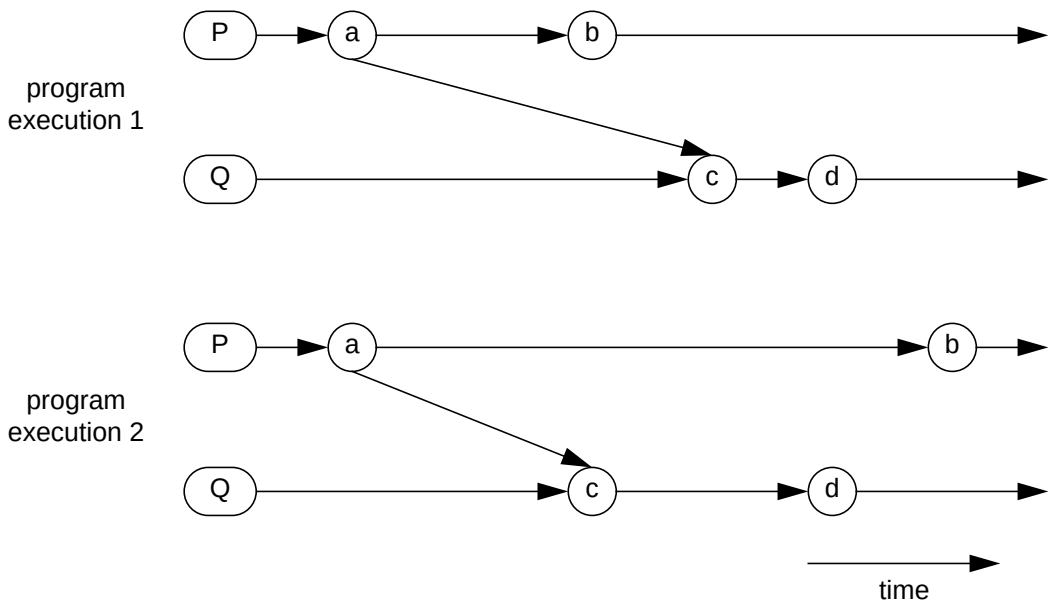


Figure 5-2. Observability effect (compare [Fidg 93])

In the parallel case, establishing a useful ordering among events is much more difficult. Clearly, the problem of ordering events on one particular process resembles the sequential order as given above. However, to extract an order of events on concurrent processes is more difficult due to the fact that some events may occur simultaneously. The following discourse about event ordering can be applied to any system exhibiting concurrency, simply because these systems allow that two or more processes to executed concurrently. This effect is called the "observability problem in concurrent systems" and can be defined as follows:

Definition 5-3: Observability problem in concurrent systems [Fidg 93]

> The observability problem is defined as the difficulty to determine the order in which events occur (or have occurred) in a concurrent system during a given computation.

Please note, that this definition does not describe how the observation of the system is being performed, and observers may be anything, from human programmers watching an animated display of system activity, or monitoring processes on the network analyzing the program's execution. In addition, observers may watch the system "live" (on-line), while the computation is in progress, or examine a post-mortem event trace generated by a monitoring tool. Whatever observer is chosen, the problem is always the proposed method of timestamping, which is used to associate timing information with each event, and thus assess event ordering.

The observability effects can be described with the two example program executions sketched in Figure 5-2. Both graphs may describe the same computation with the same input data, consisting of two parallel processes *P* and *Q*. On process *P* two events are observed, an event *a* which denotes the transmission of a message and an arbitrary local event *b*. Process *Q* also performs two events *c* and *d*, where *c* represents the event that receives the message sent by process *P*.

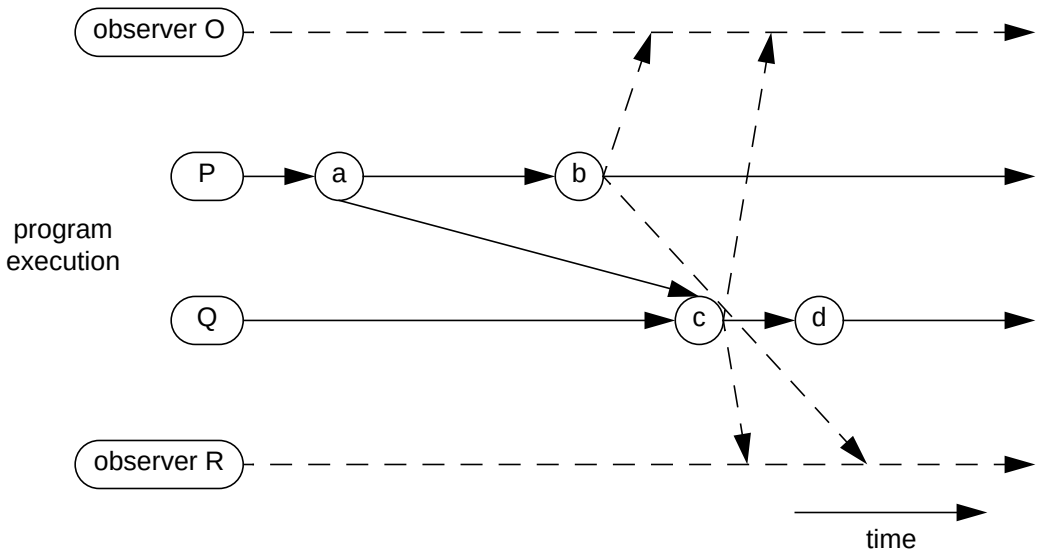


Figure 5-3. Observability effect for multiple observers (compare [Fidg 93])

In this example it is important to notice that in the absence of any further information both executions may be observed. Certainly, local events occur always in the same order, which again expresses the sequential ordering as defined above. This means, that events *a* and *b*, as well as *c* and *d*, respectively, are ordered sequentially by their time of occurrence. However, from our "omnipotent" viewpoint it is impossible to interleave independent events such as *b* and *c* on concurrent processes.

The effects as described by Fidge can be encountered by an observer in four different ways [Fidg 93]:

- Same computation[1] exhibits different orderings.
- Multiple observers see different orderings.
- Incorrect orderings are perceived.
- Arbitrary orderings are adopted.

The first problem describes exactly the situation given in Figure 5-2, where two executions of one particular program reveal different event orderings, even for the same input data. This is especially critical for debugging, when a debugging user is forced to execute the program several times in order to reveal errors during the testing and debugging cycles (see Section 4.1.2 and 4.1.3). In both instances the program reveals a different event order when observed from a global viewpoint, although it involves the same program supplied with the same data and following the same control path. Thus, this situation is also observable in case of deterministic programs, but is further exaggerated, if nondeterministic programs are taken into account. For the latter, the program may actually follow different control paths during subsequent executions and the observer may be completely frustrated because a different set of events and a different event order is observed every time the program is executed.

1. With the notion "same computation" Fidge means different executions of the same program with the same input data.

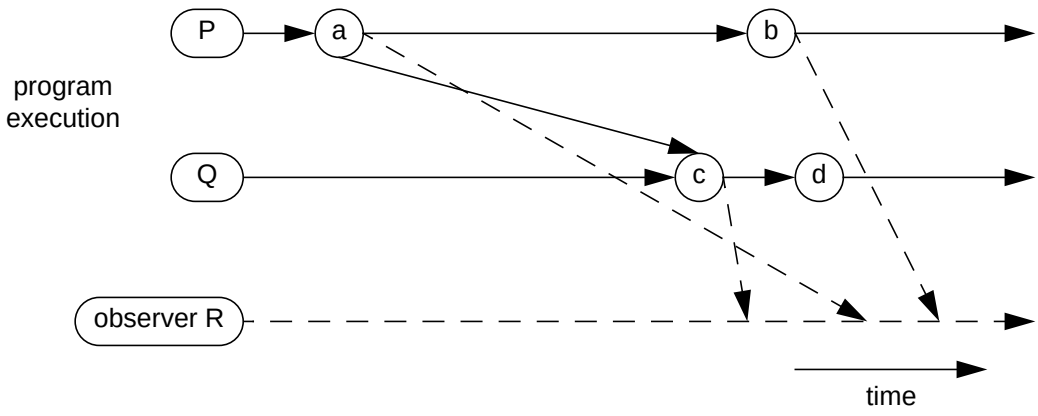


Figure 5-4. Observability effect for incorrect perceived orders (compare [Fidg 93])

The second problem occurs, whenever two or more observers target a particular computation. In this case, each of the observers may believe in a different ordering due to the propagation delay between event occurrence and observer notification. Both extracted orderings may be interpreted as valid observations, and reconciling them is rather difficult. This situation is described in Figure 5-3, where two observers *O* and *R* are added to the program's execution 1 from Figure 5-2. Due to the notification delay, observer *O* believes that event *b* occurred before event *c*, while observer *R* believes in the opposite event order.

The third problem is more serious, because it may happen that the perceived event ordering is completely incorrect. This effect may occur, if the notification delay is large enough as described in the example of Figure 5-4. In that case, observer *R* erroneously believes that event *c* occurred before event *a*, because it has been notified about the occurrence of event *c* earlier than about the occurrence of event *a*. The reasons for this notification delay can be, that retries are required for the notification message or that it is routed via blocked indirect pathways.

The remaining problem is different to the previous three problems, because arbitrary orderings may be assumed between unrelated events, even though these events are not related. This problem occurs, if observers relay on the arrival times of notifications to determine global event orderings. Again, the example is Figure 5-4, and the observer *R* is notified about the occurrence of event *c* before the occurrence of event *b*. However, this notification is simply an artifact of the notification mechanism and does not represent reality, simply because event *c* is completely independent of event *b* and event *c* may occur either before or after event *b* from a global viewpoint. This represents a severe problem, because such an arbitrary ordering is indistinguishable from a genuine enforced orderings and may influence the observer's analysis activities. For example, a debugging user witnessing event *c* occur before event *b* may mistakenly conclude that this program has sufficient interaction between processes *P* and *Q*, so that this event ordering is preserved during every subsequent execution.

5.3.2 Methods for Event Ordering

The observability problem has extensively been studied in the past, and several solutions are provided which are usually based on the original article by Lamport [Lamp 78]. In this work Lamport describes how to order events in distributed systems with different types of clocks. This basic approach has been further advanced by Fidge [Fidg 89] and Mattern [Matt 88], who independently developed a similar system for describing the global states of a concurrent sys-

tem during execution. In addition, several important contributions and clarifications have been described by Raynal et al in [Rayn 92] and [RaSi 96]. Since these methods are inherently important for the remainder of this work, we will briefly summarize the most important facts.

According to the overview article by Fidge, mainly two groups of timestamping mechanisms with two ordering approaches each have to be distinguished [Fidg 93]:

- Real-time timestamps
  - local clocks
  - global clocks
- Logical timestamps
  - totally ordered clocks
  - partially ordered clocks

These four methods are used in various existing monitoring and program analysis tools, with different consequences for the observed results. Each of these methods depends on the idea that events are timestamped whenever they occur, and this information is transmitted to the observer in notification messages. Thus, the observer relies on these data to establish the global event order, which hopefully resembles the true order in which events occurred. In the following paragraphs, we will reveal that this is not always the case, and we will describe, why some of these methods are inappropriate to cover the effects of the observability problem.

Real-time local clocks:

A first obvious approach, and one that is often applied in monitoring tools is to make use of whatever real-time clock is available in a given computing system. This may either be a local clock, which means that each processor contains its own timing hardware, or a global clock, where all processes access the same timing information from a central source.

In case of a local clock, the observability problem “Multiple observers see different orderings” can be solved, if the timestamp of each event is generated on the process associated with this event. Then, the event data and its timestamp can be transferred to multiple observers which in turn will then observe the same time ordering for each event. However, the local clocks approach does not solve any of the other problems, since clocks on different processors are not synchronized and will therefore inevitably drift, so that no ordering can be applied across processor boundaries. (This does not prevent to order events on concurrent processes or threads, if the processes are located on one particular processor and thus executed the same timing hardware!)

Real-time global clocks:

This situation is improved, if a global clock is available in the computing system. This is sometimes the case in tightly coupled systems, where processors access the same main memory as well as other shared resources. In distributed systems, e.g. distributed memory machines or networks of workstations, a global clock accessible by all processors is usually not available. The problem is to offer one source of information to all processors, which is obviously difficult in case of massively parallel machines. For these types of machines, software solutions offer a potential answer. Since clock synchronization is a large research area by itself that affects not only testing and debugging of parallel programs but a wide variety of different applications, we will not discuss possible synchronization approaches in detail. Instead, we will assume that

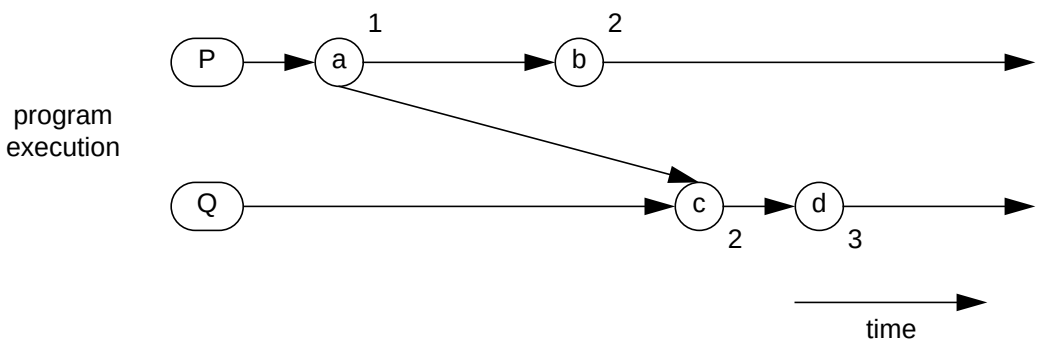


Figure 5-5. Logical clock ordering with Lamport's approach [Fidg 93])

the clocks are sufficiently synchronized with a high degree of accuracy. Please note, that such a powerful facility is not realizable to sufficient accuracy in practice [Fidg 93].

Thus, in case a global reference for real-time is available, the observability effect of "Incorrect perceived orderings" can be mastered in addition to the "multiple observers"-problem solved by local real-time clocks. With global clocks time readings become meaningful across processor boundaries and hence reflect the actual order of event occurrence. Yet, even global-clocks fail to satisfy our needs in practice, because the remaining two problems are still observable - subsequent executions may still yield different orderings and arbitrary orderings are still imposed on independent events. In fact, to answer the questions of whether one event must precede another in a particular computation with certainty would require an unbounded number of tests (see [Fidg 93] for a discussion).

Totally ordered logical clocks:

The reason for the problems of global real-time clocks is the use of absolute time for event ordering. The timing mechanism and in particular accesses to the global time are randomly influenced by a variety of hardware and operating system factors (e.g. different processor loads, different start times,...). The solution is to replace global clocks with logical clocks, which are a more objective and relative ordering mechanism. A very simple approach to logical clocks has been described by Lamport in [Lamp 78], which are often called Lamport clocks for that reason. This method is based on the following rules:

- Each process maintains an integer counter, the logical clock.
- This logical clock counter is incremented whenever an event of interest occurs.
- If the event is a send event, the process' current logical clock is "piggybacked" to this message.
- If the event is a receive event, the logical clock of the receiver is set to be greater than its current value and that of the piggybacked value in the message.

An example for the Lamport ordering mechanism is drawn in Figure 5-5. This graph represents the same execution as given in Figure 5-2, with the values of the logical clocks at each corresponding event. In this particular example, the initial values of the counter are assumed to be zero, while the increment at each event occurrence is chosen to be one[2]. Therefore, the values

2. Please note, that the amount of increment is not necessarily one but can be any positive number.

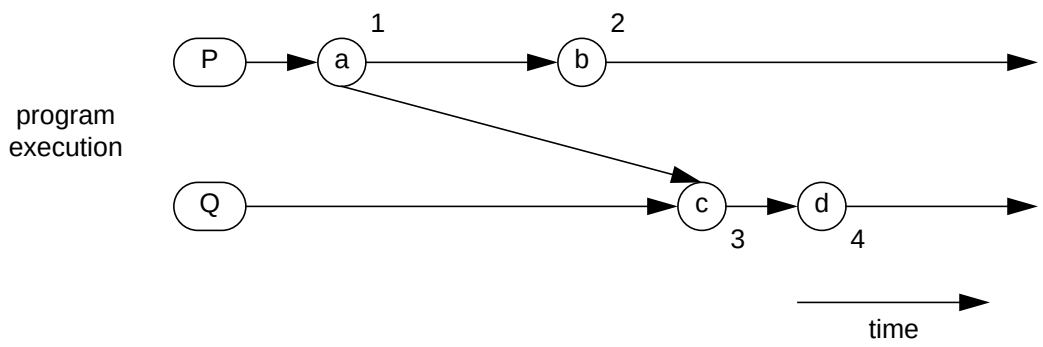


Figure 5-6. Possible total order for Lamport's approach of Figure 5-5

of event *a* and event *b* are obvious, and all the events in an arbitrary sequential programs would simply be ordered by rising numbers. On process *Q*, however, the receive event *c* is given timestamp 2 rather than 1, because it must have a higher value than the corresponding send event. This adjustment is based on the counter value of event *a*, which is piggybacked to the message transferred from process *P* to process *Q*.

Based on this algorithm, a total order is completed by adopting an arbitrary but consistent ordering among the processes, whenever two events have the same timestamp, as it is the case for event *b* and event *c*. A possible total order of events for the example of Figure 5-5 is given in Figure 5-6. In this example, the order adopted between events on different processes is established, so that events on process *Q* receive a value bigger than those on process *P*. In our case, event *b* gets total order 2 while event *c* gets total order 3.

The advantage of this approach compared to real-time clocks is, that it also precludes the possibility of the effect, that the "Same computations exhibit different orderings". Totally ordered clocks provide consistent relative timestamps associated with each event no matter how many times the computation is re-executed, and regardless of differences in the absolute timing. While this is already an improvement for testing and debugging, it still requires a record&replay mechanism for nondeterministic programs. Such record&replay techniques are in turn often implemented based on a similar approach to event ordering. For that reason and due to the simplicity in implementing totally ordered clocks, this method is actively being used in many parallel debugging systems.

There is still one observability effect remaining, namely that "Arbitrary orderings are adopted". In our example of Figure 5-6 this would be the case for event *b* and event *d*. Using the totally ordered clocks as described above, the user would mistakenly conclude, that event *b* always occurs before event *d*. However, in reality there is no interaction between process *P* and *Q* to guarantee this behavior, nor is there any causal relation between event *b* and event *d*, which may be concluded from an ordering mechanism in general. As a result, this misleading view may thwart attempts to debug problems stemming from inadequate synchronization between these events [Fidg 93].

Partially ordered logical clocks:

In order to overcome all four observability problems, Fidge proposes the usage of partially ordered clocks. With this timestamping method, the incomplete view of event causality as established by totally ordered logical clocks can be prevailed. This idea of partially ordered clocks is a straightforward extension of the method described above and was proposed by

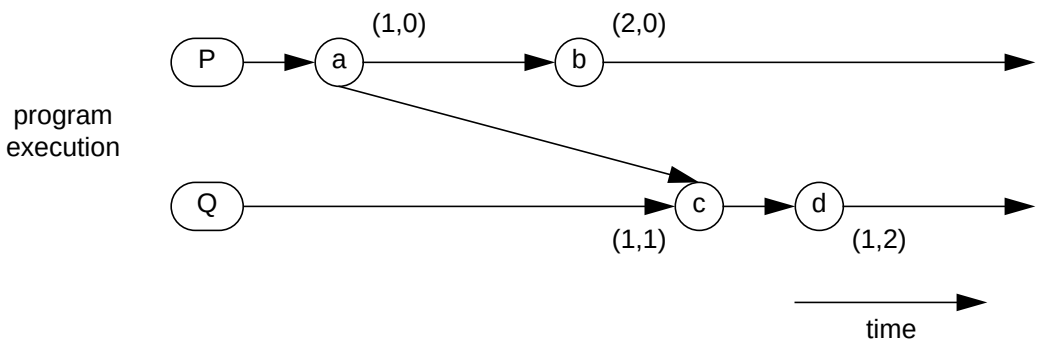


Figure 5-7. (Partial) logical vector clock ordering [Fidg 93])

Fidge [Figd 91] and Mattern [Matt 88]. The basic idea of this method is based on the following rule-set:

- Each process maintains a vector of clock counters with one element per process.
- Whenever a process performs an event of interest, its own counter value in its own vector is increased.
- If the event is a send event, the complete vector-clock is piggybacked to the transmitted message.
- If the event is a receive event, the receiver sets each element in its own vector-clock to the maximum of the current value and the corresponding value of the element in the piggybacked vector.

An example for this method is given in Figure 5-7. Again, the graph represents the program's execution from Figure 5-2, but this time both processes *P* and *Q* maintain their own vector-clock (given in parenthesis at each event). In each vector the first element represents the value of process *P*, while the second element contains the timer of process *Q*. The increments determine the ordering of the events during the program's execution.

This basic approach needs two remarks. Firstly, a vector with fixed length indicates that it is intended only for fixed numbers of processes. However, Fidge also describes an extension to his method, which would allow an application for systems with dynamic process creation (see [Figd 91] for more details). Secondly, the approaches described in [Fidg 89] and [Fidg 93] differ in one point, namely the rule for the vector-clock update at the send event. The rule described above was taken from [Fidg 93] and to our knowledge represents the latest in his series of publications about vector-clocks. The difference to his first paper on this topic is, that the sender increments its own counter at each send event, but adds another increment to the vector clock that is piggybacked to the outgoing message. This additional one in the outgoing vector is intended to represent the transmission time between the messages origin and its destination [Fidg 89]. While this minor difference has no effects on the results of the event ordering, it still needs to be addressed when implementing a monitor with a vector-clock ordering mechanism.

After these vector-clocks have been generated, each of them represents the timestamp for the corresponding event. Obviously, each vector is unique and may only occur once during execution. In addition, these vectors can be applied to determine for any given pair of events *x* and *y*, whether *x* preceded *y* or vice versa. This order is established by the following comparisons [Fidg 93]:

Condition 5-1: Precedence condition

Event *x* preceded some event *y* if and only if

- event *y* has a counter value for the process in which *x* occurred greater than or equal to the value of that process at event *x* inclusive, and
- event *x* has a counter value for the process in which *y* occurred strictly less than the value of that process at event *y* inclusive.

A proof, that the precedence condition holds for the vector clock ordering approach, is offered by Fidge in [Fidg 88]. There he shows, that this approach correctly implements the desired partial ordering for asynchronous and synchronous message passing systems.

For the example of Figure 5-7 we can conclude that event *a* preceded event *d* because event *d* knows of the occurrence of the counter value of event *a* in process *P*. Similarly we know, that event *a* precedes event *b* and event *c* precedes event *d* for the very same reason. So far, these conclusions can also be reached using totally ordered clocks. However, where the totally ordered clocks model assumed that event *b* preceded event *d*, the partially ordered model does not. In fact, with partially ordered clocks we cannot show that event *b* preceded event *d* or vice versa, but instead we may use the term “unordered” to indicate their temporal and causal relation. This means, that these two events are independent actions that (in global time) may occur in either order or even simultaneously. This situation of two independent events is either called “concurrent” as in [Lamp 78] or “simultaneous” as in [HaWe 88].

The main advantage of partially ordered clocks in contrast to all other approaches is, that partially ordered clocks overcome all the observability problems, because they reflect only “enforced” causal orderings and make the “absence” of ordering explicit [Fidg 93]. Therefore, they are very useful, especially for analysis tasks. However, there is also two problems associated with partially ordered clocks: The first problem is the amount of memory required to hold the vector clocks during monitoring. If one considers 8 bytes for each clock value, and a program running on 1000 of processes, each process would require 8000 bytes just for monitoring the timestamps. These 1000 items would have to be updated at each receive event. As a consequence, many tool developers do not apply vector clocks during monitoring, but instead stick to simply Lamport clocks with all their disadvantages. As we will see in the remainder of this work, vector clocks are needed for some special analysis tasks, but we will also show, how vector clocks can be generated post-mortem from an existing event trace (see Section 6.5.2).

The second problem associated with vector clocks is the visualization as space-time diagrams. Displaying a partially ordered set of events as a space-time diagram requires to adopt a global ordering, which of course represents only one of the possible global orderings. In fact, in every geometrical diagram time appears to be misleadingly spatialized, and we need to be especially careful with any kind of program visualization, simply because it represents an ordering that does not reflect the required freedom [Matt 92].

Please note, that the problem of event ordering is also important in many other areas where concurrent events may occur, e.g. network protocols ([Rayn 88], [Rayn 00]). For distributed databases, the original vector clocks approach has been augmented with another dimension into so-called matrix clocks ([WuBe 84], [SaLy 87]), which are used to discard obsolete information in distributed transactions. An interesting comparison of these different approaches, scalar time, vector time, and matrix time, as well as their efficient implementations is provided by Raynal et al in [RaSi 96].

5.3.3 The "Happened Before" Relation

The discussion about the "observability problems" and their possible solutions with a variety of event ordering methods leads to the conclusion, that an debugging tool analyzing program states and their relation must somehow exploit partially ordered clocks. Other ordering and timing mechanisms may be provided in addition, but their results may only accompany a generated partial order.

As side effect of this partial ordering, we are able to study the causality between events, which is our main concern because it represents the ability of one event to affect another. However, causality is much more powerful than any particular time-frame, because it helps to reason about the programs execution without limiting it to discrete values of time. Consequently, we will review e a formal method to describe the causality between events, which was introduced by Lamport during his work on event ordering. This formal mechanism is the so-called "happened before" relation, which can be defined as follows:

Definition 5-4: Happened before relation [Lamp 78]

> The "happened before" relation denoted as " $\rightarrow$ " on a set of events of a system is the smallest transitive, irreflexive relation satisfying the following two conditions for arbitrary events e_p^i :
>
> (1) If e_p^i and e_p^j are events on the same process *p*, and e_p^i occurs before e_p^j , then
>
> $$e_p^i \rightarrow e_p^j \ .$$
>
> (2) If e_p^i is the sending of a message by process *p* and e_q^j is the receipt of the same message by another process *q*, then
>
> $$e_p^i \rightarrow e_q^j \ .$$

Condition (1) is simply the sequential order $\overset{S}{\rightarrow}$ of events as defined in Definition 5-2, while condition (2) describes the relation between corresponding events on different processes. The latter may also be called the concurrent order $\overset{C}{\rightarrow}$, which is established by communication and synchronization between parallel processes.

Another requirement of the happened before relation is transitivity, which allows to establish relations between events across other interleaved events. This means that

$$\text{if } e_p^i \rightarrow e_q^j \text{ and } e_q^j \rightarrow e_o^k \text{ then } e_p^i \rightarrow e_o^k$$

Consequently, the happened before relation may also be defined as the transitive, irreflexive closure of the union of the two relations $\overset{S}{\rightarrow}$ and $\overset{C}{\rightarrow}$ as follows:

$$\rightarrow = \left(\overset{S}{\rightarrow} \cup \overset{C}{\rightarrow} \right)^+$$

In addition, we assume that $\neg\left(e_p^i \rightarrow e_p^i \right)$ for any event e_p^i, simply because systems in which an event can happen before itself do not seem to be physically meaningful. This also implies, that $\rightarrow$ is an irreflexive partial ordering on the set of all events in the system. Thus,

$$e_p^i \rightarrow e_q^j \text{ means, that } e_p^i \text{ preceded } e_q^j \text{ and } e_q^j \text{ occurred after } e_p^i .$$

A very important thing about this view is also, that

$$e_p^i \rightarrow e_q^j \text{ describes the possibility for event } e_p^i \text{ to causally affect event } e_q^j ,$$

or in other words,

$$\text{event } e_q^j \text{ may be causally affected by event } e_p^i .$$

An event e_q^j is causally affected by event e_p^i, if the state of process q at event e_q^j depends on the operation carried out at event e_p^i. This reminds us of several issues describe in Section 3.2.6. Firstly, we have defined a program's state during execution (see Definition 3-8 on page 64), and the variables used for the corresponding operation. With the happened before relation we are able to describe program states, that may contain the same set of variables, or instructions, that operate on intersecting variables. Secondly, we introduced the notions "errors of origin" and "subsequent errors". With the happened before relation we can describe, that an error of origin always happened before its subsequent set of errors, or in other words, that the subsequent errors are effected by the corresponding error of origin.

The remaining group of events are those, that do not affect each other or are not related in terms of the happened before relation[3]. Such events are reflexively defined as concurrent events:

Definition 5-5: Concurrent events

Two distinct and independent events e_p^i and e_q^j are concurrent denoted as $e_p^i \parallel e_q^j$ if

$$\neg\left(e_p^i \rightarrow e_q^j \right) \wedge \neg\left(e_q^j \rightarrow e_p^i \right)$$

5.3.4 Event Time Zones

With the simple rules given in Definitions 5-4 and 5-5, we can also characterize each event and its temporal relation to all other events in a parallel program. Obviously, for any selected event e_p^i, every other event may be either in the past or in the future, or it occurs concurrently. These attributes resemble different (logical) time zones and may formally defined as follows:

3. Please note, that sometimes the "not happened before"-relation is introduced in this context, which can be denoted by the "$\nrightarrow$" symbol.

Definition 5-6: Event time zones

Every event e_p^i establishes three time zones PAST, FUTURE, and PARALLEL for all other events occurring in a system, which may be defined as follows:

$$\text{PAST}\left(e_p^i\right) = \{e \in E \mid e \rightarrow e_p^i\}$$

$$\text{FUTURE}\left(e_p^i\right) = \{e \in E \mid e_p^i \rightarrow e\}$$

$$\text{PARALLEL}\left(e_p^i\right) = \{e \in E \mid e \parallel e_p^i \wedge e \neq e_p^i\}$$

Obviously, corresponding to one selected event, all other events of a program may always only be in one of the three timezones, which are therefore pairwise disjunct. Thus, the following rule is valid for the three timezones:

$$\text{PAST}\left(e_p^i\right) \cup \text{PARALLEL}\left(e_p^i\right) \cup \text{FUTURE}\left(e_p^i\right) \cup \{e_p^i\} = E$$

These timezones define, that all events in the set PAST may affect the current event, which in turn may affect all events in the set FUTURE. As we will see, this distinction is rather important and will help us to carry out a variety of activities for parallel program debugging, like setting breakpoints, identifying race conditions, and others. Please notice, that both timezones PAST and FUTURE may be empty sets, for example, if the first or last event on one process is selected, respectively. Similarly, the set of concurrent events PARALLEL may be empty, e.g. if all processes synchronize with one master process.

An example for the time-zones is given in Figure 5-8. It shows 20 events (*a*-*t*) occurring on four processes (P,Q,R,S). The question is to establish the three time-zones for one selected event, in our case event *c* on process *P*. According to Definition 5-6 the results of this operation are shaded and deliver the following sets:

- PAST(c) = {a,b,e,f,g,l}
- FUTURE(c) = {d,j,k,t}
- PARALLEL(c) = {h,i,m,n,o,p,q,r,s}

A similar extension is provided by Cuny et al in [Cuny 92], who introduces three more relations between events: precedes, parallels, and overlaps. These extension have been specifically tailored to the needs of their visualization system, but are in certain aspects similar to our definition of time-zones above. The precedes relation captures the notion, that one selected event occurs logically before another, which would resemble the group of events in the PAST set. Similarly, the parallels relation represents the events in the PARALLEL set. A difference is the overlaps relation, which captures the notion that events are logically intertwined, which would mean that some part of each event must occur before some part of the other. However, since

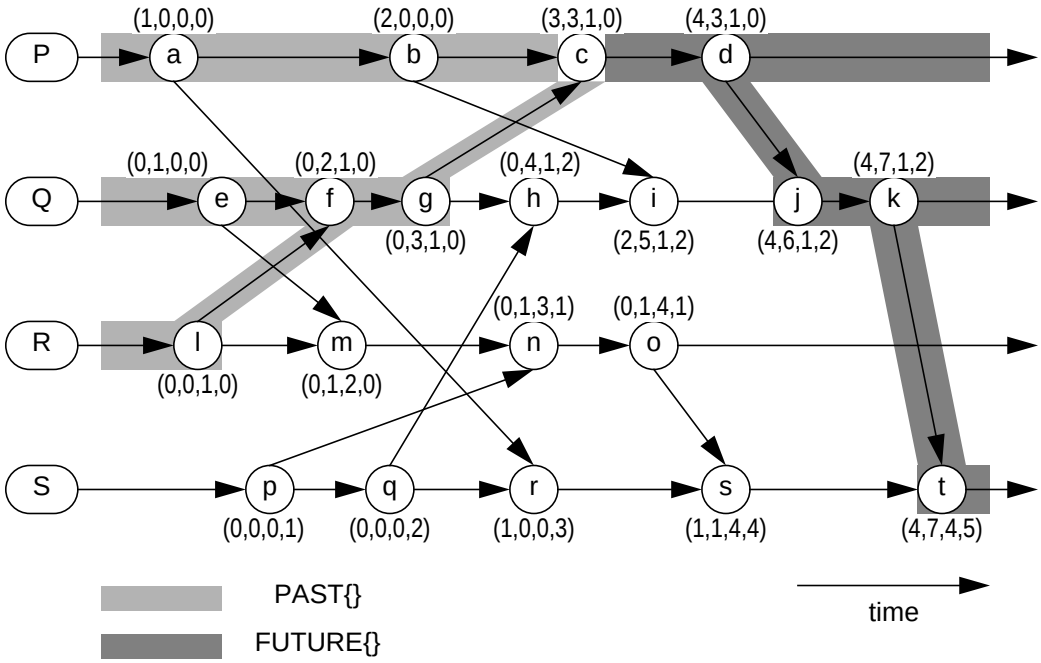


Figure 5-8. Event time-zones

events represent atomic operations or actually only manifestations of the occurrence of state changes, an overlap relation simply cannot exist in our model. However, as we will see in Chapter 10, overlapping of events may be useful when building higher level event abstractions.

5.4 The Event Graph Model

5.4.1 Abstract Definition

In this section, we will combine the events as described in Section 5.2 and the happened before relation of Section 5.3.3 to construct the so-called event graph. Such an event graph is simply a directed graph [Aho 83], which can be formally defined as follows:

Definition 5-7: Event graph [Kran 96e]

> An event graph is a directed graph $G = (E, \rightarrow)$, where E is the non-empty set of events e of G, while $\rightarrow$ is a relation connecting events, such that $x \rightarrow y$ means that there is an edge from event x to event y in G with the "tail" at event x and the "head" at event y.

The events $e \in E$ of an event graph are the events e_p^i observed during program execution. The relation connecting these events is the happened before relation as defined in Section 5.3.3. Due to the latter, events may also be characterized by the processes on which they occurred, and events on one particular process are always connected with the sequential order

relation $\xrightarrow{S}$. Similarly, events on distinct processes may be connected with the concurrent order relation $\xrightarrow{C}$, if there is some data transferred between them or one affects the other.

The vertices in such a graph also represent a state at a concrete point of time during the program's execution, while the edges are continuous due to the fact, that they represent the transition from one state change to another. Furthermore, edges are unidirectional which means that relations may always be established only in one direction. This is certainly the case for events on one process, where the edges resembles the flow of time or the sequential order, respectively. In case of data transfer between distinct processes, it may be useful to offer bidirectional communication. This is not really a problem, because any bidirectional edge can easily be replaced by two unidirectional edges, thus allowing to represent bidirectional communication features. In addition, we will later introduce higher levels of abstraction, that may offer additional types of edges connecting events in an event graph.

Besides that, the vertices of an event graph share the following properties:

- All events but the first and the last on each process have always at least one edge approaching and one edge leaving.
- Communication events or other events that initiate connections between processes may have more edges depending on the relations they are describing.

The first property is a result from the sequential happened before relation $\xrightarrow{S}$ established between the events on one process. Both events, the first and the last, are special events. The former will be called initial event, and can be defined as follows:

Definition 5-8: Initial event (on each process)

An event $e_p^0 \in E$ is called initial event on process *p*, if $\neg\exists\left(e_p^i \in E\right) e_p^i \rightarrow e_p^0$.

An initial event has no preceding event, and the set PAST(e_p^0) will be empty for each process. Furthermore, it is the first event being observed, which can be any event from the set of interesting events chosen by the user. Therefore, it does not necessarily resemble the initiation of the process, although this is often preferred.

Corresponding to the initial event, the last event on each process will be called terminal event. It can be defined as follows:

Definition 5-9: Terminal event (on each process)

An event $e_p^i \in E$ is called terminal event on process *p*, if $\neg\exists\left(e_p^j \in E\right) e_p^i \rightarrow e_p^j$.

The terminal event has no successor, and its set FUTURE(e_p^i) will contain no other events on the same process. However, there may be events in FUTURE(e_p^i) on other processes. As we will see, analyzing the last events on each process may already reveal potential program errors, for example, when events do not exhibit a correct number of edges to other events (e.g. in case of a program failure).

Please note, that there exist *n* initial events and *n* terminal events, with *n* being the number of processes applied by the program's execution. If processes are dynamically generated during execution, the number of initial and terminal events will equal the total number of processes that did exist sometimes during the program's execution.

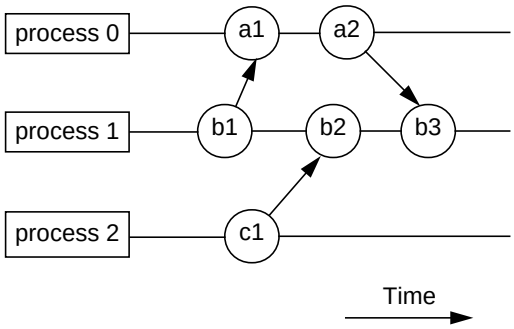


Figure 5-9. Example event graph [Kran 96a]

Remark

The existence of the initial event and the terminal event is only a characteristic of the abstract event graph model. It is not reflected in the visualized graphs contained in this work, since edges will be drawn before the initial event and after the last event. These artificial edges relate the events on one process with the name tag of the corresponding process, which is only displayed for the user's convenience.

5.4.2 Event Graph Visualization

The event graph model described above can easily be visualized as a space-time diagram. This type of visualization with all its different variations has already been discussed in Section 4.5.5. In principle, every figure in this chapter represents an event graph of some kind. Another simple example is given in Figure 5-9, which shows an event graph with three processes and six events. The events *a1* and *a2* occur on process 0, the events *b1*, *b2*, and b3 occur on process 1, while the event *c1* occurs on process 2. What is missing here, are the arrowheads at the head of the edges connecting events on each process, which would be required from the definition of the event graph. Yet, it is quite common that events occurring on one process are not connected with directed arcs, but instead to provide an indicator for the direction of time in the chosen representation. With this indicator the same information can be retrieved from the event graph visualization. In concrete, we can conclude that event *a1* happened before event *a2* and similarly event *b1* happened before event *b2* and event *b2* happened before event *b3* (and therefore event *b1* happened before event *b3*) due to the sequential order relation $\xrightarrow{S}$.

Also indicated in this event graph visualization are the connections between events *b1* and *a1*, events *a2* and *b3*, as well as events *c1* and *b2*. These events are related due to the concurrent order relation $\xrightarrow{C}$, which means, that the event at the tail of each edge affects the event at the head of each edge in some way. However, simply from this representation we cannot determine, what type of events these are. For example, in message passing systems event *a1* may be a receive event, while event *b1* is the corresponding send event. Yet, in programs based on the shared-variable paradigm, these two events may also represent the exchange of control over some shared variable that both processes are accessing. If the visualization only contains communication events like send and receive, this event graph is often called a communication graph [Grab 99]. However, a basic event graph is much more powerful than a communication graph, because it does not enforce any limits on the type of visualized events.

An important issue that must be discussed for this kind of event graph visualization is the fact, that by choosing an arbitrary visual representation the partial ordering of the abstract

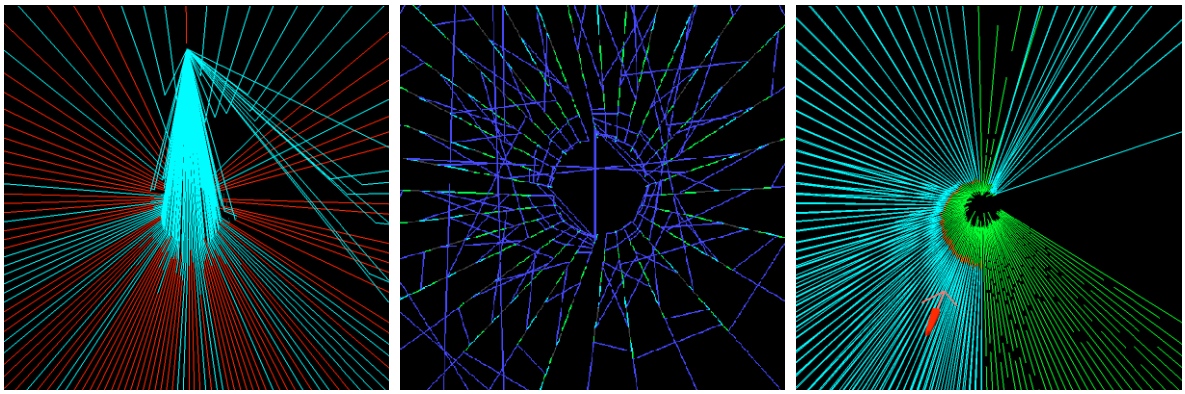


Figure 5-10. Time tunnel visualization [Reed 95]

event graph model is converted into a total ordering. Thus, such a space-time diagram displays a total ordering of a model, that is actually partially ordered. Consequently, the displayed total event order is only one of the possible orders that can be extracted from the abstract event graph model. In this example, it seems as if event *c1* happens before event *a2*, although it may event *c1* may occur at any time before event *b2* and not necessarily before event *a2* in practice.

Furthermore, please notice that the time-axis is visualized from left to right, while the process axis is arranged vertically. This is only one possible arrangement, although it is often selected by parallel debugging and tuning tools. It is also possible that the axes are turned around, or even more sophisticated like in the Time Tunnel Virtual Reality system [Reed 95]. This system uses a CAVE (CAVE Automatic Virtual Environment), which is a multi-person, room-sized, high-resolution, three-dimensional video and audio environment developed at EVL[4], for a projection of an event graph as a three dimensional tunnel [Reed 95]. Some examples for the Time Tunnel view are given in Figure 5-10. The idea behind this time tunnel approach is, that the user gets an immersive view onto the program's execution within a virtual reality environment. The processes are arranged in a circle around the center, and the time is displayed into the center of this cylinder. Using this view, the user should be able to "feel" the execution. A judgement about the usefulness of such an approach is left to the reader.

Another item is the chosen time-scale of the visual representation. Since the abstract event graph model covers only relations between objects, it is also only possible to retrieve the ordering between these events. Consequently, the distance between events on the time axis is fixed, and events on one process are apart by pre-defined intervals. This kind of visualization is often called "logical clocks"-display and has proved to be especially useful during the debugging phase. The reason is that this kind of visualization represents the causal relationship between the events and it relates events on different processes, which is required to analyze the correctness of a program [LeBl 90]. Additionally, this display does not suffer from unsynchronized clocks, where different processes with different clocks generate inconsistent timing information. A wide variety of possibilities for using logical time in visualization is offered in [Cuny 92]. There, the authors describe the advantages of using this discrete clock values for debugging, as well as their application for other domain-specific data visualizations.

However, each event may also store its actual time of occurrence in its event attributes, which may then be used in the visualization by spatially extending the edges according to the

4. For more details about the CAVE, please refer to the Electronic Visualization Laboratory at the University of Illinois at Chicago: http://www.evl.uic.edu/

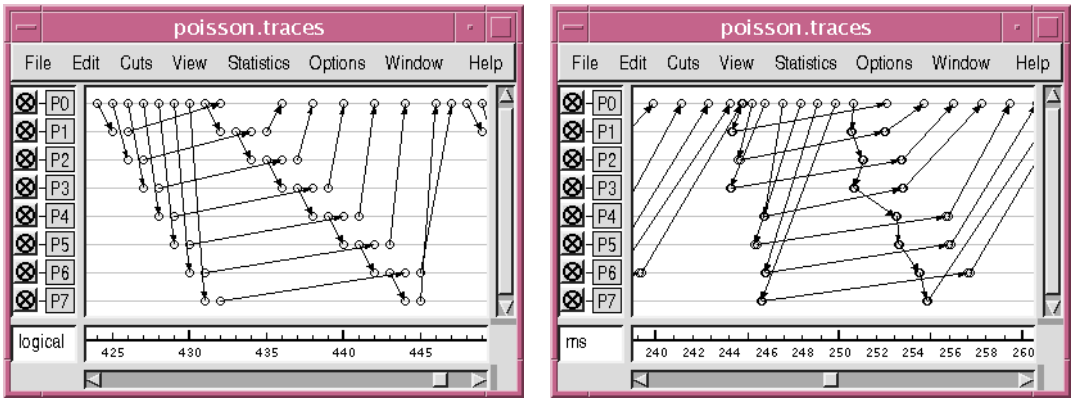


Figure 5-11. Example event graph visualization of a program execution

temporal distance between two events. This is suitable to measure the duration of operations within a process. This kind of physical time scale is usually applied during performance analysis [LeBl 90]. We can also distinguish whether these timing values have been generated by a local clock on each process, or whether they represent measurements of a global real-time clock that may be available in the system. In case of local clocks, it is only possible to use this time for the events on each process, and events on distinct processes cannot be related. This represents a major pitfall that is often overlooked or ignored.

In case of an existing global real-time clock, even events on different processes may be drawn according to their temporal distance. For example, in Figure 5-11 the distance between the displayed events resembles the time that the processes needed to proceed from one state change to another. Please remember, that this does not mean that there didn't occur any state changes or events besides the displayed events, but instead, that there may be lots of state changes represented by each edge. All this visualization tells us is the relation between our selected events of interest, which may obviously represent only a very small number of the possible state changes of a program.

An example event graph visualization for a small-scale program is displayed in Figure 5-11. These graphs have been generated with our monitoring and debugging tools from an actual program execution, in this case a parallel Poisson equation solver on 8 processes. This view represents our main interface for debugging and similar snapshots will be used in the following chapters. In Figure 5-11 the graph in the left screenshot shows the program with its logical structure, and all events are displayed within fixed intervals. The only rule is, that receive events are at least one interval after their corresponding send event. The right event graph presents the same program, but this time with actual time-stamps as measured during execution. Therefore the distance between events on one process indicates the actual amount of time. Additionally, this visualization uncovers the lack of the global clock, which results in edges pointing backwards in time. This would mean that a message is being received before sent, which is impossible.

A similar distinction for the time-scale is presented in [WeKu 98], where the authors distinguish between Lamport's scalar time, vector time or real-world time. Likewise, LeBlanc et al describe different possibilities to visualize and analyze parallel program behavior [LeBl 90]. They distinguish between logical time view and physical time view, and introduce another concept of time, the so-called phase time. This phase time is a synthetic view, since there might not be a mapping from phase time to program execution, which orders a sequence of events with respect to collections of related events or phases that occurred during execution. For

example, if a program exchanges data among processes in well-defined rounds of communication, the set of communication events in a round form a phase. As we will see, this notion will also be useful when introducing further abstraction mechanisms in Section 10.1.

5.5 Summary

In this chapter, we have described how to perform program debugging in practice. Our general idea was to extract the program's behavior during execution and analyze the occurring state changes and their relations. This is done whenever events of interest are observed, that indicate that some kind of state change has taken place. By analyzing these state changes and their relations, the reasons for errors in programs are hopefully detected.

Besides this basic idea, we have also briefly summarized the problem that the user is confronted with when analyzing program state change from a running program. Probably the biggest obstacle is the observability problem, and we have shown that only partially ordered clocks offer a suitable solution for this effect. The partial event ordering is also reflected in the "happened before" relation, which combines the set of events into our abstract event graph model. This event graph model will be used for various analysis tasks in the next chapter, while the graphical representation of the event graph as a space-time diagram will serve as the visual interface for all these debugging activities.

6. Example: A Simple Event Graph Model

In this chapter we will apply the things learned in the previous sections and describe all the steps necessary to construct an example event graph for a parallel program. The first thing is to select appropriate events of interest that should be generated during program execution. In order to accomplish this, the program's code must be instrumented to include monitoring functionality. Afterwards the monitor's operation is described which leads to the resulting event data. With these data, an event graph model can be constructed and visualized for further program analysis.

- Events of interest
- Program instrumentation
- Monitoring an execution
- Constructing the event graph
- Visualizing the graph model

6. Example: A Simple Event Graph Model

6.1 Overview of Event Modelling Approach

After introducing the theoretical basics about our event graph model, we will now describe how to generate an example event graph and its corresponding event graph visualization for an arbitrary parallel program. Since our main focus is on message passing programs, we selected a program using the MPI standard functionality for this small example. In order to build a corresponding event graph model, we have to exercise the following steps:

(1) Selection of appropriate events of interest and their attributes.
(2) Instrumentation of the program's code to generate events during execution.
(3) Execution of the program and generation of event instances.
(4) Construction of the abstract event graph model with the collected data.
(5) Visualization of the event graph model as a space-time diagram.

These steps will be described in the following subsections. Please note, that this represents only a small example and does therefore not cover the complete MPI functionality. Instead, we try to introduce how these groups of programs can be observed, how the observed data is used to construct an event graph, and how this event graph representing the program's execution can be visualized.

6.2 Selection of Events

6.2.1 Interesting Event Classes

As described in Section 4.1.3, debugging is initiated whenever incorrect results or a program failure has been observed during the testing cycle. In that case, the user instruments the program to derive analysis data about its behavior during execution. We have also discussed, that this instrumentation phase may either be placed at the beginning of the debugging cycle - as it is the case in sequential programs - or even before the testing cycle is initiated, which is necessary when investigating nondeterministic parallel programs.

Before instrumentation can be carried out, a decision has to be made about the objects of interest, or more precisely, which state information should be derived during the program's execution for subsequent analysis phases. In most cases, this decision about the interesting event classes is taken by the applied debugging tool or the tool's developers, respectively. As a consequence the complexity of the tool is reduced due to the decreased freedom, and the debugging process is expected to be simplified, because the user is released from this critical decision. However, in some cases it would be preferable that the debugging user selects the interesting objects without restrictions, simply because of the available background knowledge about the target program.

In our case, the debugging tool pre-defines a certain set of events, which are generated by the program during execution, while additionally it offers a certain degree of flexibility to inte-

grate user-defined events. The set of pre-defined events and their attributes can further be influenced by command-line switches of the debugging tool.

For our small example, we assume that the user is interested mainly in communication events and some other representative features offered by the event debugger. Instead of giving all possible event types each selected class, we will focus on abstract event classes, which are then described with one representative item of this class. The following event classes are selected:

- Nonblocking send (`send`)
- Blocking receive (`receive`)
- Nonblocking receive (`test`)
- Random number generation
- Variable inspection
- Array inspection

The first three event classes may be summarized as communication events. They are especially important in the case of message passing programs, because they represent the functionality to connect concurrently executing processes for exchanging data. These classes may not exist in shared-variable programs, or may be only of minor interest to debugger's addressing this target area. On contrary, the other three event classes may be found in almost any kind of program, and have been selected because they are interesting examples in our model, and provide a good overview of the debugging capabilities, respectively of the activities required to build an event graph. Some of the examples in the remaining chapters will apply this event graph model as a reference for further investigations.

6.2.2 Communication Events

The three selected communication events represent some of the typical functions provided in message-passing libraries as used by programmers in reality. A send function is nonblocking means, that the process initiates message transmission and returns to its control flow without checking whether the message has actually reached its target address. The transferred message may then be accepted by the target process, either with a blocking receive operation or a nonblocking receive operation. The difference between both functions is, that the blocking receive waits until a fully qualified message appears, while the nonblocking receive checks the incoming message queue and provides a return-code that indicates, whether an appropriate message is available or not. On an abstract level, these communication events may be defined as follows (compare with [Brin 95]):

Definition 6-1: Nonblocking send event

An event $e_p^i \in E$ is called a nonblocking send event, if it describes the occurrence of the following operation:

```
send(buffer, destination, tag)
```

Definition 6-2: Blocking receive event

An event $e_p^i \in E$ is called a blocking receive event, if it describes the occurrence of the following operation:

```
receive(buffer, source, tag)
```

Definition 6-3: Nonblocking receive event

An event $e_p^i \in E$ is called a nonblocking receive event, if it describes the occurrence of the following operation:

```
test(result, source, tag)
```

For these communication events, `buffer` defines the memory location of the message, and `tag` is used as a message contents identifier, often called the message type. Furthermore, the parameter `destination` of the send statement depicts the destination process of the message, while `source` of the receive and the test statement points to the origin of the message. Instead of `buffer` the nonblocking receive function test offers the above mentioned return code `result`, that indicates whether an appropriate message is available in the process' incoming message buffer or not.

6.2.3 Mapping Events to MPI Statements

Such communication statements may be available in any message passing library, which in turn may add further parameters for additional functionality. For our MPI example program, we will map this abstract communication events onto the following concrete MPI functions. These function definitions have been extracted from the MPI Message-Passing Interface standard Version 1.1, June 1995 [MPI 95] and represent the function calls for the C-language. Corresponding FORTRAN function calls are also provided by MPI.

Nonblocking send:

```
int MPI_Isend (void *buffer,
               int count,
               MPI_Datatype datatype,
               int destination,
               int tag,
               MPI_Comm comm,
               MPI_Request *request)
```

Blocking receive:

```
int MPI_Recv  (void *buffer,
               int count,
               MPI_Datatype datatype,
               int source,
               int tag,
               MPI_Comm comm,
               MPI_Status *status)
```

Nonblocking receive (and corresponding operation completion):

```
int MPI_Irecv (void *buffer,
               int count,
               MPI_Datatype datatype,
               int source,
               int tag,
               MPI_Comm comm,
               MPI_Request *request)

int MPI_Test  (MPI_Request *request,
               int *result,
               MPI_Status *status)
```

Mapping these MPI functions to the abstract functions given above is straightforward. The additional parameter `count` indicates the number of elements of type `datatype` in the message, which means that `count*sizeof(datatype)` bytes are transferred in the message. The MPI-communicator `comm` specifies the communication context for a communication operation, which can be the "communication universe" (=`MPI_COMM_WORLD`), or it is used as a method to address process groups. The parameter `request` of the `MPI_Irecv` and the `MPI_Test` function will be described in more detail below.

The receive operations also offer a parameter called `status`, which represents the `MPI_Status` structure as defined below:

```
typedef struct {
               int MPI_SOURCE;
               int MPI_TAG;
               int MPI_ERROR;
               int size;
               int reserved[2];
} MPI_Status;
```

With this structure, information about the received message can be obtained. The value of `MPI_SOURCE` indicates the message's origin, while `MPI_TAG` defines the provided message identifier. `MPI_ERROR` contains any errors occurred during message transfer, and size contains the number of bytes in the message. Unfortunately, this definition of MPI-status is not included in the MPI standard, which lead to manifold different implementations in the past. In fact, this single structure is often the main reason for problems when porting MPI software from one machine to another. The definition given above has been taken from the SGI Origin 2000, which at the time of writing offered MPI based on the standard Version 1.2 within its Cray Message Passing Toolkit MPT.

In order to understand the usage of these functions, some more information is required: Firstly, the `MPI_Isend` function performs a nonblocking send operation (the prefix I indicates "immediate"). Nonblocking means, that the send operation is initiated but the calling process does not wait for its completion. According to the MPI definition, a send operation is completed, when the message was copied out of the send buffer. Please note, that this does not mean, that the message was actually received by its destination process. Furthermore, with suitable hardware, the transport of the data out of the sender memory may processed concurrently with computations done at the sender after the send was initiated and before it was com-

pleted. In order to ensure that the send operation was completed, the functions `MPI_Wait` and `MPI_Test` are available, where the former returns only when the operation was actually completed, while the later returns a flag describing the status of the operation (completed = `true` or `false`).

In practice, user's often prefer the basic point-to-point communication function `MPI_Send` over `MPI_Isend`, and depending on the implementation, `MPI_Send` may even offer the same functionality. However, there is a minor difference which mainly stems from MPI's definition of blocking and nonblocking. In the common sense nonblocking means that the send operation may return before the receiving process actually accepts the message, and this can be achieved with both MPI send functions. According to the MPI definition, an `MPI_Send` is blocked means, that it does not return until the message data and envelope have been safely stored away so that the sender is free to access and overwrite the send buffer. In practice, the message can be copied either directly into the matching receive buffer or into a temporary system buffer for later transmission. In certain implementations it may even be necessary, that the `MPI_Send` waits until a corresponding receive operation has been started. Thus, it is better to apply the `MPI_Isend` operation, if a "real" nonblocking send is required.

The second function, `MPI_Recv` represents the standard point-to-point blocking receive function. In that case, the function only returns when an appropriate message has actually arrived in its message buffer. Of course, this implies that the program may be blocked forever, if no suitable message is delivered. As we will see in Section 7.2.3, this is a critical source for errors in message passing programs.

In contrast to `MPI_Recv`, the function-pair `MPI_Irecv` and `MPI_Test` represent the nonblocking receive operation, which can be achieved by applying both functions. The `MPI_Irecv` function is used to allocate a communication request object and associate it with the request handle identified by argument `request`. Afterwards, this request handle can be used to query the status of the communication with `MPI_Test` or to wait for its completion (with the `MPI_Wait` function). Thus, the `MPI_Irecv` function only indicates that the system may start writing data into a receive buffer. Consequently, the receiver should not access any part of the receive buffer after a nonblocking receive operation is called until the receive completes. The receive operation is completed, as soon as a call to `MPI_Test` returns the result code `true` for the corresponding communication request handle. In such a case, the status packet contains information on the completed message transmission operation.

Please note, that this set of functions available in MPI is usually also available in other message passing interfaces. For example, in the Parallel Virtual Machine PVM, `pvm_send` represents a nonblocking send, `pvm_recv` represents a blocking receive, and `pvm_nrecv` represents a nonblocking receive [Geis 96]. However, as we have seen the mapping of the MPI functions to our abstract communication events is not always direct, but instead may require to introduce more than one function to obtain the requested functionality. However, in other message passing libraries it is even possible, that a direct mapping between the abstract operations and the actual implementation may be available. This is for example the case on the nCUBE 2 multiprocessor ([nCUB 90], and [nCUB 92]), which offers one concrete function for each of our abstract operations (nonblocking send = `nwrite`, blocking receive = `nread`, nonblocking receive = `ntest`).

6.2.4 Communication Event Data

During program analysis, there are especially two issues that are important for the user. On the one hand, it is necessary to know about the occurrence of an event. On the other hand, it is use-

ful to provide some properties describing the events occurrence. The first issue addresses the place in the program that actually generated the event. As described in Section 5.2.1, there are several places where events may be generated. For example, assume that the user placed a call to the `MPI_Recv` function somewhere in the code, which is executed sometimes during execution. This would cause a blocking receive event as specified in Definition 6-2 to be generated, which could occur at several places. Some examples are given in Table 6-1.

Table 6-1: Example places for event occurrence

	level	place
time ↓	user program	before `monMPI_Recv`
	monitor	before `MPI_Recv`
	MPI-library	before `PMPI_Recv`
	low level communication-library	before operating system call
	operating system	before access to message buffer
		before message copying
		after access to message buffer
	low level communication-library	after operating system call
	MPI-library	after `PMPI_Recv`
	monitor	after `MPI_Recv`
	user program	after `monMPI_Recv`

The example provided in Table 6-1 assumes no particular MPI implementation and is only a rough extract from what may actually happen on a particular machine. In fact, there are usually several other activities involved in receiving a message. Some examples are allocation of memory, checking of message availability, error checking, and others.

Another interesting feature included in Table 6-1 is the fact, that the operating system offers several places where the event may be generated. Firstly, it may cause an event before anything in the message buffer is accessed. Secondly, it may generate an event when message copying is initiated. The delay that may occur between these two places is of major interest, since it is partially determined by the amount of blocking that occurred during program execution. Thirdly, the operating system may generate an event, when the message buffer is no longer needed. In practice, a debugging tool may even generate several events whenever a function of interest is executed. Some more details are provided in Section 6.4.1.

Additionally, it is not only important to know about the occurrence of an event, but also about its properties. These properties are called event attributes and represent everything that may be interesting for the investigator. Similar to the events, the set of attributes may be appointed by the user. In case of communication events as specified above, the user may choose to store the parameters `destination`, `source`, and `tag` as event attributes. Saving the message at address `buffer` itself seems useful at first glimpse, but results in a major

drawback due to its memory requirements and the resulting tracefile sizes. Therefore it is usually omitted, which slightly reduces the debugging potential [McHe 89].

Additional event attributes may be appointed as well. Some ad-hoc examples are real-time measurements, which may be included to deliver the event occurrence time, the duration of the associated function, or other measurements like the monitor's delay. Similarly, event attributes like source code reference, consisting of a source file name and a line number, may be useful. Some examples will be given in Section 7.4.

6.2.5 Other Events of Interest

All the selected, interesting events besides these communication operations deal with concrete memory contents during some point of the program's execution. However, there are some distinctions between these operations, that we will briefly describe in the following paragraphs.

One of these interesting events is random number generation, which is performed with the function call `random()`. This function uses a non-linear additive feedback random number generator, which employs a default table of long integers to return successive pseudo-random numbers in a pre-defined range. Besides the fact, that the returned number may be interesting for the follow-up computations of the target program, random number generation is also one of the sources of nondeterminism in a program (see Section 4.2.5). Thus, we will discuss it and how to address the nondeterminism problem in more detail later. For now, we add the following function call to the events of interest:

Random number generation:
```
long int random(void);
```

The remaining two events of interest also deliver the values of concrete memory locations, but in contrast to random number generation, they represent completely artificial events, because they are inserted by the user just for debugging or analysis purposes and would otherwise not be included in the code. The goal of both functions is to return the value of some data, whenever the program execution arrives at the following statements:

Variable inspection:
```
monVARTRACE (varAddress, varType, varSize, varNumElems)
```

Array inspection:
```
monARRAYTRACE (arrAddress, arrInfo)
```

The first operation, `monVARTRACE`, is inserted by the user whenever the value of some concrete variable should be retrieved. The variable is defined by the arguments to `monVARTRACE`, which identify the memory location of the variable (`varAddress`), the fundamental data type of the variable (`varType` = `char`, `float`, `int`,...) and its actual size as a number of bytes in memory (`varSize`). Additionally, it is possible to specify the number of elements (`varNumElems`), that should be accessed from the executing program's memory. Therefore, it is also possible to retrieve complete arrays for inspection.

The difference between `monVARTRACE` and `monARRAYTRACE`, which also offers a possibility to access arrays in memory is, that `monARRAYTRACE` operates across process boundaries. This means, that arrays can be inspected even if they are distributed across the processes of the target program. For that reason, the operation `monARRAYTRACE` requires a parameter

with a pointer to the start address of the array in memory (`arrAddress`), and a structure describing the layout of the array (`arrInfo`). Obviously, both arguments may differ for each of the processes in the running program, and the global view of the array is constructed whenever each process performs the `monARRAYTRACE` operation. The layout of the array can be described with various characteristics like data type of the elements, memory size of each element, number of array dimensions, and number of elements in each array dimension. Besides that, the layout structure `arrInfo` also requires to specify how the array is distributed across the processes, so that an analysis tool can combine the array portions on each process into one global array. The necessary parameters in `arrInfo` are the number of elements in each dimension on each process, the number of processes per array dimension, and the amount of overlapping at neighboring processes. In addition, the user may specify the array distribution in an HPF-like distribution scheme (e.g. BLOCK, CYCLIC, or no distribution) as well as possible remainders if the number of elements cannot be divided by the number of processes without remains. Additional distribution schemes may be added in the future.

With these events of interest, we may already retrieve some interesting analysis data during an arbitrary program's execution. However, as mentioned before, these are only some examples of how to describe interesting events, that are expected to occur in a program and which the user might want to analyze during debugging.

6.3 Instrumentation of the Code

6.3.1 Applying C-Preprocessor Macros

After the interesting state changes have been identified, the target program must be prepared to include the necessary monitoring code. This preparation part is called instrumentation phase, because it requires to add probes to the program that perform the monitoring activities. The basic idea of instrumentation is, that all the original statements to be observed are exchanged with monitoring statements, that effectively perform the same activity while additionally generate the important event data. Thus, any target function of interest is exchanged with the corresponding monitoring function, which in turn calls the original target function in its monitoring code. For example, assume our target function is the above mentioned nonblocking send, in concrete the `MPI_Isend` function of the MPI standard. Instrumentation means, that at each place in the code where `MPI_Isend` is called, it is exchanged by the corresponding monitoring function, which we may call `mon_MPI_Send` instead.

In our implementation, this exchange of the original functions with the monitor's functions is carried out by the C-preprocessor [KeRi 88]. In practice, this means that for each event of interest, the monitor provides a macro-definition, which maps the original function onto a monitor function as follows:

Example 6-1: C-preprocessor macro for instrumentation

This example shows, how the original `MPI_Isend` function is replaced by its corresponding `mon_MPI_Send` function. The replacement is done by the standard C-preprocessor, that is called during the first phase of the C-compiler. Its main intention is to allow inclusion of external files and the replacement of symbols by strings. The first include statement is used to insert the definition file `monitor.h` in the target code, which contains all the required macros and constants for the monitor's functionality. One part of this definition file contains all the macros, which substitute the orig-

inal functions with monitor functions as given below. Thus, all the preprocessor does is to textually substitute one function call for another, which is exactly what should be done by instrumentation.

```
#include <monitor.h>

#define      MPI_Isend(buffer,count,datatype, \
                       dest,type,comm,request) \
             \
             mon_MPI_Send(__LINE__,__FILE__,MPI_ISEND, \
                       buffer,count,datatype, \
                       dest,type,comm,request)
```

In this example we see, that the original `MPI_Isend` function with all its parameters is mapped on the `mon_MPI_Send` function. Besides the original parameters, the latter also contains some additional arguments called `__LINE__`, `__FILE__`, and the constant `MPI_ISEND`. The parameters `__LINE__` and `__FILE__` are used to establish a source code connection, which is provided by the compiler, because both identifiers are replaced by their actual values at each line this substitution takes place. Thus, whenever the C-preprocessor detects an `MPI_Isend`, it is replaced by `mon_MPI_Send` and the first two arguments are replaced by the actual line number and the corresponding filename of the original statement. This simple trick is very effective and is much less difficult to implement, than a method analyzing the program's symbol tables.

The third argument, `MPI_ISEND`, is required to identify the original communication function. It serves two purposes: Firstly, it is used as a constant for the event that is generated during execution. Thus, this third parameter is always unique for each observable event. Secondly, this parameter allows a simplification, which reduces the implementation work of the monitor. As described above, the original `MPI_Isend` function is mapped to a `mon_MPI_Send` function, instead of a `mon_MPI_Isend` function. This indicates, the both the `MPI_Isend` and the `MPI_Send` function apply the same monitoring code, and the constant parameter is used to distinguish both functions in the monitor's implementation of `mon_MPI_Send`.

6.3.2 Possibilities for Program Instrumentation

This method for instrumenting code is rather simple but powerful. In fact, in many cases this approach is already sufficient, although sometimes more advanced monitor implementations may be preferred [Holl 95]. In the scope of this work, we have also experimented with other techniques for program instrumentation. We may distinguish the following approaches:

- Application of an instrumented library.
- Textual substitution by a parser.
- Textual substitution with macros of the C-preprocessor.
- Inclusion of an additional library.
- Replacement of the original library.
- Replacement of resident run-time functions.
- Binary wrapping of object code.
- Dynamic instrumentation of executable code.

The simplest approach is to apply a library, which already contains all the necessary instrumentation. In that case, the usage of the library implicitly means, that the program generates

monitoring data during execution. In most cases, the generation of event data can also be disabled with some kind of control switch. An example for this method of implementation is offered by the Portable Instrumented Communication Library PICL [Geis 90], which addresses portability of code besides the instrumentation. A programmer applying PICL can not only monitor the program during execution, but could also execute the program on any platform supporting PICL.

The characteristic of portability and hardware independence is also one of the advantages of the second and third techniques, which both operate by textually substituting the original communication function. The drawback of the parser solution compared to the macros approach is, that in addition to the monitoring functions a parser for the corresponding language has to be implemented, which can be rather difficult in itself. The main disadvantage of these techniques is, that they provide a rather high level of monitoring overhead, and both approaches require another compilation step of the program.

The fourth and fifth instrumentation techniques, inclusion of an additional library and replacement of the original library, do not require another compilation step. However, each of them requires the program to be linked again, and both cases operate by manipulating the target system library (which in the case of a debugger for MPI programs is the MPI library). Actually, the possibility to include an additional, instrumented library is one of the features of MPI, that has been included in the standard and is called the "profiling interface" [MPI 95]. The idea of the MPI consortium was to allow programmers to insert a monitoring library between the defined function call and its actual representation in the communication library. This is achieved with a name shift mechanism, and all MPI function (e.g. `MPI_Isend`) are also accessible with the prefix "PMPI_" (e.g. `PMPI_Isend`). Those MPI functions that are not replaced by a monitor may still be linked into an executable image without name clashes. As a consequence, it is possible for the developer of a monitoring system (in MPI standard terms "profiling" system) to intercept all of the MPI calls made by the user program. With this interception, all the required event data can be collected before calling the underlying MPI implementation (through its name shifted entry points).

The fifth method, replacement of the original library, operates basically on the same idea as the MPI profiling interface. However, in this case there is no additional library added to the code, but the original communication library is replaced completely. This also requires that the original functionality (e.g. the message-passing operations) must be coded in the replacement library, which is the main reason why this method is not often applied in reality.

Similarly, the method to replace the resident run-time functions includes some critical tasks. This method assumes that the low-level communication functions are implemented as interrupt service routines. If this is the case, the communication library calls a corresponding system interrupt, whenever one of its functions requires this basic capabilities. Then, some of the parts of an operation are carried out by the operating system, which means that the code for these operations does not need to be included in the user's program. One example would be, that the communication library prepares the message in an outgoing message buffer, and calls an interrupt to initiate message transmission. The actual transmission of the message is then carried out by the operating system in combination with the system hardware, or as it is the case on the nCUBE 2 multiprocessor, completely independent of the CPU by the network communication unit NCU [nCUB 90]. If interesting events are generated by operations using this method, a possible way to intercept the program's execution is to redirect the interrupt to another interrupt service routine, which contains all the monitoring functionality required by the user. This technique is quite common on small scale personal computers, but represents a multitude of

problems for user's on multitasking, multiuser operating systems like UNIX, simply because of the possible impact of exchanging operating system function.

The remaining two possibilities for code instrumentation, binary wrapping of object code and dynamic instrumentation of executable code, are actually enhanced version of the textual substitution techniques. In fact, both methods operate by changing the executable code of a program, which is done by substituting the bytes of the object code. The difference between these two techniques is, that binary wrapping operates on the static object code that is stored on the computer before the code is being executed [CaMi 92]. Comparable tools dealing with binary instrumentation are ATOM [SrEu 94], EEL [LaSc 96], and the Just in Time Instrumenter JiTI ([RoDe 98], [Rons 00]).

On contrary, the dynamic instrumentation approach tries to replace the executable code of a program, while the code is actually executed on the computer system [Holl 94]. In fact, binary wrapping is something like the predecessor of dynamic instrumentation, and both were originally developed by the research group around Miller at the University of Wisconsin and Hollingsworth, who is now at the University of Maryland. The current results of this work are available to the public as the so-called DyninstAPI [BuHo 99], which has been developed as an integral part of the Paradyn performance analysis environment [Mill 95a]. In addition, several projects around the world apply the DyninstAPI for a variety of applications[1].

The main drawback of these binary substitutions methods is their requirement to know the structure of the available object code format. Although object files contain in principle the same information, there are many different ways to describe a program on that level. Consequently, there are manifold different object file formats (e.g. ELF [Nohr 94] and COFF [Girc 88]), and the corresponding description may even change between versions of an operating systems. Thus, the application of DyninstAPI and any other dynamic instrumentation technique is always limited to the object file formats, that are actually supported by the provider of the instrumentation. Currently, DyninstAPI Release Version 1.2 supports the following operating systems: Digital UNIX on Alpha, IBM AIX on PowerPC, SUN Solaris on SPARC and Intel x86, and Windows NT.

Other instrumentations are imaginable, and a range of techniques has been described with hardware or hybrid monitors (see Section 4.5.3 for a discussion). In [Holl 95], the authors describe operating system instrumentation and hardware instrumentation in more detail. The paper [WaRo 95] tries to offer a systematic and structured approach to instrumentation and monitoring in general, and compares several existing systems based on their characteristics. In [HoMi 99], Hollingsworth and Miller try to offer an outlook of instrumentation techniques and associated problems for metacomputing systems. However, the overview presented above offers a brief but sufficient summary of the possible instrumentation techniques of software monitors, that are applied in currently available observation and program analysis tools.

6.4 Monitoring the Program's Execution

6.4.1 Basic Operation

No matter what kind of instrumentation is applied, the result is always a program which is extended to include monitoring code. This means, that original statements are encapsulated by

1. Details about the DyninstAPI can be found at the URL http://www.cs.umd.edu/projects/dyninstAPI/

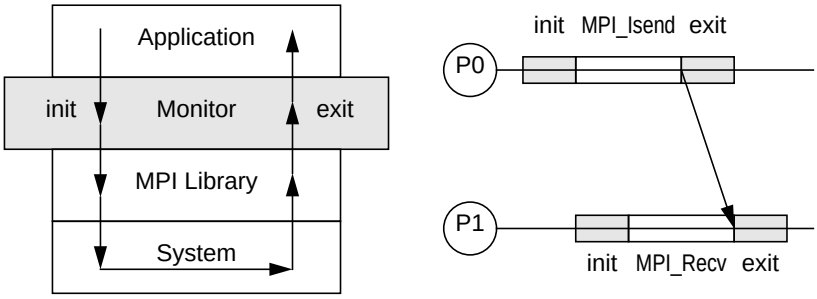


Figure 6-1. Monitoring functionality during program execution

monitoring statements, so that whenever the instrumented program would perform one of the pre-defined statements, it would first initiate the monitor which in turn calls the original functionality. This behavior is also documented in Figure 6-1. The left diagram shows a layered approach of the program during execution. The top layer in this diagram is the application layer, which contains the code as implemented by the user. Sometimes during execution, this application would initiate a function from the MPI library, which in turn would apply some of the systems function to carry out the desired functionality.

The difference between the original program and its instrumented counterpart is, that in the original code the application would call the functions of MPI directly. Yet, when the code is instrumented, all the interesting function calls to MPI are replaced with function calls into the monitoring library, so that the application does not access the MPI library without the monitor's intervention. Whenever the monitor's functions are activated, data about the program's current state can be obtained. Since the program should still perform the same functionality, the monitoring library must somehow incorporate the MPI library to perform the original functionality. After the operation, the MPI library does not return to the application, but instead replies all its data to the monitor, and some more monitoring functions can be performed afterwards again. The MPI library's results will be forwarded to the application by the monitor, which should not recognize any differences during operation with or without monitoring.

The right diagram of Figure 6-1 describes the same behavior from a temporal viewpoint. The time axis is drawn from left to right, and the diagram shows the communication between process *P0* and process *P1* with the two functions `MPI_Isend` and `MPI_Recv`. The instrumentation of the program affects each observed function at two points, firstly before the original function call and secondly afterwards. These two phases may be called the init-part (or `monInit`) and the exit-part (or `monExit`), where the former performs monitoring activities at the beginning of the communication function, while the second part performs the additional monitoring activities at its end.

In practice, this distinction between the monitor's init-part and its exit-part may seem unnecessary at first glimpse, and actually, many of the current available monitoring implementation perform monitoring either at the init-part or at the exit-part and do not perform any activities at the other part, respectively. However, if one of both parts is unused, some analysis activities may be prohibited. For example, if a program failure is observed it is undecidable, whether the error occurred before the monitored function (if the init-part is missing), during the monitored function, or after the monitored function (if the exit-part is missing). Yet, this may be crucial information for the debugging process. Thus, it is generally advisable to include both parts of the monitor. The init-part can be used to prepare the data, while the exit-part offers the ability to know about the original operation's completion.

In this context, the reader may notice one minor discrepancy with the event definition given in Section 5.2.1, where we defined that an event is an action without duration that describes a state change, or rather the occurrence of a new state. With these two monitoring parts it seems as if each function actually offers two events, namely an init-event and a corresponding exit-event. This is not really a problem. Firstly, an event still describes the state change of the program during execution, and this state change is only affected by the original function. Thus, the init- and exit-part of the monitor are solely used to generate the data associated with this event. In certain cases, it may even be beneficial that two phases exist, and the user may even choose to describe the interesting statement with two events, namely one event indicating the beginning of an operation, and another event describing statement completion. Additional events describing the availability of the communication link or the initiation of message transfer are imaginable (compare with Section 5.2.1). The actual implementation of this characteristic may be determined by the user's needs.

6.4.2 Monitor Operations

The remaining question is, what happens during the init- and the exit-phase. Although this depends on the applied monitoring tool and the required data, there are always two activities that have to be carried out. In concrete, we may identify two different parts of any monitoring implementation, a preparation part and a storage part [Kran 96a]. During the first part of the monitor, the required data are retrieved from the program's state and prepared for the second part. During the second part, these event data are stored for further processing.

Please note, that storing in this context does not necessarily mean that the data are saved to file. Instead, it means that the data are transferred to a location where the analysis tool can access them. This can be anything useful, from trace files to standard UNIX sockets. The actual connection between the monitor and the analysis tool can be either on-line (monitoring and analysis takes place simultaneously) or post-mortem (analysis is carried out after the program's execution). This issue has been briefly described in Section 4.5.3.

A major problem in this context is the monitoring delay, which is based on the following question: how much is the computing process delayed while the monitor prepares and stores the gathered data. Or in other words, what possibilities exist to reduce the monitoring overhead. For that reason, we distinguish the preparation part and the storage part corresponding to their proximity to the computing process. The preparation part is closely related to the computing process, because it requires to access its state and can therefore not be decoupled from the computing process. As soon as state data has been retrieved and prepared, there is no further need to delay the computing process. This means that the majority of the storage part may be positioned outside the computing process. In fact, the storage part may be completely separated from the computing process and may even be carried out by another process. The advantage of this approach is, that the storage part can be executed independently from the computing process, and the operations in the storage part do not delay the program's execution. The resulting benefits could be relatively big, considering that the storage part usually requires much more delay than the preparation part.

This idea is sketched in Figure 6-2. The left diagram shows the monitoring delay (= overhead) as it affects the computing process with both parts, the preparation part and the storage part. Also indicated is that the storage part requires much more time than the preparation part. In the right diagram we see an improved version of monitoring. The program is still delayed by the preparation part, but instead of the storage part, the program contains a small communication part that transfers prepared event data to a corresponding monitoring process. The actual

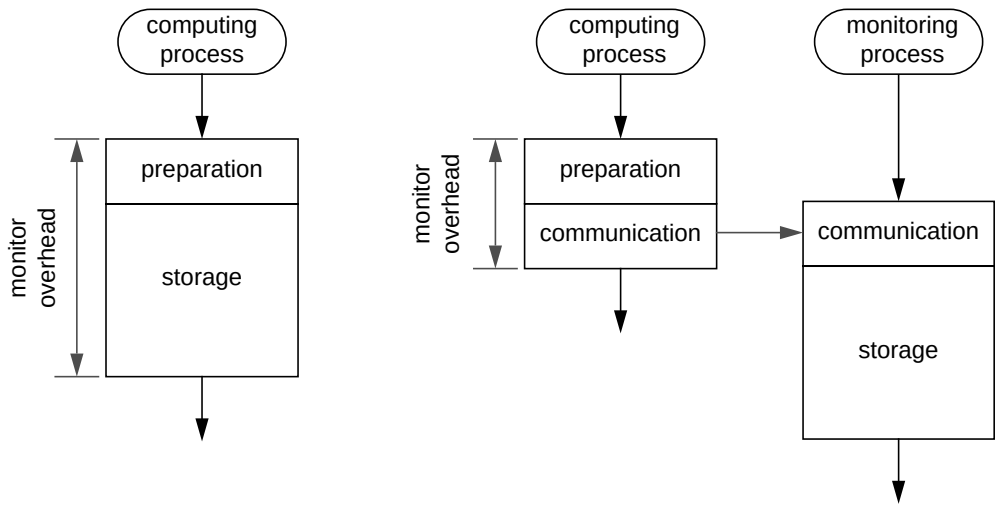


Figure 6-2. Monitoring activities [Kran 96a]

storage of the data is carried out by this dedicated task, which may be located on another process, on another processor, or even on another machine.

In our work on monitoring tools we have experimented with different solutions for decoupling the preparation and the storage part. Some ideas are to apply multitasking facilities of the computing nodes and perform data storing by the tasks on the same nodes with lower priority, or to allocate some nodes of the parallel machine solely for monitoring purposes. (Further details about these ideas are described in [Kran 96a].) Similar approaches to distinguish distinct parts of a monitor and relocate as much as possible to other processes are addressed in many existing monitoring tools, for example in the OMIS compliant monitor OCM [Wism 98a] or in the parallel and distributed debugger DDBG [Cunh 98a].

6.5 Constructing the Event Graph Model

6.5.1 Event Record Structure

After the data have been generated and transferred to the analysis tool, the event graph model can be constructed. Since transportation of event data is usually carried out with event records of some structure, the model is built on top of these records. Table 6-2 contains a corresponding example record structure for an MPI_Recv function as implemented in our tool.

Table 6-2: Example event record structure for MPI_Isend

data	example	description
type	Receive	event class
blocking	TRUE	blocking/nonblocking operation
name	MPI_RECV	original function name
source line number[a]	70	line number in original source code

Table 6-2: Example event record structure for MPI_Isend

data	example	description
source file number[a]	1	pointer to source filename table
start time[b]	694 ms	time when event was initiated
supposed parameters[c] partnerID messageType messageLength	 0 1 1	parameters as provided to operation: communication partner ID expected message type identifier expected message length
accepted parameters partnerID messageType messageLength	 0 1 1	parameters as returned from operation: communication partner ID actual message type identifier actual message length
logical clock	1	logical clock counter of event
logical clock of partner	2	logical clock counter of corresponding partner event
stop time[c]	737 ms	time when event was completed

a. optional, only when source code information is requested during monitoring
b. optional, only when selected during monitoring (amount of data level medium or maximum)
c. optional, only when selected during monitoring (amount of data level maximum)

As described in Section 5.2.2, each event is characterized by its type, its event attributes, its scope and location, and the level of event generation. In our record structure, the event type is uniquely identified by the items in the first three rows. These consist of the class the event belongs to, whether it is a blocking or nonblocking operation, and a constant identifying its original name. The other items of the record structure are the event attributes and may be divided into three groups: source code connection, timing information, and operational arguments. The source code connection is described through line number and source file name, and is only available if requested by the user. The timing information consists of data from real-time clocks and logical clocks, like start time, stop time, logical clock counter, and logical clock of partner event. Some of these timing parameters are optional and can be selected by the user during monitor operation.

The remaining data describe the operational arguments as provided at the function call. We distinguish supposed parameters and accepted parameters, which identify these values at the beginning of the operation and after the return from the underlying library, respectively. The supposed parameters are those, that are provided by the user and represent the requested behavior. In our example, the user specified this particular receive operation to wait for a message from process 0 with message type 1 and message length 1 byte. On contrary, the accepted parameters describe, what has actually happened at the receive. Although these values are often equal to those stored in the supposed parameters (as it is the case in the example), there may be actual differences. For instance, if the user wants to wait for a message from any other available process, the *partnerID* of the supposed parameters could be specified as wild card, which would be in case of MPI the constant `MPI_ANY_SOURCE`. (Similarly, the message type could be described as a wild card with `MPI_ANY_TAG`). During execution, this receive opera-

tion would wait for the next message to arrive (because *blocking* = `TRUE`), and would return the actual source process of this message as the *partnerID* in the accepted parameter structure. This feature is very useful to identify incorrect communication pairs.

The other characteristics, the scope and location of the event, and the level of event generation are not explicitly included in the event record structure. Instead, the event's scope can be derived from the event class. Obviously, blocking receive statements do only affect the process state of one particular process, and on that process only those memory locations that contain the message buffer. On the other hand, operations like broadcasts may affect all or a subset of the existing processes.

Similarly, the level of event generation is intrinsically defined by the event data structure, because we identified the `MPI_Recv` function as our target for the monitor's operation. Thus, we clearly specified, that we are interested in obtaining event information on the source code level of the program. It is also imaginable, that the level of event generation is somehow included in the event structure, although its usefulness may be questioned.

Another interesting fact about our event record structure is that it does not contain the location of the event, in this particular case the process where this event actually happened. This information is not required due to the technique of delivering the event record structure to the analysis tool. In principle, there are two main techniques which have to be distinguished. The usual way of event transfer is to submit one stream of event data to the analysis tool. In this case, each event record must also contain the corresponding event's location. In our case we can omit this information, because we transfer one event stream for each process of the parallel program. Thus, each stream is associated with one particular process and the location data of the events in each stream can be neglected. This multiple event stream approach offers some other advantages over the single event stream: Firstly, in single event streams the event records must be sorted according to their time of occurrence. This is necessary, because of the fact that multiple processes are accessing one particular file (This closely resembles the sequential consistency problem from hardware architectures and can be compared to the observability effect "incorrect perceived orders" in Section 5.3.1). Secondly, multiple event streams allow much more flexible transportation techniques. For example, in case of post-mortem tracing available parallel I/O facilities could be applied.

6.5.2 Partial Event Ordering

The event record structure given above identifies the vertices of the event graph. The remaining issue is to establish the edges between these vertices, which are represented by the "happened before" relation as described in 5.3.3. Obviously, our goal is to establish a partial ordering between the events as stored in the event record structure.

For that reason we integrated the logical clock counter and the corresponding counter of the partner event in the event record structure. From Table 6-2 we also notice, that our logical clock counters are integer numbers instead of vector clocks, which would be required for partial ordering. Thus, it can be assumed that the event record structures do not contain a partial ordering, but instead a total ordering. And in fact, although it would be possible to apply Lamport's logical clock algorithm (from Section 5.3.2) for our event records, we chose another simplification for our logical clocks. Actually, we apply the following rules for logical clocks in our event records:

(1) Each process increments its own logical clock by one whenever an event occurs. This is the value stored in “logical clock”.
(2) Whenever a process transfers data or control to some other process, it piggybacks its own logical clock value to the transmitted message.
(3) Whenever a process is connected to some other processes, e.g. at receives, it stores the piggybacked logical clock value in the “logical clock of partner”.

This simplification allows us to remove most of the clock updates from the monitor. Instead of establishing a total order as with Lamport’s algorithm, it only establishes the sequential order of the events on each process and identifies connected events on other processes by their sequential order number. Since each event has a unique sequential number on each process (providing that overrun of logical clocks cannot occur), each event is sequentially ordered to each other on the same process. In addition, connected events on other processes are identified by their unique sequence number.

Table 6-3: Example data of simplified event ordering mechanism

event	process ID	logical clock	partner process ID	logical clock of partner
a	P	1		
b	P	2		
c	P	3	Q	3
d	P	4		
e	Q	1		
f	Q	2	R	1
g	Q	3		
h	Q	4	S	2
i	Q	5	P	2
j	Q	6	P	4
k	Q	7		
l	R	1		
m	R	2	Q	1
n	R	3	S	1
o	R	4		
p	S	1		
q	S	2		

Table 6-3: Example data of simplified event ordering mechanism

event	process ID	logical clock	partner process ID	logical clock of partner
r	S	3	P	1
s	S	4	R	4
t	S	5	Q	7

An example for this technique is given with the data included in Table 6-3. This data is delivered by the monitor and consists of records with the following structure:

(event, process ID, logical clock, partner process ID, logical clock of partner)

The first element is the name of the event (a-t). The second element is the ID of the process, on which the event has occurred. The third element contains the corresponding logical clock on that particular process, which is in our case only a sequential counter. The fourth and fifth element do only exist, if the corresponding event has been a receive event. In this case, the logical clock counter of the partner process has been transferred with the message. The ID of the partner process is available as the fourth element, while the logical clock of the partner is the fifth and last element. Please note, that the monitor only stores the last three elements. The first element is only used for illustration, while the second element is implicitly provided by the monitor, since the events are always collected in one tracefile per process.

An event graph corresponding to Table 6-3 is provided with Figure 6-3. All events are annotated by the logical clocks as derived by the rules above. In addition, receive events also contain the logical clock counter of their partner process as the second number of their annotation. For instance, event *f* on process *Q* has a sequential counter on process *Q* of 2, but is also connected to the event *l* on process *R* with the sequential counter 1 on that process.

These simplified approach to event ordering offers an advantage over the proposed ordering methods described in Section 5.3.2. Firstly, it does not need complicated clock updates at the receiver and generates therefore less overhead than any other ordering method. Secondly, it is also based on simple integers, which requires much less memory than algorithms based on vector clocks. These two problems of memory requirements for vector clock approaches has been described in [Char 90] and [ChSk 93].

However, an event ordering in our simple form is not directly useful for program analysis. Thus, our simplified logical clock algorithm forwards the task of generating a partial ordering to the analysis tool. This can easily be achieved by traversing through the graph once in the direction of time from the oldest to the newest events. During this ordering step we apply the same rules as introduced in Section 5.3.2, which delivers the required vector clocks for each event. Applying this algorithm to Figure 6-3 would deliver the event graph as given in Figure 5-8. Based on this partial event ordering, all the analysis steps can be applied[2].

2. Another benefit of our approach is that it overcomes a limitation of vector clocks described by Fidge in [Fidg 97]. On systems, where overtaking of messages is allowed, vector clocks do not fully describe the relation between send and receive events. This problem arises from the usage of the maximum operation in step 4 of the original vector clock algorithm (see Section 5.3.2, page 137). With our simplified logical clocks, we are not only able to generate vector clocks, but also to provide this information even for overtaking of messages.

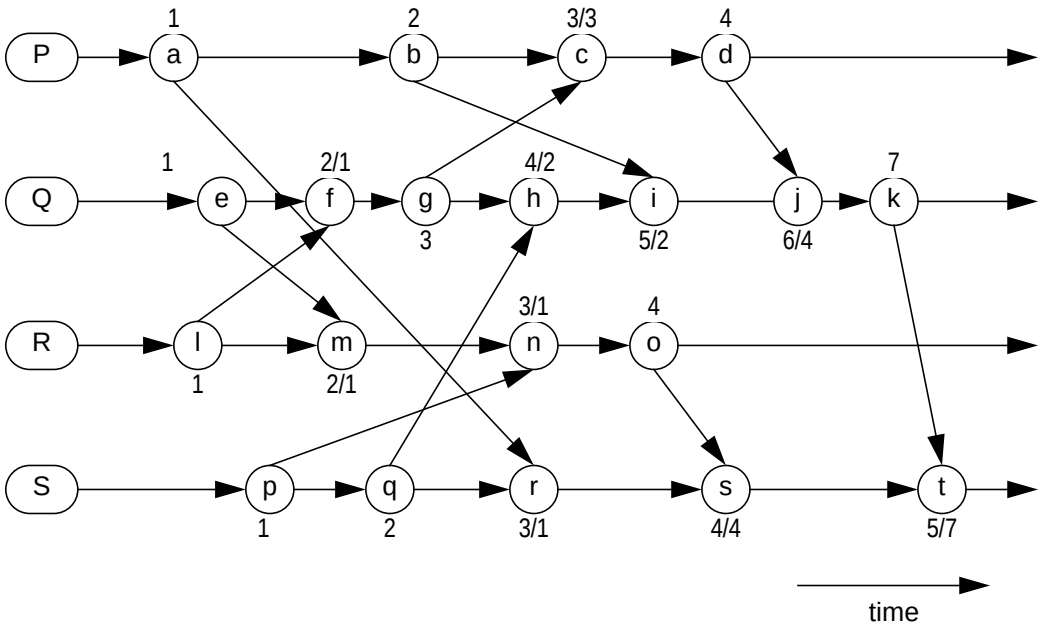


Figure 6-3. Simplified event ordering mechanism

6.6 Visualization of MPI Programs

After establishing an event graph as defined in Section 5.4.1, we can proceed with our analysis of this graph for error detection (and performance tuning). However, since we would also like to derive a visualization of our MPI program, we need to convert this partial event order as described above into a total event order for our space-time diagram. This step can be achieved with the partial ordered clocks as established above. In addition, we could use the real-time clocks as stored in the event record structures, with all the problems of visualizing real-time clock values as described in Section 5.3.1.

It is also possible to derive a total ordering from our simplified logical clocks as they are stored in the event record structure. In fact, constructing a space-time diagram with a total ordering from our event streams is relatively trivial. All we need to do is to process the events in their time of occurrence, which is given by their logical clock counters as the sequential order of the events on the processes. Since this processing is done strictly sequential on each process, it is also a feasible method for on-line visualization, when the event streams are generated simultaneously with the visualization. The method is applied as follows:

(1) Events that are not influenced by other processes (in the graph those without incoming edges) are visualized immediately, one event after each other and in predefined intervals.
(2) Events that are influenced by other processes (e.g. receive, test,...) are positioned in the first empty interval after the event that influenced them.

This kind of visualization, which we called the logical clocks display before, will be used frequently in the examples of the next chapters. One example is given in Figure 6-4, which shows

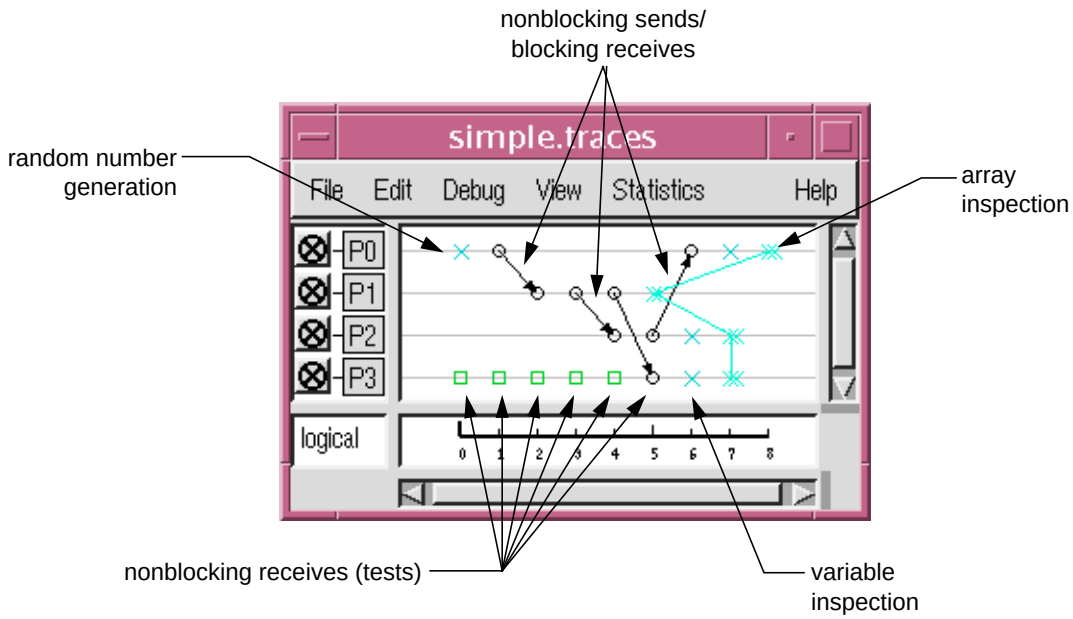


Figure 6-4. Visualization of events observed during program execution

a screenshot of our debugging tool and an event graph for an actual program execution. The main source code fragment of the corresponding program is available in Appendix A. Please note, that the program's only intention is to demonstrate it in this chapter, because it contains all the events of interest as described in Section 6.2.

The events of interest are the remaining issue in our example. We started this chapter by specifying, which events of a program should be available for program analysis. Events that are generated must somehow be accessible by the user in order to inspect their contents and derive knowledge from the program. Thus, selected events should somehow be included in the event graph.

In our example, each of these events is displayed with a corresponding symbol. The most important symbols are the nonblocking sends and their corresponding blocking receives. Since they are standard events in message passing programs, they are displayed as black circles. As we will see, the color of these events may change if anomalies are detected during analysis. All the other events in the graph are displayed with different colors for better recognition. In our example, the nonblocking receives (test) are identified by green squares, while random number generation and variable inspection are visualized as green crosses. These events share the same symbol, because both represent events that contain the data of actual memory locations. Although similar to variable inspection events, the remaining array inspection is presented to the user with another symbol, namely two overlapping cyan crosses. In addition, an array inspection event represents an event that affects all processes, which is indicated by the cyan line connected the local array inspection events on each process.

Besides these sample events there are many other events with corresponding symbols available. In addition, user's may add new events and corresponding symbols according to their needs, and in certain cases, events may not need any representation in the event graph. More examples of events will be given in the forthcoming chapters.

6.7 Summary

In this chapter, we have described how to construct an event graph in practice. For that reason we selected an example program and specified the events of interest, that should be extracted during program execution. Besides describing these events for our parallel program example, we briefly introduced the instrumentation phase, that is responsible for inserting monitoring code into the target program. Again, an example of our C-preprocessor instrumentation technique is to offered to provide a concrete picture of our approach.

The next step to be exercised was the actual execution of the program, and its simultaneous monitoring by our debugging tool. We have briefly summarized the operation of the monitor, and how it generates the data for the subsequent analysis tasks. These analysis tasks are based on constructing an event graph model, which requires to describe the event record structure and introduces our method of event ordering. Finally, the resulting event graph is used for a visualization as a space-time diagram, which serves as the main foundation and interface for the debugging functionality as described in the following chapters of this work.

7. Basic Debugging Activities with the Event Graph

In this chapter, we describe the basic debugging activities that can be carried out with the event graph model. This includes the modelling of concrete errors and their visual representation in the space-time diagram. The graphical display is used to interactively extract additional behavioral data as stored in the event attributes to increase program comprehension. Additionally, it serves as the user interface for steering the program's execution during subsequent debugging phases with an efficient and consistent breakpointing mechanism.

- Modeling errors in the event graph
- Automatic analysis of the graph
- Visualization of errors in the space-time display
- Extraction of information and event inspection
- Program re-execution and breakpointing

7. Basic Debugging Activities with the Event Graph

7.1 The Debugging Strategy Based on the Event Graph

The initial steps of our debugging strategy, monitoring the program's execution and generating the event graph model, have been completed as described in the previous sections. In many cases, these activities can automatically applied without user interactions, especially if the set of events as pre-defined by the debugging tool are sufficient for the user's expected needs.

The next step is to utilize the event graph model for error detection and to extract program execution data for increasing the user's program comprehension. These activities are carried out with a debugging strategy that consists of the following steps:

- Automatic analysis of the even graph to detect errors.
- Visualization of detected behavioral characteristics.
- Interactive inspection of the event graph display.
- Re-execution of the program and breakpointing in the event graph.

The constructed event graph model is automatically analyzed to detect errors that occurred during execution. This analysis task reveals some critical bugs which are afterwards displayed in the space-time diagram with different colors and symbols. Based on the visual representation the user can interactively explore the graph and inspect occurring events to gather further knowledge about the program's execution. Besides this inspection functionality, the visual representation is also the main part of the user interface to steer the program's behavior during subsequent execution phases. The primary activity in this context is a so-called cut mechanism, that guarantees consistent breakpoints on all processes.

Please note, that this chapter describes only the most basic features for debugging parallel programs. Special focus on nondeterministic behavior and corresponding analysis functionality is given in the next chapter.

7.2 Automatic Event Graph Analysis for Error Detection

7.2.1 Program Failures

The first step to detect errors in a program's execution is analyzing the event graph model. This analysis step can be performed while the event graph model is being constructed or at any time afterwards. The automatic error detecting feature is based on the distinction between the groups of bugs, that has already be given in Section 3.2.6, which divides bugs in (program) failures and (computational) errors (see also Figure 4-1 on page 79 and Figure 4-4 on page 103). While failures are clearly recognizable, e.g. through program break-downs or exceptions being taken, computational errors always depend on the semantic context of the results and therefore require a verification step. This verification is necessary to compare the expected results with actually obtained results, since correct results are indistinguishable from incorrect results without a given specification. Please remember, that bugs are rarely the work of a single operation. Usually there is a logical sequence between over a set of operations that lead to the

final erroneous state [Brit 98], and several computational errors may lead to an observed program failure.

The first group of bugs, program failures, can easily be integrated in the abstract event graph model. Their integration is based on the fact that failures lead to an end of the corresponding process, and are therefore always the last event on a process. This characterizes them in the event graph similar to correct program termination events as vertices that have no leaving edge. (Please note, that this does not limit the number of approaching edges.) Corresponding terminal events are introduced in Definition 5-9 on page 144. The difference between expected terminal events and program failures can easily be derived from the event's attributes.

Failure events do not need to be specified by the user, because they express a permanent and final program state that will be recognized as soon as it occurs. Thus, a failure event can be easily integrated in a debugging tool by defining an appropriate event type and its attributes, and instrumenting the corresponding exception handler. Examples for failures that should be captured by a debugger are division by zero, buffer overflow/underflow, I/O errors, and others.

If a debugger should be capable of modeling "division by zero"-failures in the event graph model, the following steps must be applied in practice: Firstly, an event type "division by zero"-failure must be defined. This definition contains event attributes like the two arguments of the division that lead to the program failure. It is also very useful to include a source code reference to identify the exact location of the failure in the original program code. Secondly, the exception handler for this "division by zero"-failures must be instrumented to contain code for generating the event defined above during program execution.

Modifying an exception handler may not be an easy task, especially without support from the computing system. Yet, modern operating systems and compilers often offer a simple method for user's to specify their own exception handler, that is executed whenever a corresponding exception is taken. An example is the C++ language, where effective exception handling is offered by language constructs. This approach provides a simple way of transferring control and information from a point in the execution of a program to an exception handler [Stro 91]. With such language constructs it is relatively easy to offer appropriate exception handling for program failures. Please note, that exception handling in C++ may also be used to cope with such failures by applying appropriate methods to react to this situation. For example, a programmer may want a program to continue and to compute the remaining results, even if a division by zero occurs. In this case, a "division by zero"-event may not be classified as a failure event, but instead as a regular computational error.

7.2.2 Computational Errors

In contrast to program failures, computational errors may occur at both, edges and vertices of the event graph. Since computational errors are not recognizable as errors without a given specification, a computational error may virtually occur everywhere in the event graph. This stems from the fact, that errors are observed at incorrect program states [Grab 99], which in turn are described by both, the edges and the vertices of the event graph (see Section 3.2.6). For that reason, we need to be prepared to detect errors at the following locations:

- Errors at communication edges.
- Errors at sequential edges.
- Errors at event vertices.

Errors at the event graph's edges have to be classified according to the type of their edge. Edges connecting events on different processes represent transmission of data or control between the corresponding processes. Since these edges are connected to events on distinct processes, and there are no intermediate program states occurring along the edges, analyzing these edges is performed by inspecting the events at the head and the tail of the edge.

Edges connecting events sequentially on one process are more difficult to analyze, because each edge may describe a series of state changes, and these state changes determine the results of a program. Analyzing the events at the head and the tail of the event may deliver only approximations of the actual error location or, at worst, no useful information at all, if the error is located somewhere on the edge in between. Therefore, complete analysis of these edges is required to detect incorrect behavior, if it is located on these edges. Since edges on one process represent only sequentially ordered state changes, the corresponding statements represent a simple sequential program. Consequently, it should be possible to apply a sequential debugger to analyze the piece of code on one process. This technique will be described in more detail in Section 7.5.

Errors at event vertices may also be detected without an additional, sequential debugger. The idea to detect corresponding errors is based on a description of expected event attributes. Given a set of events, a debugger's user may specify the constraints that must be followed by these events and their event attributes, respectively. These constraints can be automatically checked by the debugging tool, and deviations from the expected data designate incorrect or illegal operations. If these deviations are displayed in the graphical representation of the event graph, the user can immediately spot and identify possible sources of incorrect behavior.

An example for detecting communication errors that occur at the vertices of the event graph is given below. Other examples are imaginable, and again the possibilities are only limited by the user's requirements. Since the user manifests interesting events and their attributes, it should also be possible to define analysis techniques for error detection based on the event graph.

7.2.3 Example: Communication Errors

Communication errors are errors that occur due to incorrect communication operations or function calls. Grabner introduced these errors in more detail within the communication graph, and offers some explanations, why these errors may occur frequently in reality [Grab 99]. With the event graph model, which is basically an extended version of the communication graph, this group of errors can be automatically detected by analyzing and verifying the event attributes. In this section, we will briefly summarize the definitions of Grabner and how to apply them to the event graph. In concrete, we will focus on the following two types of communication errors [GrVo 93]:

- Isolated events:
  - Isolated send event.
  - Isolated receive event.
- Events with different message length at sender and receiver.

According to Grabner, all these errors may only occur due to incorrect parameters at the communication statements, which have either been provided by the user or have been computed during operation [Grab 99].

Isolated send events

Isolated events are send events without corresponding receive events or vice versa [GrVo 93]. An isolated send event occurs, if the receiver does not care for the messages delivered to its buffer. These events are critical, because the target processes may run out of message buffer memory, which eventually leads to a failure of the corresponding process. This is especially problematic, because the failure of the process and its actual source can be located far apart, and discovering the relation between the buffer overrun and some neglected isolated send events is often difficult. Formally, an isolated send event can be defined as follows:

Definition 7-1: Isolated send event

An event $e_p^i \in E$ is called isolated send event, if $\neg\exists\left(e_q^j \in E \right)\ e_p^i \overset{C}{\rightarrow} e_q^j$.

With the event graph model, isolated send events can be detected by checking the outgoing edges at each send event. One of these outgoing edges must be connected to the next event on the same process (due to the sequential happened before relation). The remaining edges, one in case of a regular send, more in case of other events like broadcasts, must be connected to corresponding receive events. If any edges is unconnected, an isolated send has been detected.

Isolated receive events

Isolated receive events are considered more critical than isolated send events, because an expected message does never occur. In case of nonblocking operation, this does usually not impose any problems, because nonblocking receives are explicitly used to check for the availability of a message. Thus, isolated receive events are assumed to occur only at blocking receive operations (e.g. in case of `MPI_Recv`). They are crucial, because the receive waits for an incoming message that never occurs and the corresponding process remains blocked until program termination. Even a chain-reaction is possible, because other processes depending on the blocked process may lock-up as well. An isolated (blocking) receive event can be defined as follows:

Definition 7-2: Isolated receive event

An event $e_p^i \in E$ is called isolated receive event, if $\neg\exists\left(e_q^j \in E \right)\ e_q^j \overset{C}{\rightarrow} e_p^i$.

Since such a receive waits forever, the isolated receive event e_p^i is also the terminal event on process *p* and the following condition is valid according to Definition 5-9 on page 144:

$$\neg\exists\left(e_r^l \in E \right)\ e_p^i \rightarrow e_r^l \ .$$

With the event graph, isolated receives can be detected immediately by checking for incoming edges at each receive event. Obviously, each receive must have at least two incoming edges, one from the previous event on the same process indicating the sequential program flow, and one from the corresponding send event. If there are less edges, an isolated receive is detected. If there exists an isolated receive that is blocking, it will be the last one on that particular process with no leaving edge.

The problem with isolated receives is, that monitoring them is tricky. The question is, where to generate the trace data, before or after the event. Consider an isolated receive event, that is blocked forever. If the data is generated by the monitor after the event has been completed, as it is the case in most existing monitoring tools, such an isolated receive event will never produce any data. For the user it seems as if this isolated receive has never been initiated. If the monitor generates event data about this isolated receive before the receive operation is carried out, the occurrence of the receive can be modelled in the event graph, but it is unclear whether this particular operation was finished or not. This is especially critical, if a program failure occurs after the selected receive, and the user may relate the program failure to the last observed event. The solution is to divide each interesting operation in two parts - an init-part and an exit-part - as described in Section 6.4.1. Then, the init part will signal that the event has actually been started, while the exit part provides information about event completion.

Table 7-1 contains the events and their data associated with the init- and exit phases of an arbitrary communication event within our monitoring tool, and corresponds to Table 6-2 on page 163. The first column specifies the level of data that is traced, which can be selected between minimum, medium, and maximum level according to the generated observation data. The corresponding data associated with its level is cumulative, which means, that the medium level contains all the data from the minimum level and its own data, while the maximum level additionally contains all the data from the minimum and the medium level. The second column gives the two phases of the monitor, with the corresponding event data in the third row. Please note, that Table 7-1 does not contain source code reference, because this optional feature can be selected independently from the other parameters of the table. If selected, the source code connection is traced during the init phase of the monitor.

Table 7-1: Communication event data

level of data	phase	event data
min	init	type blocking name
	exit	logical clock logical clock of partner (if available) accepted parameters: partnerID messageType messageLength
med	init	start time
	exit	-
max	init	supposed parameters: partnerID messageType messageLength
	exit	stop time

During monitoring, the data of the third row of Table 7-1 is generated in the corresponding phase of the second row. When a receive operation is initiated, and the user selects minimum level of data, at least event type, blocking, and event name are stored for later analysis. If maximum level is selected, even the supposed communication parameters are retrieved. Thus, when the program is terminated, at least these basic event attributes are available for the subsequent analysis phases. In principle, the monitor always tries to retrieve the necessary trace data as soon as it is available.

Another problem related to isolated receive events is associated with monitor buffers. Often monitors apply buffer memory for storing the trace data. With this buffer, the monitor's operation and especially the storage part of the monitor can be accelerated, because accessing the buffer is a lot faster than directly accessing the trace data or transmitting the data to an on-line tool. Instead, the buffer is only written to its final destination, when it is completely filled. The problem with this common approach is, that event data may still be in the monitors buffer, when the program terminates. For example, an isolated receive event is traced and the observation data is stored in the monitor buffer. During the isolated receive the program terminates for whatever reason (e.g. when the user terminates the program because of suspected blocking), and all open files are closed. Yet, the data from monitor's buffer is lost, and the debugging tool does not contain all the events that actually occurred during execution.

The problem with the monitor buffer can be addressed in various ways. One possibility is to write the data directly to the trace data storage, which may impose a heavy delay due to increased I/O activity. Another possibility would be to offer a flushing mechanism, that forces the monitor to store all its data to the traces. Thus I/O flushing can be initiated automatically, e.g. when a monitored function is blocked for a certain amount of time, or manually by defining a special monitor operation. Implementing such a flushing mechanism is a simple but effective solution. For the example communication data given in Table 7-1 this means, that the monitor writes the data to the corresponding trace storage immediately after it has been generated during the init-phase and the exit-phase, respectively. A similar solution would be to include trace file flushing in the exception handler, that is executed when the program is terminated.

Events with different message length

The remaining group of simple errors described in this section are events, where different message lengths at corresponding sender/receiver pairs have been observed. This effect can be described as follows:

Definition 7-3: Events with different message length

Two events $e_p^i \in E$ and $e_q^j \in E$ are called events with different message length, if

$$e_p^i \xrightarrow{C} e_q^j \text{ and } \text{messageLength}\left(e_p^i\right) \neq \text{messageLength}\left(e_q^j\right).$$

Based on this definition, two possibilities may be distinguished. Either the sender submits more bytes than the receiver expects, or the receiver expects more bytes than the sender transmits. Both situations reveal different consequences. The former, a sender submits more than the receiver expects, means, that the receiver did not prepare enough memory in its incoming

message buffer[1]. As a result, either some of the memory at the receiver is overwritten by the message from the sender, which leads to unpredictable behavior, or the receiver looses some of the messages data, which may be crucial for the remaining computing steps.

The second situation, where the receiver expects more bytes than the sender transmits, can also pose a critical problem, if the receiver does not check the amount of data that has actually been received. In this case, the receiver may access memory in the message that contains undefined data, which may again lead to unpredictable behavior. However, this second case can be prevented if the receiving process examines the return status from the receive operation and reacts to the actual amount of memory that has been received.

Detecting events with different message length in the event graph is rather trivial. All the debugging tool has to do is to traverse through the event graph and check the attributes of the receive events, in particular, the message length in both the supposed parameter packet and the accepted parameter packet. Since the accepted parameter packet contains information about the message as it was defined by its origin, there is no need to relate the receive event with its corresponding send event[2].

The communication errors and their detection as described in this example represent major causes for incorrect behavior in message passing parallel programs. In reality, we have observed this problem many times. Our simple and efficient detection mechanism located these bugs, that would have been much more difficult to discover without such debugging support. Other examples are imaginable and one special case, the race condition detection algorithm, is described in the next chapter.

7.3 Visualization of the Detected Behavioral Characteristics

7.3.1 Error Detection with the Space-Time Diagram

Detection of errors with the event graph model is only one part of the strategy. The next part is to attract the user's attention to these places in the code, where the incorrect behavior was observed. The approach is similar to what has been described in Section 6.6. The basic representation is the event graph visualization as a space-time diagram. Based on this diagram, different symbols and colors will be used for highlighting these behavioral properties.

An example visualization of erroneous behavior is given in Figure 7-1. It shows the execution of 8 processes from a parallel Poisson equation solver. The edges of the event graph indicate the communication between the processes. Some of these edges are highlighted to emphasize simple communication errors as described in the previous sections. Two pairs of events with different message length are given. In one case, more data is sent than received, which is characterized by a widened tail of the edge. In the other case, more data is expected at the receiver which is marked by a widened head of the edge.

1. This situation, that a process sends more bytes than another receives, may even deliver error messages from the underlying message passing system. For example, in some MPI implementations the following message is printed: `MPI_Recv: message truncated`.
2. In theory, it would be possible that the message truncation would occur at the sender. In this rare case, checking the receive events would not detect messages with incorrect message length at sender and receiver. Instead, it would be required to compare corresponding events in the event graph. However, this kind of message truncation was not observed in any of our message-passing systems.

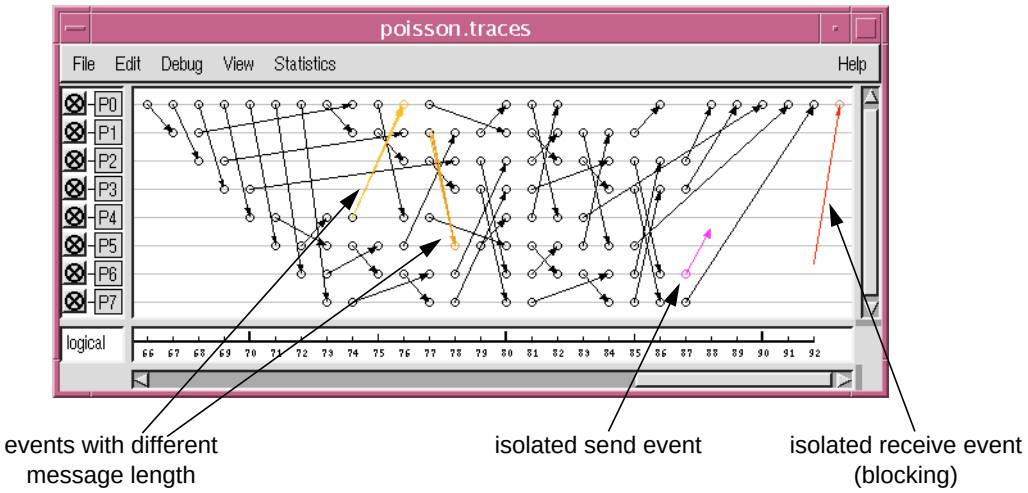


Figure 7-1. Basic event graph analysis (communication errors)

The isolated events are characterized by only one event and a corresponding edge, that is loose at the opposite side. The tail of the isolated send in Figure 7-1 is connected to its initiating send event on process P6. The head of this event points to the process that the message was actually sent to. On contrary, the isolated receive consists of the receive event on process P0 and an edge that is connected at the head to this event. The tail of the isolated receive indicates, where the message has been expected from.

In this example, the isolated receive event has been generated by a blocking receive operation. Thus, this program would have waited forever for a message from process P6, that was erroneously sent from process P6 to process P4. In order to generate the corresponding event graph, the program had to be executed with trace file flushing, and was terminated by a kill signal after some arbitrary waiting time.

7.3.2 Performance Analysis with the Space-Time Diagram

The example in Figure 7-1 uses logical clocks as time-scale for event visualization, which is rather useful for detection of communication errors as described above. Further information about the program can be obtained, if the time-scale is switched to real-time clocks, presuming that corresponding time-stamps have been retrieved during monitoring. An example is given in Figure 7-1, which displays an event graph of a program executing on 8 processes with real-time clock values. The distance between each event represents the actual time that passed between both events. The meter at the lower part of the window represents the currently selected time-scale.

Displaying an event graph with real-time clocks is very useful for performance analysis. In the example of Figure 7-1, blue bars before each event are used to represent the blocking time of each occurring event. This means, that the longer the blue bar before each event the longer this event was delayed during program execution[3]. In other words, the blocking receive operation started at the beginning of the blue bar, but data was only received at the time where the

3. An idea that needs to be investigated in this context is, whether the blocking time can be indicated with the width of the bar in front of the corresponding event. With this approach it might be possible to include simple performance data in the logical clocks display.

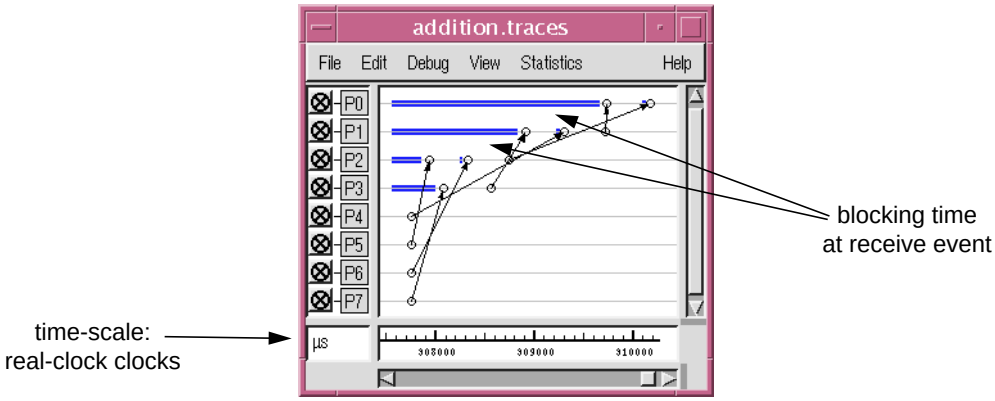


Figure 7-2. Visualization of blocking times

actual blocking receive event is drawn. It is also possible to incorporate more than one color to represent all the different states during execution. Such a display has been incorporated into many existing performance analysis tools, for instance Paragraph [HeEt 91a], Paradyn [Mill 94], and Vampir [Nage 96].

The disadvantage of this display is obviously the observability problem as described in Section 5.3.1. If the processes in Figure 7-1 are executed on different processors of the parallel system, and each of these processors possesses its own real-time clock, the time-stamps on different processes are not comparable. Thus, for our example in Figure 7-1 it was necessary to apply a clock synchronization algorithm, that establishes a correct global ordering within a given level of accuracy.

Other examples for performance analysis with the event graph model and its graphical representation are imaginable. In our debugging project we experimented with two kinds of evaluation techniques, statistical summaries and hardware oriented analysis techniques. Statistical summaries were easily computed based on the real-time clock values observed during program execution. The results of these calculations are included in the debugging tool with a dedicated menu item (“Statistics”) and include various standard statistics like mean, maximum, minimum, and standard deviation, for values like computation time, communication time, amount of monitor overhead. All the values were given for the complete program run or selected sections, as well as for certain groups of events (only send events, only receive events) or on a per process basis.

In addition, our debugging environment includes hardware oriented analysis techniques. They are based on the fact, that many performance bottlenecks occur due to unbalanced hardware-software relations. For example, the reason for the blocking at one of the processes as seen in the space-time diagram of Figure 7-1 may possibly occur due to blocking of the message at intermediate nodes of the communication network. However, detecting the cause for this problem is impossible with the space-time diagram, because the displayed data is too high-level. Therefore, we combined the space-time diagram with a tool that shows a block-diagram of the underlying hardware architecture. This visualization can be selected with the “Reference” button in the event information window. The hardware block-diagram is used to related the visualized behavior of the event graph with the underlying hardware components, and allows additional investigation techniques. For more details about our performance analysis methods incorporated in our debugging environment see [Kran 94] and [Kran 96c].

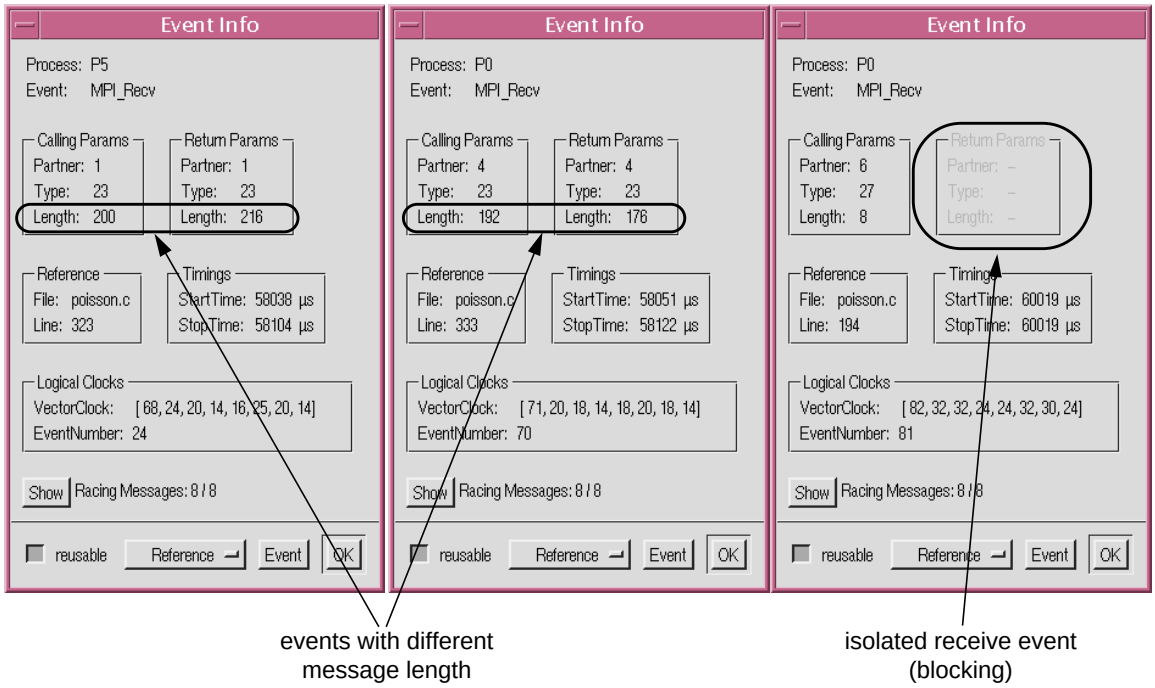


Figure 7-3. Interactive event inspection (communication errors)

7.4 Interactive Event Graph Inspection

7.4.1 Inspecting Communication Events

After the program's event graph has been visualized as a space-time diagram, the user can interactively explore this visual representation to gather more knowledge about a program's behavior. This interactive inspection of the graph is supported by basic features like scrolling forwards and backwards in time and along the process axis. Zooming is supported by selecting the number of pixels per interval in the logical time display and the corresponding time scale with the real-time clocks. For example, in Figure 7-1, twenty pixels are used for one logical clock interval. In the space-time display of Figure 7-1, thirteen pixels represent one microsecond on the real-time clock. Some more advanced possibilities of changing the basic representation of the space-time display will be given in Chapter 10.

In addition to modifications of the space-time representation, users may also interactively inspect the events occurring during operation. This is done by selecting the graphical symbols representing the events, which opens the so-called event inspection window. The data given in the event inspection window are all the event attributes that have been generated during program execution. As an example, we selected some of the erroneous events of Figure 7-1 and captured the corresponding event info windows in Figure 7-1. The left-most window shows the receive event of the incorrect message length, where process P1 sent 216 bytes to process P5 that expected only 200 bytes. Consequently, 16 bytes have been lost during this transfer, which was also indicated by the widened tail of the edge at process P1. The middle window of Figure 7-1 shows the other message length error, where 176 bytes from process P4 have been sent to process P0, which expected 192 bytes. If process P0 does not detect this disproportion and accesses some of the bytes in the message after the 176th byte, unpredictable behavior may be observed.

The right-most window in Figure 7-1 represents the event inspection window for the isolated receive of Figure 7-1. As can be seen, the event attributes reveal the calling parameters of the unfinished receive event, which indicate that process P0 was waiting for a message from process P6 with message type 27 and message length 8 bytes. The executing operation was an `MPI_Recv`, which is blocked until an appropriate message arrives in its buffer. Since a proper message never occurred, the return parameters of this receive event were unavailable, which is indicated with shades in the displayed event info window.

The event information windows of Figure 7-1 contain other available event attributes as well, which obviously depend on the displayed event and may further be extended by the user's need. Since a complete set of these data may be quite exhausting for the debugging screen, the amount of displayed data can be selected by the user. Besides the attributes described above, the examples in Figure 7-1 contain also the source code reference (filename and line-number), the timings of the real-time clock (start-time and stop-time of this event), and the logical clocks as computed during the event graph construction. The vector clock contains the actual values as computed by Fidge's algorithm (see Section 5.3.2), while the event number below represents the logical clock value of our simplified approach (see Section 6.5.2). The remaining value displaying the number of racing messages refers to the nondeterministic behavior of this particular problem, which will be addressed in the next chapter.

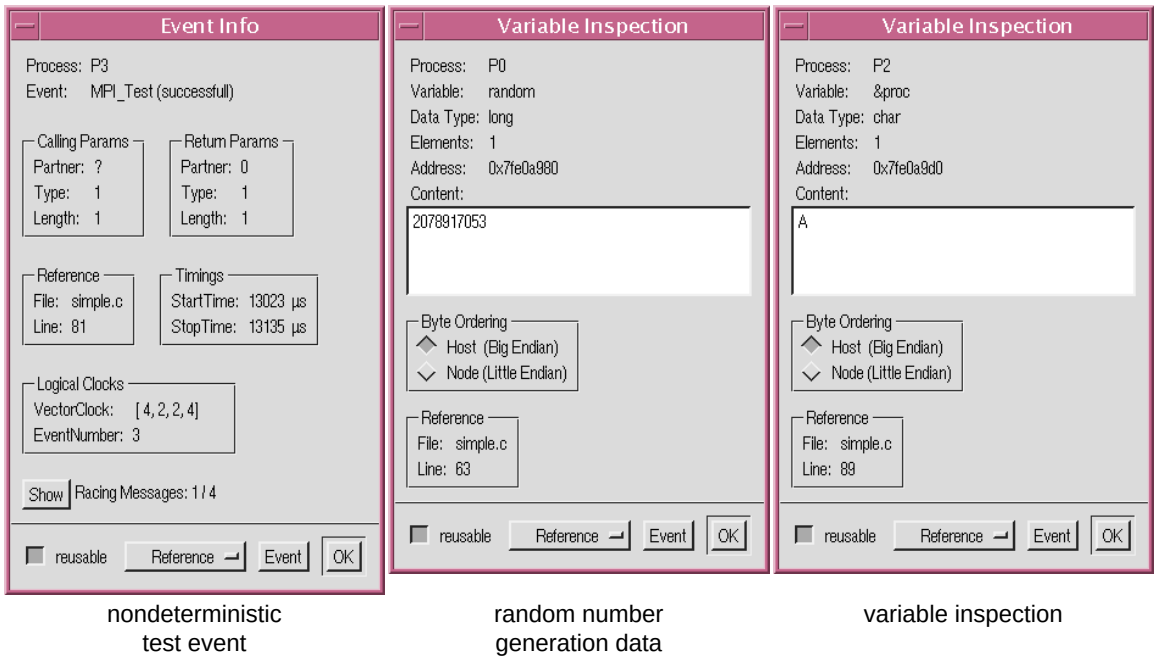
Another example is given in the left-most window of Figure 7-1, which shows the data from a nonblocking receive event as it occurred on process P3. Since the given receive operation (`MPI_Test`) is nonblocking, the event information window contains the results of the program, whether the test was successful or not. In this case, the operation was successful which means, that a message according to the calling parameters was found in the process' incoming message buffer. This receive operation is also nondeterministic, which can be attributed to the fact that the calling parameters contain a wild card (displayed as "?") for the sender process. This means, that the corresponding process P3 would receive a message from any other process, presuming that it labeled with the correct message tag (in this case "1"). Such operation which permit different messages to arrive will be studied in more detail in the next section.

7.4.2 Variable Inspection Events

Besides the communication event information as displayed in Figure 7-1, we may also retrieve information about the other events given in the example of Figure 6-4. These remaining events are a random number generation, a variable inspection event, some nonblocking receive events, and an array inspection. The results provided by the debugging tool when selecting the random number generation event are given in the middle window of Figure 7-1, which indicates, that the random number is of type long, the number of computed elements, its address in memory, and its actual contents. The parameters below allow to select a different byte ordering, which may be necessary in some cases, and the source code reference. The right-most window of Figure 7-1 represents the variable inspection window, that was added to our program shown in Figure 6-4. Since random number generation events provide the same event attributes as variable inspection windows, the middle window provides similar information as the left-most window.

7.4.3 Array Inspection

The last group of events displayed in Figure 6-4 resemble an array inspection event. This artificial event is closely related to a variable inspection event, because it also displays the con-



nondeterministic test event

random number generation data

variable inspection

Figure 7-4. Variable inspection

tents of a user-defined variable. In case of an array inspection, the variable is a multi-dimensional array, that is possibly distributed across several processes. Thus, the array inspection event as displayed in Figure 6-4 is a hook to inspecting the contents of the array and its values, respectively, at the time during execution when the array inspection code was executed by the monitor.

An example for the array inspection is given in Figure 7-5. The top-left window shows the textual data that is displayed when the user selects an array inspection object in the event graph display of Figure 6-4. This overview window contains some of the event attributes obtained during program execution, e.g. the array's name, its memory location, and data type. Also provided is the number of dimensions and the elements in the complete distributed array, as well as the upper bounds in each dimension. The given distribution resembles an HPF-like data distribution, in this case "`BLOCK`, `BLOCK`", which describes the mapping of the data array on the existing process array. The source code reference given in the example allows to trace the graphical object back to the original `monARRAYTRACE` statement.

Although the textual information is useful to retrieve some knowledge about the distributed array, it is only the starting point for another more advanced inspection mechanism. During debugging the most important data of a distributed array for the user are probably the values of the array's elements. Only by inspecting these values is it possible to determine, whether the computation of the array elements was correct or not. The problem is that inspecting one element after another requires lots of debugging interaction (and implies the problem of overlooking some erroneous elements).

A solution to the problems of inspecting distributed arrays is to display the array as a heat-diagram. In a such a diagram the values of the array's elements are mapped onto arbitrary colors, so that the color of the graphical object indicates its element's value in the array. An example is included in Figure 7-5, where the top-right diagram shows a heat-diagram of an example array. This window is opened, when selecting the reference button in the array inspection over-

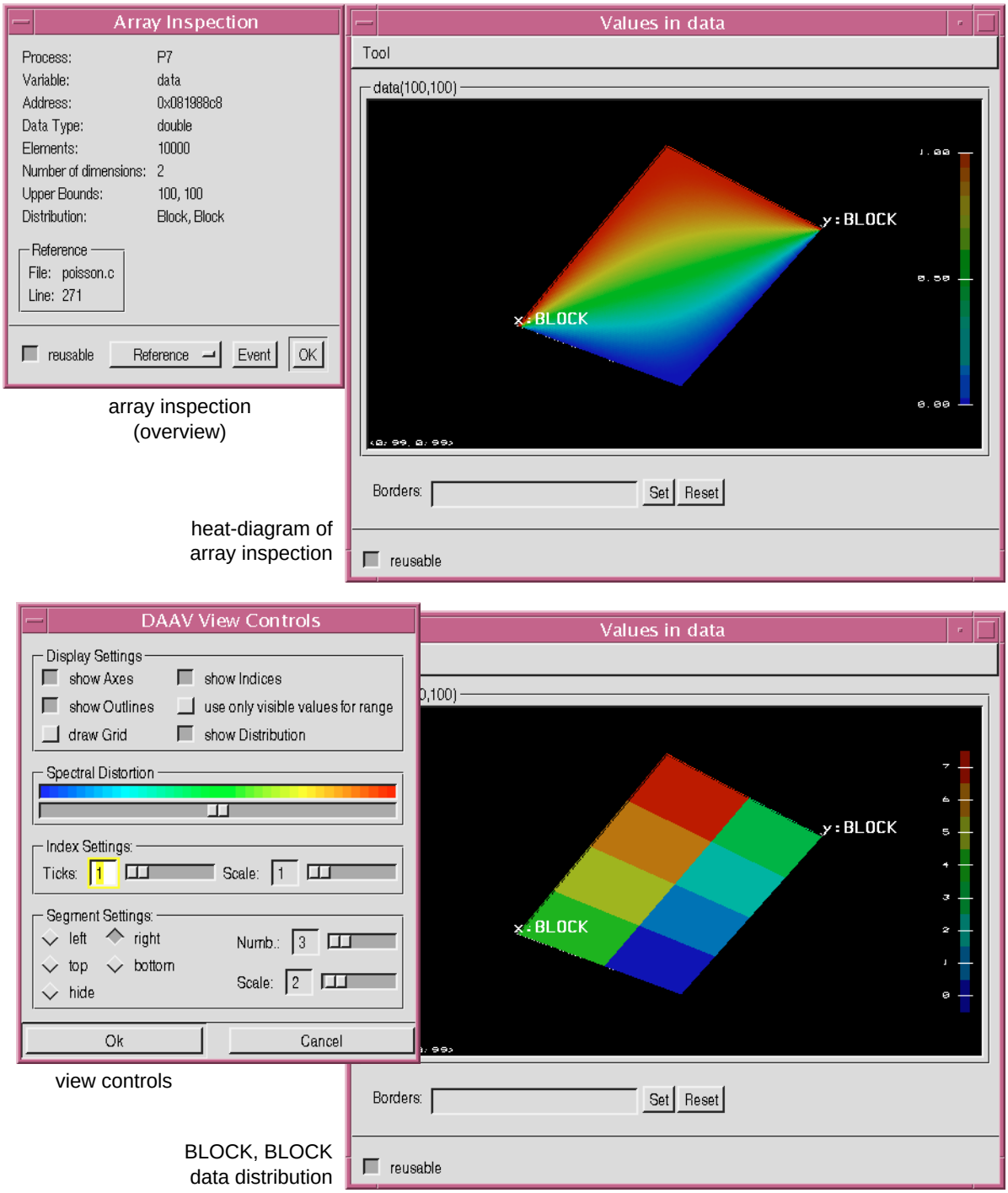


Figure 7-5. Array inspection

view window. The array in the example is two-dimensional, but the tool is also capable of displaying three-dimensional data arrays. For arrays with higher dimensions, up to three dimensions can be selected for visualization. Viewpoint manipulation features like pan, zoom, and rotation are also available and allow additional investigation functionality.

The heat-diagram view can be manipulated with the view controls shown in the bottom-left diagram of Figure 7-5. Visual attributes like axis, outlines, grid, indices and also the spectral

distortion can be modified according to the user's need. In addition, it is possible to display the data distribution instead of the data as shown in the bottom-right diagram of Figure 7-5. In this case, the elements of each process are indicated by the same color, which reveals the HPF-like BLOCK, BLOCK distribution across the 8 processes.

The example in Figure 7-5 shows a heat-diagram of a parallel solver for Poisson's equation, which uses an initial matrix where the top and left border are initialized with high values (=red), while the remainder of the array is initialized with low values (=blue). The results shown in Figure 7-5 represent the array after several iterations of applying Poisson's equation.

In contrast to Figure 7-5, the heat-diagrams of Figure 7-6 show two arrays with incorrect behavior. The upper diagram of Figure 7-6 shows an error, where the exchange of overlapping columns at process borders was not implemented properly. The lower diagram of the same Figure shows another incorrect program's execution, where one row in the computation of Poisson's equation was missing. This example reveals some of the potential of the array visualization capability, because detecting these errors is very easy compared to the debugging activities required without such a feature.

7.4.4 Control Flow Analysis

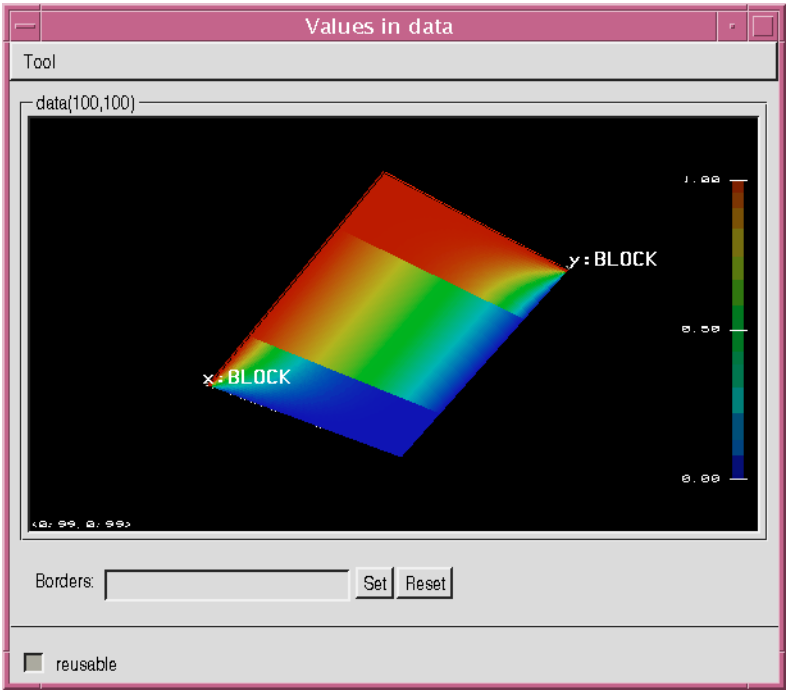
Another area that we have explored in our debugging project is the integration of additional abstraction layers that are closer to the original program code than the event graph. One useful example in this context is our control and data flow analysis tool, that was incorporated in our debugging environment to evaluate the source code of the program. This feature is also selected with the reference button in the event inspection window, and establishes a connection between the inspected event and a number of available source code representations like function call-graph and control-flow diagram. By combining the event graph as a dynamic representation of the program with representations of the original static source code, program comprehension is improved and the program's complexity is supposed to be more manageable.

Other benefits of combining the event graph with the control flow analyzer of the program will be briefly mentioned in the remaining chapters of this thesis. One example are prospective improvements for race condition analysis as discussed in [Kran 96d] and [Kran 97b], where the results of the control flow analyzer are used to reduce the number of race condition candidates (see Section 9.3.3). Another example is the application of the control flow data for program slicing, a technique that was described by Weiser in [Weis 82] and [Weis 84]. Slicing tries to reduce the total number of source lines in a program to those that believed to be relevant for debugging purposes, which has also been described by Grabner in [Grab 99]. Other advanced techniques of using control and data flow analysis for parallel program debugging are described in [Kran 97c]. Since the original approach to slicing is similar to our abstraction mechanisms, we will provide more details in Chapter 10.

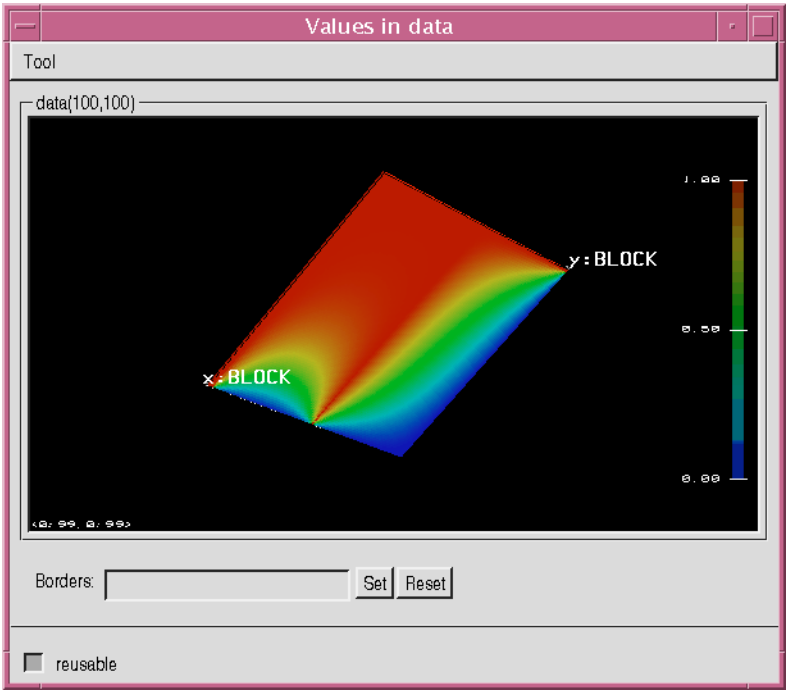
7.5 Breakpointing in the Event Graph

7.5.1 Characteristics of Breakpoints for Parallel Program Debugging

Inspecting events that are graphically displayed in the space-time diagram is a very convenient and simple way of debugging, and many errors in parallel programs can be detected with this feature. However, in many cases the event graph visualization can only serve as the first anchor



Error 1:
Incorrect left-right exchange of overlapping columns at process borders.



Error 2:
Incorrect computation (one row missing).

Figure 7-6. Errors detected with array inspection

for more in-depth analysis activities, which may be required for detecting some errors and locating the original reason for incorrect program behavior. Thus, it is necessary to provide some means of investigating the edges of the event graph, which usually represent a series of state changes, that have not been selected as events of interest by the user.

The interesting edges of the event graph are those that connect events on one process with the sequential happened before relation $\xrightarrow{S}$. This characterizes the edges between two events on one process as sequential code, and qualifies them for analysis with a traditional sequential

debugging tool. The problem is to stop the execution at the event before the selected edge, and to attach the sequential debugger to the edge in order to perform traditional debugging. The solution to this problem for parallel program debugging is based on a two phase strategy: Firstly, a breakpoint is set in the parallel debugger to stop the program's execution. Secondly, control over the selected process is transferred to an arbitrary sequential debugger. This strategy identifies some of the following characteristics of breakpointing mechanisms for parallel program debugging:

- Breakpoints are only meaningful for a running program.
- Breakpoints must be set at places in the code that are actually executed.
- Breakpoints on one process may affect other processes in the program.

The first characteristic defines that the target program has to be executed when breakpoints are applied. This does not require, that the program is executed when the breakpoint is placed. In fact, it should be possible to place the breakpoints in the program before its execution is initiated. However, for breakpoints to come into effect it is necessary that the program is re-executed. This may impose severe problems in the nondeterministic case, because these programs are affected by the "irreproducibility" effect, which means that subsequent executions of the program may reveal different results even if the same input is provided. The "irreproducibility" effect and possibly solutions based on record&replay mechanisms have been described in Section 4.2.6 and will be addressed in respect to our approach in the next chapter. For the remainder of this chapter we assume, that the target program is deterministic and delivers the same behavior during each subsequent execution, provided the same input data are given.

The second characteristic of a parallel breakpointing mechanism is equally important in the sequential case. The problem is to find a position in the code, where the breakpoint can be set, so that the program's execution is stopped at the correct place. This can be difficult in some cases, and is even more critical in parallel programs due to their increased complexity. For that reason we apply the breakpointing mechanism within the graphical display, which should help to manage the complexity of the parallel program. This approach defines another characteristic, namely that breakpoints in parallel programs can only be set at events that occurred during program execution, which limits the possibilities for breakpointing to all the events of interest selected by the user.

7.5.2 Setting Breakpoints in Parallel Programs

The most problematic characteristic of parallel breakpointing mechanisms is that a breakpoint on one process may affect other processes. For example, if a breakpoint is set before a send statement, the corresponding receive operation cannot be satisfied, until the execution on the process with the send is continued. Therefore, a useful breakpointing mechanism should stop the execution on the process with the send statement and on the process with the receive statement, accordingly. As a consequence, breakpoints in parallel programs are usually not considered in isolation, but instead implicitly define breakpoints on other processes as well. Such a set of breakpoints is called a cut, and breakpointing in parallel programs is equivalent to generating corresponding cuts. These cuts may affect all or only a subset of processes, which allows us to distinguish complete and partial cuts. Complete cuts can be defined as follows:

Definition 7-4: Complete cut

A complete cut C in a parallel program executing on *n* processes is a set of breakpoints $b_p \in E$ on each process *p*, so that

$$C = \{b_p \in E\} \qquad 0 \le p < n.$$

These kinds of cuts are called simple cuts according to Grabner [Grab 99]. An overview of comparable breakpointing mechanism is described with a classification of parallel breakpoints by Kacsuk in [Kacs 00]. The top-level distinction divides breakpoints in local breakpoints, message breakpoints, and global breakpoints. Local breakpoints are local to one process and are therefore comparable to our breakpoints b_p. Message breakpoints are established on processes that participate in the same communication event [Wism 97a]. Global breakpoints are those affecting several processes. For this group, Kacsuk also provides additional subclasses to and identifies strongly complete collective breakpoints as the most useful [Kacs 00]. This class of breakpoints is equivalent to our complete cut.

However, complete cuts are only a special case of partial cuts, which describe sets of breakpoints where $|C| < n$. These partial cuts can be defined as follows:

Definition 7-5: Partial cut

A partial cut *C* in a parallel program is a set of breakpoints b_p on each process *p*, where $b_p \in E$ or b_p = *undefined*.

This means, that partial cuts may establish breakpoints only on a subset of the available processes. If a breakpoint on a particular process in a selected partial cut is undefined, it is also undefined what the process will do when the cut is reached. In the best case, the process with the undefined breakpoint will execute until its termination. In the worst case, it will be blocked due to some causal relation to another process containing a breakpoint in the partial cut.

As a consequence, it is obviously preferable to establish only meaningful partial cuts, where the activity of all processes is defined when the cut is reached. For the set of breakpoints to be meaningful, a partial cut is constructed for a selected event in the event graph, and the event responsible for the cut generation is called the breakpoint event. The following activities are required to establish a partial cut in an event graph:

(1) Select a breakpoint event from the event graph, where the program should stop during program execution.
(2) Establish breakpoints on other processes, so that each process stops on a reachable event concerning the breakpoint event selected in (1).

The first requirement describes the fact, that every event in the event graph may possibly be selected as the breakpointing event, which means, that a program may stop anywhere in the event graph. The question is, how to establish breakpoints on other processes, which are useful for the debugging process. The answer to this question is based on what the user wants to do at the breakpoint, which is in the general case not predictable. (A special case of a partial cut with a pre-defined activity at the breakpoints is described in the next chapter for the event manipulation technique.)

7.5.3 Generating Cuts in the Event Graph

In practice, a user may manipulate the program's state so that its behavior is completely different to what has been observed in advance. Therefore, the cut for a breakpointing event can only be generated at events, that cannot be affected by the user's activities after this selected event. This identifies all the events, that are in the timezone PAST of the breakpointing event (see Definition 5-6 on page 142).

Another requirement is, that the breakpoints on the other processes may not stop the execution before an event, that affects the execution of the selected breakpointing event. If the execution would be stopped at such an event, the breakpointing event may possibly not be reached, because it may causally depend on the previous event. This means, that a breakpoint may only be selected at the last event in the timezone PAST. A function for obtaining the last event of a set of events on a process can be defined as follows:

Definition 7-6: Last event

Given the set of events E and a process p, the function last(E,p) is defined as

$$\text{last}(E, p) = \max(E(p)) \text{ with } E(p) = \{e_q \mid q = p\} .$$

$$\text{If } E(p) = \{\ \} \Rightarrow \text{last}(E, p) = \textit{undefined} .$$

The given set of events E can be the complete set of events in an event graph or any possible subset. If E contains all events, last(E,p) delivers the terminal event on process p as specified in Definition 5-9 on page 144. The partial set $E(p)$ is the totally ordered set of events on process p according to the sequential happened before relation. The function max($E(p)$) delivers the last event e_p^i in this set, so that

$$e_p^j \xrightarrow{S} e_p^i \quad \forall e_p^j \in E .$$

Based on this definition the breakpoints of a partial cut for a selected breakpointing event can be generated by providing the set of events in timezone PAST for a selected breakpointing event as follows:

Definition 7-7: Partial cut for breakpointing

The partial cut for a selected breakpointing event $e_p^i \in E$ is the set C consisting of e_p^i on process p, and the events $e_q^j \in E$, so that

$$e_q^j = \text{last}\left(\text{PAST}\left(e_p^i\right), q\right) \text{ for all processes } q \neq p .$$

If the set PAST(e_p^i) contains no elements for a process q, the breakpoint event on process q in the cut is *undefined* as specified in Definition 7-6. This means, that the execution of process q is stopped before its initial event. On contrary, if all events on a process q are included in

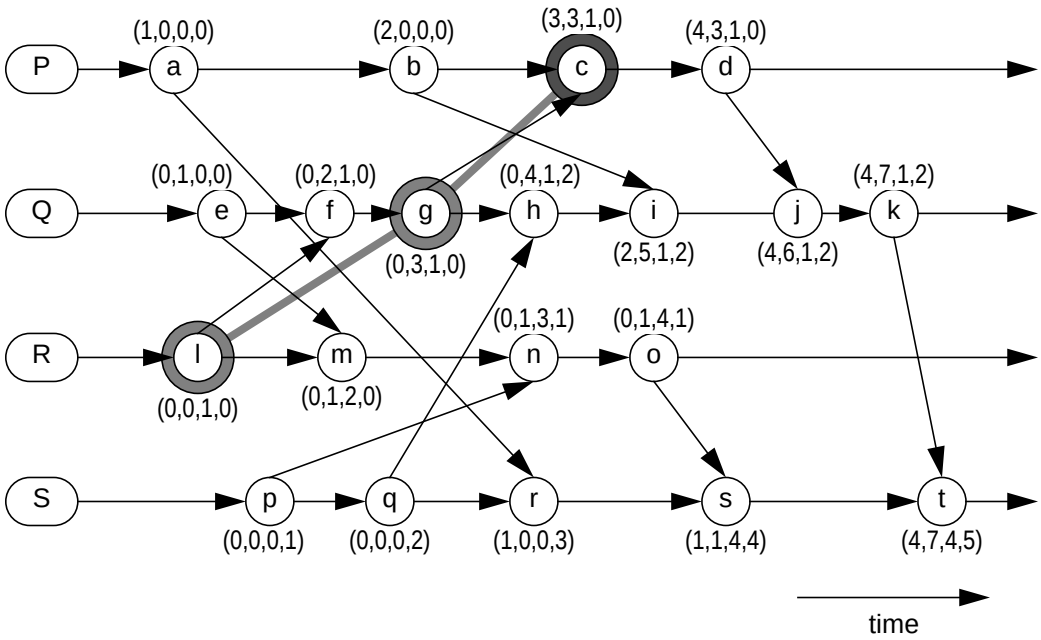


Figure 7-7. Setting a cut (breakpoints on all processes)

PAST(e_p^i), the execution of this particular process is not affected by the breakpoint and the process may actually terminate independently from the activities at the selected breakpoint event.

An example for a cut corresponding to this definition can be found in Figure 7-7. The displayed event graph resembles the example from Figure 5-8 on page 143, where the timezones for a selected event were introduced. In Figure 7-7, the same event *c* is selected as the breakpointing event for a cut. According to Definition 7-7, this event is already one member of the cut, and the others have to be constructed by the debugging tool. The resulting partial cut C is therefore:

$$C = \{c,g,l\}$$

Comparing this example with Figure 5-8 reveals, that the events on processes *Q* and *R* are actually members of the set PAST(*c*), and each of the events *g* and *l* is really the last event on each process. Additionally, the set PAST(*c*) is empty on process S, and no event on this process is included in the cut. All the events left of the cut are necessary to reach the cut during program execution. The events right cannot be used for a cut at breakpointing event *c*, because the program may be manipulated after this event, which could affect all the events in the set FUTURE(*c*) and PARALLEL(*c*).

Algorithm 7-1 provides some more details about the implementation of the cut mechanism. With this simple algorithm, an cut is generated with its breakpoint on each process in the vector *cut[]*. The breakpointing event is stored in the object *breakpoint*, which is an instance of class *event*. Three methods of this class are required in our algorithm, *process_id()*, *vector()*, and *next()*. The method *process_id()* delivers the process identifier of its event object, while *vector()* provides the vector clock as produced by Fidge's algorithm (see Section 5.3.2). With the *next()* function, the successive event on the same process can be selected. Besides these three methods of class *event*, our algorithm also requires an operation ("<") to compare vector

clocks, which effectively checks the precedence condition (see Condition 5-1 on page 139). Additionally, a function to access the first event on a process (= *first()*) is used.

Algorithm 7-1: Constructing a partial cut

```
for (i=0; i < numProcesses; i++) {
        if (i = breakpoint->process_id())
                cut[i] = breakpoint;
        else {
                event = first(i);
                while(event & (event->vector() < breakpoint->vector()))
                        event = event->next();
                cut[i] = event;
        }
}
```

7.5.4 Additional Notes on Breakpoint Cuts

After the cut has been constructed, the remaining problem is where to stop the execution. Clearly, a cut is identified by events which have no duration but only express the occurrence of a state change. Thus, if an event is marked as a breakpoint, the question is whether the program's execution should be stopped before the operation responsible for the event is executed or afterwards. For our breakpoints as defined above, the answer is afterwards. This means, that the event is only the starting point for further activities, e.g. when analyzing the successive edge of the graph with the sequential debugger. The reason for choosing to stop afterwards is the fact, that the breakpointing event must have occurred for the breakpoint to come into effect. If a debugger would stop the program before the operation at the selected breakpoint is executed, as it is the case in sequential debuggers, some events would probably never occur. For example in Figure 7-7, if the debugger stops the execution before event *h* occurs on process *Q*, event *c* on process *P* cannot occur until process *Q* is continued.

In this context, Grabner also distinguishes between inauspicious and auspicious cuts [Grab 99]. With inauspicious cuts, send events may exists that occur before cuts, while their corresponding receive events occur after it. This identifies the so-called pending messages, which are defined as follows:

Definition 7-8: Pending messages

A message initiated by a send event $e_p^i \in E$ with a destination at receive event $e_q^j \in E$ is called a pending message, if the event $b_q \in E$ is a breakpoint on process *q*, so that

$$b_q \xrightarrow{S} e_q^j .$$

In the example event graph of Figure 7-7, the pending messages for the cut at the breakpointing event *c* are the messages sent at the events *a*, *b*, and *e*. Corresponding cuts are called unfavorable cuts by Grabner [Grab 99], because a debugging tool must provide dedicated functionality to deal with pending messages. In concrete, he proposes to save the complete

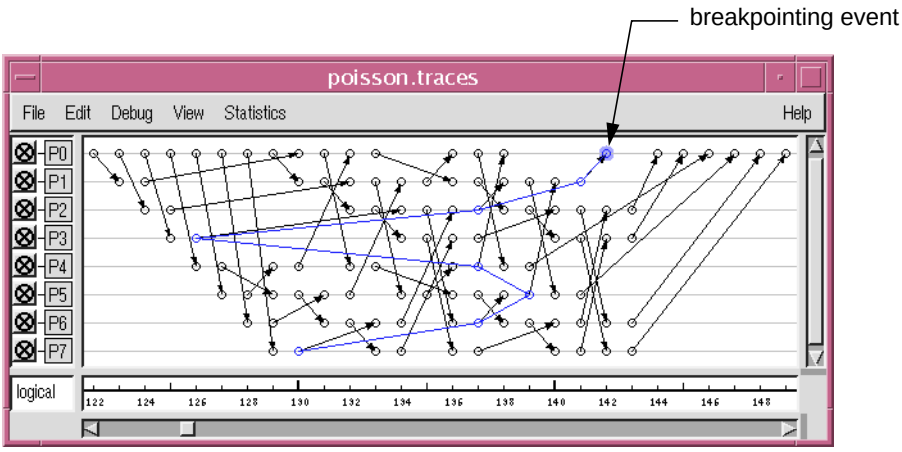


Figure 7-8. Setting a cut in the event graph display

pending message in a temporary buffer, and restore it from there when the program's execution is continued. Consequently, his approach prefers auspicious cuts, which can however not be guaranteed during generation.

With our approach, pending messages do not provide any problems. Handling these messages is completely managed by the underlying MPI system with all its message buffers, which has proved sufficient for all debugging activities, even for analysis techniques concerning racing messages as described in the next chapter. With a sequential debugger it is even possible to manipulate the messages in the message buffer, if the message passing library has been compiled with corresponding debugging information.

An example for our cut implementation is shown in Figure 7-7. The displayed program is again the parallel Poisson equation solver implemented in MPI, that has been executed on 8 processes. In this example, a breakpointing event was selected on process P0. All the other breakpoints required for the cut are automatically generated by the debugging tool. In addition, the events determining a cut are connected by a blue line to indicate the breakpoint.

During re-execution the program is stopped right after the corresponding event has been executed. This is achieved by comparing the current events logical number with the event number given in the cut vector. If both numbers match, the program's execution is stopped and a sequential debugger is attached to the process. Since each process stops at a breakpoint, each process requires its own instance of the sequential debugger. For simplification, the event graph display is also used to manage these multiple instances of the debugger.

Other related areas for cuts are imaginable. In the next chapter we will describe how to use cuts for analysis of nondeterministic program behavior, while Section 10.5.1 provides an idea on how to decrease the program's re-execution time with this debugging feature. Similar to cuts are also consistent checkpoints as described in [KiPa 93][Wang 95].

7.6 Summary

This chapter describes how to use the event graph for parallel program debugging. Based on a simple debugging strategy, different activities for error detection with the event graph model are described. The initial step in our approach is automatic analysis of the event graph, which reveals program failures and communication errors. The results of this analysis task are inte-

grated in the space-time visualization of the event graph, with different colors indicating different effects. The visual representation of the event graph is also the anchor-point for further debugging activities, which are interactively carried out by the user. The most basic feature is interactive inspection of the graphical object, so that event attributes can be investigated. One of the advanced features integrated in our debugging tool, the array inspection facility, is demonstrated together with two examples of typical errors occurring in parallel programs. Besides these inspection activities, the event graph also serves as the interface for placing breakpoints. Since breakpoints in parallel programs may affect more than one process, a corresponding cut mechanism and its implementation is described. Besides these basic debugging features, the event graph is also useful for other activities related to error detection in parallel programs. Some of these advanced error detection facilities will be described in the next chapters.

8. Analysis of Nondeterministic Behavior

As a first extension to the basic debugging activities, we will show that the event graph model is also well suited for analyzing nondeterministic program behavior. The set of nondeterministic events and possible candidates for race conditions are generated automatically. The results of this analysis phase can be visualized and deliver useful information about the possible consequences of nondeterminism established by the code. In addition, event ordering can be manipulated in visual representation of the event graph, and the resulting event order can be verified during subsequent replay steps. This requires a slightly modified breakpointing mechanism and artificial re-executions of the target program.

- Modeling nondeterministic behavior with the event graph.
- Automatic detection of nondeterministic events and race condition candidates.
- Visualization of nondeterministic events and extraction of information.
- Event manipulation and setting of cuts.
- Execution replay of nondeterministic programs.

8. Analysis of Nondeterministic Behavior

8.1 Modeling Nondeterministic Behavior with the Event Graph

8.1.1 Nondeterministic Program Behavior

An important characteristic of parallel programs that can be described with the event graph is nondeterminism. The theoretical foundations and the most basic definitions for this class of programs are discussed in Section 4.2.4. Such programs produce different results in successive program runs, even if the same input data are provided. Therefore, it is unpredictable, which results will be obtained. The term "results" in this context consists of the final results, intermediate results, and interaction patterns between concurrent processes. Correspondingly, a nondeterministic parallel program can be defined as follows:

Definition 8-1: Nondeterministic parallel programs

A program P with a valid input x is defined to be nondeterministic, if

$$\exists x \quad y_1 = P(x) \wedge y_2 = P(x) \wedge y_1 \neq y_2$$

any y_1 and y_2 are either final or intermediate results obtained by executing P on x, or if there exist two event graphs $G_1 = (E_1, \rightarrow)$ and $G_2 = (E_2, \rightarrow)$ for a corresponding execution of $P(x)$, so that

$$G_1 \neq G_2 .$$

Thus, for a program to be nondeterministic, it is sufficient that either the final results, or some intermediate results, or the event graphs of two program executions with the same input are dissimilar. This does not imply, that two subsequent executions of a program may actually exhibit different results, but only that there is a possibility of different results to occur during some program executions. This possibility occurs with a certain probability, and programs with high probability are often characterized as highly nondeterministic programs.

The event graph model cannot capture all the different executions of a program at once, because an event graph describes exactly one execution and each possible execution requires one corresponding event graph. Yet, the event graph allows to describe some basic characteristics of nondeterministic programs, e.g. where nondeterministic behavior may be observed and what determines the different executions. The basic idea is to display the sources of nondeterminisim, and to perform some in-depth studies of the nondeterminism in parallel programs. In concrete, the following activities may be applied:

- Visualization and inspection of nondeterministic events and corresponding race condition candidates.
- Re-execution of nondeterministic programs with previously observed behavior.
- Investigation of artificially manipulated program executions.

These activities will be described in more detail below. Nondeterministic events are those events that may lead to different executions of the observed program, while race condition candidates depict the different possibilities that may occur at nondeterministic events. Program re-execution means, that a program is executed again with the same input data provided. With the artificial manipulation of a program's behavior, different program re-executions will be obtained for one particular set of inputs.

8.1.2 Responsible Events

Before discussing possible investigations strategies based on the event graph model, we have to introduce the set of events that are of interest in this context. The problem is to detect those events, that are actually responsible for nondeterministic behavior, by investigating the following question:

What events may change the state of a program in an unpredictable way?

The answer to this question delivers all the responsible events that must be studied during nondeterminism analysis. These so-called "nondeterministic events" depend on the selected system. For example, on shared memory architectures nondeterminism is introduced as soon as two processes access a shared resource like the system's memory (compare with Definition 4-6 on page 92). Depending on which process accesses the memory first, different behavior will be observed.

Since our approach is targeting massively parallel computing systems applying the message passing paradigm, the following three events are the most important responsible events:

- Random number generation
- Blocking receives
- Nonblocking receives

Random number generation modifies a memory location of the calling process by computing a random number value within a predefined interval. This represents a dedicated function to introduce nondeterminism in a program. The other two operations, the blocking and nonblocking receive, may also modify some memory contents of the calling processes in unpredictable way, if it is possible that different messages are accepted at a selected receive statement. In addition to affecting a process' memory, the nonblocking receive introduces nondeterminism since it is often unpredictable when exactly a message arrives at a corresponding process. These responsible events will be described in more detail below, and we will discuss what other events may be of interest and why most events are not important for nondeterminism analysis.

Please note, that most related approaches in this area treat only a subset of the possible sources of nondeterminism. For example, the technique described in [Neym 99] considers only blocking receives. Consequently, programs including nonblocking receives (or random number generation) may still reveal nondeterministic behavior and are thus irreproducible by their analysis technique. This problem has been noted in [Chas 99], which offers a sophisticated method for any kind of nondeterministic receive operation. Yet, even this approach does not deal with random number generator events and other system calls.

8.1.3 Random Number Generation

Random number generation is an obvious source for nondeterministic behavior that has been explicitly chosen by the user. Some reasons for choosing such a nondeterministic operation have been described in Section 4.2.4, and a corresponding definition can be found in Section 6.2.5.

In principle, random number generation is applied to deliver an arbitrary value within a certain range. The resulting number can then be used to calculate some intermediate results, which may influence the final results of a program. Therefore, such a random number generation may actually represent one of the reasons, why different program executions are observed with the same input data applied. The usage of random number generation does not necessarily influence the structure of the event graph. In fact, with random number generation, equal event graphs may be obtained although different results have been observed.

Debugging a program with a random number generation imposes two important issues. Firstly, for a complete test of the program, all possible results of the random number generator within the specified range would have to be tested in order to guarantee correct behavior for each value. While this can theoretically be achieved with an arbitrary debugging tool, that allows the user to manually set the resulting value of the random number generator before any subsequent computations are performed, it is usually impractical due to the number of possible values within the given range.

8.1.4 Nondeterministic Receives

The other two types of events responsible for nondeterministic behavior are blocking and nonblocking receive events, which are called nondeterministic receive events, accordingly. Chassin et al, who have implemented their MPL* record&replay technique for the same type of receives, call blocking receive events “promiscuous receives” and nonblocking receive events “nonblocking tests” [Chas 99]. In both cases, these events describe the receipt of a message on a particular process. The reason for nondeterminism is established by the fact, that the corresponding statement for these receive events may allow different messages to be accepted. If different messages are accepted, it is also possible that different program executions and/or different intermediate and final results are obtained, assuming that the control- and data-flow after the receive depends on the contents of the delivered message.

Depending to the applied receive operation, a nondeterministic receive event may either be a blocking receive (`receive`) or a nonblocking receive (`test`). The corresponding events have been introduced in Definitions 6-2 and 6-3 on page 152 as follows:

$$e_p^i = \texttt{receive(buffer, source, tag)}$$

$$e_p^i = \texttt{test(result, source, tag)}$$

In case of a blocking receive, the origin of the message is identified with the parameters `source` and `tag`. Thus, a blocking receive is a source of nondeterminism, if there exists more than one message with fitting parameters for `source` and `tag`, that can be accepted when the receive operation is executed. In this case, the program’s behavior is affected by the

contents of the message stored in `buffer`, and different executions may be observed depending on the transferred message.

The nonblocking receive operation is in many aspects similar to its blocking counterpart, and nondeterminism is again introduced if more than one message matches the parameters `source` and `tag`. However, the behavior of a process after the nonblocking receive may not only be affected by the transferred message, but also whether a corresponding message is available or not. Therefore, different program executions may be observed depending on the value `result` of the nonblocking receive operation. We conclude, that nonblocking receive events are nondeterministic for two reasons: Firstly, they may accept a message from different processes with different message tags. Secondly, repeated executions of nondeterministic receives may accept a message during different instances, leading to different event graphs and probably different results, even if the same input data are provided and the parameters of the receive operation identify exactly one message. A debugger addressing nondeterministic behavior at nonblocking receive operations must clearly deal with both cases, and often both cases occur simultaneously.

8.1.5 Additional Nondeterministic Events in Parallel Programs

The responsible events described above represent the most critical reasons for nondeterministic behavior in parallel programs. All other possible events can be either described with similar features as the three groups above or they can never exhibit nondeterministic behavior. For example, we will not consider other sequential sources of nondeterminism as described in [Rons 00], e.g. system calls like `gettimeofday()` or operating system interrupts and signals. We believe, that these reasons can be treated comparable to the random number generator calls.

In fact, there is only one more group of events that may also express nondeterministic behavior: all global operations that collect data from the executing processes[1]. Such operations may lead to nondeterministic behavior, if the arrival of the data from the addressed processes may influence the program or the occurring events in an undefined way. Yet, these operations can be handled comparable to the receive operations defined above, since they are often implemented by combining several basic operations. A crucial issue in this context is the possibility of overlapping, where different operations are concurrently executed and may influence each other, or when user messages interfere with control messages of the underlying system. These complex situations will not be treated any further in this work.

All other events occurring in message passing programs need not be addressed during nondeterminism analysis, because these events may never lead to unpredictable and irreproducible program executions. This assertion stems from the fact, that only nondeterministic events as described above change the state of process through operations, that cannot be determined by the target process itself. All other events deterministically compute their follow-up state.

For example, a send event does not influence the process executing it in a nondeterministic way, because it performs a well-defined operation. It will always lead to the same subsequent state of a process, if the state of the process before the send operation is the same. This statement holds for all other events, that have not been mentioned so far. Therefore, these events

1. The MPI Standard Version 1.1 (http://www.mpi-forum.org/docs/mpi-11-html/mpi-report.html) explicitly states two other function calls - `MPI_CANCEL` and `MPI_WAITANY` - as additional sources of nondeterminism. The former allows to cancel pending messages, while the later waits for any specified send or receive to complete.

can also be re-produced during program re-executions and do not need any special treatment during nondeterminism analysis.

8.1.6 Problems During Debugging

Testing and debugging a program containing any of the events responsible for nondeterministic behavior can be a serious challenge. Firstly, it is possibly that only one of the possible executions is observed all the time, for example, that a nondeterministic receive always accepts the same message during subsequent execution. In this case, the user may not even be aware that different program executions may exist for the same set of input data. Consequently, critical errors occurring only during some of the possible program executions may never be investigated during testing. Secondly, the program may reveal one execution during testing, but may always deliver another execution during debugging, which may be very frustrating and confusing for the user looking for an error in a previously observed execution. Many other examples are imaginable and program's containing several of these responsible events may exhibit a variety of different execution, which are only limited by the number of possible combinations at each nondeterministic event.

In this context, we must also mention a fundamental misbelief associated with nondeterminism, which labels nondeterministic events as errors per se. However, only the statements and their parameters that appoint a nondeterministic event, or the behavior that is effect by this event, may be erroneous, but not the nondeterministic event itself. Thus, for debugging it is important to detect these places of nondeterministic events that occurred during a program's execution. This triggers one of the following reactions[2]:

- Unintended nondeterministic events can be removed.
- Intended nondeterministic events can be ignored.
- All other nondeterministic events are subject to further investigations.

Nondeterministic events that have been placed by accident, whether parameters were used or computed incorrectly or due to programmer's laziness, can be removed immediately. On contrary, if the user is aware about the existence of a nondeterministic event, e.g. when using random number generation or implementing a FIFO queue (first in first out) with blocking receives, the detected nondeterminism can be ignored.

The remaining group of nondeterministic events are those, where the consequences of their existence are initially unknown to the user. This is possible, if the user investigates large programs with very complex program flows or structures, or programs which have been composed by different users. These nondeterministic events are clearly a starting point for further investigations, and the event graph offers some useful investigation mechanisms to study the consequences of theses source of nondeterminism.

Another most critical group of nondeterministic events includes those, that are intended by the user, but reveal erroneous behavior in certain situations. The special problem in this case is, that the user faithfully believes in the program's correctness, and may often not be aware of the consequences. Although our investigations mechanisms can certainly be applied for this

2. The distinction between unintended and intended nondeterministic events is comparable to the same notions for race conditions introduced by Damodaran-Kamal and Francioni in [DaFr 94]. However, they do not identify those races, which cannot be classified as either unintended or intended.

group, their initiation depends entirely on the user choice. Here our only advice can be to apply these mechanisms with the first moment of suspicion.

8.2 Detection of Nondeterministic Events

8.2.1 Evaluation of Responsible Events

With the event graph model obtained for a parallel program's execution, detection of nondeterministic events is a straight-forward process. All events of the graph are checked, and those identifying responsible events like random number generation and message receives are exposed. The random number generation events and the nonblocking receives represent nondeterministic events in any case. The remaining blocking receive operations have to be divided into nondeterministic receives and deterministic receives. Nondeterministic receives are those that may accept different messages during subsequent re-executions, while deterministic receives will always only accept one specific message. This distinction can only be made for one particular execution of a program with a specified set of input data. A receive statement responsible for a nondeterministic receive event during a program execution with one set of input data may deliver a deterministic receive event during a program execution with another set of input data.

Evaluating nondeterministic receives is based on the following two-step strategy, which inspects every receive event of a particular program execution with a given set of input data.

(1) Check, whether the parameters provided at the corresponding receive statement permit different messages to occur.
(2) Verify, if there exists more than one message that may actually be received at the occurrence of the selected receive event.

The first check is used to distinguish between receive statements, that permit only one particular message and those, that would accept different messages to arrive during program re-executions with the selected set of input data. The first set of events are disqualified as possible places for nondeterministic behavior and do not need to be examined any further.

The second verification step is only applied to the group of receives, that could possibly accept more than one message during program re-execution. For each of these receives the actual number of feasible messages are determined, and only receives with more than one potential message are subject for further investigations.

8.2.2 Detecting Nondeterministic Receive Events

Evaluation of nondeterministic receive events starts by checking the parameters of each receive operation. Based on the definition of the generic receive, the important parameters are the `source` and the `tag` parameter. Table 8-2 shows the possible values of these parameters in the standard case as provided by an arbitrary message passing interface. The first column of this table contains the possible values for the parameter `source`, which identifies the origin of a message. The second column contains the possible values for parameter `tag`, which defines the message contents identifier.

Table 8-1: Receive parameters (standard case)

`source`	`tag`	candidate	case	accepted messages
?	?	yes	(a)	from any process with any tag
?	fixed	yes		from any process with fixed tag
fixed	?	yes	(b)	from fixed process with any tag
fixed	fixed	yes		from fixed process with fixed tag

The possible values for source and tag are the constants "?" and the term "fixed". The former represents a so-called "wild card", while the latter depicts any other acceptable but fixed value, which can be a process identifier like the process number for parameter `source` and a valid integer for parameter `tag`. The usage of the wild card constant "?" allows the user to retain the origin and/or the type of the incoming message. Instead, a receive with a wild card for the `source` parameter will accept a message from any process, while a wild card for the `tag` parameter will accept a message with any possible message type. Such receives are called "wild card" receives to indicate this characteristic. In MPI, a wild card receive with an unidentified source is specified with `MPI_ANY_SOURCE`, while `MPI_ANY_TAG` represents a wild card message type.

The existence of such a wild card receive does not necessarily mean, that the program exhibits nondeterministic behavior. For example, if there is only one message being sent to a wild card receive, it will always only be this message that can be accepted. In addition, it does not require that there are any messages being sent to this receive during one particular execution, especially in the case of the nonblocking receive operation. (As mentioned before, the blocking receive operation will infinitely block the process' execution, if no appropriate message arrives).

Checking these parameters for every receive operation will reveal, whether the receive is a possible source for nondeterministic behavior or not. Only receives that qualify as a nondeterministic event, will be checked for the number of arriving messages. In the standard case, every receive operation may accept more than one message, which characterizes each receive as a possible candidate for nondeterminism analysis. This is indicated with the third column of Table 8-2, which contains a "yes" in every row of the table. The consequences of this situation are, that for all receive events the number of incoming messages must be evaluated, which can be quite tedious in reality. As we will see below, most message passing systems include a prerequisite that reduces the number of possible candidates.

The fourth column relates Table 8-2 to Figure 8-1, which contains simplified examples for all possible nondeterministic situations occurring at receive operations. In principle, there are only two situations, that have to be considered. Either, there are two messages arriving from different processes as in case (a), or there are two messages arriving from one process as in case (b).

The event graph in case (a) consists of three processes, with receive event *a* on process *P*. If the source parameter of event *a* contains the wild card constant as indicated in the first two rows of Table 8-2, the corresponding receive operation may accept either the message from send event *b* on process *Q* as in the left event graph, or the message from send event *c* on process *R* as shown in the right event graph. Both event graphs are possible, if the parameter tag is a wild card (as in the first row of Table 8-2) or both messages from events *b* and *c* contain the same fixed tag that is expected at receive event *a* (as in the second row of Table 8-2).

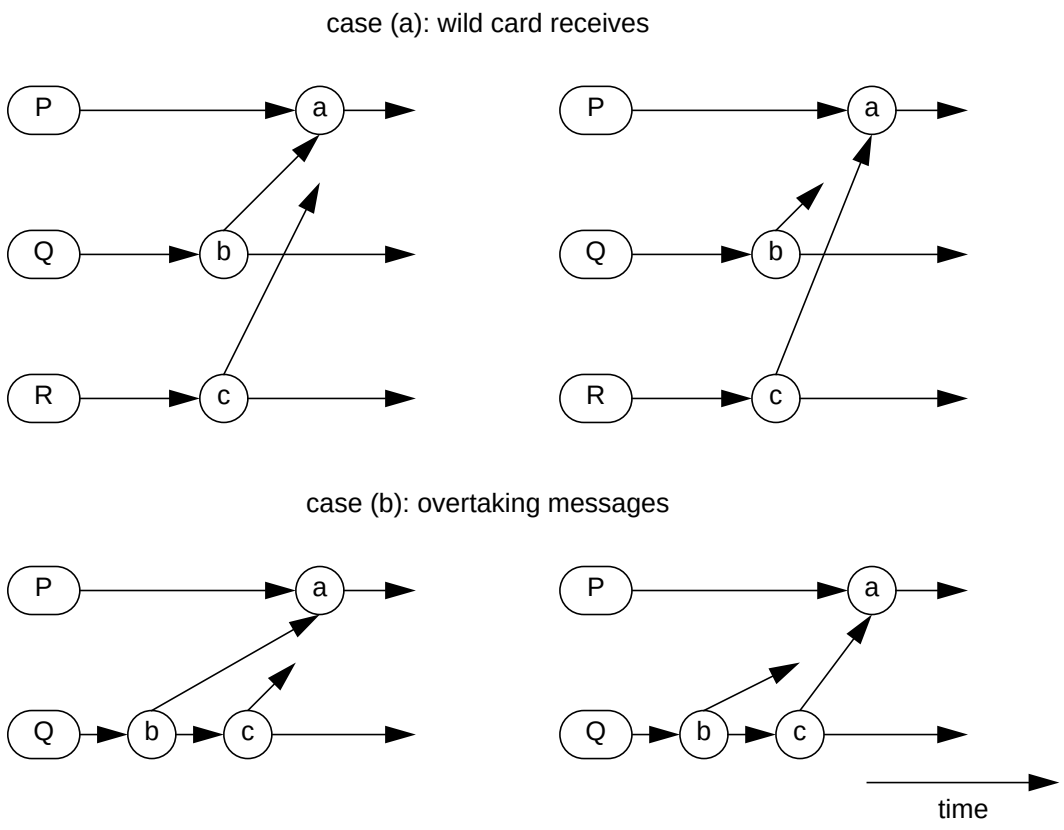


Figure 8-1. Nondeterministic receive events

Example 8-1 contains a small code fragment, that would exhibit the behavior displayed in case (a) of Figure 8-1 if executed on three processes. The applied send and receive statement correspond to the abstract communication events as described in Section 6.2.2 on page 151. The function *process_id()* is used to identify the process on which this code segment is executed. The transferred message is stored in variable *buffer*, and the used message type *tag* is predefined as the constant “1”.

Example 8-1: Nondeterministic (blocking) receives

```
if (process_id() == P) receive ( buffer, ?, ? );
else send ( buffer, P, 1);
```

Case (b) shown in Figure 8-1 displays only two processes, where again the receive event *a* is placed on process *P*. In this case, event *a* accepts only a message originating from process *Q*, but it may accept the message from event *b* or event *c*, if the message type parameter uses a wild card or the tag parameters specified at events *b* and *c* equal the expected, fixed message type parameter provided at receive event *a*. Such situations, when each of two messages originating from one particular process may arrive first at the corresponding receive operation, is called “message overtaking”. Nondeterminism as indicated in case (b) may only occur, if the underlying message passing system permits overtaking of messages, which is not the case in many implementations.

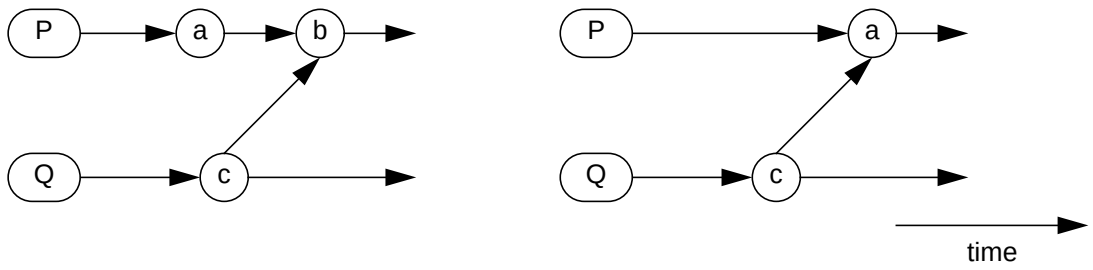


Figure 8-2. Nondeterministic nonblocking receive events

8.2.3 Some Remarks on Nonblocking Receives

The evaluation of the receive events as described above is valid for both, blocking and nonblocking receives. However, there is one characteristic of nonblocking receives, that may have serious consequences for a parallel program's behavior and the corresponding testing and debugging phases. This characteristic stems from the fact, that the mere arrival of a message may influence a process' behavior after a nonblocking receive. This has been ignored in most previous approaches for investigating nondeterministic behavior.

An example is given in Figure 8-2, which again shows the event graphs of two program executions. In the left event graph, two nonblocking receives (events *a* and *b*) occur on process *P*. When the first nonblocking receive is executed, the message from send event *c* on process *Q* has not arrived and thus, event *a* returns without accepting a message. In the right event graph, the message from event *c* has already been delivered, and event *a* can return with a result indicating successful delivery of the corresponding message.

In this example, it does not matter whether the receive specifies a wild card for the origin of the message or not. The program reveals nondeterministic behavior even without a wild card for the sending process. Such a situation, where the receive specifies a concrete origin for a message and the program still exhibits nondeterministic behavior, is given in Example 8-2.

Example 8-2: Nondeterministic behavior at nonblocking receives

```
result = 0;
if (process_id() == P) {
        do {
                result = result + 1;
        } while (test ( buffer, Q, 1 ));
        }
else send ( buffer, P, 1);
```

The output of this code fragment is stored in the variable *result*, which is initialized at the beginning of the segment. The function *process_id()* is again used to distinguish between the sender of the message and the receiver, which performs a nonblocking receive operation (*test*) in a *while*-loop, as long as no message according to its parameters has arrived. These parameters identify the origin of the message as process *Q* and its message tag as the constant "1". The result of this example code depends on the number of iterations, that process *P* waits before the message from process *Q* arrives.

8.2.4 Evaluation of Racing Messages

As mentioned before, random number generation events and nonblocking receives are always subject to further investigations. For nondeterministic receive events that may accept more than one message due to their operational parameters, we have to determine the number of messages, that may actually occur during a program execution with a given set of input data. One of these messages is the so-called "winning message", that has actually been accepted by the selected receive event. Only if there exist more messages than the winning message for a selected receive statement, further investigations have to be applied. This set of messages is called the "racing messages", because they compete for the arrival at a receive, and the message arriving first is the one that will be accepted. Formally, we may define the set of racing messages as follows:

Definition 8-2: Racing messages

The set of racing messages M for a selected receive event $e_p^i \in E$ with

$$e_p^i = \texttt{receive(buffer, source, tag)}$$

consists of the send event $e_q^j \in E$ responsible for the "winning message", where

$$e_q^j \overset{C}{\rightarrow} e_p^i$$

and of all events $e_r^k \in E$, where

$$e_r^k = \texttt{send}(\texttt{buffer}, p, \texttt{tag}) \wedge e_r^k \in \text{PARALLEL}\left(e_p^i\right).$$

We assume, that the parameter `source`, the process identifiers p and q, and the parameter `tag` are set according to Table 8-2. Besides that Definition 8-2 is specified with a generic blocking receive operation, but is also equally valid for generic nonblocking receives by replacing the receive statement with the following test statement:

$$e_p^i = \texttt{test(result, source, tag)}$$

An important issue of Definition 8-2 for the evaluation of the racing messages besides the parameters described in Section 8.2.2 is the fact, that messages are uniquely identified by their send event, and the set of racing messages M actually contains the send events associated with these racing messages. In addition, only messages from send events in the set PARALLEL(e_p^i) race with the "winning message" that has been accepted during program observation[3]. Although this seems trivial at first glimpse, it is very important for nondeterminism analysis. Only messages initiated by send events that did not happen before the selected nondeterministic receive, and that do not depend on this receive, may deliver messages to that receive. Therefore, each element in PARALLEL(e_p^i) would require a dedicated testing cycle in order to verify program correctness. As we will see below, it is the number of elements in

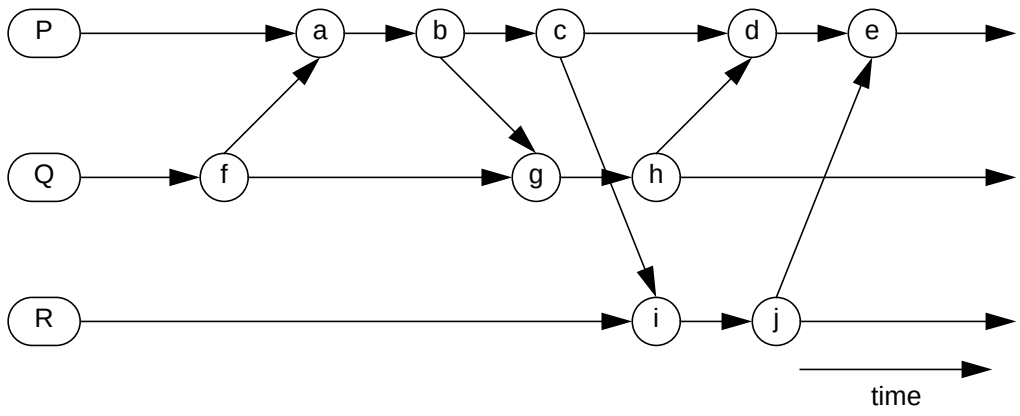


Figure 8-3. Nondeterministic receive without effect

PARALLEL(e_p^i) that may impose a heavy load on users debugging parallel programs, and it is of major importance to reduce this number as much as possible.

Based on the set of racing messages, the receive operation that may reveal nondeterministic behavior can be identified with Definition 8-3. Such nondeterministic receives are called race condition candidates, because they represent a condition that establishes the race between the incoming messages. Therefore, a race condition candidate represents the actual instance of a race condition as defined in Definition 4-5, or related to message passing systems as message race identified in Definition 4-7.

Definition 8-3: Race condition candidates

> An nondeterministic receive event $e_p^i \in E$ with a set of racing messages M is a race condition candidate, if $|M| > 1$.

This means, that the set of racing messages M is only meaningful, if $|M| > 1$. All receives where $(M = \{\ \}) \vee (|M| = 1)$ can be ignored, because they do not reveal nondeterministic behavior for the selected set of input data, even though the receive operation would permit it in the first place. In case, there is only one message than can be accepted at a particular nondeterministic receive, this message is always the winning message and the corresponding receive event cannot exhibit nondeterministic behavior during subsequent re-executions.

An example for a situation, where only one racing message exists, is shown in Figure 8-3. The space-time diagram shows an event graph with 3 processes. Assuming that event *a* is a nondeterministic receive event, than this receive event will always only receive the message from send event *f*, because all other send events transmitting messages to process *P* (e.g. events *h* and *j*) happened after event *a*. Consequently, their messages cannot be received at event *a* and its set of racing messages *M* contains only the message transmitted by event *f*.

With the critical nondeterministic receives identified with Definition 8-3, and the other responsible events, random number generation events and nonblocking receives, evaluated in

3. Please note, that there is also the possibility of messages racing on the process, that contains the selected nondeterministic receive. For example, if the process sends two successive messages to itself. In that case, the set of racing messages also contains all the messages, that have been sent but not received, when the target receive is inspected. However, this special case of message passing is rather questionable in practice, and we won't consider such racing messages in the remainder of this work.

Section 8.2.1, we have identified all the events required for nondeterminism analysis. After these events have been detected, they can easily be highlighted in the graphical representation of the event graph to attract the users attention.

8.2.5 Simplifications for Evaluation of Race Condition Candidates

The approach described in the previous sections is valid for any message passing system, that offers similar point-to-point communication operations. However, during our practical implementations we noticed a characteristic available in message passing systems, that allows to include some simplifications and improvements over the standard case. This idea is described in the following paragraphs and is connected to the fact, that many systems do not permit overtaking of messages as mentioned in Section 8.2.2.

Overtaking of messages is a feature that imposes a high potential for erroneous behavior. As can be seen from Table 8-2, it means, that any program containing two sends with equal destination process and equal message type and a corresponding receive may reveal nondeterministic behavior. Since this is very likely, most parallel programs would be nondeterministic, corresponding errors would occur more frequently, and nondeterminism analysis would be much more emphasized. Consequently, most message-passing implementations are prepared for this eventuality. For example, the standard Message Passing Interface [MPI 95] clearly states, that every valid MPI implementation must offer non-overtaking message transfer:

Excerpt: from the MPI Standard 1.1,
Section 3.5. “Semantics of point-to-point communication”,
http://www.mpi-forum.org/docs/mpi-11-html/node41.html

> *“A valid MPI implementation guarantees certain general properties of point-to-point communication, which are described in this section.*
> *Order: Messages are non-overtaking: If a sender sends two messages in succession to the same destination, and both match the same receive, then this operation cannot receive the second message if the first one is still pending. If a receiver posts two receives in succession, and both match the same message, then the second receive operation cannot be satisfied by this message, if the first one is still pending. This requirement facilitates matching of sends to receives. It guarantees that message-passing code is deterministic, if processes are single-threaded and the wild card* `MPI_ANY_SOURCE` *is not used in receives.”*

A similar prerequisite is provided by the Parallel Virtual Machine PVM [Geis 96], which defines the message order of point-to-point communication functions as follows:

Excerpt: from the PVM 3.3 Manual pages,
`pvm_send` - Immediately sends the data in the active message buffer,
http://www.epm.ornl.gov/pvm/man/pvm_send.3PVM.html

> *“The PVM model guarantees the following about message order. If task 1 sends message A to task 2, then task 1 sends message B to task 2, message A will arrive at task 2 before message B. Moreover, if both messages arrive before task 2 does a receive, then a wild card receive will always return message A.”*

Besides the fact, that the standard MPI and the pseudo-standard PVM explicitly require non-overtaking of messages, an implementation of this feature is rather trivial. In principle, the goal is to establish the happened before relation on distinct processes as defined by Lamport [Lamp 78] and introduced in Definition 5-4 on page 140. If the sending process transmits all messages with a piggybacked sequence number that is increased after transfer initiation, all the receiver has to do is to accept only messages with the correct follow-up sequence number.

Sometimes, the message order is implicitly taken care of by the underlying hardware architecture. An example is the nCUBE 2 distributed memory multiprocessor machine, which offers its own proprietary communication library for message passing programming [nCUB 92]. On this platform, messages are non-overtaking because of the applied message routing algorithm, which determines the path a message takes through the interconnection network. In case of the nCUBE 2, path creation is established with the deterministic e-cube routing mechanisms. Using this technique, every message will travel the same path through the network, and two messages transmitted from one node of the nCUBE to another one will always arrive in the same order.

Based on the abstract event graph model, non-overtaking message transmission can be defined as follows:

Definition 8-4: Non-overtaking message transmission

For events $e_p^i, e_p^j \in E$ with $i \neq j$ on process p and events $e_q^k, e_q^l \in E$ on process q

let $e_p^i = \texttt{send}(\texttt{buffer}, q, \texttt{tag}) \wedge e_q^k = \texttt{receive}(\texttt{buffer}, p, \texttt{tag})$ so that

$$e_p^i \overset{C}{\rightarrow} e_q^k$$

and $e_p^j = \texttt{send}(\texttt{buffer}, q, \texttt{tag}) \wedge e_q^l = \texttt{receive}(\texttt{buffer}, p, \texttt{tag})$ so that

$$e_p^j \overset{C}{\rightarrow} e_q^l$$

then the underlying message passing system offers non-overtaking message transmission, if it guarantees that

$$(i < j) \Rightarrow (k < l) \ .$$

This definition of non-overtaking message transmission also affects the set of messages, that may be accepted at a particular receive event. In concrete, only case (a) from Figure 8-1 is still a valid case of nondeterministic behavior. Case (b) from Figure 8-1 is no longer a race condition and the receive event *a* on process *P* will always receive the message sent by event *b* on process *Q*. Consequently, row three and four of Table 8-2 have to be modified, because receives with fixed source processes will always receive only one message and therefore do not qualify as a race condition candidate. The modification of Table 8-2 required by non-overtaking message transmission are reflected in Table 8-2, which shows the parameters `source` and `tag` for a receive event in message passing systems, that prohibit message overtaking.

Table 8-2: Receive parameters (non-overtaking case)

`source`	`tag`	candidate	accepted messages
?	?	yes	from any process with any tag
?	fixed	yes	from any process with fixed tag
fixed	?	no	next available message from fixed process with any tag
fixed	fixed	no	next available message from fixed process with fixed tag

The non-overtaking characteristic also affects the set of racing messages (and thus the number of race condition candidates), so that $|M| \le n$, where n is the number of available processes. The resulting set of racing messages for a selected receive event e_p^i will be called "restricted racing messages" and contains the following elements:

- The send event e_q^j responsible for the "winning message", where

$$e_q^j \overset{C}{\rightarrow} e_p^i$$

- All events $e_r^k \in E$, where

$$e_r^k = \texttt{send}(\texttt{buffer}, p, \texttt{tag}) \wedge e_r^k = \text{first}\left(\text{PARALLEL}\left(e_p^i\right), q\right).$$

An important issue is the function first(PARALLEL(e_p^i)), which delivers the first element of the timezone PARALLEL for a given event $e_p^i \in E$ as defined by the happened before relation. The definition of this function is similar to the function "Last event" as specified in Definition 7-6 on page 189, which has been introduced to establish a partial cut in a parallel program:

Definition 8-5: First event

Given the set of events E and a process p, the function first(E,p) is defined as

$$\text{first}(E, p) = \min(E(p)) \text{ with } E(p) = \{e_q \mid q = p\}.$$

$$\text{If } E(p) = \{\ \} \Rightarrow \text{first}(E, p) = \textit{undefined}.$$

The given set of events E can be the complete set of events in an event graph or any possible subset. If E contains all events, first(E,p) delivers the initial event on process p as specified in Definition 5-9 on page 144. The partial set $E(p)$ is the totally ordered set of events on process p according to the sequential happened before relation. The function min($E(p)$) delivers the first event e_p^i in this set, so that

$$e_p^i \xrightarrow{S} e_p^j \quad \forall e_p^j \in E \ .$$

The restricted set of racing messages determines also the race condition candidates as specified in Definition 8-3. With fewer numbers of racing messages, we will also obtain fewer race condition candidates. The corresponding race condition candidates are called “restricted race condition candidates” to emphasize this characteristic.

The benefits of this reduction on systems that permit message overtaking are obvious. Firstly, programs using receives are only nondeterministic, if the receive applies a wild card for the source parameter (see Table 8-2). Secondly, since the number of racing messages is often far less, which also means reduced possibilities for nondeterministic behavior. Thirdly, the evaluation of racing messages is simplified, because only the first message of each process qualifies as a racing message for a selected race condition candidate.

Besides that, non-overtaking is a feature that is implicitly provided by most available message passing systems, and it seems appropriate to limit our focus on these kinds of systems. Therefore, the remainder of this work will focus mainly on nondeterministic behavior in systems that prohibit message overtaking, and the terms “racing messages” and “race condition candidates” will be used to identify “restricted racing messages” and “restricted race condition candidates”, respectively. If necessary, remarks referring to systems with message overtaking will be made explicitly.

8.2.6 Visualization of Nondeterministic Behavior

After determining the race condition candidates and the racing messages as described in the previous sections, the resulting events can be highlighted in the graphical display of the event graph. The presented space-time diagram can be analyzed and events can be inspected as described in Section 7.4.1. Examples for a random number generation event and several non-blocking receive events are included in Figure 6-4 on page 169.

Nondeterministic (wild card) receives and racing messages are shown in Figure 8-4, which contains an event graph for a simple parallel algorithm as executed on 8 processes. The program under consideration is an implementation of standard distance doubling algorithm:

Example 8-3: Distance doubling

> Consider a MPI program that is executed on n processes and each process holds some arbitrary local data. The goal of distance doubling is to determine some global characteristics of these distributed data, so that afterwards each process holds the same global result. Some applications for this algorithm would be to determine the maximum or minimum of some integer values.

The code of this example can be found in Appendix B. Every receive operation is carried out with a wild card as source parameter, which is also visible in the top-left screenshot. The top-right window shows the event information corresponding to a receive event, that has been selected on process *P4*. This window contains the number of racing messages (in this case: 3) out of the maximum number of racing messages possible for that particular receive (in this case: 8). By clicking on the “Show”-button in the event information window, the space-time diagram is updated to display the racing messages for the currently selected receive event. The resulting diagram is presented in the bottom-left screenshot, which shows the three racing messages originating from process *P0*, process *P2*, and process *P3*. These are the messages that

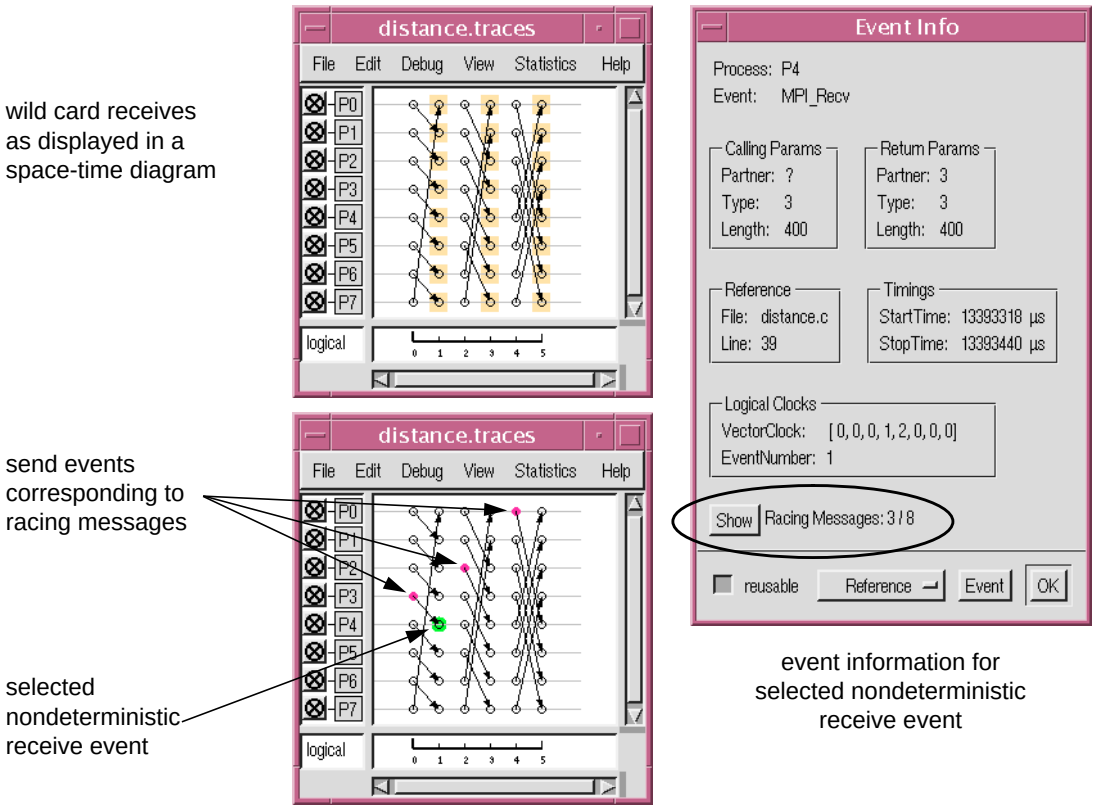


Figure 8-4. Wild card receives and racing messages

race to be accepted at the receive event on process *P4* and may occur in different order during program re-executions. During the observed execution the "winning message" was the one from process *P0*. Please note, that instead of highlighting the edges that represent the racing messages, the corresponding send events are displayed with a pre-defined color.

In the example of Figure 8-4 each of the $n \log n$ receive events, with $n = 8$ the number of processes, is a wild card receive. Consequently, the shown event graph represents only one possible execution, and many different executions could be observed in reality. Some other example executions of the same program are given in Figure 8-5. While the event graph of Figure 8-4 shows the communication pattern that is intended by the distance doubling algorithm, the patterns in Figure 8-5 represent erroneous executions, where some messages are retrieved before intended. In practice, the user may always retrieve the execution shown in Figure 8-4, and may mistakenly assume, that the program will always deliver the correct results. However, when decreasing the time between the communication events, other executions as shown in Figure 8-5 may emerge and affect the results of the program. (The presented example represents an actual case as observed during programming courses of students attending a lecture on parallel software engineering at the Johannes Kepler University Linz).

The benefits of the visualization are obvious. Often, the mere graphical representation offers some very interesting information about the program's nondeterministic properties, especially to distinguish between unintended nondeterminism and intended nondeterminism as described in Section 8.1.6. On the one hand, events that exhibit nondeterministic behavior not intended by the user can be eliminated immediately. A possible solution is to remove the wild card from the `source` parameter and to replace it by the correct address of the sending process. On the other hand, nondeterministic events that have been intended by the user may be

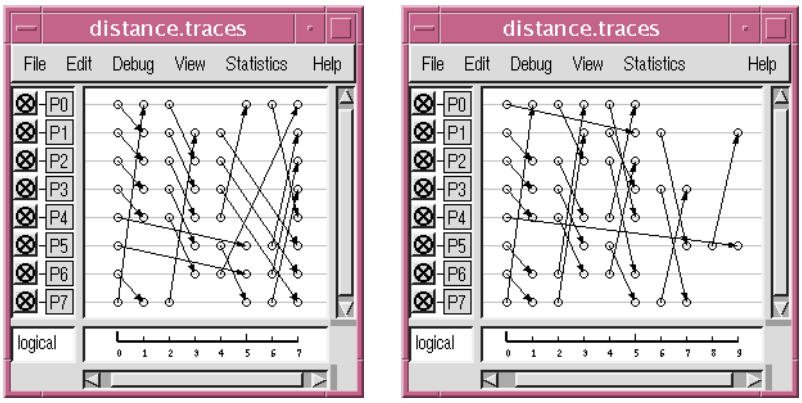


Figure 8-5. Distance doubling: incorrect nondeterministic executions

left out of account. This is the case for example when implementing a FIFO (first-in-first-out) queue with wild card receive events.

The problematic nondeterministic events are only those, that are neither unintended nor intended, or at least may not be identified as one of these two groups immediately. For these events it is undecidable, whether changes of the order arrival of racing messages may lead to an erroneous execution or not. As described in Section 8.1.6, these are the events that are subject for further investigations. Our approach to investigate these situations is based on two activities, program re-execution and event manipulation, which will be described in the next two sections.

8.3 Re-execution of Nondeterministic Programs

8.3.1 Equivalent Execution

Detection of nondeterministic behavior and visualization of the corresponding event graph model is an important step for understanding a program's execution during error detection. At the same time it is often only a first step for more in-depth analysis of the target program's behavior. In case of nondeterministic programs this analysis targets the following two distinct activities:

- Analyzing a program execution as soon as erroneous behavior has been observed.
- Testing different instances of a program execution for a selected set of input data.

These two activities are also represented in Figure 4-4 on page 103, where the traditional testing cycle with iterations for different sets of input data is accompanied by two other cycles, the debugging cycle and the nondeterminism testing cycle, one for each of the two areas mentioned above.

The necessity to re-execute a program and the irreproducibility effect associated with nondeterministic code defines the requirement for a facility that ensures equivalent program executions. Such a facility is necessary for repeated executions of a program with a selected set of

input data. A solution is offered by record&replay mechanisms, which have been discussed together with related work in this area and the foundations of the irreproducibility effect in Section 4.2.6. In this section, we have also stated a first definition of an equivalent execution as offered by Leu [Leu 90]. Based on the event graph model this definition can be refined as follows:

Definition 8-6: Equivalent execution (based on the event graph model)

> Two executions of an arbitrary parallel program are equivalent, if their event graphs are isomorphic. This means that there exists a bijection $G \rightarrow G'$ that preserves the happened before relation and all existing vertices.

This adapted version of Definition 4-8 also outlines the basic requirement for record&replay mechanisms, which must guarantee such an equivalent execution during repeated program runs with the same input data. In order to fulfil this requirement, record&replay methods apply the following two-step approach:

- Derive the order of critical elements at nondeterministic events during an initial record phase.
- Enforce the same order of the critical elements at the corresponding nondeterministic events during subsequent replay phases.

Related to the event graph model, the critical elements that must be derived during the initial record phase are the results of those events, that may lead to a different order during program re-execution. These events are the responsible events that have been defined in Section 8.1.2. Since the occurrence of these responsible events may change the program's behavior even if the same input data is provided, a replay phase must enforce the same behavior to ensure an equivalent execution.

8.3.2 Record Phase

The first phase of a record&replay mechanism is performed by a dedicated monitoring tool. In principle, any monitoring tool can be applied as long as it retrieves the data required to enforce the same program run during the subsequent replay phases, and most monitoring tools sufficiently fulfil this requirement. However, in Section 9.2.4 we will show, that it may be beneficial to limit the monitors data to the minimal set of items that permits program re-execution.

Besides the requirement to trace responsible events, it is also important to integrate the monitor in the testing cycle instead of the debugging cycle. This requirement is also visible in Figure 4-4, which shows the testing and debugging cycles for parallel programs. For deterministic parallel programs, the monitor phase will not be needed until an error has been detected, since these programs can be re-executed at will and always deliver an equivalent execution. On contrary, a nondeterministic program requires an equivalent execution to test different program executions for a previously selected set of input data. This can only be assured, if the monitor performing the record phase is integrated in the testing cycle at the very beginning of the testing and debugging phase.

Corresponding to the responsible events, the record phase must deliver the following data associated with the nondeterministic events, which also represent the minimum requirements for equivalent program re-execution:

- Random number generation event:
  - Generated return value
- Blocking receives:
  - Accepted parameter partnerID (=`source`): origin of received message.
  - Accepted parameter messageType (=`tag`): type of received message.
  - Logical clock of partner process.
- Nonblocking receives:
  - Value `result`: status of message receipt.
  - Accepted parameter partnerID (=`source`): origin of received message.
  - Accepted parameter messageType (=`tag`): type of received message.
  - Logical clock of partner process.

The example described in Chapter 6 already contains all these necessary event data without explicit focus on a program's nondeterministic behavior. Therefore, the generated trace files will already provide sufficient information to re-execute the observed program execution.

The simplest nondeterministic item is random number generation, and its important data is the generated random number value. If the same random number is returned during each re-execution, the program's behavior will always be the same.

The required parameters for both types of receive events have been chosen according to Tables 8-2 and 8-2, and are also included in Table 6-2 on page 163. These data represent the unique characteristics for the accepted message, which means that a message can be characterized by this small set of parameters. The replay mechanism can easily identify the correct message during re-execution, while all other event attributes will be regenerated.

Another distinction concerns the underlying message passing system, which carries out the receive operations. Systems permitting message overtaking require the availability of all these parameters for each receive operation. On system where message overtaking is prohibited, blocking and nonblocking operations have to be distinguished. In this case, only blocking receive operations containing wild cards (for `source` and/or `tag`) require the stated parameters, because all other receives will deliver the same message during each re-execution. For nonblocking receives, the parameter `result` will always be required, while the other parameters are only needed for operations applying wild cards, if `result` returns successful message arrival (`result` = `true`).

Please remember, that for each communication statement we distinguish supposed parameters and accepted parameters as described in Section 6.5.1. During the record phase, the supposed parameters are checked whether they contain a wild card or not. If a wild card has been applied, the accepted parameters describe which process transmitted the accepted message and which message type was provided. Therefore, these accepted parameters are needed during the replay phase to identify the message, that must be accepted.

8.3.3 Replay Phase

The record phase of a record&replay mechanism is completed, if the program's execution is terminated. At that point, corresponding trace data is available, that can be used as a constraint for subsequent replay phases. This second phase is carried out by a dedicated replay tool, which enforces the same order of events as observed during the initial record phase. Such a replay tool performs to some extent similar activities as a monitoring tool:

- Instrumentation
- Observation
- Steering

The first two activities of a replay tool are instrumentation and observation. Instrumentation is required to include the replay functionality in the target application, while observation performs the program monitoring during the re-execution, which is necessary to verify the equivalence of the initial record phase and subsequent replay phases. Both tasks are almost identical to the record phase, and in fact, the main difference is the required functionality for program steering. As a consequence, the instrumentation phase for the monitor and the replay tool is often implemented similarly or even simultaneously.

The record&replay tool integrated in our debugging environment is the NOndeterministic Program Evaluator NOPE (see Section 4.6), which represents such a combined approach. As described in Section 6.3.1, this tool instruments the code by substituting the original function calls (e.g. `MPI_Isend`) with a corresponding monitor function (e.g. `mon_MPI_Send`). During execution, this monitor function performs the same operations as the original function call, and additionally carries out the program's observation as required by the monitor. In practice, the original function is substituted with a function pointer, that gets initialized during program start-up with a function corresponding to the intended operation. If NOPE is used during the record phase, `mon_MPI_Send` will point to a function performing program observation. If NOPE is used during the replay phase, an appropriate replay function will be appointed instead. This function pointer approach offers several benefits:

- It allows to integrate the monitor with the replay tool, which allows to share code and reduces implementation work.
- It allows to switch between different operations during program execution.
- It is a very convenient way to appoint different functions depending on the required operation.

The advantage to integrate the record and the replay phase within one code have been described above. The second benefit to switch between different operations during program execution is a basic requirement needed for the event manipulation feature described in the next section. The last possibility to implement different functions which can be hooked into the executable during the program's start-up allows to provide dedicated functions with different characteristics for different activities. This was especially useful when testing different monitoring techniques or methods to determine the amount of observation data and the overhead generated by the monitoring tool. In practice, we implemented over 20 different monitor functions for the basic communication events, and at least 5 functions for each observed event.

Besides performing the original operation, the record and the replay phase have to accomplish their dedicated functionality. We have discussed this for the record phase in Section 6.4 and Section 8.3.2. During replay the required functionality is to perform an equivalent execution, which is what defines the term "steering" above. These steering activities are applied to guide the re-execution of the program at nondeterministic events. This is achieved with two activities at each event:

- The original behavioral data as observed during the record phase and stored in the event attributes is retrieved from the tracefiles.
- The event attributes are used as a constraint for the operation of the event.

The replay mechanism must adapt itself to the characteristics of each event. If the current event is a random number generation event, the replay tool must return the same random number value as has been observed during the initial record phase. Similarly, a nonblocking receive must always return `false`, when the event trace indicates that its equivalent counterpart in the original function returned `false`, because the requested message was not available when the nonblocking receive was executed. If the parameter `result` of a nonblocking receive is `true`, which means that an appropriate message has arrived, or the event under consideration is a blocking receive using wild cards, the replay tool must return the same message, that was received at the corresponding event during the initial record phase. This is achieved by replacing the wild cards of the original function call with the accepted parameters for `source` and `tag` as stored in the event trace[4]. Since these parameters and the sequence number of the current event uniquely identify the accepted message, the replay tool will deliver an equivalent message to the calling process as originally observed. Please note, that this may require to transform the original nonblocking receive into a blocking receive, if the required message is not available when the nonblocking receive is executed.

Table 8-3 shows an example for the record&replay mechanism. The first row presents the event observed during the record phase, that must be enforced during the replay phase. The second row shows an arbitrary function call, that may be responsible for generating the event in the first place. Wild card parameters have been specified with the “?” constant. The remaining third row displays results, that may have been observed during the initial execution, when the function call responsible for the observed event was executed. This last row represents the data, that must be generated during re-executions of the program in order to guarantee an equivalent execution.

Table 8-3: Example for record&replay mechanism

event type	function call	results
random number generation	`random()`	`random()=5`
nonblocking receive	`test(result,source=?,tag=?)`	`result=false`

nonblocking receive	`test(result,source=?,tag=?)`	`result=true`
		`source=P`
		`tag=3`

blocking receive	`receive(buffer,source=?,tag=?)`	`source=P` `tag=3`

4. In message-passing systems, where overtaking of messages is permitted, the replay tool must also take care of the event order from messages originating from one process, which can be achieved with the accepted logical clock value of the partner process. In this case, the replay tool has to manage the incoming message buffer and must restore the same message order as observed during the initial record phase.

8.4 Debugging in the Presence of Nondeterminism

8.4.1 Evaluation of Race Condition Candidates

In the previous sections of this chapter, we have shown how to analyze nondeterministic programs and how to re-execute them for in-depth analysis with a debugging tool. Another essential part of this in-depth analysis concerns nondeterministic events, which are characterized as events with unknown consequences that cannot be classified as intended or unintended. The central questions for studying this group of events can be formulated as follows:

> What would have happened, if the choice taken at a nondeterministic event would have been different from what has been observed?

Corresponding to our set of nondeterministic events this means:

- What would have happened, if the random number generator would have delivered another possible number?
- What would have happened, if a message was available/not available when a nonblocking receive was executed?
- What would have happened, if messages arrived in different order at a wild card blocking/nonblocking receive?

Answering these questions for each nondeterministic event is a non-trivial problem which requires dedicated functionality from a debugging tool. The principal idea of our unique solution to this problem is to modify the originally observed results and enforce artificially generated results during subsequent replay phases. In practice, we propose the following steps for analyzing nondeterministic parallel programs:

(1) Instrument the program before testing to include the record functionality.
(2) Execute the program during the testing cycle to generate event data.
(3) Automatically evaluate nondeterministic events and racing messages.
(4) Visualize the resulting event graph and highlight the results of step (3).
(5) Remove unintended race conditions and ignore intended race conditions.
(6) Inspect the remaining events and modify the observed event attributes.
(7) Re-execute the program with the modified event graph as a constraint.
(8) Evaluate the consequences of the artificially introduced modifications.

The eight steps described above can also be related to our extended testing and debugging cycles as introduced in Section 4.3. For illustration purposes we have extended Figure 4-4 into Figure 8-6 with annotations corresponding to the eight steps. In addition, we have split up the event manipulation step of Figure 4-4 to reflect steps (3) to (6).

The two main activities to answer the questions given above are event manipulation and re-execution of the artificially generated program flow. These two activities will be described in more detail below. Besides that, we will introduce a modified cut mechanism, which is required to distinguish between those parts of the program, that shall be executed as defined in the event graph, and those parts that may reveal unpredictable behavior due to the applied modifications.

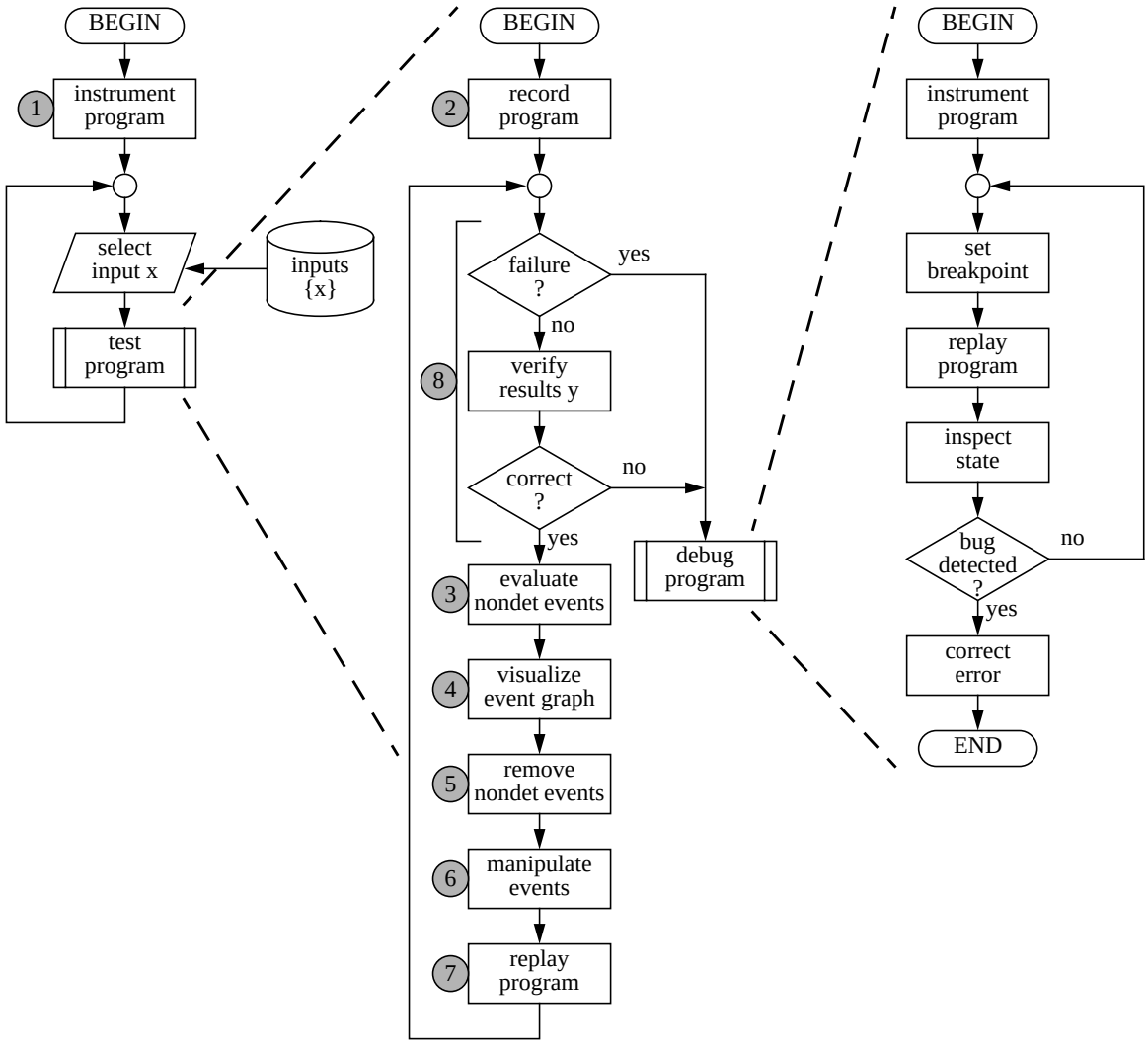


Figure 8-6. Testing and debugging in case of nondeterminism

8.4.2 Event Manipulation

The idea of the event manipulation feature is to modify the results observed during the initial program execution and enforce these modifications during re-executions by the record&replay mechanism. The target events of these modifications are the nondeterministic events discussed above, and the data items that can be manipulated are the characteristic event attributes as described in Section 8.3.3. Modifying other event attributes or even deterministic events would vigorously affect the program's behavior and deviate the replay phase from the original record phase, so that the program may not execute properly and may deliver useless data.

A simple example of how to apply event manipulation is provided by random number generation event. If this event occurs during the initial program execution, the record phase delivers the resulting value that has been observed. With the event manipulation feature these observed results can be modified in the tracefiles within the event inspection window as displayed in Figure 7-1 on page 179. Afterwards a corresponding re-execution of the program with the record&replay tool will reveal the program's behavior with the modified random

number value that has been provided by the debugging user. The modifications that can be applied must be within certain limits. For random number generation events only values within the range of this function can be applied.

The manipulation of communication events is comparable to random number generation event, and again affects only the data important for nondeterministic execution. For example, a nonblocking receive that returned `false` during the record phase, may be modified to return `true` during the replay phase. The limitation for this modification is, that a corresponding message must actually be available at the selected receive operation. The same is true for blocking receives with wild cards for message origin and message type, and in both cases the values that can be used during event manipulation identify racing messages, that have been derived during the nondeterministic analysis steps.

Within our tool, the application of event manipulation for nondeterministic receive events is applied as follows:

(1) Select a nondeterministic receive event from the set of possible race condition candidates.
(2) Select a send event corresponding to the racing message, that should be received at the selected race condition candidate.
(3) Save the applied event manipulation and initiate program re-execution.

The first step is to select a receive event that should be manipulated. For that reason, all race condition candidates are visualized, and the user may choose a corresponding exchange event from the graphical display. This exchange event identifies the point-of-exchange (POE), which is the location where event manipulation should be carried out during replay. After selecting the requested race condition candidate, the tool evaluates the corresponding racing messages and displays the responsible send events in the space-time diagram. Examples for both steps are included in Figure 8-4. After the manipulation has been performed, the user may save the modifications and initiate a corresponding program re-execution to derive the consequences of the event manipulation.

During these three steps, the user is effectively supported by the underlying debugging tool. Firstly, the set of possible race condition candidates has to be evaluated. Secondly, the racing messages corresponding to each nondeterministic receive must be determined. In addition the tool has to determine a so-called exchange cut, which is used to distinguish those parts of the event graph, that happened before or concurrently to the event manipulation, and those events, that may be affected by the manipulation during program re-execution. And finally, the debugging tool has to perform the artificially generated program execution in order to verify the program's behavior.

8.4.3 Cut Placement for Event Manipulation

The placement of a cut has already been described for breakpointing in Section 7.5. An exchange cut is in many respects similar to a breakpoint cut, but there exist minor differences. A breakpoint cut divides a program's execution into a part that must be executed to reach the breakpoint, and a part that may be affected by the debugging activities applied at the breakpoint event. An exchange cut divides a program's execution into a part that is not affected by the event manipulation, and a part that may be affected by the modifications applied at the point-of-exchange.

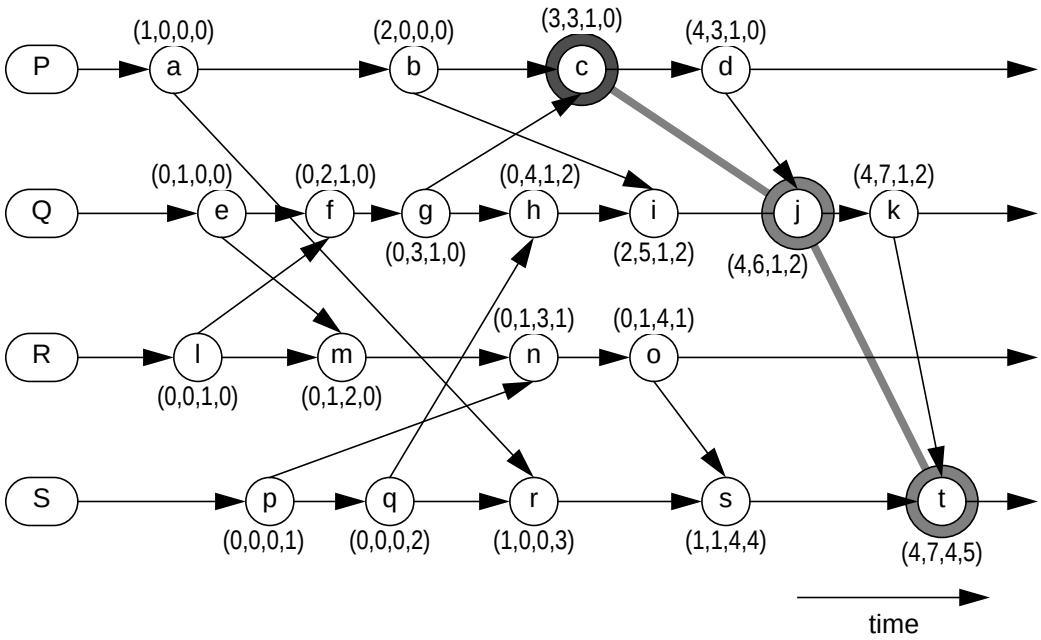


Figure 8-7. Placement of exchange cut for event manipulation

In order to apply an exchange cut to an event graph, we need to formally define its properties. An event graph is also a partial cut, and Definition 7-5 can be taken as specified on page 188. The difference between partial cuts for breakpointing and partial cuts for exchange are the selected breakpointing events. The events affected by the event manipulation are those that occur in its timezone FUTURE, which has been specified in Definition 5-6 on page 142. Then we can adapt Definition 7-7 on page 189, which was used to describe a partial cut for a selected breakpointing event, by changing the function to determine the timezone of the breakpointing events.

Definition 8-7: Partial cut for event manipulation

The partial cut for a selected exchange event $e_p^i \in E$ is the set C consisting of e_p^i on process p, and the events $e_q^j \in E$, so that

$$e_q^j = \text{first}\left(\text{FUTURE}\left(e_p^i\right), q\right) \text{ for all processes } q \neq p.$$

If the timezone FUTURE(e_p^i) used in Definition 8-7 does not contain any elements for an arbitrary process q, first(FUTURE(e_p^i,q) will deliver *undefined*, a result that identifies process termination. This means, that the corresponding process is not affected by the event manipulation and will be executed until its end under the control of the replay mechanism. For all other processes, the set C identifies those events, where the event manipulation affects the process' behavior.

An example for an exchange cut can be found in Figure 8-7, which resembles the event graph with the timezones from Figure 5-8 on page 143, and the breakpoint cut as described in Figure 7-7 on page 190. Again, event c is selected as an exchange event, and we assume that

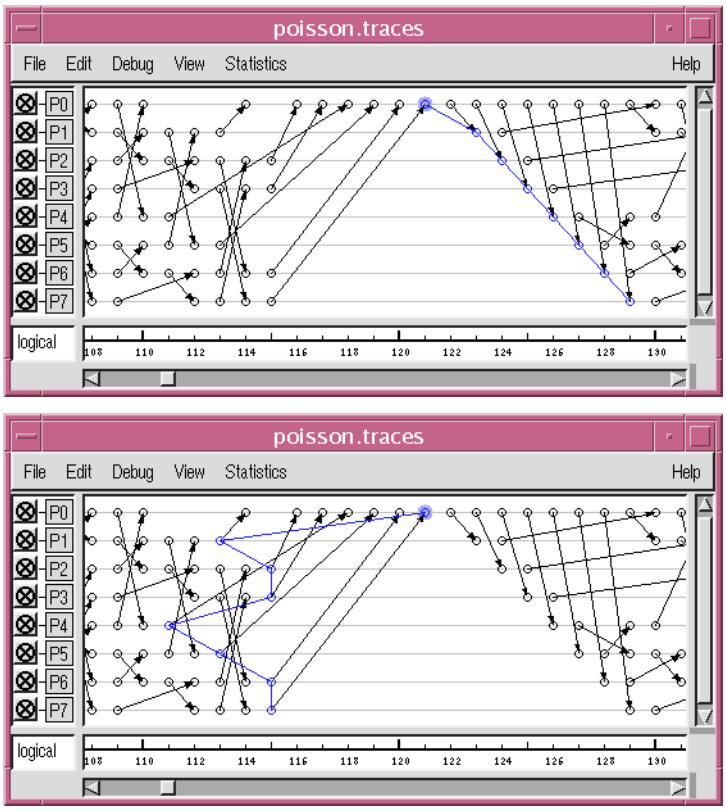


Figure 8-8. Cut for exchange (above) and cut for breakpointing (below)

there are no invisible racing messages (e.g. on process *R*). The resulting exchange cut *C* for the selected event *c* would be:

$$C = \{c,j,t\}$$

As can be seen in Figure 8-7, the events on processes *Q* and *S* are actually members of the timezone FUTURE(*c*), and each of the events *j* and *t* is the first event in this set on the corresponding process. There is no breakpoint event on process *R*, because no event in this process' future is visible that happens after event *c*. All events left of this exchange cut are not affected by the event manipulation, while all events right of the exchange cut including the events comprising the cut are possibly affected. The latter are those events that cannot be considered for the current exchange cut, because it is unpredictable if they would even occur after event manipulation has been applied at the point-of-exchange.

The example screenshots of Figure 8-8 also show the differences between the two kinds of cuts for an actual program execution. The program under consideration is again the parallel solver for Poisson's equation as mentioned in Section 7.5.4. An arbitrary event has been selected on process P0 for both types of cuts. The upper screenshot of Figure 8-8 shows the cut that would be obtained if event manipulation is performed, while the lower screenshot of Figure 8-8 shows the cut for breakpointing.

A corresponding algorithm for placing an exchange cut is given below. This algorithm closely resembles Algorithm 7-1 from Section 7.5.3, which was used to establish a consistent cut for breakpointing in an arbitrary parallel program. Again, the resulting cut is stored in the elements of vector *cut[]*, while the exchange event is stored in the object *breakpoint*. The basic method is again the method *vector()* of the class *event*, which provides the vector clock as pro-

duced by Fidge's algorithm (see Section 5.3.2). The difference between Algorithm 7-1 and Algorithm 8-1 is only the overloaded operation determining the results of the precedence condition (see Condition 5-1 on page 139). This operation is the "<" for the setting a cut for breakpointing, while it is a "<=" for establishing an exchange cut. The resulting differences have already been described in Figure 8-8. For the exchange cut in the upper screenshot, the breakpoints are set at the first event that happened after the selected exchange event on process *P0*. For the breakpoint cut, the breakpoints are placed after the last event that happened before the breakpointing event on process *P0*.

Algorithm 8-1: Constructing a partial cut for event manipulation

```
for (i=0; i < numProcesses; i++) {
        if (i = breakpoint->process_id())
                cut[i] = breakpoint;
        else {
                event = first(i);
                while(event & (event->vector() <= breakpoint->vector()))
                        event = event->next();
                cut[i] = event;
        }
}
```

8.4.4 Additional Notes on Exchange Cuts

Similar to the remarks on breakpoints, exchange cuts require some more discussion. Figure 8-9 shows an example where the placement of the exchange cut is not trivial. The two event graphs describe a program's execution on 6 processes (*P*-*U*), where events *a*, *b*, *e*, and *f* are send events, while events *c*, *d*, *g*, and *h* are nondeterministic receive events. The left event graph shows the program's execution as observed by the monitor. In this case, event *c* received the message from event *b*, event *d* received the message from event *a*, event *g* received the message from event *f*, and event *h* received the message from event *e*. However, since event *c* and event *g* are nondeterministic receives as defined above, it is also possible that event *c* receives the message from event *a*, or that event g receives the message from event *e*.

Analyzing this event graph with the debugging tool immediately identifies the nondeterministic receive events and their corresponding racing messages. The actual consequences of these nondeterministic events can be evaluated by the user by performing corresponding event manipulations. In the example described above, the event manipulation is performed at receive event *c*, so that event *c* receives the message from event *a* during the subsequent re-execution phase. This re-execution would deliver the event ordering on processes *P*, *Q*, and *R* as expected, which is shown in the right event graph of Figure 8-9. The problem is the placement of the cut on processes *S*, *T*, and *U*, and the example graph shows three possible cut positions.

A breakpoint cut for the selected event *c* would be established immediately after the process initialization and before event *e* on process *S*, event *g* on process *T*, and event *f* on process *U* occur, assuming that these events are the first events on their processes. These breakpoint events must be selected, because the user may modify the states of processes *S*, *T*, or *U*, so that they exhibit a completely different behavior.

Placing the exchange cut is more difficult. The problem is to decide, whether the breakpoint event for the exchange cut is set before the nondeterministic event g, or after event *h* and

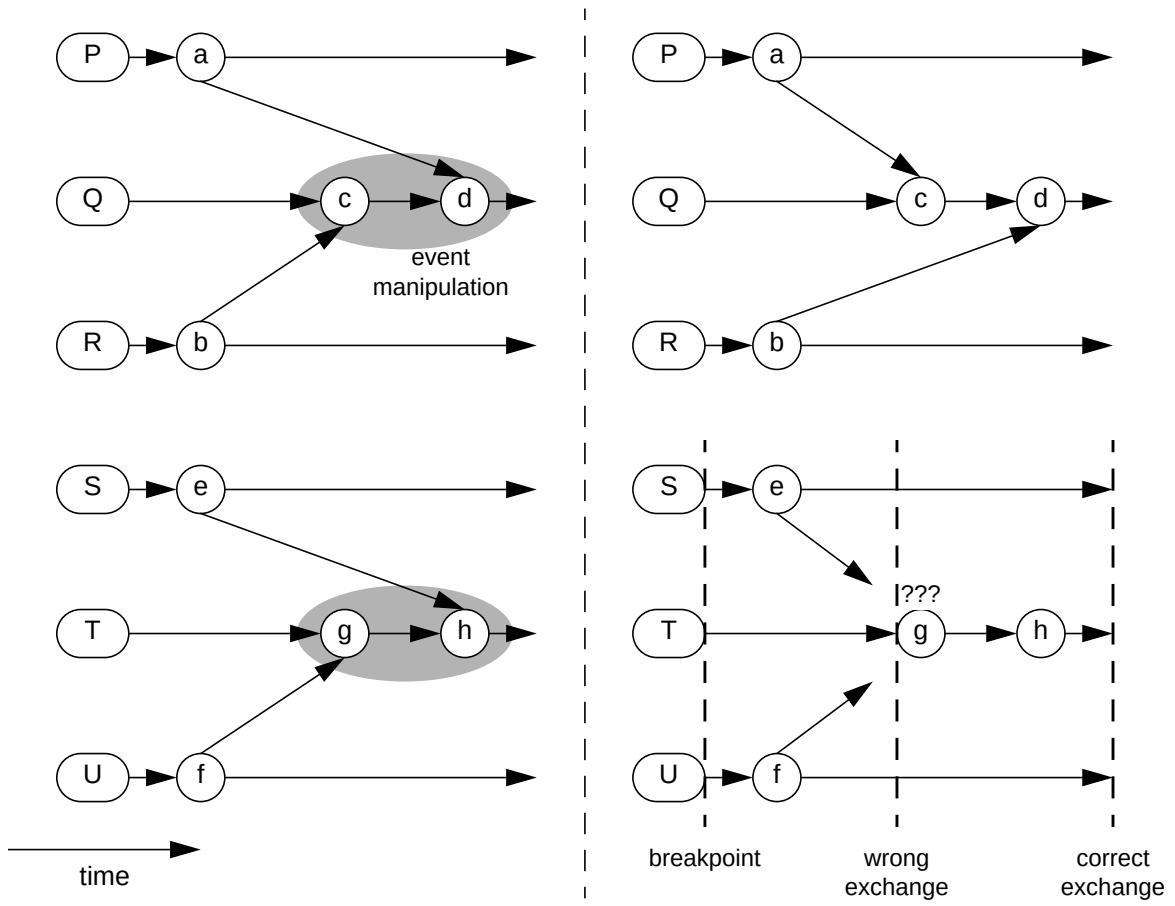


Figure 8-9. Placing cuts for event manipulation

before the next event on process *T*. An argument for placing the breakpoint before event *g* is, that the program's behavior of process *Q* after event *c* may actually establish a send event, that may direct a message to process *T*, that could be received at the nondeterministic receive event *g*. However, this is not the correct place, because it would permit the following "flip-flop" effect:

Assuming, that the event manipulation is performed at event *c*, so that event *c* receives the message from event *a* as described in the right event graph of Figure 8-9. If the breakpoint for the exchange cut is set on process *T* before event *g*, it is possible that event *g* receives the message from event *e* during program re-execution. After complete re-execution, the following two event graph's can be observed:

- The original event graph as shown in the left diagram of Figure 8-9, and also in the top-left diagram of Figure 8-10.
- The event graph after event manipulation as shown in the top-right diagram of Figure 8-10.

After this first event manipulation, the user may choose to analyze either the second nondeterministic event in the top-left event graph of Figure 8-10, or the two nondeterministic events in the top-right event graph Figure 8-10. With the cut placement before event *g* (and event *c*, respectively), it is possible that always one of these two event graphs could be obtained. In

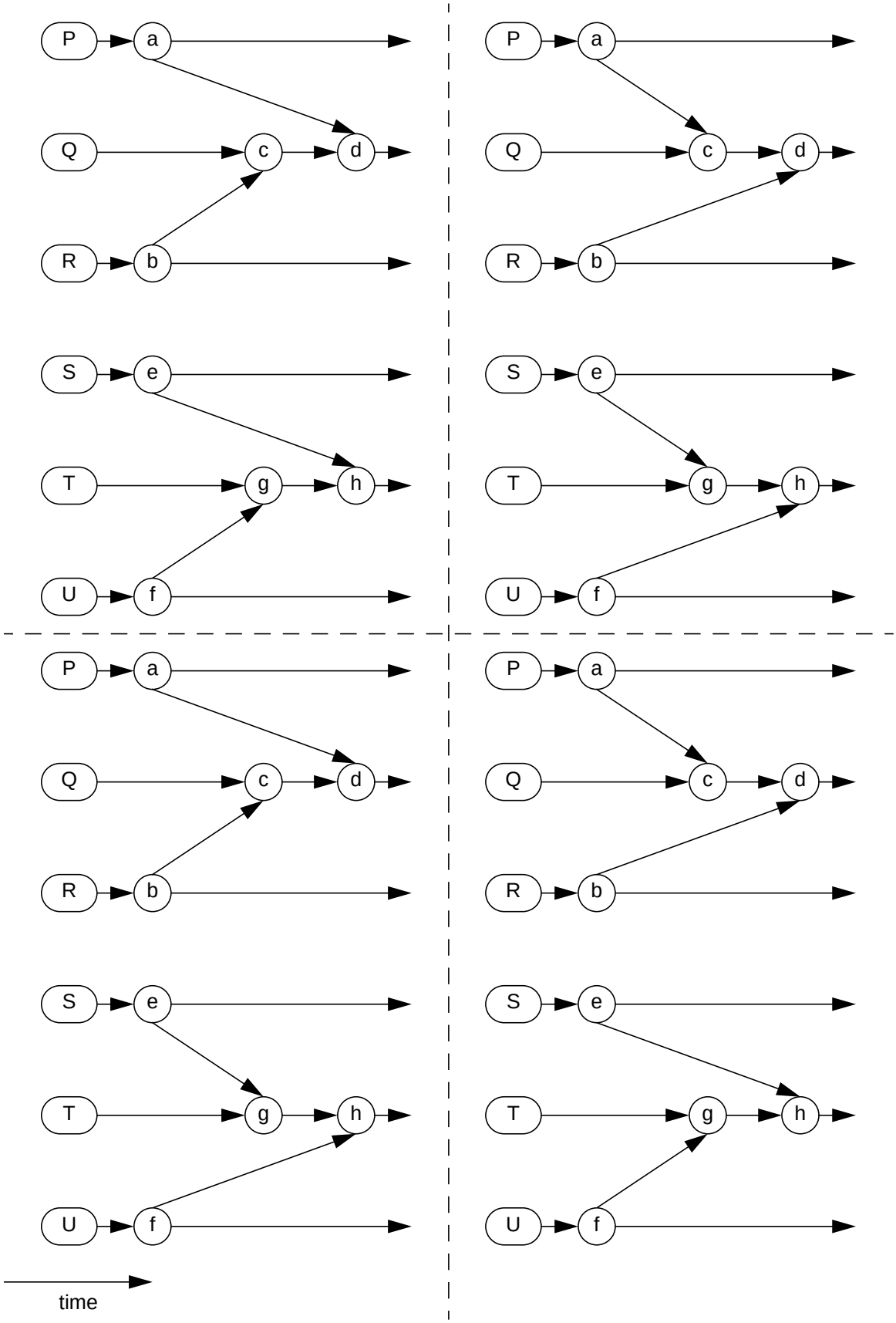


Figure 8-10. Possible program executions (resulting from Figure 8-9)

fact, after three event manipulations at the latest, the user could assume that all possible executions of the selected program have been observed, and quit performing program analysis[5]. However, in reality this approach overlooked 50 percent of the possible results, which are shown in the bottom diagrams of Figure 8-10.

A solution to this problem is to set the cut in this example after event *h*, so that event *g* and event *h* are performed as specified in the tracefiles. This guarantees, that the event manipulation at event *c* as applied in the top-right diagram of Figure 8-10 actually delivers the event graph as shown in the bottom-right diagram of Figure 8-10. By performing the remaining event manipulations on these two event diagrams, all four event graphs of Figure 8-10 will be revealed. This example proves, that it is feasible to set the cut after nondeterministic events (e.g. events *g* and *h*), although these events may be modified by the applied event manipulation. Yet, since event manipulation is performed iteratively, all possible cases will be addressed during some place of the program analysis.

8.4.5 Artificial Program Re-execution

The remaining issue that has to be discussed with respect to the event manipulation feature is artificial program re-execution. The event manipulation applied at the point-of-exchange enforces a program execution, that has possibly not been observed before. The exchange cut as defined above divides the program's execution into a pre-defined part that must be replayed as previously observed, and an undefined part that is determined by the event manipulation. This describes the activities, that have to be carried out, when the exchange cut is reached during program re-execution. With a given exchange cut, the artificial program execution is performed as follows:

- Before the exchange cut: re-execute the program under the control of the replay tool with the tracefiles as a constraint.
- After the exchange cut: execute the program without any constraints and generate tracefiles for subsequent analysis phases.

These two activities are carried out on each process. In addition, the process containing the point-of-exchange must apply the manipulation, when the execution arrives at the point-of-exchange. The important task at the exchange cut is to switch the functions calls from the replay functions used during the first part of the re-execution to the corresponding monitoring functions, which will be executed during the second part of the re-execution. The function switching is necessary because the program's execution after the exchange cut is undefined, which is based on the fact that the consequences of the event manipulation are unknown. This is also a difference between exchange cuts and breakpoint cuts, because the former is applied before the event comprising the cut is executed, while the latter must perform the event before halting the process' execution.

If the functions calls have been implemented with function pointers as described in Section 8.3.3, the switching can be performed by exchanging the replay functions with corresponding monitoring functions. The events after the selected exchange cut cannot be replayed, because it

5. This example is based on some actual experiences that have been encountered with a preliminary version of our debugging set, which established an incorrect exchange cut. This simple example indicates, how critical this problem is in reality.

is unpredictable if they would even occur, simply because their occurrence is affected by the applied event manipulation.

There are three problems related to artificial program re-execution. The first problem is based on the fact, that there might be innumerable race condition candidates or countless numbers of racing messages for each nondeterministic receive. In this case it may be impractical to apply nondeterminism analysis as offered by event manipulation and artificial replay.

In addition, the artificial re-execution may reveal new and previously unknown program executions, which are again subject to testing and debugging in general, and nondeterminism analysis in particular. Therefore, event manipulation has to be applied iteratively and in the worst case it is even imaginable, that a program reveals a new executions infinitely. For example, if a program waits for a special sequence of events for program termination, and this sequence is always disturbed by the event manipulation.

The last problem of artificial program re-execution is the fact, that the execution after the exchange cut may require additional input from the user, which may not have been required during the initial execution. Thus, the artificial re-execution requires a consistent presence of the debugging user to provide input data when requested, which may also be a major drawback of this approach.

8.5 Summary

This chapter describes, how to perform nondeterminism analysis with the event graph model of a parallel program. For that reason, we have identified the events responsible for nondeterministic behavior, and how they can be detected. A major part of this discussion focuses on nondeterministic receives, which permit more than one message to be received during subsequent program executions. In this context, we have also distinguished the standard case of message transmission and message passing systems permitting overtaking message transfer. The latter is very relevant in practice as existing implementations have shown and offers some benefits for simplifying the analysis task.

After the nondeterministic events have been detected, they can be highlighted in the space-time diagram to provide a first basis for more sophisticated analysis activities. These activities are supported by a record&replay mechanism, which guarantees equivalent re-execution of nondeterministic parallel programs.

The second part of this chapter deals with the problem of predictability, which occurs at nondeterministic events that cannot be classified as intended or unintended. For that reason we have adapted the event manipulation approach of Grabner for the event graph model, and we show how artificial program re-executions can be achieved. This approach requires a special version of a partial cut, which is used to distinguish those parts of the event graph, that are affected by the event manipulation from those, that are not.

9. Automatization of Nondeterminism Analysis

Nondeterminism analysis as introduced in the previous chapter requires user interaction for event manipulation, which may represent a massive problem for programs with large numbers of nondeterministic events or racing messages, respectively. This chapter discusses possible solutions for this problem and how to perform nondeterminism analysis automatically. Besides the basic strategy for automatic event manipulation and artificial re-execution, this approach also requires the integration of keys to distinguish executions, which we called finger-prints. As a side-effect of these finger-prints we obtained a substantial improvement for the basic record&replay mechanism. Although the solutions described in this chapter represents a useful extension of the initial event manipulation approach, it may still lead to huge numbers of executions and large amounts of trace data, especially for large programs. Since these extensive requirements cannot be provided under regular conditions, we briefly describe some ideas that have been investigated in this context.

- Idea: automatic evaluation of nondeterministic behavior.
- Practical application for testing nondeterministic programs.
- Required fingerprints and minimal tracefiles.
- Remaining problem: number of executions and amount of trace data.
- Solutions for the completeness problem.

9. Automatization of Nondeterminism Analysis

9.1 Problems of Manual Nondeterminism Analysis

9.1.1 Complete Nondeterminism Testing

Nondeterminism analysis as described in Chapter 8 is a unique technique to cope with nondeterministic behavior of parallel programs. Our approach, which has been specifically adapted to the characteristics of the event graph model, represents a revised and improved version of the original event manipulation approach of Grabner and Volkert ([GrVo 93] and [GrVo 96]). Each of these approaches offers a practical solution to the irreproducibility effect as described in Section 4.2.6, and allows to address the completeness problem as described in Section 4.2.7.

One difference between the irreproducibility effect and the completeness problem is that the former describes the problem of re-executing a nondeterministic program for debugging, while the latter addresses the problem of complete testing of nondeterministic source code. In this section, we are mainly interested in the completeness problem, and how we can perform the testing of nondeterministic parallel programs automatically. Therefore, our focus is on the nondeterminism testing cycle as sketched in Figure 4-4 on page 103, which allows to evaluate the program's behavior by computing all possible results for a given set of inputs. This is achieved with event manipulation and artificial re-execution, which allow to enforce different results at nondeterministic events during re-execution as describe in Section 8.4.2.

To perform complete testing as described in Section 3.2.2, each result that may possibly be observed at a nondeterministic event during an execution has to be enforced during one iteration of the nondeterminism testing cycle. This means, that one corresponding re-execution has to be enforced for each nondeterministic event $e_p^i \in E$, so that

... e_p^i delivers each possible random number within the given range, if e_p^i is a random number generator event,

... e_p^i delivers `result=FALSE`, if e_p^i is a nonblocking receive which delivered `result=TRUE` during the initial execution,

... e_p^i delivers `result=TRUE` for each message that may actually arrive at e_p^i, if e_p^i is a nonblocking receive which delivered `result=FALSE` during the initial execution,

... e_p^i delivers each racing message at e_p^i, if e_p^i is a blocking receive.

The number of possible executions that can be derived from an initial execution depends on the number of nondeterministic events and the possible results these nondeterministic events may return during execution. Since each derived execution requires a dedicated event manipulation carried out by the user, the practical application of this manual approach is limited to small scale problems with few nondeterministic events and a corresponding small number of return values.

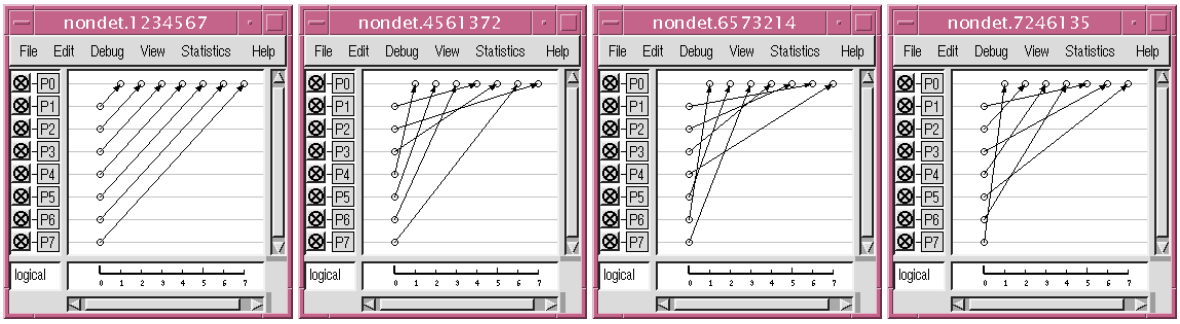


Figure 9-1. Four executions of a nondeterministic all-to-one communication program

9.1.2 Example Problem 1: All-To-One Communication

A very simplified example that may be difficult to test with manual event manipulation and artificial re-execution is the situation, where all processes are sending data to one process. The program under consideration represents a code fragment, that can be found in many parallel applications, e.g. when first-in-first-out queues are needed or for program termination. This divides the processes in two groups, a master process and several slave processes. While the slaves' only task is to send the data to the master, the master must receive all messages and process the data accordingly. Depending on the program's goal, the order of arrival may determine the program's results or influence the subsequent program flow.

Example 9-1: All-to-one communication

> Consider a MPI program that is executed on n processes. Assume, that process 0 is the master process, while all other processes are slaves. The goal of the master process is to collect data from all the slaves. The messages will be accepted in their order of arrival, which can be implemented by specifying a wild card receive statement with `MPI_ANY_SOURCE` on the master process. As a simplification, the slaves are delayed randomly before sending their own process identification. The master prints the accepted message data. The program's result is the order of incoming messages.

An actual implementation of Example 9-1 can be found in Appendix C. Executing this program several times in succession will probably deliver different results. Four different executions respectively their event graphs are shown in Figure 9-1. Please note, that each of these executions may serve as the initial execution for the subsequent testing cycles.

Complete testing of this simple example code means to establish each possible message order during re-execution. The message order for each re-execution can be determined by event manipulation. The question to be solve is:

How many possible executions do exist?

The set of executions can be derived from an initial execution by event manipulation is called the descendant executions. The number of descendents immediately reflects the number of needed event manipulations. Only if the amount of descendents is within manageable limits, manual event manipulation and artificial re-execution as described in Section 8.4.2 is meaningful. The number of executions for Example 9-1 can be derived as follows.

(1) Identify the number of nondeterministic events.
(2) Identify the number of possibilities for each nondeterministic event.

Assuming that the number of processes for Example 9-1 is *n*, then the number of nondeterministic events is *n-1*, and each of these events is a blocking wild card receive. Consequently, the possibilities for each of these nondeterministic events is the corresponding set of racing messages, which can be given as follows:

1st receive: *n*-1 racing messages
2nd receive: *n*-2 racing messages
...
$(n\text{-}2)^{th}$ receive: 2 racing messages

Please note, that the $(n\text{-}1)^{th}$ receive is no race condition candidate according to Definition 8-3 on page 206, since it can always only accept one message. This is the only message that has not been received before.

For each of the race condition candidates, we require one re-execution for each racing messages. Thus, the number of executions that can be derived from the initial executions can be computed as follows:

$$\text{no. of executions} = (n-1) + (n-2) + \ldots + 2 = \sum_{i=1}^{n-2} n - i$$

For the example given in Figure 9-1 with *n*=8 processes, this means that 27 executions do exist and 26 descendant executions can be derived from each of these initial executions. However, since each derived descendant represents a valid execution for the given program, each descendant is again subject for nondeterminism analysis. This means, that we have to distinguish two kinds of descendants, direct descendants that are derived from one execution, and indirect descendants, that are derived from descendants. Consequently, the total number of possible executions is the sum of the initial execution, its direct descendants, and all possible indirect descendants.

For Example 9-1 the total number of possible execution is thus computed based on the number of processes *n* as follows:

$$\text{no. of executions} = (n-1)!$$

This means, that the example executions given in Figure 9-1 with *n*=8 processes deliver 5040 different executions, which requires 5039 event manipulations in order to perform complete testing during the nondeterminism testing cycle. The number of executions for other configurations are given in Table 9-1:

Table 9-1: Number of executions corresponding to number of processes

number of processes	number of executions
4	6
5	24
6	120

Table 9-1: Number of executions corresponding to number of processes

number of processes	number of executions
8	5.040
10	453.600
12	59.875.200

The results in Table 9-1 indicate, that complete testing of Example 9-1 may be possible for up to 5 processes, but gets virtually impossible for more than 6 processes. An argument against this worst-case example is, that implementing such a communication pattern in a parallel program is usually intended by the user, and the user may not be interested in performing these tests at all. However, this does not remove the potential havoc that may be created by this or a similar program.

In addition, this example only contains *n*-1 receives on one process. Consider all possible executions of a program, if there are other receives on other processes. In this case, the number of combinations may raise even faster. A corresponding example for this situation will be given in the next subsection.

9.1.3 Example problem 2: Data Circulation

The example described in this subsection is slightly more complicated than the previous example, but it shows some additional characteristics of nondeterminism testing. The code under consideration is data circulation on hypercube architectures as it is applied in various numerical algorithms [Deke 81]. Its goal is to circulate data items distributed on the processes so that these data visit each of these processes exactly once.

Example 9-2: Data circulation

> Consider a MPI program that is executed on n processes of a hypercube multiprocessor [HwXu 98], where $n=2^d$ with d the dimension of the hypercube, so that each process is executed on one node of this architecture. Each process holds an arbitrary data item, which is used for some computation. After the computation, the data items are exchanged with the other processes, and the received data items are used for another computation. The goal of the data circulation algorithm is to exchange the data, so that each process receives each data once and communication between the process is as efficient as possible.

An implementation of the code for Example 9-2 is included in Appendix D. This program operates in three phases:

(1) Compute operation with available data.
(2) Send data to a specific process.
(3) Receive data from a specific process.

For nondeterminism analysis, the important point in this code is the function for selecting the communication partners. Since this algorithm is targeted at hypercube architectures, each transmission should always take place between directly connected neighbors along one hypercube dimension. For that reason, Dekel et al defined the exchange sequence X_p as follows:

Definition 9-1: Exchange sequence for hypercube data circulation [Deke 81]

The exchange sequence X_d for data circulation in a hypercube multiprocessor with $n = 2^d$ processes is defined as

$$X_1 = 0 \qquad d = 1$$

$$X_d = X_{d-1}, d-1, X_{d-1} \qquad \forall d > 1 .$$

The results of an exchange sequence describe the dimension of the hypercube, which should be used for the communication during a corresponding iteration [RaSa 88]. This sequence essentially treats a *d* dimensional hypercube as two *d*-1 dimensional hypercubes. Data circulation is then performed in three steps:

- Circulate data in each *d*-1 hypercube with the exchange sequence X_{d-1}.
- Swap the data along dimension *d*-1.
- Circulate data in each *d*-1 hypercube with the exchange sequence X_{d-1}.

Based on this sequence, we obtain the destination address to be used in step (2) of our algorithm. However, while it is clear that sends have to know exactly where to transmit the message, users may assume that this is not necessary for the receive operations. It seems, as if a wild card receive may be appropriate for step (3), which is also underlined by the cyclic behavior of the algorithm.

In fact, users choosing a wild card receive for distance doubling may actually observe correct program behavior as shown in the top-most screenshot of Figure 9-2. In this case, the communication patterns performs data circulation as specified above. However, with wild card receives the only reason for this correct behavior is the computation time needed for step (1). If the time spent for computing the data is long enough, a correct pattern as given in the top-most diagram of Figure 9-2 may be observed all the time. Yet, if the computation time is to short, some processes may already process step (2) of a subsequent iteration, while others are still performing step (3) of a previous iteration. The results are incorrect communication patterns as observed in the other two event graphs of Figure 9-2, where some of the data items are not delivered to all processes[1]. Consequently, using a wild card receive for data circulation represents an error due to “programmer’s laziness” as described in Section 4.2.4.

Performing the nondeterminism testing cycle on this program is trivial. Only one event manipulation has to be applied to the correct event graph as shown in the top-most diagram of Figure 9-2 to reveal another execution that returns incorrect results. However, for demonstration purposes we will try to determine the number of executions for this example, which is not as trivial as for Example 9-1. Again, we have to determine the number of nondeterministic events and the number of possibilities for each nondeterministic event.

1. This data circulation example is another case that has actually been experienced during a course on parallel software engineering. The students’ task was to implement data circulation with MPI and some of them applied `MPI_ANY_SOURCE` at the receive operation. However, the students observed only correct program behavior, since the applied multiprocessor machine was “slow enough” during the computation step. Unfortunately, the submitted work was tested on a much faster machine, which decreased the cycle time and lead to incorrect program behavior during each execution.

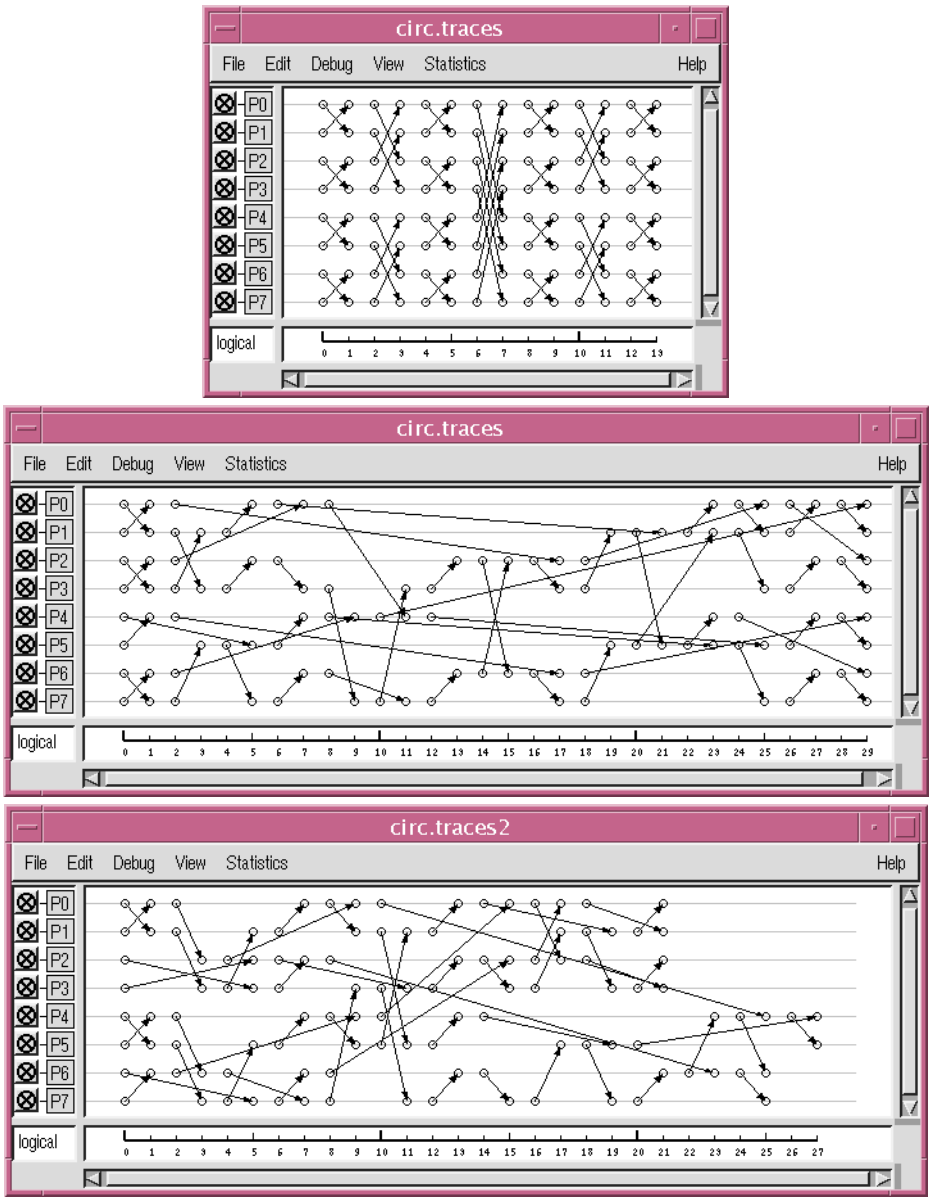


Figure 9-2. Three executions of a nondeterministic data circulation program

The number of nondeterministic receives is $n-1$ per process, which accumulates to $n(n-1) = n^2 - n$ for all processes. For the number of racing messages, we need to distinguish between racing messages in the standard case (see Definition 8-2 on page 205) and restricted racing messages in case of non-overtaking message transmission (see Definition 8-4 on page 208). In the standard case, it would be possible that two messages from one process to another arrive in reverse order due to overtaking. Thus, we can determine the following numbers of racing messages for the receives on one arbitrary process *p*:

1st receive on process *p*: *n*-1 racing messages
2nd receive on process *p*: *n*-2 racing messages
...
(n-2)th receive on process *p*: 2 racing messages
last receive on process *p*: no race condition candidate according to Definition 8-3

Since the set of racing messages in the standard case is equal to the set of racing messages as determined for Example 9-1, we conclude that the number of executions that can be derived with event manipulations on process *p* is $(n-1)!$. Yet, in contrast to Example 9-1 we must now derive each possible permutation for all processes, which leads to a total number of executions as follows:

$$\text{no. of executions} = n^{(n-1)!}$$

This means, that an example execution on 8 processes as given in Figure 9-2 would probably reveal $8^{(8-1)!} = 8^{5040}$, which is clearly out of scope of any testing mechanism. However, there are three remarks needed in this case:

(1) Many of these executions are only possible due to message overtaking. If message transmission is non-overtaking as characterized in Definition 8-4 on page 208, the number of possible executions is highly reduced (see below).
(2) The given number of executions represents only the upper limit. Not every execution is possible, e.g. at least one process must receive the first message from another process to start sending the second message.
(3) It is certainly questionable, how likely it is that the last message on an arbitrary process arrives before the first one of the same process. This remark will be discussed in more detail in Section 9.3.4.

Relating to Remark (1) we will now evaluate the number of racing messages for a message passing system that prohibits message overtaking, which is much more relevant in practice, especially for MPI programs. In this case, the number of racing messages per receive on an arbitrary process *p* are:

1st receive on process *p*: $\log n$ racing messages
2nd receive on process *p*: $\log n$ racing messages
...
$(n-2)^{th}$ receive on process *p*: 2 racing messages
last receive on process *p*: no race condition candidate according to Definition 8-3

This means that the number of racing messages per wild card receive is limited with $\log n$, which leads to the following number of executions for the non-overtaking case:

$$\text{no. of executions} = \left(n^2 - n\right)^{\log n}$$

Similar to Remark (2) given above, this number is a better upper limit for the number of executions, because some executions are again not possible. Yet, for the example execution of Figure 9-1 a maximum of 175.616 executions may be observed, which is certainly impossible with manual event manipulation but is also impractical with an automated approach.

9.1.4 Example Problem 3: Distance Doubling

Another problem case to demonstrate nondeterminism analysis is the distance doubling algorithm as introduced in Section 8.2.6. We have chosen this example to describe how complete testing of such a program can be performed, since it can easily be down-scaled in terms of process numbers and number of receive events. In concrete, we will show how to perform event manipulation and how to determine the total number of executions necessary for 2 iterations of distance doubling on 3 processes. In addition, we will present results as derived during experiments with $3 \leq p \leq 10$ processes.

In order to perform complete testing of distance doubling, we need to obtain an initial execution of the program. For that reason, we instrument the code and initiate a program run to generate tracefiles. Example executions for 3 processes with 2 iterations of the distance doubling algorithm are given in Figure 9-3. Although any of these executions may be chosen as the initial execution for nondeterminism testing, we selected the left-most execution.

Based on this initial execution, we have to determine the number of nondeterministic events and the number of possibilities that may be observed at each of these events. The nondeterministic events are again the blocking receives which specify a wild card for the messages origin. As can be seen in Figure 9-3, each process contains 2 receives, which totals 6 receives on 3 processes. However, only the first receive is a nondeterministic event, since the second receive may only accept the one message, that has not been accepted at the first receive. As a consequence, we may perform exactly 3 event manipulations on our initial execution, which deliver the 3 executions in the middle column of Figure 9-3. These three executions are direct descendents of the initial executions, because only one event order has been exchanged.

In the next step, the direct descendants are again subject to nondeterminism analysis. Since each iteration offers the same characteristics as the initial execution, we may again perform 3 event manipulations on each execution in the middle column of Figure 9-3. If we perform the 3 event manipulations on the top-most execution of the middle column, we obtain the two iterations in the right-most column as indicated by the arrows, and the initial execution in the left-most column. Similarly, each of the other two executions in the middle column of Figure 9-3 delivers two descendants in the right-most column and the initial execution.

The executions in the right-most column represent direct descendants of the execution in the middle column, and indirect descendants of the initial execution. Applying the nondeterminism cycle to any of these direct or indirect descendants will only deliver executions that have already been observed. Thus, we conclude that we have completely tested the given program. This means that 6 executions can be derived from 1 initial execution of 2 iterations of distance doubling on 3 processes.

The dependencies between an initial execution and its descendants can also be described graphically with a tree of executions, which can be defined as follows:

Definition 9-2: Tree of executions

> A tree of executions describes the relations between an initial execution of a nondeterministic program and all its direct and indirect descendants that can be derived with event manipulation and artificial replay. The root node of this tree represents the initial execution, while all the other nodes represent the corresponding descendants.

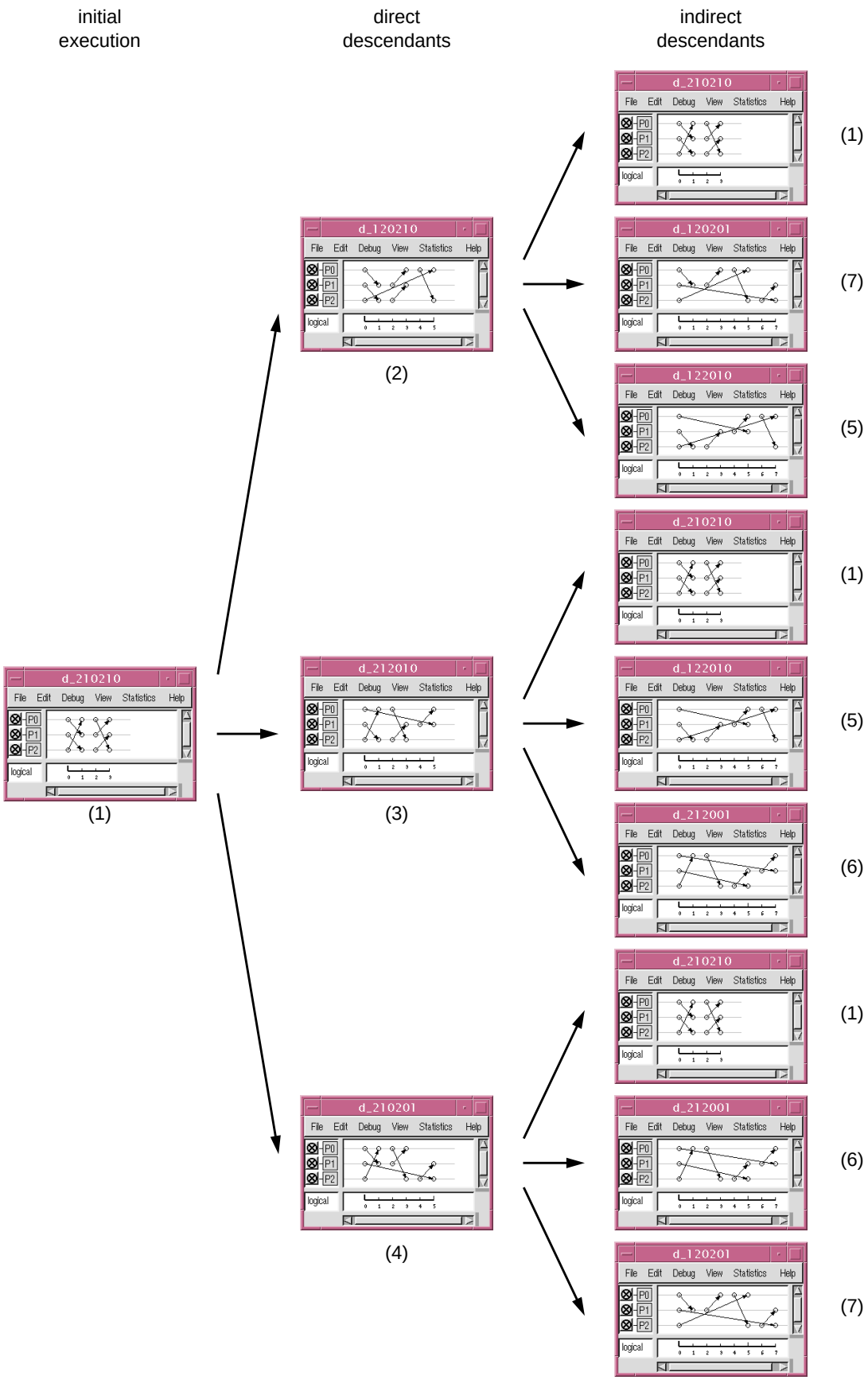


Figure 9-3. Executions of a nondeterministic distance doubling program

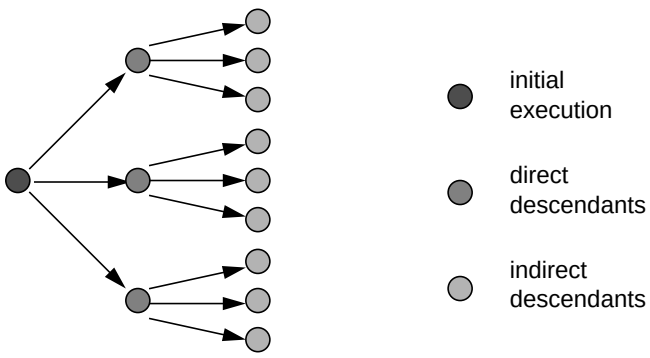


Figure 9-4. Tree of executions

An example tree of executions for the distance doubling code on 3 processes with 2 iteration is shown in Figure 9-4. It can easily be mapped onto the executions shown in Figure 9-3. An interesting characteristic of this example is that the tree of executions is similar for each possible initial execution, which needs not be the case in practice.

Table 9-2: Experimental results for 2 iterations of distance doubling

processes	executions	memory consumption
3	7	6.321 bytes
4	9	10.836 bytes
5	31	46.655 bytes
6	49	88.494 bytes
7	127	267.589 bytes
8	225	541.800 bytes
9	513	1.384.299 bytes
10	961	2.892.610 bytes

Please note, that we can always only perform one event manipulation per iteration of the nondeterminism testing cycle. Two or more event manipulations applied to one initial execution may lead to a program run that cannot be performed during artificial program execution. For example, we could manipulate each of the first blocking receives in the initial execution of Figure 9-3 to accept only a message from the second send of the corresponding processes, since each of them is a racing message. Yet, the program would be blocked forever, because no process can proceed to the second send before receiving one message. As a consequence, we cannot derive more than 12 executions from the initial execution. The 13th execution would be the one, where all three messages are exchanged.

Some more numbers for this example are included in Table 9-3, where column one contains the number of processes, while column two contains the total number of executions for these processes. Besides that, we determined the memory consumption for the traces. The results indicate, that even this small example cannot be tested by manual event manipulation and arti-

ficial replay. Consequently, we need to offer some means of automatization, which are described in the next section.

9.2 Automatic Nondeterminism Analysis Strategy

9.2.1 Principal Approach

The example shown above indicates, that manual approach to event manipulation and artificial replay as described in Section 8.4 is often not applicable in practice due to the required user interaction. As a consequence it seems desirable to provide some means of automatization for performing nondeterminism analysis. Such an automatic debugging feature must then comply with the following characteristics:

For a given initial execution, it must

(1) detect all nondeterministic events.
(2) determine the set of possibilities for each nondeterministic event.
(3) derive the corresponding descendants by performing event manipulation and artificial replay for each possibility of each nondeterministic event.
(4) repeat step (1)-(3) for each descendant.

These requirements lead to our automatic nondeterminism analysis strategy, which is applied as shown in Figure 9-5. The shown diagram replaces the central nondeterminism testing cycle of Figure 4-4 on page 103. It starts with a record phase of the target program for the selected set of inputs to obtain the initial execution. This execution is stored as one of the possible program run's.

Based on the initial execution, automatic nondeterminism analysis performs steps (1)-(4) as described above with three loops. One of the two inner loops is needed for each nondeterministic event in the given execution, and a second one is needed for each possibility corresponding to a selected nondeterministic event. For each combination of these loops, the nondeterminism analysis performs event manipulation and a corresponding artificial replay phase, which generates a corresponding program execution. After retrieving all combinations of nondeterministic events and their possibilities, automatic nondeterminism analysis performs the outer-most loop by injecting every descendant to the analysis cycles.

A remark that has already been discussed in Section 8.4.5 concerns the selected input. The input applied to the initial execution may change for the artificial replay phases, if it is affected by decisions at nondeterministic events. Since it could be practically impossible to request user interactions whenever different input is needed, we assume, that the input data set does not change during the automatic nondeterminism analysis.

9.2.2 Equivalence of Executions

A problem of automatic nondeterminism analysis that needs some more detailed discussion is the equivalence of executions derived from different ancestors. As shown in Figure 9-5, each replay phase delivers a new program execution. Based on Definition 8-1 on page 196 about "Nondeterministic parallel programs" and Definition 8-6 on page 213 about "Equivalent exe-

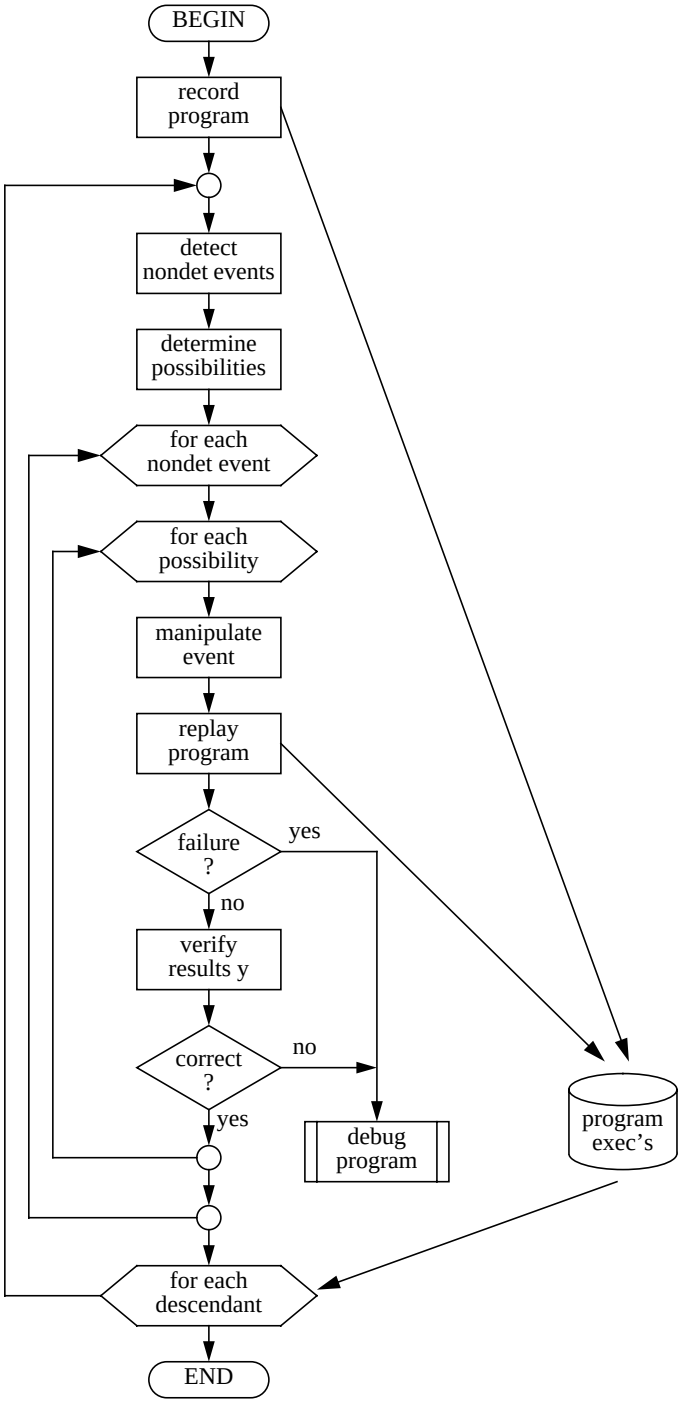


Figure 9-5. Testing cycle for automatic nondeterminism analysis

cution (based on the event graph model)" it is clear, that each artificial replay derived from one arbitrary program execution by event manipulation will lead to a different program run. However, this needs not be the case for artificial replays that are based on different executions (e.g. the initial execution and one of its descendants). For instance, the example of Section 9.1.4 will deliver equal descendants for different initial executions. Since it is not useful to analyze an execution that has already been investigated, it is also useless to keep equivalent executions.

A tool, that cannot identify equal executions may not only analyze the same execution several times, it may even deliver new executions infinitely as described in Section 8.4.4.

As a consequence, an automated approach to nondeterminism analysis requires the introduction of a unique identifier for the generated executions. The problem in this context is to characterize, what identifies an execution and what makes two executions different. These two questions can be answered based on Definition 8-6 on page 213, which described the equivalent execution as needed for re-execution of nondeterministic programs. In this definition, two executions are defined to be equivalent, if their event graphs are isomorphic. This means, that both event graphs are equivalent concerning their vertices and the established happened before relation.

A straight-forward idea would be to compare the tracefiles of two executions, since they contain all the data needed for constructing the event graph. However, this is not useful, because tracefiles contain much more data than just the vertices, and even the data of the vertices, the so-called event attributes, may differ for two equivalent executions. For example, consider the event attributes of a send event as given in Table 6-2 on page 163. Clearly, the available timing information (start time and stop time) is different for two executions of a program, even if the two executions are equivalent or the program is deterministic.

For that reason, it is important to identify those data items that may differ even if two executions are considered equivalent according to Definition 8-6. The remaining data are those that must be identical for two executions to be equivalent. For the send event of Table 6-2 these are all event attributes besides start and stop time. These data have to be compared after the replay phase of Figure 9-5, and only if no previous executions is equivalent to the current descendant, it will be added to the set of descendants.

The problem of this approach is, that the comparison of executions represents another major processing step, considering that each execution must be compared with every previously obtained execution. In order to reduce the comparison step, we introduce a key, that

- is a unique identifier for every execution of a nondeterministic program,
- is as small as possible in terms of memory consumption,
- can efficiently be constructed.

Two equivalent execution would be identified with the same key and duplicate entries could be removed from the testing cycle. Since the comparison of executions depends on the number of available executions, it is a basic necessity to provide an efficient key that can be constructed either post-mortem or during monitoring.

9.2.3 Finger-printing of Executions

Defining a key to distinguish executions is a non-trivial activity and clearly depends on the system under investigation. Such a key must be based on the nondeterministic events and their associated results, so that it describes the execution of a nondeterministic program based on the decisions taken at nondeterministic events. This key is called a finger-print and can be defined as follows:

Definition 9-3: Finger-print of a nondeterministic execution

> The finger-print FP(x) of a nondeterministic parallel program execution with a given set of input data x is the set

$$\mathrm{FP}\,(x) \;=\; \{\,\mathrm{fp}_p \mid 0 \le p < n\}$$

where n is the number of processes, and

$$\mathrm{fp}_p \;=\; \{\,f\left(e_p^i\right)\}$$

is the finger-print on process p for all nondeterministic events $e_p^i \in E$, with $f(e_p^i)$ being the result that occurred at each of these nondeterministic events during execution.

The finger-print fp on each process is the sequentially ordered set of results of each occurring nondeterministic event, while finger-print entries on distinct processes are unordered. Based on the set FP(x) we are able to construct the partial ordering of all nondeterministic events, which is unique for a particular nondeterministic program execution with a given set of input data.

The results of each nondeterministic event depends on the type of event. If e_p^i is a random number event, then the result $f(e_p^i)$ contains the value of the random number returned by the random number generator. If e_p^i is a nondeterministic nonblocking receive event, then $f(e_p^i)$ indicates whether a corresponding message was available or not. If such a message was available, or e_p^i is a nondeterministic blocking receive event, then $f(e_p^i)$ identifies the winning racing message that was received during execution (see Definition 8-2).

The message identification at nondeterministic receive events depends on the underlying system. On systems providing message overtaking (see Section 8.2.2), the winning racing message is identified by its sender process and the sequential order of its send event, which is also captured by the logical or vector clock piggybacked to the message on transfer. On systems, where message overtaking is not possible, the winning racing message is identified by its sender process only, since only the next message from this process can be received. This idea, which we have originally published in [KrVo 98] and later improved in [KrVo 99], has also been integrated for the MPL* record&replay system described in [Chas 99].

Finger-prints can also be used to transform the trees of executions into a graph of executions, which is defined as follows:

Definition 9-4: Graph of executions

A graph of executions describes the relations between all the possible executions of a nondeterministic parallel program for a given set of input data. Each node in such a graph represents a particular execution as identified by a unique finger-print. If two nodes are connected by an edge, the corresponding executions can be mutually derived by event manipulation and artificial replay.

Constructing the graph of executions for the distance doubling program of Section 9.1.4 shown in Figure 9-3 delivers the set of executions shown in Figure 9-6. In Figure 9-3, each execution is displayed 3 times, which is also indicated by the numbers in parenthesis. In Figure 9-6, these equivalent executions are identified and only one execution remains in the graph. This means, that the distance doubling program delivers only 7 distinct executions.

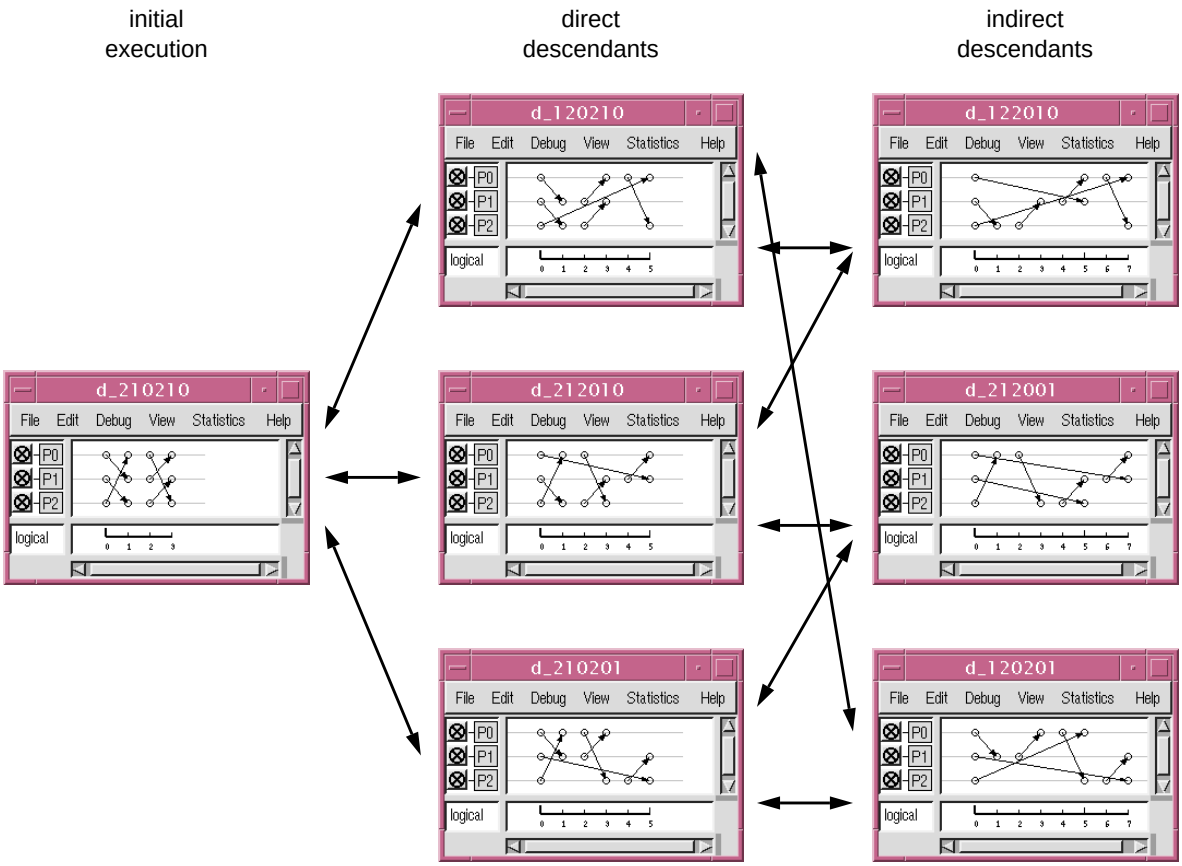


Figure 9-6. Complete set of executions of distance doubling program

A very simplified finger-print mechanism has been integrated in our debugging tools. The finger-prints are stored as ASCII-files, with one file corresponding to each process. This takes the unordered events on different processes into account and allows usage of parallel I/O facilities for finger-print generation. In addition, the sequential order in each file represents the order of occurrence of the nondeterministic events. Therefore, the files only contain the result $f(e_p^i)$ of each nondeterministic event, one after the other as they occurred during execution. An example finger-print of the distance doubling program discussed in Section 9.1.4 is given in Table 9-3:

Table 9-3: Example finger-print file

	Record 0	Record 1
Process P0	2	1
Process P1	0	2
Process P2	1	0

This finger-print file resembles the left-most event graph of Figure 9-6 and shows the results of the nondeterministic events, in this case only blocking receives. Since the underlying message passing system is the standard MPI, which is per definition non-overtaking (see Section 8.2.5), each record of this file identifies the origin process of the accepted message, and in practice

each record is stored as one line. If the system permits message overtaking, each record would need an entry for the logical or vector clock as described above.

Another important characteristic of the finger-print mechanism is, that deterministic events do not generate any data. Thus, processes delivering empty finger-print files are deterministic, and if no finger-print files are generated, a program is guaranteed to be deterministic for the given set of input data.

Generating the finger-prints for a nondeterministic execution can be performed in two ways. On the one hand, it is very easy to extract the finger-print information from the tracefiles, because the data needed for the finger-print is only a subset of the data provided in the tracefiles. A side-effect of this relation between the finger-prints and tracefiles is, that finger-prints can also be used for compression. The finger-prints represent the compressed data, while the traces represent the uncompressed data. Uncompressing the finger-prints is achieved by replaying the program and generating complete traces during an equivalent execution. Although the observed timings during the uncompressing execution may differ from the original execution, the resulting traces will still describe the same program behavior. This is a very useful feature if filespace is limited and many executions are generated automatically.

On the other hand, it is also possible to generate solely the finger-print files during the record phase, because they contain all the data necessary to perform an equivalent execution. In this case, the record phase provides functionality to generate the finger-print files as specified above. Afterwards a replay phase can be used to generate the additional event graph data necessary for program analysis.

This possibility is also anticipated in Figure 4-4 on page 103, where the testing cycle and the debugging cycle each contains an instrumentation phase. Thus, during the instrumentation phase of the testing cycle (the left-most cycle of Figure 4-4), the program is instrumented to generate only the finger-print data. These data are obtained during the record phase of the nondeterminism cycle (the central cycle of Figure 4-4). If the program's execution is subject to debugging, the program is instrumented again to generate additional trace data and uses the finger-print as a constraint during the replay phase of the debugging cycle (the right-most cycle of Figure 4-4). This strategy allows to generate only a minimum amount of data and a minimum amount of monitor overhead during the initial record phases, while all the other data can be obtained during more intrusive replay phases of the debugging cycle. This approach is similar to incremental tracing as described in [Choi 91].

9.2.4 Possible Optimizations

The finger-prints as implemented in our strategy represent a very efficient way of storing data necessary for equivalent execution of nondeterministic parallel programs. However, the approach is not optimal, since it sometimes stores data that are not needed for re-execution. For instance, the last entry in each finger-print file of Table 9-3 is not needed for re-execution, because it only the remaining message has not been received so far. In fact, the last entry in the example finger-print above is not even a race condition candidate according to Definition 8-3 on page 206. However, it is included in our finger-print file because the monitor does not know that this record can be omitted.

The problem of this simple finger-print approach is, that some nondeterministic receives can be deterministic due to the structure of the program. This fact is also taken into account in Definition 8-3 on page 206, which distinguish between nondeterministic events and race condition candidates. In this definition, race condition candidates are only those nondeterministic

receive events with more than one racing message. Yet, in Table 9-3 the last receive on process P0 has only one racing message and is thus not a race condition candidate.

The problem of optimal storage of trace data for nondeterministic programs has been addressed several times in the past. Netzer describes a corresponding scheme for shared-variable programs in [Netz 93], which was also adapted together with Miller for message-passing systems in [NeMi 92b]. Additional improvements to this techniques have been achieved with the ROLT method (Reconstruction of Lamport Timestamps), which was introduced by Levrouw et al in [Levr 94] and [LeAu 94]. Ronsse et al evaluated these methods on different levels of granularity (from monitor operations to individual memory accesses) as evidenced in [RoLe 95], and included a simple but efficient compression scheme as shown in [Rons 95].

In cooperation with Ronsse, we adapted the original shared-variable approach of ROLT to message passing systems and named it $ROLT^{MP}$. A detailed description of $ROLT^{MP}$ can be found in [RoKr 98]. In addition to the simple finger-printing method described above, $ROLT^{MP}$ distinguishes between nondeterministic receives and race condition candidates, and produces only trace data for the latter. Its main advantage over traditional tracing techniques is, that it introduces only a minimal time and space overhead during monitoring due to applying Lamport clocks instead of vector clocks. Furthermore it is possible to apply the efficient compression techniques described in [Rons 95] for additional trace size minimization.

A comparison of $ROLT^{MP}$ with the simple finger-printing method described above also reveals, that the former may be applied to any kind of message-passing system, while the latter can only be used on non-overtaking systems like MPI and PVM. However, by sticking to non-overtaking systems the simple finger-printing method described above will generate even less monitor overhead during execution than $ROLT^{MP}$ and it is therefore sufficient for our needs, even though it sometimes generates data for nondeterministic events that can be ignored.

9.3 Ideas for Test Reduction

9.3.1 Overview of Ideas

The approach described in the previous section allows to perform automatic testing of nondeterministic programs, so that each possible execution can be evaluated for a given set of input data. Assuming that enough memory for storing all the finger-prints is available, and the time for performing the testing cycles is unlimited, an arbitrary program can be completely tested for a given set of input data. If no errors are revealed during these testing cycles, the program can be characterized as correct for the given set of input data.

The problem of this approach is, that available memory is usually limited and testing has to be completed within certain time constraints. This problem of test completion, also called the halting problem of testing, has been described in Section 3.2.2. A possible solution is described in Definition 3-1 on page 58 with the Good Enough testing strategy of Bach [Bach 98], which demands to apply testing within reasonable costs and time constraints, so that the resulting program quality is still sufficient. Thus, it is necessary to develop strategies to reduce the number of testing cycles.

Our ideas to reduce the testing amount are based on a comparison between automated testing and manual testing. Most tasks of automatic nondeterminism analysis are equal to the traditional manual approach. In fact, the biggest difference between these two techniques is the selection of the nondeterministic event and the corresponding possibility for event manipula-

tion. In the traditional manual approach, this selection is performed by the user, while in the automatic approach the tool simply chooses every possibility for each nondeterministic event once. Thus, the user may hold additional knowledge about a program's behavior, and event manipulation can be applied only to the most critical events observed during the initial execution. This knowledge is a priori not available, but it seems beneficial to integrate some additional knowledge in our automatic analysis approach. In concrete, we identified the following solutions for reducing the number of test cases:

(1) Integrating initial user feedback for removing intended nondeterminism
(2) Combining the event graph model with control- and data-flow characteristics
(3) Estimating the probability of message arrival due to observed timings
(4) Estimating the probability of message arrival due to hardware characteristics
(5) Investigating characteristics of observed event graph patterns

With these ideas we hope to reduce the number of iterations needed for testing a program as shown in Figure 9-5, while still providing a sufficient degree of program reliability. The differences between these solutions are also described by the following three characteristics:

- Scope: which cycle is affected by the reduction?
- Range: which types of nondeterministic events are affected by the reduction?
- Potential: how much reduction can be expected?

Firstly, a solution has a scope according to the cycle it affects, which identifies the reducible objects. Some solutions may either reduce the number of nondeterministic events (1,2), or the number of possibilities for some of the nondeterministic events (3,4), or both (5). Secondly, these solutions may be divided into two groups according to their range. The range of a solution is defined by the types of events that are removed. Some solutions affect every possible nondeterministic event (1,2,5), while others affect only nondeterministic receives (3,4). Thirdly, the solutions offer different potential in terms of reduction and amount of reducible objects. Some ideas allow to decrease the amount of testing highly, while others may only offer limited reduction possibilities. Yet, determining the potential absolutely for each of these solutions is impossible, because it is at least always affected by the target program, and sometimes even determined by the user's knowledge (e.g. 1). Thus, every statement about the potential of a test case reduction technique can always only be an estimate based on our previous experience.

A major problem of all these test case reduction techniques is the fact, that they have to be applied iteratively for the initial execution and each of its descendants. On the one hand, this may be again a critical issue, if test case reduction needs more processing power than testing every possibility for each nondeterministic event. On the other hand, it may be difficult for semi-automatic approaches like (1), where users are involved to reduce test cases based on their knowledge. For that reason, we assume that the test case reduction technique applied to the initial execution is equally valid for each of its descendants. In any case, applying automatic nondeterminism testing and test case reduction has to be performed wisely due to the required processing power.

9.3.2 Integrating Initial User Feedback

Including initial user feedback in the automatic nondeterminism analysis technique seems obvious from the problems described above. If users support the automated approach with some knowledge about the expected nondeterministic behavior, the tools may reduce the number of needed test cases.

The idea of initial user feedback is to distinguish between different kinds of nondeterministic events according to their introduction in the code. Such a distinction has been described in Section 8.1.6, where we introduced three kinds of nondeterministic events, unintended, intended, and the remaining that cannot be added to one of the other groups (compare also [DaFr 94]). The latter represents the only group of events, that needs to be addressed during nondeterminism analysis. Thus, if users remove all nondeterministic events that are obviously unintended, and afterwards identify all nondeterministic events that are intended, only those events that need to be investigated will remain.

The range of this solution are clearly all types of nondeterministic events. User may distinguish intended and unintended events for random number generator events, nondeterministic nonblocking receives, and nondeterministic blocking receives. The scope of this solution is only the set of nondeterministic events, because it does not distinguish between different possibilities for identifying intended and unintended events. The potential of this technique cannot be expressed by any estimation, because it depends entirely on the chosen application and the user's knowledge about the expected program behavior.

Integrating this distinction in the testing and debugging cycle of Figure 4-4 on page 103 and in the automatic testing cycles of Figure 9-5 on page 239 can easily be achieved. Since this distinction can only be made based on an observed execution for a given set of input data, it needs to be included after the initial execution was observed (first record phase) and before event manipulation is applied.

In practice, the initial user feedback can be provided by displaying the event graph of the initial execution as a space time diagram and visualizing all nondeterministic events (as shown in Figure 8-4 on page 211). Afterwards, the user may choose to identify those events, that are subject for automatic testing, and those events that can be ignored. The latter is removed from the set of events that are addressed in the second loop of the right diagram of Figure 9-5 ("for each nondet event").

9.3.3 Exploiting Control- and Data-Flow Characteristics

Control- and data-flow analysis represents a different technique to analyze a program than the event graph model. The former concentrates on static analysis of the source code and does not take into account an actual program execution. On contrary, the latter investigates only one instance of the program as observed during an execution with a given input. As discussed in Section 7.4.4, we have combined both approaches in order to benefit from each of these techniques. Concerning automatic nondeterminism analysis, the benefits of this combined approach as described in [Kran 97b] are:

- Improved program understanding
- Identification of uncritical nondeterministic events.

Understanding a program can be improved by adding different kinds of visual representations. For instance, instead of showing only the space-time diagram of the event graph model and the

source code reference of selected objects, control- and data-flow analysis may offer additional representations like function call graph and control flow graph.

An example for this combination is shown in Figure 9-7, where an event has been selected in the event graph display (top-right window) and the corresponding source code line is shown in the integrated filebrowser. In addition, the control-flow of this functional block is shown in the bottom-middle window and the function call-graph is shown in the bottom-right window.

The advantage of improved program understanding is, that the user may be able to identify more nondeterministic events as either intended or unintended, which allows to remove them from the testing cycle as shown in Section 9.3.2.

The second possibility of using the control- and data-flow information is to identify uncritical nondeterministic events. These are nondeterministic events, which do not affect the subsequent code fragments and may therefore not modify the behavior of the investigated program.

A very simplified case is discussed with the two code segments of Example 9-3 and Example 9-4 [Kran 97b]. In both example, the code fragment is only presented for process *P0*, which performs `N`-1 a wild card receive operation, where `N` is the total number of available processes. The remaining processes send a message to process *P0*, which stores the contents of the message in the variable data.

Example 9-3: Uncritical code fragment

```
Process P0:

     ...
     for(i=1;i<N;i++) {
             receive(data,?,?);
             }
```

Example 9-4: Code fragment which may expose different behavior

```
Process P0:

     ...
     for(i=1;i<N;i++) {
             receive(data,?,?);
             do_something(data);
             }
```

In Example 9-3, the code on process *P0* ignores the contents of the message completely. In fact, each subsequent message overwrites the contents of the previous one. Assuming, that the variable `data` is not accessed after the given loop, this program will always reveal the same output although the messages can be accepted in any order. Please note, that the program is still nondeterministic as defined in Section 4.2.4, since the event graphs for two executions may differ if more than 2 processes are available. An example usage of this code fragment would be a synchronization point, where each of the processes announces that it reached a certain location in the code by sending the synchronization message to process *P0*.

In contrast to Example 9-3, the code fragment of Example 9-4 cannot be identified as uncritical for nondeterminism analysis, since the function call `do_something` may reveal different behavior depending on the contents received by the incoming messages stored in variable data. If function `do_something` is actually affected by the contents of data, the pro-

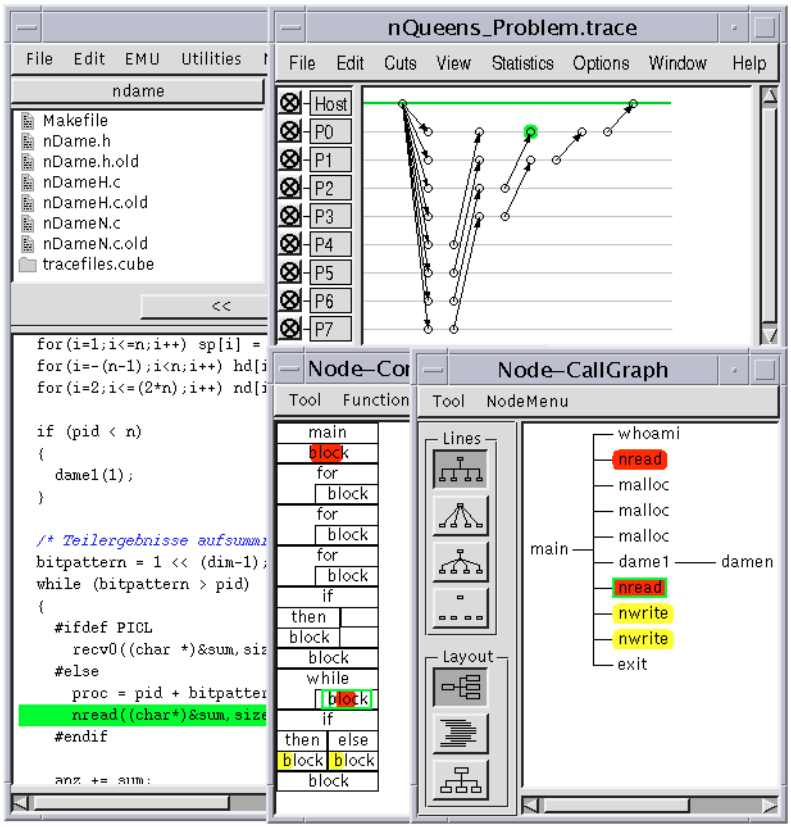


Figure 9-7. Control- and data-flow analysis and event graph [Kran 97b]

gram may deliver different output corresponding to the arrival order of the messages on process *P0*.

Although these two examples are very simple, they still show the potential of applying control- and data-flow analysis for testing nondeterministic parallel programs. By identifying uncritical code fragments, that are not affected by the results observed at nondeterministic events, the set of events needed for testing can be decreased by removing uncritical events from the testing cycles.

Similar to the solution discussed in Section 9.3.2, the range of this solution are all types of nondeterministic events, since the subsequent program flow may be affected by random number generator events or communication events, respectively. Furthermore, the scope of this idea is only the set of nondeterministic events, because different possibilities are not addressed by the control- and data flow analysis tool. A drawback of this approach is that its usage is restricted to few algorithms (e.g. all-to-one communication), which obviously limits its application for test case reduction.

9.3.4 Estimating Message Arrival Order with Observed Timings

While the solutions for test case reduction shown so far describe nondeterministic events precisely, the solution discussed in this subsection is much more fuzzy. Besides that, the range of this approach is limited to the set of nondeterministic receives, while the scope of this approach is limited to the set of possibilities observed at these types of events. These possibili-

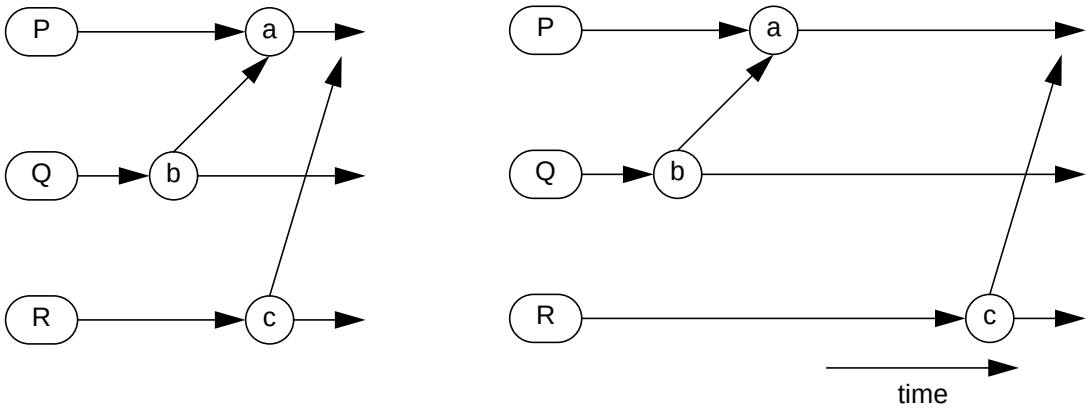


Figure 9-8. Probability of message arrival

ties are the arrival of a message for nonblocking receives and the order of the incoming messages for blocking receives.

The principal idea is to investigate the probabilities of the different possibilities to occur at an arbitrary nondeterministic receive event during subsequent program executions with the same input data. For instance, if two racing messages arrive at a process, it seems to be more likely that they will appear in different order if the arrive closely after each other. On contrary, if two racing messages arrive far apart from each other it seems that most times the first one will arrive at the corresponding receive event.

An example for this idea is shown in Figure 9-8 with two processes (*Q* and *R*) sending a message to process *P*, and the receive operation on P is assumed to potentially accept each of these racing messages. In the left diagram of Figure 9-8 the message from send event *c* on process *R* arrives shortly after the message from send event *b* on process *Q*, while in the right diagram the message from send event *c* arrives much later. Therefore, the probability of these two messages arriving in reverse order is much higher in the left diagram than in the right diagram.

A first approach based on this idea was described by Grabner and Volkert in [GrVo 93], who introduced an epsilon-interval with a user adjustable size. During race condition detection only those racing messages where highlighted as critical, that arrived within the given interval. The problem of this approach is, that it offers only a very rough estimate for the probability of different message arrival, since racing messages are either within the interval or not. Thus, it seems a useful improvement to appoint different probabilities for each racing message of a nondeterministic receive event according to the corresponding likelihood of different message ordering. This means, that the event graph has to be scanned so that each racing message is weighted according to its probability of arriving at a nondeterministic receive event first. Afterwards, the user may choose to perform testing only on those possibilities with the highest probability, and all other possibilities can be removed from the testing cycle.

Although this idea seems very useful at first glimpse, it faces several problems that question its results. Firstly, most monitors measure only the time when the message is accepted by the corresponding receive operation. This does not take into account, that a message may be available in a process' message buffer long before it is extracted by a receive statement. In addition, racing messages generated by isolated send events are often ignored completely[2]. This problem can be solved by improving the monitor to provide the actual times of message arrival in

2. This problem was also overlooked in the approach described in [GrVo 93], because it applied the interval only between receive operations that have actually been observed.

the message buffer of the destination process, which is not a simple matter due to its proximity to low level communication functionality.

Secondly, the timing data used for determining reducible objects is based on a single execution of the target program. This execution may be affected by other applications being executed on the same hardware architectures, therefore falsifying the results of the estimates. As described in Section 4.2.2, the influence of the hardware on parallel program analysis is much bigger in parallel systems than in sequential computers. In order to improve the quality of the estimations, a possible solution is to perform repeated executions of the target program with the same input data. Afterwards the critical estimations can be based on accumulated estimates of the obtained set of executions.

Thirdly, the observed execution may not be equivalent to the execution that is revealed when monitoring is turned off. This effect is called the Heisenberg (Uncertainty) Principle or Probe Effect, which have been introduced in Section 4.2.1 and Section 4.2.8. Yet, since monitoring is only applied during testing and debugging, the important executions are those that would occur without attached monitor. Thus, the probability estimates are based on incorrect timing data and may expose possibilities that are not relevant in practice.

The solution to this problem is to determine the execution that would most likely occur without the monitor. This can be achieved with monitor overhead removal techniques applied to the initial execution. Such techniques have been described in Section 4.2.8, and the approach discussed in [Kran 99b] and [Scha 99] has been developed especially for the event graph model by combining monitor overhead removal with sophisticated clock synchronization as well as automatic event manipulation and artificial replay. Since nondeterministic programs may expose several possible executions with almost equal probability to occur when monitoring is turned off, and the accuracy of monitor overhead removal algorithms is limited, the monitor overhead removal algorithm mentioned above delivers not only one possible execution, but the set of executions that are most likely to occur in reality.

9.3.5 Estimating Message Arrival Order with Hardware Characteristics

The solution described in this section is similar to the idea mentioned above, because it also tries to estimate the arrival order of racing messages. Therefore, it exposes the same scope and range, which means that the reducible objects are the possibilities at nondeterministic receive events. In contrast to the previous idea, which is based on observed message arrival times, we discuss the influence of certain hardware characteristics on a program's behavior. The basic idea is to investigate, how the hardware affects the message arrival order.

Our assumption is, that some messages would be favored over others due to certain characteristics, like distance between sender and receiver, different latency times on the communication channel, and blocking in the network. In order to prove this hypothesis and to understand the influence of the hardware onto the program behavior, we conducted several experiments of which we selected one for this work. The selected experiment was to execute the code of Example 9-1, "All-to-one communication", 10 million times without monitoring or any other constraints on different hardware architectures with 8 nodes. We selected Example 9-1 due to its simplicity and repeated the experiment for different numbers of processes and under different system loads, always obtaining comparable useful data. For 8 processes, the program may expose 5040 different results as described in Table 9-1 on page 230. By analyzing the observed executions and deriving the accumulated heuristics of these data, we are able to identify characteristics that may be useful for test case reduction. The results of these experiments are summarized in [Kran 99a].

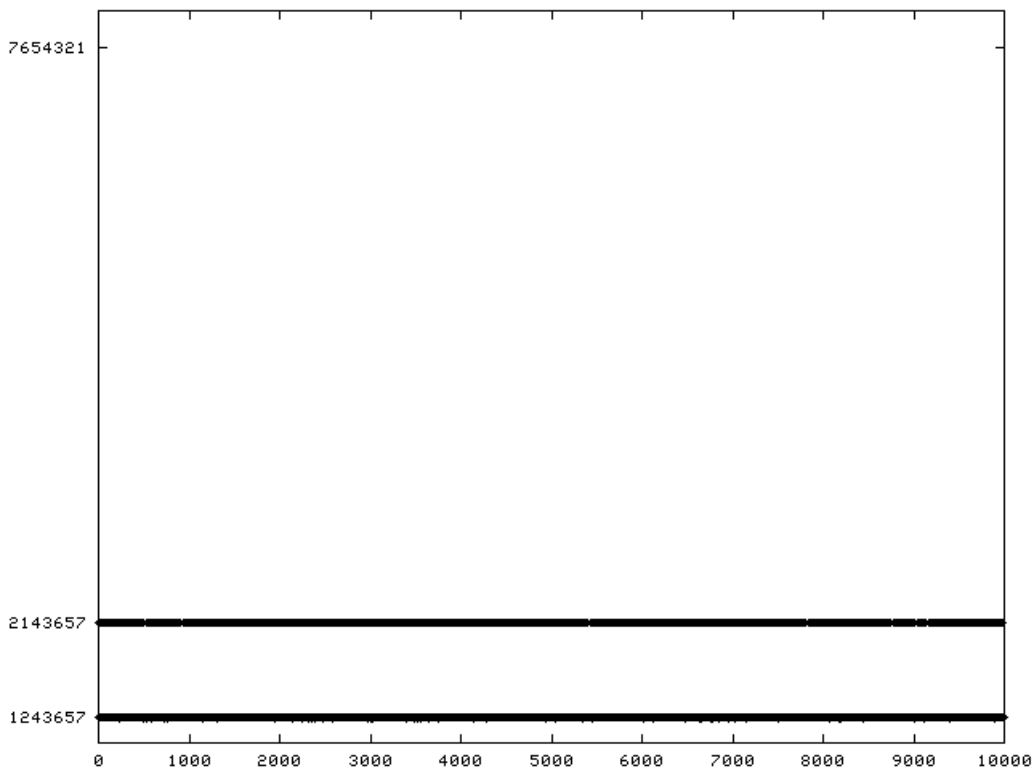


Figure 9-9. Results on the nCUBE 2 multiprocessor [Kran 99a]

A portion of 10.000 results of this experiment is shown in Figures 9-9, 9-11, and 9-13. The x-axis contains the observed execution numbered 1 to 10.000, while the y-axis contains the results from 1234567 to 7654321. Please remember, that the results of Example 9-1 is the order of the incoming messages. The dots in the diagram indicate, which result has been observed during the selected execution.

The results presented in Figure 9-9 have been obtained on the nCUBE 2 multiprocessor architecture. This is a MIMD multiprocessor, where proprietary nCUBE 2 nodes (64-bit CISC CPUs with FPU (Floating Point Unit) and 14 interconnection ports) are connected via a hypercube network topology [Hwan 93]. The important characteristics of such a hypercube architecture are, that each node is connected with *n* other nodes, where *n* is the number of dimensions included in the topology [HwXu 98].

The experimental data of Figure 9-9 are very promising, because they emphasize the validity of our assumption. It seems, that from the 5040 different execution, only results in two areas (around “1243657” and “2143657”) will be observed in practice. In concrete, for all 10 million executions of this example, we observed only 116 different executions, which represents only a share of 2.3% of the possible executions. Furthermore, almost 44% of the execution delivered the result “1243657”, and only 7 execution deliver over 96% of the observed results. The distribution of this 7 executions is shown in the diagram of Figure 9-10.

The observed results can be attributed to one main reason, namely the distance between the sending nodes (process *P1* on node 1 to process P7 on node 7) and the receiving node (process *P0* on node 0). In a hypercube architecture, each node is only connected to a limited number of neighbors. Communication with one of the remaining nodes can only be established by transferring the messages via intermediate nodes. The number of links between two nodes is repre-

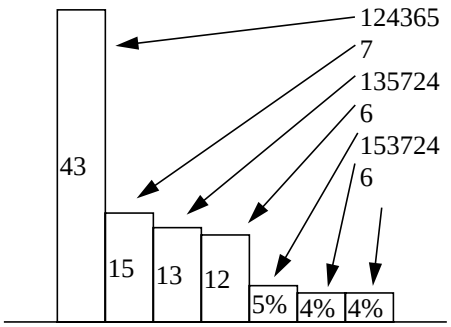


Figure 9-10. Cumulative results on the nCUBE 2 multiprocessor [Kran 99a]

sented by the Hamming distance. For the experiment on the nCUBE 2 multiprocessor this means, that nodes with smaller Hamming distance will deliver their messages earlier to the master process due to their shorter communication path. This is indicated in Figure 9-9 and Figure 9-10, where the messages from process *P1*, *P2*, or *P4* always arrive first.

The same experiment was also executed on the SGI Origin 2000 multiprocessor, which is currently more a state-of-the-art machine than the nCUBE 2. Some representative machines are even included in the TOP 500 supercomputer list [Dong 99], like the SGI ASCI Blue Mountain, which is composed of 48 SGI Origins with 128 processors each (see Table 2-1 on page 40). The hardware characteristics of the SGI Origin 2000 represents a CC-NUMA architecture (Cache Coherent - Non-Uniform Memory Access) [HwXu 98]. It was announced in October 1996 by SGI/Cray and also known as S^2MP (Scalable Shared-memory MultiProcessing) system. The nodes consist of two processors (MIPS R10000 or higher), that are by default connected via a so-called fat hybercube topology [HwXu 98].

The results observed on the SGI Origin 2000 multiprocessor are different to that observed on the nCUBE 2 architecture as shown in Figure 9-10. In concrete, around 100 executions represent almost 80% of the 10 million observed results. A reason for the different behavior of the target program is certainly the increased complexity of this machine and its features, like symmetric multiprocessing, virtual shared memory, and others. However, the SGI Origin offers basically the same hypercube topology as the nCUBE 2, which was mainly responsible for observing the results shown in Figure 9-9, and it seems as if our assumption may be too weak for this architecture.

There are several reasons for this discrepancy. A first reason is that the hypercube architecture incorporated in the SGI Origin is more advanced compared to the nCUBE 2's hypercube. Its main difference is, that each node in the SGI Origin's hypercube contains 4 processors instead of 1 processor as on the nCUBE 2. These 4 processors are located closely together and access the same shared memory. Besides that, the connections to the other processors is established via different links, resulting in different communication bandwidth. The block diagram of Figure 9-12 shows the machine topology of our SGI Origin with 32 processors arranged as 8 nodes as available during these experiments at the Johannes Kepler University Linz.

A second reason for the unsatisfying results is, that the nodes in the nCUBE 2 are usually used exclusively by the user's application, while program's on the SGI Origin have to share the processors and other resources with applications from other users. Besides that, the operating system on the SGI Origin (IRIX) includes several mechanisms, to re-arrange the application to processor mapping during runtime in order to obtain a better load balance.

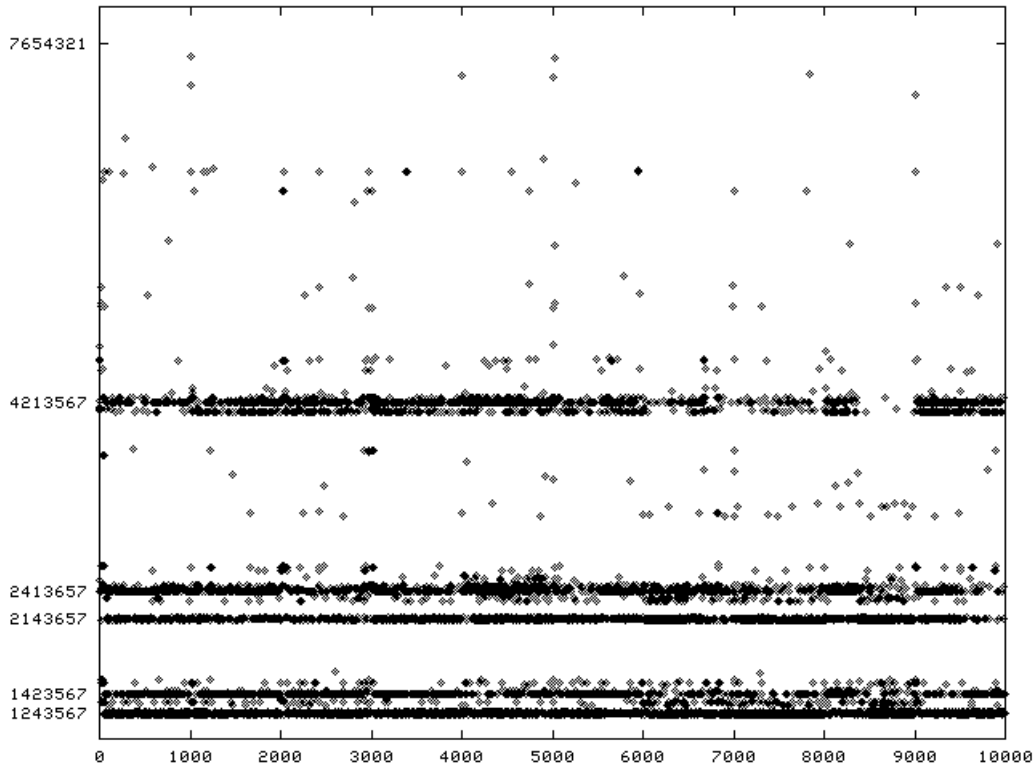


Figure 9-11. Results on the Origin 2000 without process placement [Kran 99a]

As a consequence, we have repeated our experiments on the SGI Origin during system time, so that the small test application from Example 9-1 was provided with exclusive access to the system's resources. In addition, we disabled the process placement of the operating system by appointing the program's processes to explicit nodes in the hypercube. The results of this revised experiment as shown in Figure again support our hypothesis above. In fact, only two

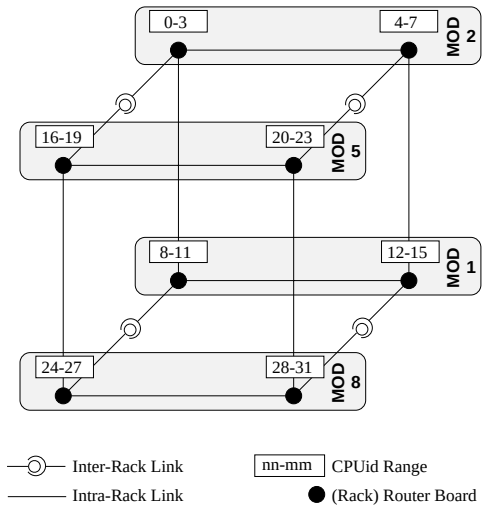


Figure 9-12. Origin 2000 hypercube architecture [Kran 99a]

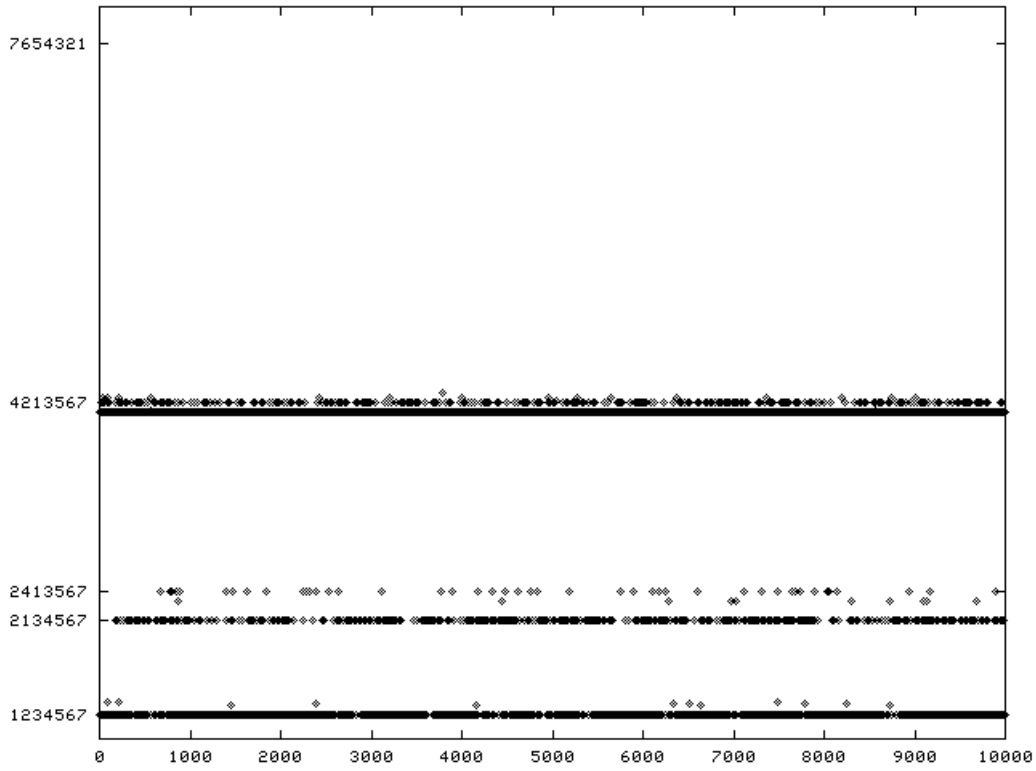


Figure 9-13. Results on the Origin 2000 with process placement

combinations delivered over 92% of the possible executions, and one of these two executions ("4123567") represented over 70% of the observed results. Additionally, less than 100 different executions were observed during the 10 million executions, and the majority of these executions represented less than 1% share.

This small experiment is only a small excerpt from the on-going research in this topic. However, the results indicate that the characteristics of the underlying hardware architecture may actually deliver useful hints for reducing the number of needed test cases. If only those executions are tested that accept the privileged racing messages first, it seems as if the most important test cases are covered. The next steps in this research project will be to integrate this results into the automatic nondeterminism analysis approach. It remains to be seen, whether the causes for this different probabilities can be determined by a tool automatically and for other hardware architectures as well.

9.3.6 Exploiting Observed Event Graph Patterns

The solutions described so far apply test case reduction either by applying user knowledge or by deriving knowledge about a program's execution from the source code, the timing data of events, or the characteristics of the underlying hardware. The reduction technique described in this section focuses on the shape of the event graph. Since the event graph consists of vertices displaying nondeterministic events and of edges describing their possibilities, this solution's scope are both inner cycles of Figure 9-5, while its range are all types of nondeterministic events.

The principal idea of exploiting observed event graph patterns is that geometric features of the event graph can be utilized for identifying similarities in a program's behavior. This idea is supported by our past experience, with many event graphs revealing patterns (e.g. Figure 9-2 on page 233), and by the fact that programming model's like Single-Program-Multiple-Data (SPMD) incorporate similarity during program construction and execution (see Section 2.3.2). Some examples of using shape characteristics of the event graph for test case reduction are described below. Other more advanced applications for pattern matching in the event graph are discussed in the next chapter.

A very simple idea for test case reduction based on event graph patterns is to detect loops in the graph, which indicate repeated behavioral cycles during program execution. If such a loop can be detected, automatic nondeterminism analysis can be limited to one solely iteration of this loop, instead of testing the same loop over and over. Other examples consider groups of processes, where nondeterministic behavior observed in one group also emerges in other groups, so that only one of these groups need to be tested. Extending this idea finally leads to a point, where arbitrary symmetric features of the event graph can be exploited in order to remove the number of testing cycles.

A high potential for test case reduction with pattern detection is Example 9-2, "Data circulation". As discussed in Section 9.1.3, this small program would require to generate $\left(n^2 - n\right)^{\log n}$ executions to perform complete testing, where n is the number of available processes. However, as can be seen in the top-most diagram of Figure 9-2 on page 233, the program reveals completely symmetric features across all processes, because each process is actually doing the same thing during communication, albeit with different communication partners. With this detection it is possible to reduce the number of executions needed to $(n-1)^{\log n}$, with $(n-1)$ being the number of nondeterministic receive events, while $\log n$ represents the number of racing messages expected at each of these receives. Furthermore, if symmetric properties along the horizontal time axis are considered, only $(\log n)^{\log n}$ executions remain (see [KrVo 99]). This means, that instead of obtaining 175.616 executions for the data circulation program on 8 processes, only 27 different executions are needed. All other executions can be derived by renaming process names and combining the results on the horizontal axes.

Besides the patterns observed with blocking receives as described above, similar features can be detected for other nondeterministic events like random number generator events and nonblocking receives. The problem of this approach is only, that some of the processes may exhibit different behavior due to special conditions at the borders. However, this knowledge may again be provided by the user, so that a combined approach with the solution described in Section 9.3.2 seems most useful.

9.4 Summary

This chapter describes how to perform nondeterminism analysis automatically. The usefulness of such a technique is explained with the distance doubling program example, that reveals a high number of possible executions although the actual application consists only of a few lines of code. The required number of event manipulations can easily overwhelm the user, making the traditional manual approach to nondeterminism analysis impossible.

The automatic approach tries to relieve the user from applying manual event manipulation as described in Chapter 8 by performing the necessary steps automatically. This shifts the lim-

its of nondeterminism analysis from small scale applications with few possible executions into an area, where testing is only bounded by constraints in processing time and available memory.

Two main characteristics shape the automatic approach to nondeterminism analysis. Firstly, it consists of three cycles, one for each derived descendant execution, one for each nondeterministic event in each execution, and one for each possibility at each nondeterministic event. Secondly, it requires a sophisticated key to identify equivalent executions. The latter was introduced by finger-printing of applications, which allows to compare executions easily and efficiently. A side-effect of this finger-printing mechanism are several optimizations that have been integrated in the original testing and debugging cycles.

Finally, this chapter discusses some solutions for reducing the number of iterations performed in the three cycles of nondeterminism analysis. The basic idea of these solutions is to perform testing only on those executions, that are most likely to occur in reality. Consequently, the best results can be expected, if all these solutions are combined for test case reduction. This is especially important for software engineers that have to meet deadlines while at the same time guaranteeing a certain degree of reliability. Thus, it is certainly justified to perform additional investigations in this area in order to integrate more knowledge into the automatic testing cycle.

10. Advanced Event Graph Techniques

The features discussed in this chapter supplement the debugging techniques described in the previous chapters by addressing two problems occurring especially during debugging of massively parallel programs, the high numbers of processes and the long execution time. The presented solution is the introduction of techniques for further abstraction of the event graph, for reduction of the processed debugging data, and for modifying the visual representation of the event graph. With these techniques, the event graph data is reduced to a small amount of data, which is assumed to contain the crucial information. Most of these advanced techniques can be applied either post-mortem, which targets at the displayed event graph visualization, or on-line during re-execution of the target program. Most techniques may either be applied automatically by the tool or manually according to the users needs.

- Problems during debugging of massively parallel programs
- Program slicing and event graph abstraction
- Classification of abstraction techniques
- Post-mortem abstraction techniques (horizontal/vertical)
- On-line abstraction techniques (horizontal/vertical)

10. Advanced Event Graph Techniques

10.1 Requirement and Ideas

10.1.1 Problem: Numbers of Processes and Execution Time

The technique described in the previous section uses the event graph model for automatization of nondeterminism analysis. Another important feature that should be offered by a debugging approach is a set of techniques, which addresses problems associated with the amount of debugging data generated by massively parallel programs. As described in Section 4.4.2, this problem can be attributed to two distinct characteristics of such applications:

- The numbers of processes applied in massively parallel programs are very large.
- The execution times of massively parallel programs are very long.

Concerning the number of processes, massively parallel programs may scale up to hundreds or even thousands of processes. At the same time, the execution times of such applications are in the range of days, weeks, or even month. The results are huge amounts of analysis data, which have to be stored and processed for testing and debugging. If the observed data can be stored, constructing a corresponding event graph model may be at least very time-consuming if not impossible. Displaying the event graph model as a space-time diagram may be even more useless, because only small fragments of the graph can be displayed on the screen. This problem of displaying all available information is well-known. It is told that Niklaus Wirth once commented: “Details are the jungle in which the devil hides” [Cai 93].

The problems associated with the high amount of debugging data are illustrated with the examples shown in Figure 10-1. Both space-time diagrams represent the event graph models of program executions, which executed a relatively long time. The number of available processes was limited to 32 due to restrictions of the available multiprocessor architecture, the capabilities of the visualization tool, and the available screen size for the screenshots. Although this represents only a massively parallel program of small size, it is sufficient for our purposes. Besides that, with the techniques described in the subsequent chapters, we hope to reduce the displayed data to an even smaller amount.

The left diagram of Figure 10-1 shows the event graph model of a parallel Gaussian elimination, which contains approximately 6000 events. Due to the regularity of this algorithm, the shape of the event graph is not very complex and it seems possible to debug this program even on the given scale with traditional techniques.

This is probably not the case in the right diagram of Figure 10-1, which shows an event graph visualization of a parallel finite element solver. In total, this application generated over 34000 events during 200 iterations. The resulting event graph shape seems to reveal no patterns at all, and debugging this program may obviously be very difficult and time-consuming.

One solution to these problems is certainly down-scaling of the application, which means to reduce the number of processes and decrease the applied problem size. This idea is also suggested by Geist et al in [Geis 94a], and has been briefly described in Section 4.3. Although this approach may be useful in many cases, it cannot eliminate the necessity to test the program at scales that will be applied during everyday operation. Therefore, a testing and debugging tool

Gaussian Elimination Solver Finite Element Solver

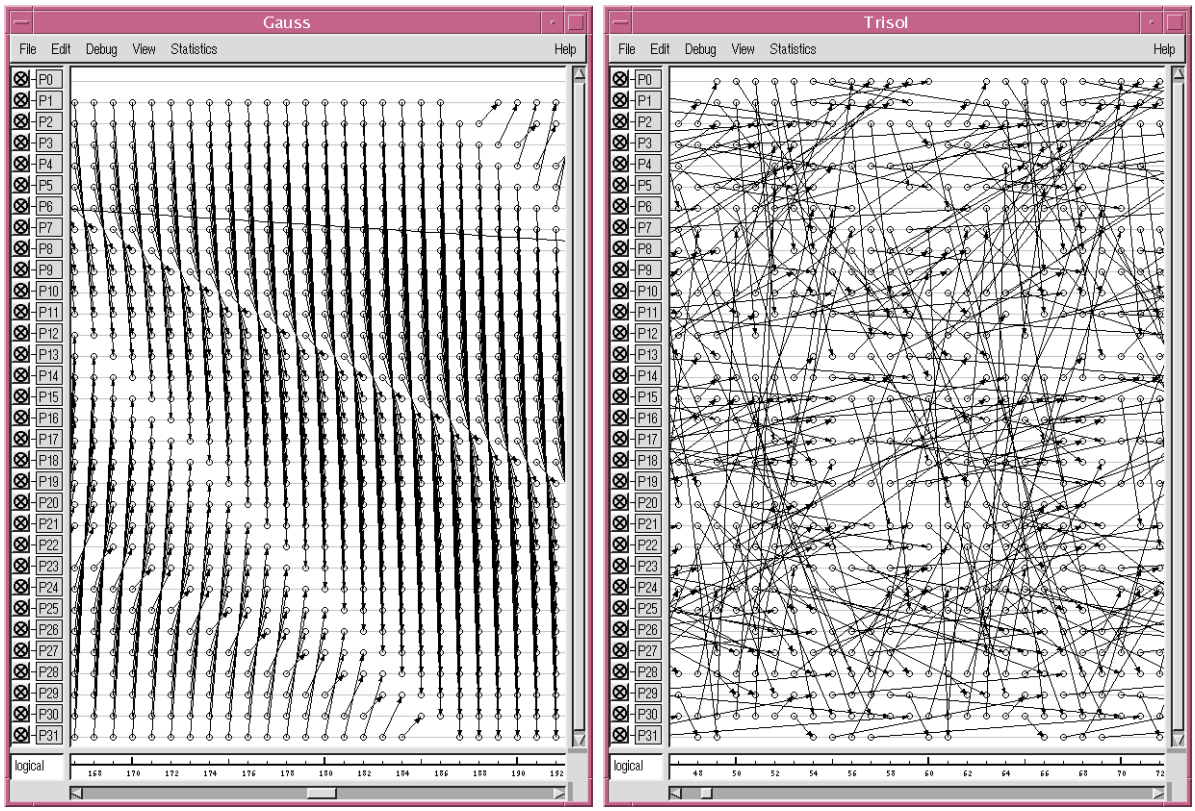


Figure 10-1. Complex event graph displays observed in massively parallel programs

should offer some means of managing these amounts of data, provided that the data can be stored for processing.

10.1.2 Principal Idea: Focus Only on Important Aspects

For the subsequent sections we assume, that the application is observed in its original size as intended by the user. This means, that the number of participating processes is rather high and the total execution time is very long. As a consequence, the reduction must take place based on the analysis data included in the event graph model, either during observation or during visualization.

The principal idea of our approach is similar to program slicing, which was originally introduced by Weiser in [Weis 84]. Program slicing is a reduction method that tries to decompose programs by analyzing their data flow and control flow. In concrete, slicing reduces the program to a minimal form, called a “slice”, which is an essentially independent program guaranteed to represent a specified subset of the original program behavior. A slice is constructed with respect to a given variable (or a set of variables), so that it consists of those parts of the program which may directly or indirectly influence the value of that variable at a selected program point.

The approach of Weiser focuses on static slicing of sequential programs. A static program slice is computed through static data and control flow analysis and is valid for all possible executions of a program. This technique has been further refined by Agrawal et al as described in

[Agra 93] and [Agra 94]. Computing a dynamic slice for one particular program execution is described in [KoLa 90] and [AgHo 90].

Slicing of parallel and distributed programs can be found in [KoFe 92], [Chen 93], and [Dues 92]. Kamkar at al describe how to apply an interprocedural dynamic slicing algorithm as introduced in [Kamk 93] to parallel message-passing programs [Kamk 96]. In [Kran 96d], we describe our approach to variable backtracking in message-passing programs, which generates slices across process boundaries by combining the event graph model with a simple slicing technique. Another application of slicing is described by Grabner in [Grab 99], who compares the flowback analysis technique of Choi [Choi 91] with the slicing technique of [Weis 84] and describes, how debugging can benefit from a combination of both techniques. Grabner also distinguishes between two different approaches, forward slicing and backward slicing [Grab 99].

One of the main applications of slicing is certainly program debugging [Kamk 96]. Weiser even claims, that slicing is exactly what user do during debugging [Weis 82]. They mentally decompose the program into slices and follow the slice thought responsible for the erroneous behavior. Thus, the important idea is to concentrate on the most important parts of a program - those containing the error - and ignore all other parts. This is also one of the main ideas of using the event graph model as described in Section 5.1.2, which contains only objects of interest. The main differences between slicing and modelling a program with the event graph are:

- Slicing is based on control- and data flow analysis, while the event graph describes program state changes of one particular execution.
- Slicing focuses on the textual source code of a program, while the event graph consists of arbitrary events occurring during execution.
- Slicing connects lines of code that affect one selected object of interest, while the event graph connects its events with the happened-before relation.

Consequently, the method applied for constructing the event graph is not slicing as described by Weiser [Weis 84], although it also follows the idea of reducing the amount of analysis data. For the event graph model, a more appropriate term is “abstraction”. The question is, whether the level of abstraction incorporated in the event graph model described so far is appropriate for all debugging activities, or whether it may be useful to reduce the event graph model through even higher levels of abstraction. Therefore, we describe some techniques for increasing the level of abstraction in order to reduce the debugging data. With the resulting higher levels of abstraction users are be able to cope with the huge amounts of debugging data offered by massively parallel programs.

Another characteristic, that may lead to abstraction mechanisms is the fact that differences in the code of parallel programs are often rather small, even if a program is running on thousands of processes. In most cases, the program has been implemented using data-parallelism as offered by the SPMD (Single Program Multiple Data) programming model as described in Section 2.3.2, and the code executed on all the processes is actually the same code, although processes may execute different code blocks concurrently. Even if a program is implemented using the task-parallel MPMD (Multiple Program Multiple Data) programming model, there are usually groups of processes doing comparable work and thus revealing similar behavior. The main distinction between the processes comes either from the responsibility of a process (e.g. master-slave) or the data they are working on (e.g. process groups). With the abstraction mechanisms described below, the analysis data revealed by these programs can be reduced to the necessary parts. For instance if there is an error in the master process, why bother about

visualizing all the slave processes. Or if there is an error in one group, why visualize all the other groups as well.

10.2 Classification of Event Graph Abstraction Techniques

The idea of using higher levels of abstraction is used in many other approaches, not only for error detection and performance tuning, but for a lot of different analysis activities. A brief overview of related work in this area is presented in Section 4.5.6. All these abstraction mechanisms target one major goal, to help the user in coping with large amounts of debugging data. In concrete, we can derive two important benefits from most existing abstraction techniques:

- Reduction of data and associated complexity.
- Improvement of program understanding.

In order to provide an overview of techniques, which may be applied to the event graph model, we will now introduce our classification scheme. To our knowledge, such an overview does not exist in literature, which may be attributed to several reasons. Firstly, most of these techniques are specific to one problem or one particular tool, respectively. Secondly, such an overview can never provide a complete set of possibilities, no matter how many items are included. In this context, Bates and Wileden noted, that no particular abstraction mechanisms can be appropriate for all problems [BaWi 83b].

Consequently, our ideas have been developed during our work on the event graph model and are certainly biased towards our approach. However, due to the event graph's universal characteristic we believe that most of these techniques are equally useful in other debugging approaches as well.

An important remark concerns the techniques being included in our classification. Although our principal focus is on abstraction techniques, where abstraction usually means some kind of mental generalization, we will include several other related ideas under this hull. Some of these ideas are transformation, reduction, extraction, isolation, and filtering, as well as techniques for modifying the visual representation of the graph.

Remark

The term abstraction is used in this work as a kind of generalization for different kinds of event graph operations and transformation techniques addressing problems associated with huge amounts of debugging data.

The basic characteristics of our abstraction techniques can be divided into three groups according to the following questions:

- How to apply abstraction?
- When to apply abstraction?
- Which dimension to use for abstraction?

The first question discusses, how to apply abstraction techniques in general. This distinguishes between three possibilities:

- Manual abstraction
- Semi-automatic abstraction
- Automatic abstraction

Manual abstraction mechanisms are those, where all the responsibility for the abstraction is taken by the user. The user decides not only, which kind of abstraction mechanism to apply, but also how it is applied. Since this method relies heavily on user interaction, its applicability is certainly limited to a small applications. As an extension, semi-automatic abstraction techniques relieve the user from performing the abstraction of the data. Yet, the user has still to decide, which abstraction techniques to perform on a given event graph. Although this represents a huge improvement over the manual approach, it is even more preferable that abstraction techniques are applied without any user interaction. Such automatic abstraction mechanisms could try to apply as many techniques as possible, and the user only needs to decide, whether to investigate the abstracted or the original analysis data. Please note, that some abstraction mechanisms cannot be applied automatically and are therefore limited to the group semi-automatic approaches.

The second question targets at the time, when abstraction mechanisms are applied. In most cases, abstraction mechanisms are applied when analysis data has been generated and is available for processing. Yet, the visualization is only one possibly target for abstraction mechanisms. Another possibility is the execution of the target program, when the analysis data is actually generated. In that case, abstraction may be applied to reduce the program to smaller portions of time or to execute it only with parts of the totally needed processes. Therefore, we distinguish two possible times for applying abstraction mechanisms:

- On-line (on the re-execution) when the analysis data is generated
- Post-mortem (on the display) after the analysis data has been generated

The third question addresses the range of the abstraction mechanisms. Since the event graph model contains only events of interest, this set of events already represents one range [McHe 89]. The other two abstraction ranges can be applied on the two dimensions incorporated in the event graph, which delivers the following abstraction techniques:

- Abstraction on time axis
- Abstraction on process axis
- Abstraction on time axis and process axis simultaneously

Reduction on the time axis concerns the processed time-interval in case of on-line techniques and the number of displayed events in case of post-mortem techniques. Similarly, abstraction techniques operating on the process axis reduce the number of executed processes for on-line approaches, and the number of displayed processes for post-mortem approaches. The highest reduction factor can be gained by abstraction mechanisms, that operate on both axes at the same time. This group of abstraction mechanisms covers techniques that are either combinations of the one-dimensional techniques mentioned above or specific approaches targeting both dimensions.

With the basic characteristics of abstraction techniques as described above, we defined the classification given in Table 10-1. This table contains some important examples of abstraction mechanisms, which are grouped according to their characteristics. The rows of this table distinguish between post-mortem and on-line abstraction techniques, while the columns distin-

guish the two axes and their combination. Please note, that the given abstraction techniques represent only examples, and many other techniques do exist.

Table 10-1: Classification of abstraction mechanisms

	time axis	process axis	combined
post-mortem	loop detection collapsing	process grouping process hiding	zooming/scaling filtering error extraction pattern matching
on-line	sliced execution	process isolation	sliced execution of isolated processes

The post-mortem abstraction mechanisms mentioned in the first row of Table 10-1 are only briefly discussed in Section 10.3. For a more extensive discussion please refer to [Kran 96b]. Probably the most powerful technique is pattern matching, which will be briefly described in Section 10.4. The group of on-line abstraction mechanisms, which is shown in the second row of Table 10-1, is certainly not as advanced compared to its post-mortem counterparts. Some ideas in this area will be summarized in Section 10.5.

10.3 Basic Techniques for Post-Mortem Abstraction

10.3.1 Abstraction Along Time-Axis

The post-mortem abstraction techniques can be divided into three groups according to the two dimensions and their combination. These abstraction techniques are always carried out in the visualization of the event-graph, where we use the horizontal axis as the time axis and the vertical axis as the process axis. Consequently, abstraction across time-axis may also be called horizontal abstraction techniques. In our classification we have included the following two examples:

- Loop detection
- Collapsing replicated events

Loop detection operates on the displayed events. It tries to remove repeated segments from the space-time diagram, which occur when programs compute their data within loops. Since loops are quite common in many programs, the potential benefits of this method are rather high. With loop detection, the space-time diagram contains only one iteration, instead of displaying each consecutive iteration. An example where loop detection exhibits its benefits is the finite element solver shown in Figure 10-1, because only one of the 200 observed iterations will be displayed. Consequently, the displayed space-time diagram will be less complicated and smaller in size.

Besides the possibility to remove loops from the display, this method also allows to detect deviations occuring in some loops. Since only identical loops will be removed from the display, even minor discrepancies will be revealed. This allows the user to notice incorrect itera-

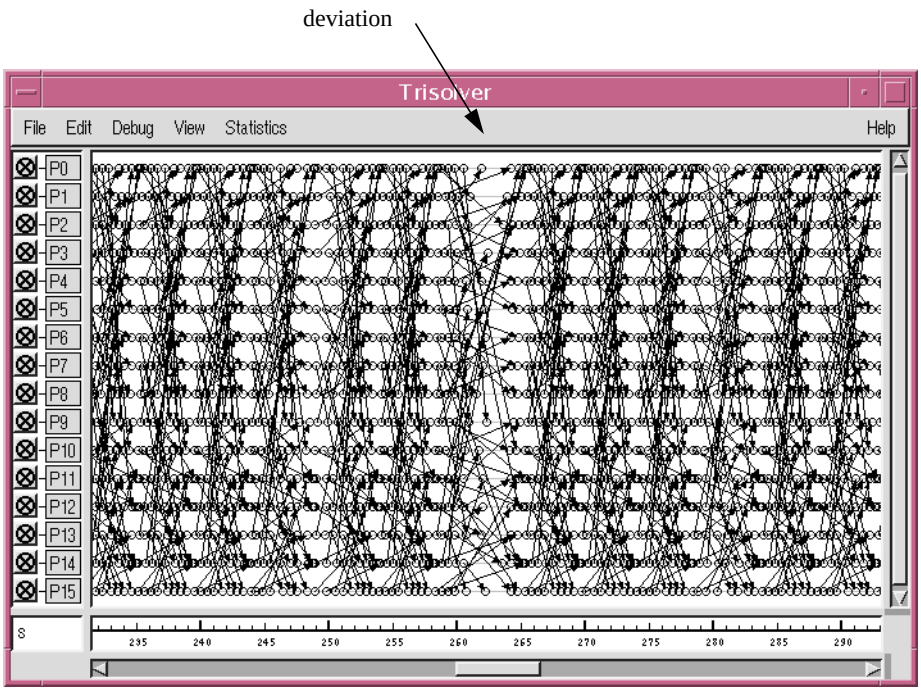


Figure 10-2. Detecting loop deviations

tions immediately which may be overlooked otherwise. An example is given in Figure 10-2, again with the parallel finite element solver of Figure 10-1 and its 200 iterations, but this time only on 16 processes. By extracting one iteration and comparing it with all other iterations, anomalous behavior have been detected at one of the iterations. Please note, that in contrast to all other examples, the event graph in Figure 10-2 is displayed with measured time stamps to display the effect more easily.

Implementing a loop detection mechanism may be performed with one of two methods. On the one hand, the event graph model can be investigated with pattern matching algorithms (see Section 10.4). This method tries to identify loops by investigating the connections between the events. On the other hand, it may be even easier to extract loop behavior by investigating the event attributes, especially the source code connection. In that case, it is sufficient to traverse through the graph once and analyze the sequence of source code lines for each process. The differences between these two methods are, that the pattern matching method may reveal even those loops, that are not explicitly specified in the source code.

The second horizontal abstraction mechanism, collapsing of replicated events, follows the same idea as loop detection. It tries to compress events with equal attributes to build concatenated events. A straight-forward example for applying this method is given for nonblocking receive events. These so-called test events are produced at statements, which probe the input buffer for incoming messages (see Section 6.2.2). Abstracting of consecutive test events is useful, if many unsuccessful probes have been carried out successively. Then only one object has to be produced for each series of unsuccessful returns. The advantage of this collapsing mechanism is the fact that the user usually doesn't need to see all the events, but only the existence of more than one successive events and a number indicating the replication factor.

Implementing a collapsing mechanism is similar to the loop detection mechanism. In our approach, two different kinds of graphical symbols - in this case squares - are used for such events. For example, single, large squares (green) indicate each successful test event, which

means that a message with the specified parameters is available in the message buffer, while small (red) squares define unsuccessful test events. The visualization of several consecutive, unsuccessful nonblocking receive events is done with a set of three small squares instead of one square for each unsuccessful event. When inspecting these three events, the number on unsuccessful tries will be added to the event information window. Please note, that nonblocking receive events are only one possible example for removing replicated events. Certainly, the same technique may be applicable to other types of events as well.

10.3.2 Abstraction along Process-Axis

In contrast to the abstraction mechanisms along time-axis, the mechanisms operating on the process axis are called vertical abstraction mechanisms. These mechanisms are intended to decrease the complexity of the event graph in terms of numbers of processes and interprocess dependencies. Again, the abstraction is applied post-mortem on the displayed space-time diagram. Two distinct ideas have been investigated:

- Process grouping
- Process hiding

Process grouping follows the idea, that parallel programs often assign groups of processes working together on a particular subproblem, and the processes within these groups perform similar activities. An example is an algorithm, that uses several master processes, and each master process employs several slave processes. If the master-slave relation is the same for all the processes, it is not necessary to analyze all processes together. Instead it might be sufficient to analyze only relations between the masters (intergroup communication) and relations between one master and his slave nodes (intragroup communication). In addition, grouping of processes can be performed hierarchically, which increases flexibility, because errors can be detected errors on higher levels of abstraction, before following them down to the actual faulty processes.

An example for an algorithm that uses groups is given in Figure 10-3. The displayed event graph shows a small fraction of a parallel raytracing algorithm. This algorithm assigns parts of the computational work to different groups of processes, in this case two groups with four processes each (0-3, 4-7). With a group detection mechanism, the graph can be transformed in several other graphs. Firstly, it would be possible to display only intergroup communication, where every communication that takes place between any members of different groups is displayed, while messages between processes of the same group are removed. Secondly, only intragroup communication within one process group could be displayed. Thirdly, a mixture of both group structures could be useful, if a graph is constructed that contains the intragroup communication of one group together with its intergroup communication to the other groups.

Another vertical abstraction mechanism is process hiding, which reduces the complexity of the event graph by removing all processes that are uninteresting for certain investigations. Only the remaining processes are used for further analysis. Consequently, process hiding is also often called process extraction, because only interesting processes are extracted.

In the current implementation of our debugging tool, there are four different modes, how processes can be hidden and communication between these processes is displayed. The mode selection is done manually with the symbols in front of the process identification field. Possible hiding methods are:

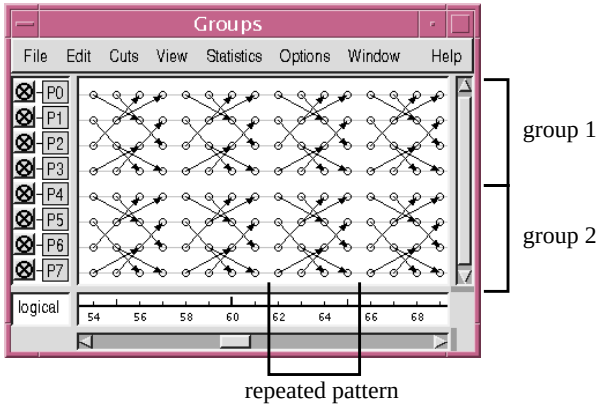


Figure 10-3. Process grouping

- Visualize a process (default mode): The process and all its communication (send and receive messages) is displayed, as long as the partner process is not hidden.
- Visualize only events, if partner process is completely available (mode above): Communication between events with this mode enabled is not displayed.
- Hide events and only suggest communication: The events are not displayed and communication to these events is visualized to or from the upper window edge.
- Hide processes: The processes and their events are completely hidden.

An example is visualized in Figure 10-4, where process grouping is applied on the event graph displayed in Figure 10-2. The displayed event graph contains only 5 processes, and only process P7 is completely visualized. On the other processes, only those events are visualized that establish connections with process P7. The remaining processes, that do not communicate with the completely visualized process P7, are removed from the display and are therefore invisible. Comparing Figure 10-2 and 10-4 reveals the advantages of this method, since only 5 processes are displayed instead of 16.

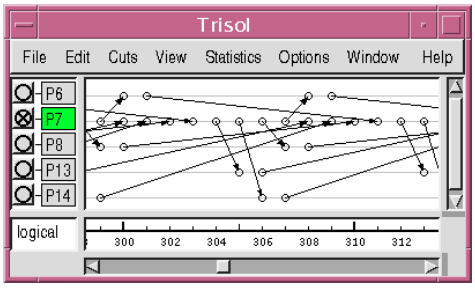


Figure 10-4. Process extraction and hiding

The process hiding method described so far is applied manually by the user, because only the user knows which processes are interesting. A possibility to perform process hiding and extraction automatically is, if only one of the processes contains an error. In this case, it can be assumed that this erroneous process contains the most important data, and the user's attention should be directed to this critical process. Besides that, it seems feasible to display only processes, which are actually communicating with the selected process, and which may therefore be responsible for incorrect behavior.

10.3.3 Combined Post-Mortem Abstraction

The approaches described in this section operates post-mortem on both axes of the event graph display. For that reason, some of these techniques are extensions or combinations of the horizontal and vertical abstraction mechanisms described before. Some example combined abstraction mechanisms are:

- Zooming/scaling
- Filtering
- Error extraction
- Pattern matching

Zooming and scaling are the most simplest approaches to modify the displayed event graph data. In fact, for these two operations the term abstraction may actually be misleading, because their modifications concern only the displayed space-time diagram. Zooming means, that the scale of the space-time diagram is increased or reduced to emphasize on certain details or to obtain a coarse overview. Scaling is similar to zooming, but operates on the given time or process scale. For example, the number of pixels per logical clock display may be modified or the given time scale may be changed. Besides these basically visual operations, there are other techniques e.g. scrolling that help to navigate through the displayed data.

An example, where these simple methods provide some clues for the user is given with the finite element solver of Figure 10-1 and Figure 10-2. Both event graphs contain the same program, but the loops visible in Figure 10-2 are not evident in Figure 10-1. However, one difference between both figures is the chosen time scale. In Figure 10-2 there are many more events displayed per interval than in Figure 10-1. Thus, the resulting event graph visualization contains a bigger fraction of the program's execution, and the loops can be spotted by the user.

Another traditional abstraction technique is filtering, which tries to reduce the displayed data by focusing on specific subsets as provided by filters. Filtering means, that all but a designated subset of events from a program are deleted, which requires a mechanism to filter out irrelevant events in the formation of a higher level event, and thus also reduces the amount of displayed information [Cai 93]. Therefore, filtering is an intrinsic feature of the event graph, because only events of interest are included for the user.

Collapsing of replicated events and process grouping are possible examples for filtering on the horizontal or vertical axis, respectively. More advanced filtering techniques would allow to focus on certain types of events or on particular event attributes. For example, users may choose to display only events with a given message tag, which may then reduce the data of the event graph on both axes simultaneously.

An extension of the vertical abstraction mechanism process hiding is error extraction, which can be performed automatically by the debugging tool. Again, the goal is to direct the users attention to places of incorrect behavior in the event graph. With error extraction, only a limited surrounding of the detected erroneous events is extracted. A simplified operation of automatic error extraction is to evaluate the history of events that finally resulted in the detected bug. Therefore, it may only be necessary to display the communication partners of the corresponding process, and this only for a certain interval of time.

An example for error extracting would be to isolate a runtime-failure (e.g. division by zero), that occurred sometimes during a program run- As soon as this happens, the debugger initiates failure detection as described above. The error on process is isolated, and only processes that

communicated with process in its most-recent history are considered. Based on this history buffer, the detected error and its surroundings is displayed.

10.4 Pattern Matching

10.4.1 Overview and Principal Idea

One of the most sophisticated ways of event graph analysis considers the communication patterns of the graph itself. This technique also operates post-mortem and simultaneously in both dimensions of the event graph. The basic idea of pattern matching is, that often, a set of corresponding send and receive events resembles more or less complex communication patterns, like broadcast, scatter, gather, and butterfly [Kran 96b]. These characteristics can be detected automatically with simple pattern matching algorithms, and it seems sufficient to visualize detected patterns in compressed form (with a dedicated symbol) or completely remove them from the display [Kran 96b].

Besides these automatic pattern matching approach, it seems also useful to integrate the debugging user's knowledge into the process. The concrete idea is, that the user specifies expected behavior of the program by defining communication patterns which should be visible in the event graph display. These patterns can be checked automatically and deliver one of the following results:

- A pattern was detected
- A pattern was not detected
- An almost complete pattern was detected

As an extension of the automatic pattern matching approach for massively parallel programs, the expected and unexpected patterns are weighted according to their importance. Again, this allows to direct the user's attention to the most important places in the event graph, and leave less important places for latter investigations. At present, the weight of a pattern is simply indicated by a number, with higher numbers meaning higher priority in the pattern checking algorithm.

10.4.2 Specification of Patterns

In our approach, specifying the patterns can be done with PatternTool [Grub 96], an interactive tool for creating scalable hierarchical graphs. The original intention of PatternTool was to assist programmers during Computer Aided Parallel Software Engineering (CAPSE) by using task graphs to model algorithms in terms of scalable patterns. Although this goal is not exactly the same as targeted by our approach, PatternTool has been applied successfully. Of course, it also contains some features as well as predefined patterns, which are useless for debugging. One of our future goals is therefore to adapt and modify PatternTool in order to provide the necessary functionality and extensions required only for parallel program debugging. Besides that, we will investigate graph specification languages, that may be much more powerful.

A main characteristic of patterns is scalability, because patterns must be applicable for different numbers of processes. Therefore, we distinguish four items during the definition of patterns:

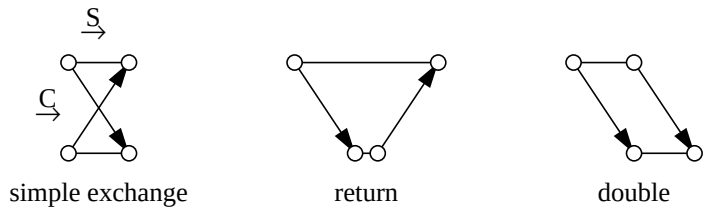


Figure 10-5. Basic event graph patterns

- Basic patterns
- Selection operations
- Relational operations
- Macro nodes

Basic patterns are the starting point for every pattern definition. These patterns are already scalable and define vertices and corresponding edges with relative number, relative position, and relative index. The relative values are used as variables, which will be filled as soon as the number of participating processes has been defined. Therefore, the parameterization defines the scale of the pattern, which should be applied during pattern matching.

For the user's convenience, the tool already provides some basic patterns as shown in the examples of Figure 10-5, where the number of participating processes is always 2. More complicated basic patterns are round robin, binary and arbitrary trees, and butterfly networks [Kran 96b] (see examples in Figure 10-6).

The left-most pattern of Figure 10-5 is called a *simple exchange* pattern, which describes the situation, where two arbitrary processes mutually exchange some kind of data item. Using our event graph formalism, we can define this kind of pattern as follows:

Definition 10-1: Simple exchange pattern

Given two events $e_p^i, e_p^{i+1} \in E$ on process p and two events $e_q^i, e_q^{i+1} \in E$ on process q with $q \neq p$, then the graph

$$\mathrm{EX}(i, p, q) = (\mathrm{EX}_{\mathrm{ev}}(i, p, q), \mathrm{EX}_{\mathrm{rel}}(i, p, q))$$

$$\text{with } \mathrm{EX}_{\mathrm{ev}}(i, p, q) = \{e_p^i, e_p^{i+1}, e_q^i, e_q^{i+1}\} \text{ and}$$

$$\mathrm{EX}_{\mathrm{rel}}(i, p, q) = \{\left(e_p^i \xrightarrow{S} e_p^{i+1}\right), \left(e_q^i \xrightarrow{S} e_q^{i+1}\right), \left(e_p^i \xrightarrow{C} e_q^{i+1}\right), \left(e_q^i \xrightarrow{C} e_p^{i+1}\right)\}$$

is called a *simple exchange* pattern.

As shown in Definition 10-1, the simple exchange pattern is an event graph consisting of four events, which are connected either by the sequential order relation $\xrightarrow{S}$ (if placed on the same process) or the concurrent order relation $\xrightarrow{C}$ (if placed on distinct processes).

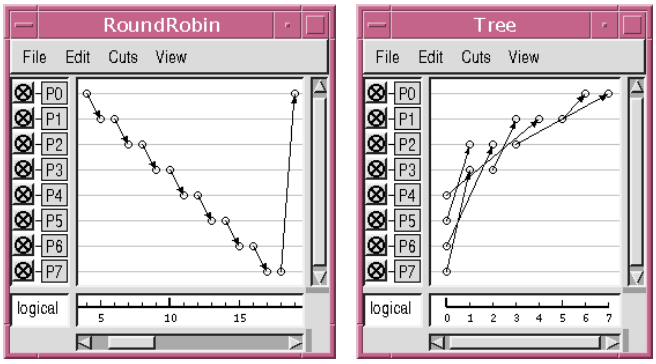


Figure 10-6. Simple event patterns (round robin, tree)

Remark

The usage of event index *i* in Definition 10-1 for the first event on both processes, *p* and *q*, represents a minor simplification of the exchange pattern. In reality, it is not necessary that the first event of a pattern on each process is identified by the same event index, e.g. when the number of preceding elements is different on distinct processes. However, adding a second index to Definition 10-1 is rather trivial, but is left out for readability. We will apply the same simplification to the two following Definitions 10-2 and 10-3 below.

Based on such basic patterns, selection operations and relations are defined. Selection operations allow to identify groups of events, that are accumulated for particular operations. For example, the right-most pattern in Figure 10-5 may identify all send events as one group of events, with a special parameter defining the number of occurrences. In addition, relational operations allow to access vertices by their relative position to other vertices of the graph. For example, the position of a vertex in the event graph is always defined by its index. Thus, it is possible to identify the previous and next event.

With these basic patterns and the defined operations, more complex patterns can be implemented with macro nodes. A macro node is a collection of possibly different event graph patterns, that allows to construct arbitrary complex patterns for any imaginable algorithm. As an example, we will describe the pattern of the hypercube data circulation algorithm as described by Dekel at al in [Deke 81]. The algorithm has already been introduced in Example 9-2 on page 231, and is included in the code of Appendix D. An example of the pattern revealed by this algorithm and its connection to the simple exchange pattern described in Definition 10-1 is given in Figure 10-7. As can be seen in this pattern, every value of every participating process is shuffled between all available processes.

An important characteristic of the hypercube data circulation algorithm is, that it performs data exchanges only between neighboring processes. For that reason, the nodes of an arbitrary hypercube are usually numbered according to the Gray code scheme [Haye 88]. In a hypercube, two nodes are adjacent, if the binary representation of their node addresses differs in exactly one bit, which is equivalent to a Hamming distance of 1. Consequently, two nodes can communicate along dimension *d*, if their binary addresses differ exactly in bit *d*.

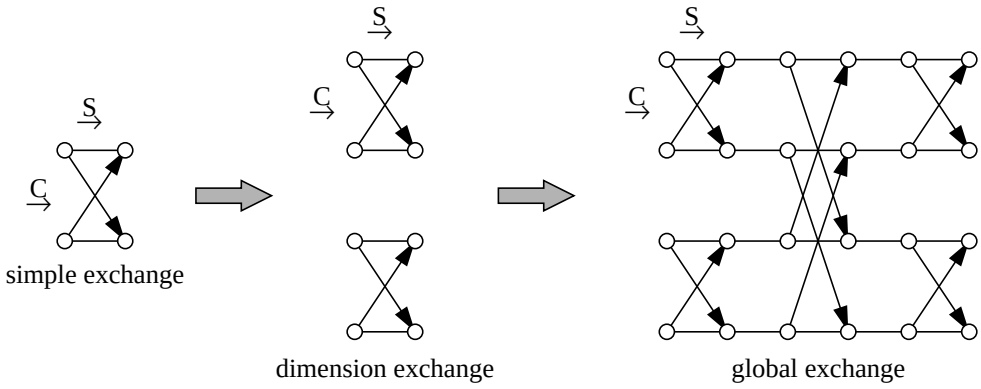


Figure 10-7. Scalable patterns

Based on this characteristic, every process performs a similar data exchange during each cycle of the Dekel's data circulation algorithm. For simplification, we can define the exchanges along one dimension of the hypercube during one cycle of the data circulation as follows:

Definition 10-2: Dimension exchange pattern

Given the event index i of event $e_p^i \in E$ on process p and the total number of processes n with hypercube dimension d where $n = 2^d$, then the graph

$$\mathrm{DEX}(i, d') = (\mathrm{DEX}_{\mathrm{ev}}(i, d'), \mathrm{DEX}_{\mathrm{rel}}(i, d')) \text{ with}$$

$$\mathrm{DEX}_{\mathrm{ev}}(i, d') = \{ e_q^{i+j} \mid \forall (0 \le j < 2, 0 \le q < n) \} = \bigcup_{0 \le p, q < n} \mathrm{EX}_{\mathrm{ev}}(i, p, q)$$

$$\text{and } \mathrm{DEX}_{\mathrm{rel}}(i, d') = \bigcup_{0 \le p < n} \mathrm{EX}_{\mathrm{rel}}\left(i, p, p \oplus 2^{d'}\right)$$

is called a *dimension exchange* pattern along dimension d' with $0 \le d' < d$.

The communication between neighboring processes in a hypercube is established by the communication partners p and $p \oplus 2^{d'}$, where $\oplus$ is the exclusive-or operation performed on the binary representation of the address of process p. In concrete, $p \oplus 2^{d'}$ means that bit d' of the address of process p is inverted, revealing the address of the required communication partner.

During data circulation, that dimension exchange pattern is iteratively applied as specified with the recursive exchange sequence of Definition 9-1 on page 232. Applying the same rules with our event graph formalism, we can obtain a global exchange pattern for the data circulation algorithm as follows:

Definition 10-3: Global exchange pattern

Given the event index i of event $e_p^i \in E$ on process p and the total number of processes n with hypercube dimension d where $n = 2^d$, then the graph

$$\mathrm{GEX}(i, d) = (\mathrm{GEX}_{\mathrm{ev}}(i, d), \mathrm{GEX}_{\mathrm{rel}}(i, d)) \text{ with}$$

$$\mathrm{GEX}_{\mathrm{ev}}(i, d) = \{e_p^{i+j} \mid \forall\left(0 \le j < 2^{d+1} - 2, 0 \le p < n\right)\} \text{ and}$$

$$\mathrm{GEX}_{\mathrm{rel}}(i, 1) = \mathrm{DEX}_{\mathrm{rel}}(i, 0)$$

$$\begin{aligned}\mathrm{GEX}_{\mathrm{rel}}(i, d) = {} & \mathrm{GEX}_{\mathrm{rel}}(i, d-1) \cup \\ & \{\left(e_p^{i+2^{d-1}-1} \xrightarrow{S} e_p^{i+2^d}\right) \mid \forall (0 \le p < n)\} \cup \\ & \mathrm{DEX}_{\mathrm{rel}}\left(i + 2^{d-1}, d-1\right) \cup \\ & \{\left(e_p^{i+2^{d-1}+1} \xrightarrow{S} e_p^{i+2^{d-1}+2}\right) \mid \forall (0 \le p < n)\} \cup \\ & \mathrm{GEX}_{\mathrm{rel}}\left(i + 2^{d-1} + 2, d-1\right)\end{aligned}$$

is called a *global exchange* pattern.

An example space-time diagram for such a complex global exchange pattern is shown in Figure 10-8, where 8 processes are applied in a template matching algorithm. Other occurrences of this global exchange pattern can be detected in many parallel programs. In addition, the defined pattern is often applied several times throughout the algorithm.

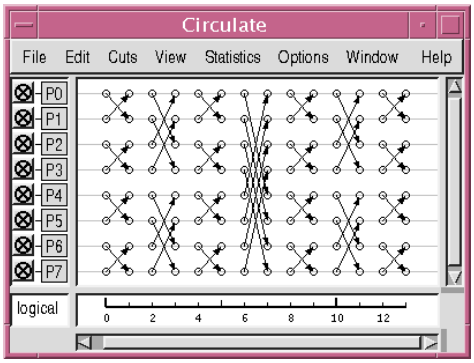


Figure 10-8. More complex pattern

10.4.3 Future Perspectives and Benefits

All the patterns that have ever been defined with PatternTool (or its successor) are collected in a standard pattern library. Of course, if loops are detected, one iteration of a loop is also a good candidate for this database. Therefore the database is consistently extended with newly detected patterns from various fields of parallel programming. By adding new patterns to the database and providing this information for future users, the task of specifying patterns is decreased significantly. The consequences are, that the user only has to detect the expected pattern in the database, and modify its weight factor to indicate the corresponding importance for the target algorithm.

A side effect of the pattern definition is, that an appropriate pattern matching algorithm can detect almost complete patterns. Again, this is especially complicated for large patterns on many processes. While users probably overlook small inconsistencies of event graph patterns, the matching algorithm will detect anomalies and attract the user's attention. With this simple technique, many errors occurring at communication statements due to minor programming errors can easily be detected.

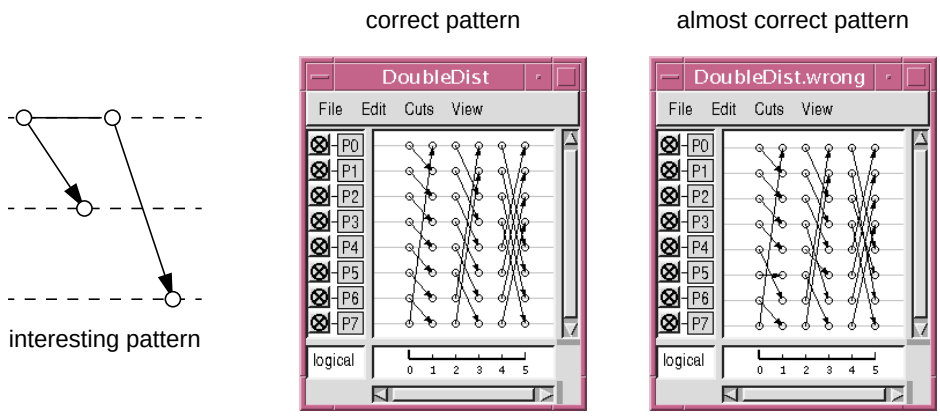


Figure 10-9. Detecting almost correct patterns

An example is given in Figure 10-9, where the existence of an interesting pattern in an event graph is evaluated. The pattern under consideration is displayed in the left diagram. In the pattern database, it would probably be parameterized for the number of processes and the number of iterations. The event graph containing only the correct pattern is displayed in the middle diagram, while an incorrect pattern with only a minor deviation is given in the right diagram. The deviation occurs on process P5 and process P6.

10.5 On-line Abstraction Technique

10.5.1 On-line Abstraction Along Time-Axis

In contrast to the post-mortem abstraction techniques discussed so far, which operate on trace data after a program's execution has terminated, on-line abstraction techniques are applied while the program is still running. Therefore, the target of the abstraction is shifted from the data processed for generating the event graph display to the program being executed. Conse-

quently, applying on-line abstraction means to reduce the execution time of a program, or the number of processes being debugged, or both.

A reduction of the execution time operates along the time axis of the event graph. Our idea in this area focuses on integrating checkpointing and debugging. The idea of checkpointing is to periodically save the states of a program in order to allow restarts at any time of the execution, in case the execution fails for some reason. This shortens a programs requirements to computation resources, because it needs not be restarted all over again from the beginning. Hence, massively parallel programs running a long time are good candidates for checkpointing, and some kind of checkpointing mechanism is available in most cases.

Checkpointing itself, often called checkpoint and rollback recovery [BoPr 93], is a large research area and consequently many different approaches have been proposed, mostly associated with fault-tolerant computing [ChRa 72]. Besides the principal problem to store a program's state during execution by saving the memory contents and the processors state to a stable storage device, two main issues are the problems of determining a consistent global state and the amount of data needed for a set of checkpoints. The former is often achieved with distributed snapshots as described in [ChLa 85]. However, since our cuts already represent a consistent global state established on the communication structure of a message passing program (see Definition 7-5 on page 188), it seems useful to apply the cut mechanism for checkpointing. Similarly the second problem of checkpointing, how to cope with the required storage consumption, which is addressed by the incremental checkpointing approach described in [Plan 95] may benefit from the event relations described with the event graph model.

The advantage obtained by combining some generic checkpointing algorithm with a debugging strategy is, that the program is cut at consistent places during the execution into arbitrary regions of time, which may be termed time slices. Therefore, the slice between two checkpoints of a program qualifies as an independent execution, which may be analyzed during debugging without connection to the other time slices. As a benefit, the debugger does not need to focus on the complete execution, but only on a small part of it, which means a big reduction in the amount of event data to be processed. Additionally, program re-execution with a debugger seems feasible, because it can be initiated at the last checkpoint. An example for a debugging session on *n* processes is given in Figure 10-10. It shows, that the program observation starts at the last checkpoint and gathers data for the analysis until the next checkpoint is reached.

An additional improvement can be obtained by distinguishing between a process' recent history and its former past. On the one hand, on-line failure detection does not require all event data since the last checkpoint, but only its recent past. In order to achieve this, the debugging tool provides a user-defined history buffer, which establishes an inspection region as displayed in Figure 10-10. Within this inspection region, event attributes are available for inspection, in case a program failure occurs. Thus, the debugger has the possibility to analyze the program's recent history during on-line failure detection.

On the other hand, the record&replay mechanism necessary for analysis of nondeterministic programs requires information about event ordering to perform program re-execution starting at the last checkpoint. However, the necessary data for replay is far less than required during inspection (see Section 9.2.4). Therefore, it is possible to filter the analysis data as soon as it drops from the inspection region, and store only ordering information in some kind of replay buffer (see Figure 10-10). Yet, while the inspection region can be implemented as a FIFO queue, the replay buffer must preserve enough memory for the event ordering information. This is not a problem, if compression algorithms as discussed in Section 9.2.4 are integrated in the debugging strategy.

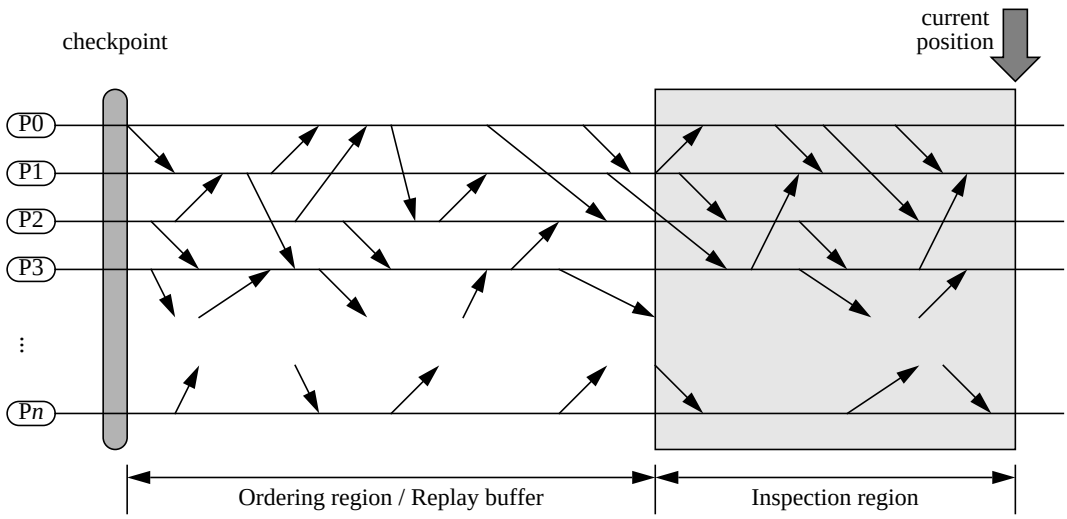


Figure 10-10. Debugging and checkpointing

With the availability of the replay buffer, it is then possible to re-execute the program from the last checkpoint. Since the checkpoint slice represents an stand-alone execution, additional activities like nondeterminism analysis with event manipulation as described in Chapter 8 can be performed. While it would be impossible to steer a complete long-running program, it is certainly feasible for small checkpoint slices.

In addition, the checkpointing mechanism can benefit from the replay technique. In conventual approaches, the checkpointing mechanism may vigorously influence the execution of the program, since storing the checkpoint data requires a certain amount of computing power and I/O bandwidth. As a consequence, the execution of a program after each checkpoint may be disturbed by the checkpointing mechanism comparable to the probe effect (see Section 4.2.8). This problem can be addressed by the combination of the event manipulation approach and the artificial replay in the same way, as in program executions without checkpointing.

10.5.2 On-Line Abstraction Along Process-Axis

Similar to the on-line abstraction techniques along time-axis, on-line abstraction along process-axis can be applied. In this case, the abstraction reduces the number of processes applied during execution. Instead of debugging all the processes of a program, only a small set of processes or even only one selected process is used for debugging. An example for this so-called process isolation is shown in Figure 10-11. There, a process *Px* is selected from the complete set of processes and debugging is carried out only on process *Px*.

With such an abstraction mechanism, most sequential debugging techniques could be adopted for parallel program debugging. Although this seems very promising, it raises some problems for which certain precautions have to be taken: Firstly, the behavior of the system

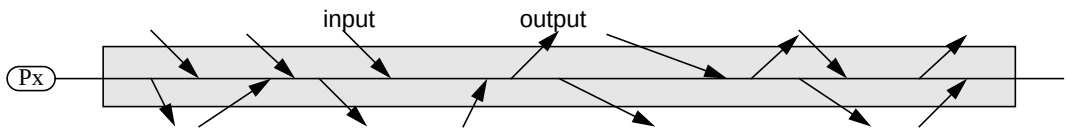


Figure 10-11. Process isolation

surrounding process *Px* must be simulated, which may be rather difficult in some cases [McHe 89]. Yet, it is not necessary to simulate the complete program, but only the behavior affecting the execution of process *Px*. Secondly, the process under investigation may still reveal nondeterministic behavior and may affect other processes behavior. Thus, a record&replay method is needed for process execution. In concrete, we propose the following four-step approach for debugging isolated processes:

(1) Execute and monitor the whole program on its intended scale and with the corresponding amount of data to generate traces sufficient for equivalent execution.
(2) Select process *Px* for debugging (probably from the event graph display).
(3) Re-execute complete program and monitor surroundings of process *Px*, that affect process *Px* and that are affected by process *Px*.
(4) Perform debugging on process *Px* and simulate external behavior with the data obtained in (3).

The problem of this approach is that the program must be executed at least two times before process isolation can be applied. The first execution is needed to observe the behavior of the complete program and guarantee equivalent execution. The second execution is needed to trace the surroundings of the process under consideration. During this execution the monitor switches from event-driven monitoring to data-driven monitoring [LeMe 87]. Instead of recording only the occurrence of a communication event, the actual contents of the message have to be recorded - at least at receive events. Systems performing data-driven monitoring are described in [CuWi 82], [LeRo 85], and [Smit 84]. Obviously, the second execution can be neglected, if the surroundings of all processes are captured. Yet, this is certainly infeasible in the general case due to the amount of observation data.

A possible idea to improve the problem of simulating the target processes with the observation data has been described by Volkert[1], which assumes that the program under consideration has been composed according to the SPMD programming model. In that case, each process executes the same code and most processes will perform similar activities. Consequently, if a process performs pair-wise communication operations (e.g. when exchanging borders), the receive operation can be fed with the data provided to the send operation. In this case, only those data items that are not processed by the process itself need to be traced during step (3) of our process isolation approach.

Another technique included in the classification of abstraction mechanisms as specified in Table 10-1 is the combination of both on-line abstraction mechanisms. This can easily be achieved by integrating the checkpointing approach described in Section 10.5.1 with the process isolation approach described above without any additional problems. The resulting technique would allow to debug only small time slices of the total execution time on a small subset of processes, while still analyzing the applications execution on its original scale. An example for this combined approach is given in Figure 10-12, where a single process *Px* is being debugged between two arbitrary checkpoints.

Consequently, it may also be interesting to investigate combinations of on-line abstraction mechanisms with post-mortem abstraction mechanisms. For instance, applying pattern match-

1. This idea for debugging isolated processes has been discussed in an informal way. It has therefore never been described in any paper, because publication would need some more in-depth analysis of the feasibility of this approach. However, this approach seems rather promising in context with the problems described above and has therefore been included in this work.

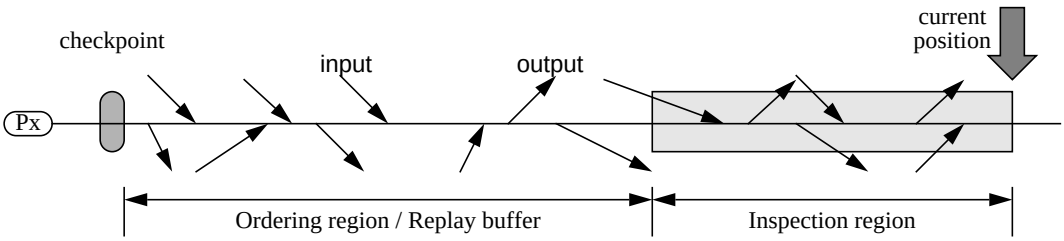


Figure 10-12. Combined on-line abstraction

ing as described in Section 10.4 during monitoring may help to reduce the amount of generated trace data, when patterns are stored instead of each occurring event. Many other ideas for future work in that area are imaginable. However, we always need to keep in mind, that on-line abstraction mechanisms, however useful they are, tend to perturb the program's execution and raise the probability of malign probe effects much more than those abstraction mechanisms, that are applied post-mortem.

10.6 Summary

The advanced techniques described in this chapter address the problems associated with the amount of debugging data generated by massively parallel programs. Included in our event graph abstraction mechanisms are all kinds of methods, like reduction, transformation, extraction, isolation, and filtering, which try to raise the level of abstraction by reducing the number of processes and the amount of time included in the event graph model.

The principal idea of this approach is to focus only on the most important aspects of a program's behavioral data, which can be compared to the traditional program slicing idea based on the data flow and control flow of a program. The main benefits of these abstraction techniques are, that the data and the associated complexity can be reduced, and the user's understanding for program behavior can be improved. We try to offer a classification scheme based on the main characteristics of these approaches, with some examples for each group. Besides basic abstraction techniques we describe our pattern matching approach and a small set of interesting on-line abstraction techniques in more detail.

This chapter can only give a very brief overview of the perspectives offered by the abstract event graph model. Some of the described features have actually been implemented and evaluated, as shown in the example screenshots and the cited references. Other ideas (e.g. the on-line abstraction techniques) do only exist as theoretical approaches and remain therefore open for discussion as well as for our future work in this area.

11. Summary and Conclusions

Massively parallel computing is considered as the most powerful high performance computing technology applied to solve problems of Grand Challenge scale. Corresponding applications try to achieve high levels of performance by exploiting large numbers of processes and huge amounts of memory provided by these computing architectures. Yet, developing efficient applications is rather difficult and can only succeed, if appropriate software engineering techniques and corresponding tool support is provided. In fact, writing a parallel program and getting it to work properly can be a significant intellectual challenge.

An important phase of the software life-cycle is testing and debugging, which determines an applications quality in terms of correctness and reliability. In this work we have concentrated on testing and debugging of parallel applications, which suffers from severe problems like the multiplicity of sequential bugs and the existence of additional anomalous effects. This emphasizes the demand for suitable techniques and strategies, that support users during error detection and correction. For that reason, we have formally described the abstract event graph model, which tries to fill this gap.

The event graph model represents the state changes of arbitrary parallel programs by describing events and their interactions occurring on currently executing processes. The applied relation is the happened before relation, which is established either through sequential program flow or corresponding communication and synchronization operations. It expresses not only temporal characteristics of events but also their causal relations.

A visual representation of the event graph can easily be constructed as a space-time diagram, displaying the time on one axis and the available processes on the other axis. This rather natural way of describing a parallel program's behavior serves as our main interface to the user, where all activities of the testing and debugging phase are applied. We believe that our graphical representation with its level of abstraction is better suited for parallel program debugging than the textual representations of other traditional approaches, because text is often inappropriate to reveal the complex relationships established among concurrently executing processes.

The construction of the event graph model is based on an observation of the target program's execution. Consequently, the observation must be tailored to a particular class of programs and the set of events occurring in this programs. We have chosen parallel programs applying the message passing paradigm, because this programming model is currently the most important one for massively parallel architectures. However, it is certainly feasible to use the abstract event graph model for other classes of programs, like the popular shared variable approach, as well.

Analyzing the program's behavior for error detection is based on investigating the shape of the event graph and inspecting the event attributes associated with occurring states and state changes. Besides the usage of the event graph model for these simple analysis and inspection tasks, we have also adapted some traditional debugging techniques as well as integrated unique new features. The resulting debugging strategy can be used to perform many different debugging activities, like automatic error detection and consistent breakpointing.

One of our main points of interest are nondeterministic parallel programs, which yield different program behavior even if the same input data are provided. The principal characteristics of nondeterministic parallel programs and their crucial consequences during debugging, for example annoying effects like the irreproducibility problem, the completeness problem, and the probe effect, have been described in more detail. These problems are covered with dedi-

cated techniques based on the event graph model, like our record&replay approach for equivalent re-execution. Additionally we have integrated our sophisticated event manipulation and artificial replay mechanisms for steering the nondeterminism in parallel program executions.

The basic event graph model is also a starting point for several on-going research topics. Two issues of major importance - automatization and abstraction - have been briefly described. An example for automatization is our automatic nondeterminism analysis technique, which can be used to solve the completeness problem. With this technique every possible execution of an arbitrary nondeterministic program can be derived and tested. Since the practical application of this approach is restricted by the available computing power and storage capacity, some ideas for reducing the number of test cases have been described. One of these ideas tries to apply our techniques for solving the probe effect, which occurs when parallel programs are observed.

The abstraction techniques integrated in our debugging approach operate on different levels. On the one hand, the event graph model of a particular program represents an abstraction itself, since it focuses only on selected events of interest. On the other hand, the event graph serves as the basis for additional more advanced abstraction techniques, which try to cope with the large amounts of debugging data generated by massively parallel programs and to improve the user's understanding for huge and complex program executions. Some interesting ideas for extended abstraction techniques are pattern matching and combined on-line abstraction.

The main contribution of this work for the field of parallel debugging comes from the universal character of the event graph, not only as the main interface to the user but also for all possible functions of error detection, even though only a subset of features has been explicitly described. Although there exists some proposals for using graph-like structures for a variety of analysis techniques in this area (e.g. the communication graph, or the diversity of filtering and abstractions mechanisms), there has never been a model with a comparable formalism throughout all the different kinds of debugging activities.

Besides this formal approach to parallel program debugging, we have also investigated and developed several techniques, that may be useful in other approaches. For monitoring of parallel programs we have shown that instrumentation with the macros allows to generate source code information without accessing system-dependent symbol tables. Additionally, we have developed a logical clock ordering mechanism, that is much simpler than vector clocks and generates less monitoring overhead in time and space. Afterwards, the partial ordering can be generated from our logical clock ordering without influencing the program.

For the event graph analysis, we have described two different kinds of consistent cut mechanisms, which guarantee that a breakpoint event is reached on each affected process. One of this mechanisms allows to investigate the program comparable to a traditional breakpointing approach. The other one is needed for event manipulation, where the artificial replay mechanism needs to distinguish between those parts of the program that need to be executed equivalent to the initial program run, and those parts of the program that are possibly influenced by the event manipulation.

During this work on nondeterminism analysis, we have investigated different kinds of nondeterministic events like blocking and nonblocking nondeterministic receives and the random number generator. For the messages racing towards nondeterministic receives we have revealed some important features related to the non-overtaking characteristic inherent to widely used message passing interfaces. These features allow to decrease the number of racing messages and therefore reduce the amount of required testing.

For all these different ideas we have implemented dedicated tool prototypes or performed the necessary experiments to verify our assumptions. Many of these techniques have also been

integrated in our debugging environment, which can be applied freely by users around the world. Some open issues, mainly those that are described in the future work chapter, are currently being implemented by our diploma thesis students or are waiting for someone interested.

In this respect, we may also remark that the relevance and impact of this work is still increasing, although we probably may need to change our programming model (probably to shared variable approaches) or the underlying hardware architecture (probably to distributed or cluster environments). It is accepted as a fact, that parallelism is moving into everyday applications, even though some critiques said and still proclaim, that parallelism is on a dead-end road and will die someday due to the fact of ever increasing computing power of sequential machines, it is moving steadily into all areas of computing. Modern graphics cards can only provide their processing speed due to parallelism. Processors like the Pentium II with its MMX extension provide vector codes and pipelining to increase the processing power in personal computers. Dual-Pentium processors are increasing in sales volume and may most likely replace the current desktop machines in the near future.

In the high performance computing domain, lots of efforts are currently spent on clustering and metacomputing infrastructures. Their idea is to couple supercomputing power with high-speed network connections to achieve even higher levels of computational performance. Now imagine the software development and event management problems occurring in these computational grids, where pieces of code may be running on systems scattered across the country, and those systems themselves may be running many thousands of different problems across the entire grid that have absolutely no relationship to one another. Dealing with such complex systems on this scale - with the number of resources, heterogeneous platforms, geographic distances, scope, and diversity of use - leads quickly into uncharted territory, particularly also in the field of parallel program debugging. We are looking forward to these developments and their impact on mankind and hope to offer possibilities to cope with the errors occurring in these systems.

12. References

[Abra 95] Abrams, M., Lee, T.J., Cadiz, H., Ganugapati, K., “Beyond Software Performance Visualization”, in: Concurrency: Practice and Experience, John Wiley & Sons, Vol. 7, No. 8, pp. 737-764 (December 1995).

[Abra 96] Abrams, M., Ribler, R., Mathur, A., “Two Performance Tool Design Issues and CHITRA’s Solutions”, Proc. SPDT’96, ACM SIGMETRICS Symposium on Parallel and Distributed Tools, Philadelphia, PA, USA, pp. 98-107 (May 1996).

[AbSo 95] Abramson, D., Sosic, R., “A Debugging Tool for Software Evolution”, Proc. CASE-95, 7th International Workshop on Computer-Aided Software Engineering, Toronto, Canada (July 1995).

[Adve 91] Adve, S.V., Hill, M.D., Miller, B.P., Netzer, R.H.B., “Detecting Data Races on Weak Memory Systems”, The 18th Annual Intl. Symposium on Computer Architecture, Toronto, Canada (1991), reprinted in: ACM SIGARCH Computer Architecture News, Vol. 19, No. 3, pp. 234-243 (May 1991)

[AgCh 96] Aggarwal, J.K., Chillakanti, P., “Software for Parallel Computing - A Perspective”, in: Zomaya, A.Y., “Parallel Computing: Paradigms and Applications”, Intl. Thomson Computer Press, pp. 357-375 (1996).

[Agra 94] Agrawal, H., “On Slicing Programs with Jump Statements”, Proc. PLDI ’94, ACM SIGPLAN ’94 Conference on Programming Language Design and Implementation, Orlando, Florida, USA, pp. 302-312 (June 1994)
reprinted in: ACM SIGPLAN Notices, Vol. 29, No. 6 (1994).

[Agra 91] Agrawal, H., DeMillo, R.A., Spafford, E.H., “An Execution Backtracking Approach to Debugging”, IEEE Software, Vol. 8, No. 3, pp. 21-26 (May 1991).

[Agra 93] Agrawal, H., DeMillo, R.A., Spafford, E.H., “Debugging with Dynamic Slicing and Backtracking”, IEEE Software - Practice and Experience, Vol. 23, No. 6, pp. 589-616 (June 1993).

[AgHo 90] Agrawal, H., Horgan, J.R., “Dynamic Program Slicing”, Proc. PLDI ’90, ACM SIGPLAN ’90 Conference on Programming Language Design and Implementation, New York, NY, USA, pp. 246-256 (June 1990)
reprinted in: ACM SIGPLAN Notices, Vol. 25, No. 6 (1990).

[Aho 83] Aho, A.V., Hopcroft, J.E., Ullman, J.D., “Data Structures and Algorithms”, Addison-Wesley Series in Computer Science and Information Processing, Addison-Wesley (1983).

[AhBa 99] Ahr, D., Bäcker, A., “Project Workspaces for Parallel Computing”, in: Amestoy, P., Berger, P., Dayde, M., Duff, I., Fraysse, V., Giraud, L., Ruiz, D., (Eds.), “Euro-Par’99 Parallel Processing”, Proceedings 5th Intl. Euro-Par Conference, Lecture Notes in Computer Science, Vol. 1685, Springer-Verlag, Toulouse, France, pp. 98-107 (Aug./Sept. 1999).

[AlVe 93] Alagar, S., Venkatesan, S., “Hierarchy in Testing Distributed Programs”, in: Fritzson, P.A., (Ed.), “Automated and Algorithmic Debugging”, Proc. of AADEBUG'93, 1st Intl. Workshop on Automated and Algorithmic Debugging, Lecture Notes in Computer Science, Vol. 749, Springer-Verlag, Linköping, Sweden, pp. 101-106 (May 1993).

[Alle 91] Allen, D., Bowker, R., Jourdenais, K., Simons, J., Sistare, S., Title, R., “The Prism Programming Environment”, Proc. Supercomputer Debugging Workshop '91, Albuquerque, NM, USA, pp. 1-7 (November 1991).

[AlPa 87] Allen, T.R., Padua, D.A., “Debugging Fortran on a Shared Memory Machine”, Proc. of the 1987 Intl. Conference on Parallel Processing, St. Charles, IL, USA, pp. 721-727 (August 1987).

[AlGo 94] Almasi, G.S., Gottlieb, A., “Highly Parallel Computing”, 2nd Edition, The Benjamin/Cummings Series in Computer Science and Engineering (1994).

[Appe 90] Appelbe, B., “Software Tools for Visualization of Performance”, in: Simmons, M., Koskela, R., (Eds.), “Performance Instrumentation and Visualization”, Frontier Series, ACM Press, Addison-Wesley, Chapter 10, pp. 147-155 (1990).

[ApMc 88] Appelbe, W.F., McDowell, Ch.E., “Integrating Tools for Debugging and Developing Multitasking Programs”, Proceedings of the ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed

Debugging, May 5-6, 1988, University of Wisconsin, Madison, Wisconsin, USA, SIGPLAN Notices, Vol. 24, No. 1, pp. 78-88 (January 1989).

[ASCI 99] US Dept. of Energy, "ASCI - Accelerated Strategic Computing Initiative program", Sandia National Labs, Lawrence Livermore National Lab, Los Alamos National Lab, http://www.llnl.gov/asci (1999).

[AuLe 95] Audenaert, K.M., Levrouw, L.J., "Space Efficient Data Race Detection for Parallel Programs with Series-Parallel Task Graphs", Proceedings EUROMICRO PDP '95, 3rd EUROMICRO Workshop on Parallel and Distributed Processing, IEEE Computer Society Press, San Remo, Italy, pp. 508-515 (January 1995).

[Augu 95] Auguston, M., "Program Behavior Model Based on Event Grammar and its Application for Debugging Automation", Proc. AADEBUG '95, 2nd International Workshop on Automated and Algorithmic Debugging, Saint Malo, France, pp. 277-291 (May 1995).

[Augu 98] Auguston, M., "Building Program Behavior Models", Proc. ECAI-98, European Conference on Artificial Intelligence, Workshop on Spatial and Temporal Reasoning, Brighton, England, pp. 19-26 (August 1998).

[Augu 99] Auguston, M., "Tools for Program Dynamic Analysis, Testing, and Debugging Based on Event Grammars", Technical Report, NMSU-CSTR-9906, Computer Science Department, New Mexico State University, Las Cruces, New Mexico, USA (1999).

[Aydt 92] Aydt, R.A., "The Pablo Self-Defining Data Format", Technical Report, Department of Computer Science, University of Illinois, Urbana, Illinois (May 1992).
latest revision: ftp://bugle.cs.uiuc.edu/pub/Pablo.Release.5/Documentation/SDDF.ps.gz (1997).

[Baba 95] Babaoglu, Ö., Fromentin, E., Raynal, M., "Debugging Distributed Executions Using Language Recognition", Proc. ICPP '95, 24th International Conference on Parallel Processing, Oconomowoc, Wisconsin, USA, Vol. II, pp. 55-62 (August 1995).

[Bach 98] Bach, J., "A Framework for Good Enough Testing", IEEE Computer, Vol. 31, No. 10, pp. 124-126 (October 1998).

[Bach 99] Bach, J., "Risk and Requirements-based Testing", IEEE Computer, Vol. 32, No. 6, pp. 113-114 (June 1999).

[Baec 97] Baecker, R., DiGiano, Ch., Marcus, A., "Software Visualization for Debugging", Communications of the ACM, Vol. 40, No. 4, pp. 44-54 (April 1997).

[Balz 69] Balzer, R.M., "EXDAMS - EXtendable Debugging and Monitoring System", Proceedings of the AFIPS Spring Joint Computer Conference, pp. 567-580 (1969).

[BaEi 96] Ball, T. and Eick, S.G., "Software Visualization in the Large", IEEE Computer, Vol. 29, No. 4, pp. 33-43 (April 96).

[Bate 88] Bates, P., "Debugging Heterogeneous Distributed Systems Using Event-Based Models of Behavior", Proceedings of the ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging, May 5-6, 1988, University of Wisconsin, Madison, Wisconsin, USA, SIGPLAN Notices, Vol. 24, No. 1, pp. 11-22 (January 1989).

[Bate 95] Bates, P., "Debugging Heterogeneous Distributed Systems Using Event-Based Models of Behavior", ACM Transactions on Computer Systems, Vol. 13, No. 1, pp. 1-31 (February 1995).

[BaWi 83a] Bates, P., Wileden, J.C., "An Approach to High-Level Debugging of Distributed Systems", ACM SIGPLAN Notices, Vol. 18, No. 8, pp. 107-111 (August 1983).

[BaWi 83b] Bates, P., Wileden, J.S., "High-Level Debugging of Distributed Systems: The Behavioral Abstraction Approach", Journal of Systems and Software, Vol. 3, No. 4, pp. 255-264 (December 1983).

[Begu 93a] Beguelin, A., Dongarra, J.J., Geist, A., Sunderam, V.S., "Visualization and Debugging in a Heterogeneous Environment", IEEE Computer, Vol. 26, No. 6, pp. 88-95 (June 1993).

[Begu 93b] Beguelin, A., Dongarra, J.J., Geist, A., Manchek, R., Moore, K., Sunderam, V.S., "PVM and HeNCE: Tools for Heterogeneous Network Computing", in: Dongarra, J.J., Tourancheau, B., (Eds.), "Environments and Tools for Parallel Scientific Computing", Elsevier Science Publishers (1993).

[Bemm 92] Bemmerl, T., "Programmierung skalierbarer Multiprozessoren", BI-Wissenschaftsverlag, Reihe Informatik, Bd. 84, Mannheim, Germany (1992) [in German].

[BeBr 93] Bemmerl, T., Braun, P., “Visualization of Message Passing Parallel Programs with the TOPSYS Parallel Programming Environment”, Journal of Parallel and Distributed Computing, Vol. 18, No. 2, pp. 118-128 (June 1993).

[Bern 66] Bernstein, A.J., “Analysis of Programs for Parallel Processing”, IEEE Transactions on Electronic Computers, Vol. EC-15, No. 5, pp. 757-763 (1966).

[BeSt 94] Bernstein, K.L., Stark, E.W., “Operational Semantics of a Focusing Debugger”, Proc. MFPS ’95, 11th Conference on the Mathematical Foundations of Programming Semantics, New Orleans, LA, USA, pp. 21-40 (March 1995).

[BeZi 98] Berrendorf, R., Ziegler, H., “PCL - The Performance Counter Library: A Common Interface to Access Hardware Performance Counters on Microprocessors”, Technical Report FZJ-ZAM-IB-9816, Research Center Jülich (Oct. 1998).

[BlWa 96] Blum, M., Wasserman, H., “Reflections on the Pentium Division Bug”, IEEE Transactions on Computers, Vol. 45, No. 4, pp. 385-395 (April 1996).

[BlEi 92] Blume, W., Eigenmann, R., “Performance Analysis of Parallelizing Compilers on the Perfect Benchmarks Programs”, IEEE Transactions on Parallel and Distributed Systems, Vol. 3, No. 6, pp. 643-656 (1992).

[Bode 95] Bode, A. “Klassifikation paralleler Architekturen”, in: Waldschmidt, K., (Ed.), “Parallelrechner: Architekturen - Systeme - Werkzeuge”, Leitfäden der Informatik, Teubner, Stuttgart, Germany, Chapter 2, pp. 11-40 (1995) [in German].

[BoHä 83] Bode, A., Händler, W., “Rechnerarchitektur II - Strukturen”, Springer Verlag, Berlin (1983) [in German].

[Boeh 76] Boehm, B.W., “Software Engineering”, IEEE Transactions on Computers, Vol. C-25, No. 12, pp. 1226-1241 (Dec. 1976).

[Boeh 83] Boehm, B.W., “Seven Basic Principles of Software Engineering”, Journal of Systems and Software, Vol. 3, No. 1, pp. 3-24 (March 1983).

[Born 95] Born, T., Obelöer, W., Schäfers, L., Scheidler, C., “The Monitoring Facilities of the Graphical Parallel Programming Environment TRAPPER”, Proc. 3rd. Euromicro Workshop on Parallel and Distributed Processing, Sanremo, Italy, pp. 555-562 (Jan. 1995).

[BoPr 93] Bowen, N.S., Pradhan, D.K., “Processor and Memory-Based Checkpoint and Rollback Recovery”, IEEE Computer, Vol. 26, No. 2, pp. 22-31 (February 1993).

[BoMo 77] Boyer, R.S., Moore, J.S. “A Fast String Searching Algorithm”, Communications of the ACM, Vol. 20, No. 10, pp. 762-772 (October 1977).

[Boyl 99] Boyle, J.M., Resler, R.D., Winter, V.L., “Do You Trust Your Compiler?”, IEEE Computer, pp. 65-73 (May 1999).

[Brez 99a] Brezany, P., Grabner, S., Sowa, K., Wismüller, R., “DeHiFo -- An Advanced HPF Debugging System”, Proc. EUROMICRO PDP’99, 7th Euromicro Workshop on Parallel and Distributed Processing, Funchal, Portugal (February 1999).

[Brez 99b] Brezany, P., Czerwinski, P., Koppler, R., Sowa, K., Volkert, J., “Advanced Visualization and Data Distribution Steering in an HPF Parallelization Environment”, Proc. ParCo ’99, Conference on Parallel Computing, Delft, The Netherlands (September 1999).

[Brin 95] Brinch Hansen, P., “Studies in Computational Science - Parallel Programming Paradigms”, Prentice Hall, Englewood Cliffs, New Jersey, USA (1995).

[Brit 98] Britcher, R.N., “Software Reliability: Numbers Aren’t the Whole Story”, IEEE Computer, Vol. 31, No. 10, pp. 127-128 (October 1998).

[Broc 96] F.A. Brockhaus GmbH, “Brockhaus - Die Enzyklopädie: in 24 Bänden”, 20th Edition, Leipzig-Mannheim (1996).

[Brow 97] Brown, J., Francioni, J., Pancake, C., “White Paper on Formation of the High Performance Debugging Forum”, available at: http://www.ptools.org/hpdf/meetings/mar97/whitepaper.html (February 1997).

[BrHe 91] Brown, M.H., Hershberger, J., “Colour and Sound in Algorithm Animation”, Technical Report 76a, The Systems Research Center, Digital Equipment Corporation, Palo Alto, California, USA (August 1991).

[Brow 95] Browne, J.C., Hyder, S.I., Dongarra, J., Moore, K., Newton, P., “Visual Programming and Debugging for Parallel Computing”, in: IEEE Parallel & Distributed Technology, Vol. 3, No. 1, pp. 75-83 (Spring 1995).

[Brow 99] Browne, S., Dongarra, J.J., London, K., “Review of Performance Analysis Tools for MPI Parallel Programs”, Technical Report, NHSE, National High-performance Software Exchange project, Computer Science Department, University of Tennessee, Knoxville, TN, USA, http://www.cs.utk.edu/~browne/perftools-review/review.html (1999).

[Brun 99] Brune, M., Gehring, J., Reinefeld, A., “Linking Message-Passing Environments”, in: Buyya, R., (Ed.), “High Performance Cluster Computing - Programming and Applications”, Vol. 2, Prentice-Hall PTR, New Jersey, USA, Chapter 4, pp. 87-100 (1999).

[BuHo 99] Buck, B., Hollingsworth, J.K., “An API for Runtime Code Patching”, Computer Science Department, University of Maryland, College Park, MD, USA, http://www.cs.umd.edu/~hollings/papers/apijournal.pdf (1999).

[Buth 99] Buth, B., Peleska, J., Shi, H., “Combining Methods for the Livelock Analysis of a Fault-Tolerant System”, in: Haeberer, A.M., (Ed.), “Algebraic Methodology and Software Technology”, Proc. of the 7th Intl. Conference, AMAST 98, Lecture Notes in Computer Science, Vol. 1548, Springer-Verlag, Amazonia, Brazil, pp. 124-139 (January 1999).

[BuLu 94] Butler, R.M., Lusk, E.L., “Monitors, Messages, and Clusters: The p4 Parallel Programming System”, Journal of Parallel Computing, Vol. 20, pp. 547-564 (Apr. 1994).

[Buyy 99a] Buyya, R., (Ed.), “High Performance Cluster Computing - Architectures and Systems”, Vol. 1, Prentice-Hall PTR, New Jersey, USA (1999).

[Buyy 99b] Buyya, R., (Ed.), “High Performance Cluster Computing - Programming and Applications”, Vol. 2, Prentice-Hall PTR, New Jersey, USA (1999).

[Cai 93] Cai, W., Milne, W.J., Turner, S.J., “Graphical Views of the Behavior of Parallel Programs”, Journal of Parallel and Distributed Computing, Vol. 18, pp. 223-230 (1993).

[CaTu 94] Cai, W., Turner, S.J., “An Approach to the Run-Time Monitoring of Parallel Programs”, The Computer Journal, British Computer Society, Vol. 37, No. 4, pp. 333-345 (1994).

[CaMi 92] Cargille, J., Miller, B.P., “Binary Wrapping: A Technique for Instrumenting Object Code”, ACM SIGPLAN Notices, Vol. 27, No. 6, pp. 17-18 (June 1992).

[CaSc 89] Caspary, E., Scherson, I.D., “A Self-Balanced Parallel Ray-Tracing Algorithm”, in: Dew, P.M., Earnshaw, R.A., Heywood, T.R., (Eds.), “Parallel Processing for Computer Vision and Display”, Papers presented at the International Conference on Computer Graphics, University of Leeds, Great Britain (January 1988), Addison-Wesley, pp. 408-419 (1989).

[CaTa 91] Carver, R.H., Tai, K.C., “Replay and Testing for Concurrent Programs”, IEEE Software, Vol. 8, No. 2, pp. 66-74 (March 1991).

[ChLa 85] Chandy, K.M., Lamport, L., “Distributed Snapshots: Determining Global States of Distributed Systems”, ACM Transactions on Computer Systems, Vol. 3, No. 1, pp. 63-75 (February 1985).

[Chan 83] Chandy, K.M., Misra, J., Hass, L.M., “Distributed Deadlock Detection”, ACM Transactions on Computing Systems, pp. 144-156 (May 1983).

[ChRa 72] Chandy, K.M., Ramamoorthy, C.V., “Rollback and Recovery Strategies for Computer Programs”, IEEE Transactions on Computers, Vol. 21, pp. 546-556 (June 1972).

[Char 90] Charron-Bost, B., “Concerning the Size of Clocks”, in: Guessarian, I., (Ed.), “Semantics of Systems of Concurrent Processes”, Lecture Notes in Computer Science, Springer-Verlag, Vol. 460, pp. 176-184 (1990).

[Chas 99] Chassin de Kergommeaux, J., Ronsse, M., De Bosschere, K., “MPL*: Efficient Record/Replay of nondeterministic features of message passing libraries”, in: Dongarra, J., Luque, E., Margalef, T., (Eds.), “Recent Advances in Parallel Virtual Machine and Message Passing Interface”, Proc. EUROPVM/MPI 99, 6th European PVM/MPI Users’ Group Meeting, Lecture Notes in Computer Science, Vol. 1697, Springer-Verlag, Barcelona, Spain, pp. 141-148 (September 1999).

[ChSt 00a] Chassin de Kergommeaux, J., Stein, B., “Paje: An Extensible and Interactive Environment for Visualizing Multi-Threaded Program Executions”, Rapport de Recherche, No. 3919, INRIA, Institut National de Recherche en Informatique et en Automatique, France (April 2000).

[ChSt 00b] Chassin de Kergommeaux, J., Stein, B., “Paje: An Extensible Environment for Visualizing Multi-Threaded Program Executions”, in: Bode, A., Ludwig, Th., Karl W., Wismüller, R., (Eds.), “Euro-Par 2000 Parallel Processing”, 6th International Euro-Par Conference, Lecture Notes in Computer Science, Springer-Verlag, Vol. 1900, Munich, Germany, pp. 133-140 (August/September 2000).

[ChHo 94] Cheng, D., Hood, R., “A Portable Debugger for Parallel and Distributed Programs”, Proc of Supercomputing’94, Washington, D.C., pp. 723-732 (Nov. 1994).

[Chen 93] Cheng, J., “Slicing Concurrent Programs - A Graph-Theoretical Approach”, in: Fritzson, P., (Ed.), Proc. AADEBUG ’93, First International Workshop on Automated and Algorithmic Debugging, Lecture Notes in Computer Science, Springer-Verlag, Vol. 749, pp. 223-240 (1993).

[ChSk 93] Cheriton, D.R., Skeen, D., “Understanding the Limitations of Causally and Totally Ordered Communications”, Operating Systems Review, pp. 44-57 (December 1993).

[Choi 91] Choi, J.-D., Miller, B.P., Netzer, R.B., “Techniques for Debugging Parallel Programs with Flowback Analysis”, ACM Transactions on Programming Languages and Systems, Vol. 13, No. 4, pp. 491-530 (Oct. 1991).

[ChMi 91] Choi, J.-D., Min, S.L., “RACE FRONTIER: Reproducing Data Races in Parallel-Program Debugging”, ACM Press, Proc. of the 3rd ACM SIGPLAN Symposium on Principles & Practise of Parallel Programming PPOPP, Williamsburg, Virginia (April 1991),
reprinted in: ACM SIGPLAN Notices, Vol. 26, No. 7, pp. 145-154 (July 1991).

[Clem 95a] Clemencon, C., Endo, A., Fritscher, J., Müller, A., Rühl, R., Wylie, B.J.N., “The ‘Annai’ Environment for Portable Distributed Parallel Programming”, in: El-Rewini, H., Shriver, B.D., (Eds.), Proc. HICSS ’28, 28th Hawaii Intl. Conference on System Sciences, Vol. 2, IEEE Computer Science, Maui, Hawaii, USA (January 1995).

[Clem 95b] Clemencon, C., Fritscher, J., Rühl, R., “Visualization, Execution Control and Replay of Massively Parallel Programs within Annai’s Debugging Tool”, Proc. High Performance Computing Symposium, HPCS ’95, Montreal, Canada, pp. 393-404 (July 1995).

[Conw 63] Conway, M.E., “A Multiprocessor System Design”, Proc. AFIPS Fall Joint Computer Conference, Vol. 4, pp. 139-146 (1963).

[Coy 92] Coy, W. et al, (Eds.), “Sichtweisen der Informatik”, Vieweg, Braunschweig, Wiesbaden (1992) [in German].

[Craw 96] Crawford, D., “ASCI Academia Strategic Alliances Program: Research Interests in Computer Systems and Computational and Computer Science Infrastructure”, Sandia National Laboratories, USA (Dec. 1996).

[Cunh 98a] Cunha, J.C., Lourenço, J.M., Antão, T., “An Experiment in Tool Integration: the DDBG Parallel and Distributed Debugger”, EUROMICRO Journal of Systems Architecture, 2nd Special Issue on Tools and Environments for Parallel Processing, Elsevier Science Publisher (1998).

[Cunh 98b] Cunha, J.C., Lourenço, J.M., Vieira, J., Moscão, B., Pereira, D., “A Framework to Support Parallel and Distributed Debugging”, In Proc. of HPCN’98, Amsterdam, Netherlands (1998).

[Cuny 97] Cuny, J.E., Dunn, R., Hackstadt, S.T., Harrop, Ch., Hersey, H., Malony, A.D., Toomey, D.R., “Building Domain-Specific Environments for Computational Science,: A Case Study in Seismic Tomography”, Journal of Parallel Computing, Dongarra, J.J., Tourancheau, B., (Eds.), “Environments and Tools for Parallel Scientific Computing III”, Elsevier Publisher, Vol. 23, No. 1-2 (April 1997).

[Cuny 92] Cuny, J.E., Hough, A.A., Kundu, J., “Logical Time in Visualizations Produced by Parallel Programs”, Proceedings of Visualization ’92, pp. 186-193 (1992).

[Cuny 93] Cuny, J.E., Forman, G., Hough, A., Kundu, J., Lin, C., Snyder, L., Stemple, D.W., “The Ariadne Debugger: Scalable Application of Event-Based Abstraction”, Proc. of the 3rd ACM/ONR Workshop on Parallel and Distributed Debugging, San Diego, CA, USA (May 1993),
reprinted in: ACM SIGPLAN Notices, Vol. 28, No. 12, pp. 85-95 (December 1993).

[CuWi 82] Curtis, R.S., Wittie, L.D., “BugNet: A Debugging System for Parallel Programming Environments”, Proc. 3rd Intl. Conference on Distributed Computing Systems, Miami, FL, USA, pp. 394-399 (October 1982).

[CyLe 93] Cypher, R., Leu, E., “The Semantics of Blocking and Nonblocking Send and Receive Primitives”, IBM Technical Report RJ 9527 (93405) (September 1993).

[Damo 94] Damodaran-Kamal, S.K., “Testing and Debugging Nondeterministic Message Passing Programs”, PhD Thesis, University of Southwestern Louisiana, USA (1994).

[DaFr 93] Damodaran-Kamal, S.K., Francioni, J.M., “Nondeterminacy: Testing and Debugging in Message Passing Parallel Programs”, Proc. 3rd ACM/ONR Workshop on Parallel and Distributed Debugging, San Diego, CA, USA (May 1993),
reprinted in: ACM SIGPLAN Notices, Vol. 28, No. 12, pp. 118-128 (Dec. 1993).

[DaFr 94] Damodaran-Kamal, S.K., Francioni, J.M., “Testing Races in Parallel Programs with an OtOt Strategy”, Proc. 1994 Intl. Symp. on Software Testing and Analysis, Seattle, WA, USA (August 1994).

[Daup 92] Dauphin, P., Hofmann, R., Klar, R., Mohr, B., Quick, A., Siegle, M., Sötz, F., “ZM4/SIMPLE: A General Approach to Performance-Measurement and -Evaluation of Distributed Systems”, Technical Report 1/91, IMMD VII, University Erlangen-Nürnberg (January 1991)
reprinted in: Casavant, T., Singhal, M., (Eds.), “Readings in Distributed Computing Systems”, IEEE Computer Society Press, Los Alamitos, California, USA, Chapter 6. pp. 286-309 (January 1992).

[DaKe 82] Davis, A.L., Keller, R.M., “Data Flow Program Graphs”, IEEE Computer, Vol. 15, pp. 26-41 (1982).

[Deke 81] Dekel, E., Nassimi, D., Sahni, S., “Parallel Matrix and Graph Algorithms”, SIAM Journal of Computing, Vol. 10, No. 4, pp. 657-675 (Nov. 1981).

[DeRo 98] DeRose, L., Zhang, Y., Reed, D.A., “SvPablo: A Multi-Language Performance Analysis System”, Proc. Performance Tools ’98, 10th International Conference on Computer Performance Evaluation - Modelling Techniques and Tools, Palma de Mallorca, Spain, pp. 352-355 (September 1998)

[DiSc 90] Dinning, A., Schonberg, E., “Detecting Access Anomalies in Programs with Critical Sections”, Proc. of PPOPP ’90, 2nd ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming, Seattle, WA, USA, pp. 1-10 (1990).

[Dolp 98] Dolphin Interconnect Solutions Inc., “TotalView 3.8”, Document available via WWW at: http://www.dolphinics.com/tw/download/tv3.8.0-webdocs/User_Guide.ps (1998).

[DoMu 97] Dominque, J., Mulholland, P., “Fostering Debugging Communities on the Web”, Communications of the ACM, Vol. 40, No. 4, pp. 65-71 (April 1997).

[Dong 94] Dongarra, J.J., “Performance of Various Computers Using Standard Linear Equations Software”, Technical Report, UT-CS-89-85, Computer Science Dept., University of Tennessee, TN, USA (1994).

[Dong 99] Dongarra, J.J., Meuer, H.W., and Strohmeier, E., (Eds.) “TOP 500 Supercomputer Sites - 13th Edition”, Technical Report, RUM 58/99, Computing Center University of Mannheim, Germany, and UT-CS-99-425, Computer Science Dept., University of Tennessee, TN, USA (June 1999).

[Dubo 99] Dubois, P.F., “Scientific Components are Coming”, IEEE Computer, Vol. 32, No. 3, pp. 115-117 (March 1999).

[Dues 92] Duesterwald, E., Gupta, R., Soffa, M.L., “Distributed Slicing and Partial Re-execution for Distributed Programs”, Proc. of the 5th Workshop on Languages and Compilers for Parallel Computing, New Haven, Connecticut, USA (August 1992).

[EaZh 96] Eades, P., Zhang, K., (Eds.), “Software Visualization”, Series on Software Engineering and Knowledge Engineering, Vol. 7, World Scientific Publishing, Singapore (1996).

[ElRe 93] El-Rewini, H., Lewis, T., Shriver, B., “Parallel and Distributed Systems - From Theory to Practice”, IEEE Parallel and Distributed Technology, Vol. 1, No. 3, pp. 7-11 (Aug. 1993).

[EmPa 88] Emrath, P.A., Padua, D.A., “Automatic Detection of Nondeterminacy in Parallel Programs”, Proceedings of the ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging, May 5-6, 1988, University of Wisconsin, Madison, Wisconsin, USA, SIGPLAN Notices, Vol. 24, No. 1, pp. 89-99 (January 1989).

[Emra 92] Emrath, P.A., Ghosh, S., Padua, D.A., “Detecting Nondeterminacy in Parallel Programs”, IEEE Software, Vol. 9, No. 1, pp. 67-77 (January 1992).

[Enge 88] Engesser, H., (Ed.), “Duden ‘Informatik’ - Ein Sachlexikon für Studium und Praxis”, Dudenverlag, Mannheim (1988).

[FaCh 95] Fagot, A., Chassin de Kergommeaux, J., “Formal and Experimental Validation of a Low-Overhead Execution Replay Mechanism”, in: Haridi, S., Ali, K., Magnusson, P., (Eds.), Euro-Par ’95 Parallel Processing, Lecture Notes in Computer Science, Vol. 966, Springer-Verlag, pp. 167-178 (August 1995).

[FaCh 96] Fagot, A., Chassin de Kergommeaux, J., “Systematic Assessment of the Overhead of Tracing Parallel Programs”, Proceedings EUROMICRO PDP ’96, 4th EUROMICRO Workshop on Parallel and Distributed Processing, IEEE Computer Society Press, Braga, Portugal, pp. 179-186 (January 1996).

[FaSu 93] Fan, J.-J., Su, K.-Y., “An Efficient Algorithm for Matching Multiple Patterns”, IEEE Transactions on Knowledge and Data Engineering, Vol. 5, No. 2, pp. 339-351 (1993).

[Feld 88] Feldman, S.I., Brown, Ch.B., “Igor: A System for Program Debugging via Reversible Execution”, Proceedings of the ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging, May 5-6, 1988, University of Wisconsin, Madison, Wisconsin, USA, SIGPLAN Notices, Vol. 24, No. 1, pp. 112-123 (January 1989).

[Fers 92] Ferscha, A., “A Petri Net Approach for Performance Oriented Parallel Program Design”, Journal of Parallel and Distributed Computing, Vol. 15, No. 3, pp. 188-206 (July 1992).

[Feyn 96] Feynman, R.P., “Feynman Lectures on Computation”, edited by: Hey, A.J.G., Allen, R.W., The Advanced Book Program, Addison-Wesley Publishing Company, Reading, Massachusetts, USA (1996).

[Fidg 88] Fidge, C.J., “Timestamps in Message-Passing Systems That Preserve the Partial Ordering”, Australian Computer Science Communications, Vol. 10, No. 1, pp. 56-66 (February 1988).

[Fidg 89] Fidge, C.J., “Partial Orders for Parallel Debugging”, Proceedings of the ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging, May 5-6, 1988, University of Wisconsin, Madison, Wisconsin, USA, SIGPLAN Notices, Vol. 24, No. 1, pp. 183-194 (January 1989).

[Figd 91] Fidge, C.J., “Logical Time in Distributed Computing Systems”, IEEE Computer, Vol. 24, No. 8, pp. 28-33 (August 1991).

[Fidg 93] Fidge, C.J., “Fundamentals of Distributed System Observation”, Technical Report 93-15, Software Verification Research Centre, Dept. of Computer Science, The University of Queensland, Australia (November 1993)
reprinted in: IEEE Software, Vol. 13, No. 6, pp. 77-83 (November 1996).

[Fidg 97] Fidge, C.J., “A Limitation of Vector Timestamps for Reconstructing Distributed Computations”, Technical Report 97-43, Software Verification Research Centre, Dept. of Computer Science, The University of Queensland, Australia (1997).

[FlZü 97] Floyd, C., Züllighoven, H., “Softwaretechnik”, in: Rechenberg, P., Pomberger, G., “Informatik-Handbuch”, Carl Hanser Verlag, München, Wien, pp. 641-667 (1997) [in German].

[Flyn 72] Flynn, M.J., “Some Computer Organizations and Their Effectiveness”, IEEE Transactions on Computers, Vol. 24, No. 9, pp. 948-960 (Sept. 1972).

[Flyn 96] Flynn, M.J., “Parallel Processors Were the Future ... and May Yet Be”, IEEE Computer, Vol. 29, No. 12, pp. 151-152 (Dec. 1996).

[FlRu 96] Flynn, M.J., and Rudd, K.W., “Parallel Architectures”, ACM Computing Surveys, Vol. 28, No. 1, pp. 67-70 (March 1996).

[Fost 94] Foster, I.T., “Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering”, Addison-Wesley Publishing Company, http://www.mcs.anl.gov/dbpp (1995).

[FoKe 98] Foster, I., Kesselman, C., “The Globus Project: A Status Report”, Proc. IPPS/SPDP’98 Heterogeneous Computing Workshop, pp. 4-18 (1998).

[FoKe 99] Foster, I., Kesselman, C., (Eds.), “The Grid: Blueprint for a New Computing Infrastructure”, Morgan Kaufmann Publishers, San Francisco, California, USA (1999).

[Fran 96] Francioni, J.M., “Determining the Effectiveness of Interfaces for Debugging and Performance Analysis Tools”, in: Hayes, A.H., Simmons, M.L., Brown, J.S., Reed, D.A., “Debugging and Performance Tuning for Parallel Computing Systems”, IEEE Computer Society Press (1996).

[FrJa 93] Francioni, J.M., Jackson, J.A., “Breaking the Silence: Auralization of Parallel Program Behavior”, Journal of Parallel and Distributed Computing, Vol. 18, pp. 181-194 (1993).

[Fren 95] French, R.S., Lam, M.S., Levitt, J.R., Olukotun, K., “A General Method for Compiling Event-Driven Simulations”, Proceedings of the 32nd ACM/IEEE Design Automation Conference, San Francisco, CA, USA (1995).

[From 94] Fromentin, E., Raynal, M., Garg, V.K., Tomlinson, A.I., “On-The-Fly Testing of Regular Patterns in Distributed Computations”, Proc. ICPP ’94, 23rd International Conference on Parallel Processing, St. Charles, IL, USA, pp. 73-76 (August 1994).

[From 95a] Fromentin, E., Jard, C., Jourdan, G.-V., Raynal, M., “On-The-Fly Analysis of Distributed Computations”, Technical Report Nr. 859, IRISA, France (September 1994).
reprinted in: Information Processing Letters, Vol. 54, No. 5, pp. 267-274 (1995).

[From 95b] Fromentin, E., Jard, C., Jourdan, G.-V., Raynal, M., “Recognition of Patterns on Control Flows for Debugging Distributed Executions”, Proc. AADEBUG ’95, 2nd International Workshop on Automated and Algorithmic Debugging, Saint Malo, France, pp. 221-232 (May 1995).

[From 95c] Fromentin, E., Plouzeau, N., Raynal, M., “An Introduction to the Analysis and Debug of Distributed Computations”, Proc. IEEE Conference on Architectures and Algorithms for Parallel Processing, Brisbane, Australia (1995).

[Frum 98] Frumkin, M., Hood, R., Yan, J., “On the Information Content of Program Traces”, Technical Report NAS-98-008, http://www.nas.nasa.gov/Research/Reports/Techreports/1998/nas-98-008.pdf, NAS Parallel Tools Group, NASA Ames Research Center, Moffet Field, California, USA (March 1998).

[Furn 86] Furnas, G.W., “Generalized Fisheye Views”, Proc. of Conference on Human Factors in Computing Systems”, pp. 16-23 (April 1986).

[Gait 86] Gait, J., “The Probe Effect in Concurrent Programs”, IEEE Software - Practice and Experience, Vol. 16, No. 3, pp. 225-233 (March 1986).

[Gann 94a] Gannon, J.A., Williams, K.J., Andersland, M.S., Casavant, T.L., Lumpp, J.E., “Trace Recovery in Multiprocessing Systems: Architectural Considerations”, Proc. ICPP’94, Intl. Conference on Parallel Processing, Vol. II, pp. 99-102 (August 1994).

[Gann 94b] Gannon, J.A., Williams, K.J., Andersland, M.S., Lumpp, J.E., Casavant, T.L., “Using Perturbation Tracking to Compensate for Intrusion Propagation in Message Passing Systems”, Proc. 14th IEEE Computer Society Intl. Conference on Distributed Computing Systems, Poland (1994).

[Geis 90] Geist, G.A., Heath, M.T., Peyton, B.W., Worley, P. H., “PICL: A Portable Instrumented Communication Library, Reference Manual”, Technical Report ORNL/TM-11130 and “A users’ guide to PICL”, Technical Report ORNL/TM-11616, Oak Ridge National Laboratory, Tennessee, USA (1990).

[Geis 94a] Geist, G.A., Beguelin, A., Dongarra, J.J., Jiang, W., Manchek, R., and Sunderam, V.S., “PVM3 - User’s Guide and Reference Manual”, Technical Report, Oak Ridge National Laboratory, Tennessee, MIT Press, Cambridge, MA, USA (1994).

[Geis 94b] Geist, G.A., Kohl, J., Papadopoulos, P., “Visualization, Debugging, and Performance in PVM”, Proc. of Visualization and Debugging Workshop (Oct. 1994).
reprinted in: Simmons, M.L., Hayes, A.H., Brown, J.S., Reed, D.A., “Debugging and Performance Tuning for Parallel Computing Systems”, IEEE Computer Society Press, Los Alamitos, CA, USA, pp. 65-77 (1996).

[Geis 96] Geist, G.A., Kohl, J.A., Papdopulos, P.M., “PVM and MPI: A Comparison of Features”, in: Calculateurs Paralleles, Vol. 8., No. 2 (1996).

[GeSu 92] Geist, G.A., Sunderam, V.S., “Network-based Concurrent Computing on the PVM System”, in: Concurrency - Practice & Experience, Vol. 4, No. 4, pp. 293-311 (1992).

[Geof 99] Geoffray, P., Lefevre, L., Pham, C.D., Prylli, L., Reymann, O., Tourancheau, B., Westrelin, R., “High-speed LANS: New Environments for Parallel and Distributed Applications”, in: Amestoy, P., Berger, P., Dayde, M., Duff, I., Fraysse, V., Giraud, L., Ruiz, D., (Eds.), “Euro-Par’99 Parallel Processing”, Proceedings 5th Intl. Euro-Par Conference, Lecture Notes in Computer Science, Vol. 1685, Springer-Verlag, Toulouse, France, pp. 633-642 (Aug./Sept. 1999).

[Germ 84] German, S.M., “Monitoring for Deadlock and Blocking in Ada Tasking”, IEEE Transactions on Software Engineering, Vol. SE-10, No. 6, pp. 764-777 (November 1984).

[Gern 98] Gerndt, M., Mohr, B., Pantano, M., Wolf, F., “Automatic Performance Analysis for CRAY T3E”, Proc. CPC ’98, Seventh Workshop on Compilers for Parallel Computers, University of Linköping,

Sweden, pp. 69-78 (June/July 1998)
reprinted in: Technical Report FZJ-ZAM-IB-9808, Forschungszentrum Jülich, Zentralinstitut für Angewandte Mathematik, Germany (May 1998).

[Gers 94] Gerstel, O., Hurfin, M., Plouzeau, N., Raynal, M., Zaks, S., "On-the-Fly Replay: A Practical Paradigm and Its Implementation for Distributed Debugging", Proceedings SPDP '94, 6th Symposium on Parallel and Distributed Processing, IEEE Computer Society Press, pp. 266-272 (1994).

[Gill 58] Gill, S., "Parallel Programming", The Computer Journal, Vol. 1, pp. 2-10 (Apr. 1958).

[Girc 88] Gircys, G.R., "Understanding and Using COFF", Nutshell Series, 1st Edition, O'Reilly & Associates, Sebastopol, CA, USA (1988).

[GlSh 80] Gligor, V.D., Shattuck, S.H., "On Deadlock Detection in Distributed Systems", IEEE Transactions on Software Engineering, Vol. SE-6, No. 5, pp. 435-439 (May 1980).

[Gorl 91] Gorlick, M.M., "The Flight Recorder: An Architectural Aid for System Monitoring", Proc. of the ACM/ONR Workshop on Parallel and Distributed Debugging, New York, NY, USA (1991)
reprinted in: ACM SIGPLAN Notices, Vol. 26, No. 12, pp. 175-183 (December 1991).

[Grab 95] Grabner, S., Kranzlmüller, D., Volkert, J., "Debugging Parallel Programs using ATEMPT", Proc. HPCN Europe 95 Conference, Milano, Italy, pp. 235-240 (May 1995).

[GrVo 93] Grabner, S., Volkert, J., "Debugging Parallel Programs Using Event Graph Manipulation", Proc. VECPAR '93, 1st Intl. Meeting on Vector and Parallel Processing, Porto, Portugal (Sept. 1993),
reprinted in: Computing Systems in Engineering, Vol. 6, Nos. 4/5, Elsevier Science Ltd., Great Britain, pp. 443 - 450 (1995).

[GrVo 96] Grabner, S., Volkert, J., "Debugging Distributed Memory Programs Using Communication Graph Manipulation", Proc. HPCS'96, 10th Annual International Symposium on High Performance Computing Systems, Canada (1996).

[Grab 99] Grabner, S., "Strategien zur Fehlersuche in parallelen Programmen", Dissertation, GUP Linz, Dept. for Graphics and Parallel Processing, Technical Computer Science and Telematics, Johannes Kepler University Linz (Jan. 1999) [in German].

[Gran 89] "Grand Challenges to Computational Science", Future Generation Computer Systems 5, Special Double Issue, No. 2&3 (1989).

[Gran 92] Committee on Physical, Mathematical and Engineering Sciences, "Grand Challenges: High Performance Computing and Communications - the FY 1992 U.S. Research and Development Program", Technical Report, U.S. Government Office of Science and Technology (1992).

[Gran 96] "High Performance Computing and Communications: Foundation for America's Information Future", HPCC FY 96 Blue Book (1996).

[Gran 98] NASA ESS (Earth and Space Sciences) project, "Grand Challenges", NASA High Performance Computing and Communications (HPCC) Program, JPL's High Performance Computing Group, http://www-hpc.jpl.nasa.gov/NGCS (1998).

[Gran 99a] San Diego Supercomputer Center (SDSC), "Grand Challenge Equations", University of California, San Diego, http://www.sdsc.edu/GCequations (1999).

[Gran 99b] National HPCC Software Exchange (NHSE), "Definition of Grand Challenges", High Performance Computing and Communications, http://www.nhse.org/grand_challenge.html (1999).

[Grim 98] Grimshaw, A., Ferrari, A., Lindahl, G., Holcomb, K., "Metasystems", Communications of the ACM, Vol. 41, No. 11, pp. 46-55 (November 1998).

[Gris 91] Griswold, V.J., "Core Algorithms for autonomous Monitoring of Distributed Systems", Proc. of the ACM/ONR Workshop on Parallel and Distributed Debugging, New York, NY, USA (1991)
reprinted in: ACM SIGPLAN Notices, Vol. 26, No. 12, pp. 36-45 (December 1991).

[Grop 94] Gropp, W., Lusk, E., Skjellum, A., "Using MPI: Portable Parallel Programming with the Message-Passing Interface", MIT Press, Cambridge, MA, USA (1994).

[Grub 96] Gruber, B., Haring, G., Kranzlmüller, D., and Volkert, J., "Parallel Programming with CAPSE - A Case Study", Proc. 4th EUROMICRO Workshop on Par. and Distr. Programming, Braga, Portugal, pp. 130-137 (Jan. 1996).

[GuEi 94] Gu, W., Eisenhauer, G., Kraemer, E., Schwan, K., Stasko, J., Vetter, J., “Falcon: On-line Monitoring and Steering of Large-Scale Parallel Programs”, in: Proc. Frontiers’95, Fifth Symposium on the Frontiers of Massively Parallel Computation, McLean Virginia (February 1995).

[Gu 94] Gu, W., Vetter, J., Schwan, K., “An Annotated Bibliography of Interactive Program Steering”, ACM SIGPLAN Notices, Vol. 29, No. 9, pp. 140-148 (September 1994).

[Gust 98] Gustafson, J., “Computational Verifiability and Feasibility of the ASCI Program”, IEEE Computational Science & Engineering, Vol. 5, No. 1 (1998).

[HaWe 88] Haban, D., Weigel, W., “Global Events and Global Breakpoints in Distributed Systems”, Proc. HICSS ’21, 21st Annual Hawaii Intl. Conference on System Sciences, Hawaii, USA, pp. 166-175 (January 1988).

[HaWy 90] Haban, D., Wybranietz, D., “A Hybrid Monitor for Behavior and Performance Analysis of Distributed Systems”, IEEE Transactions on Software Engineering, Vol. 16, No. 2, pp. 197-211 (February 1990).

[HaMa 94] Hackstadt, St.T., Malony, A.D., “Next-Generation Parallel Performance Visualization: A Prototyping Environment for Visualization Development”, Proc. PARLE ’94, Parallel Architectures and Languages Europe Conference, Athens, Greece, pp. 192-201 (July 1994).

[HaGa 97] Hamilton, S., Garber, L., “Deep Blue’s Hardware-Software Synergy”, IEEE Computer, Vol. 30, No. 10, pp. 29-35 (Oct. 1997).

[Hao 93] Hao, M.C., Karp, A.H., Mackey, M., Singh, V., Chien, J., “On-the-Fly Visualization and Debugging of Parallel Programs”, Hewlett Packard, HP Laboratories Technical Reports, HPL-93-95, http://www.hpl.hp.com/techreports/93/HPL-93-95.html, Palo Alto, USA (October 1993).

[Haye 88] Hayes, J.P., “Computer Architecture and Organization”, 2nd International Edition, McGraw-Hill Book Company, Singapore (1988)

[Heat 93] Heath, M.T., “Recent Developments and Case Studies in Performance Visualization using ParaGraph”, in: Haring, G., Kotsis, G., (Eds.), “Performance Measurement and Visualization of Parallel Systems”, Proc. of the Workshop on Performance Measurement and Visualization Moravany, Czechoslovakia (October 1992), Elsevier Science Publishers, Amsterdam, pp. 175-200 (1993).

[HeEt 91a] Heath, M.T., Etheridge, J.A., “ParaGraph: A Tool for Visualizing Performance of Parallel Programs”, Technical Report ORNL/TM-11813, Oak Ridge National Laboratories, Oak Ridge Tennessee, USA (May 1991).

[HeEt 91b] Heath, M.T., Etheridge, J.A., “Visualizing the Performance of Parallel Programs”, IEEE Software, Vol. 8, No. 5, pp. 29-39 (September 1991).

[Heat 95a] Heath, M.T., Malony, A.D., Rover, D.T., “The Visual Display of Parallel Performance Data”, IEEE Computer, Vol. 28, No. 11, pp. 21-28 (November 1995).

[Heat 95b] Heath, M.T., Malony, A.D., Rover, D.T., “Parallel Performance Visualization: From Practice to Theory”, IEEE Parallel & Distributed Technology, Vol. 3, No. 4, pp. 44-60 (Winter 1995).

[HeMc 96] Helmbold, D.P., McDowell, C.E., “Race Detection - Ten Years Later”, in: Simmons, M.L., Hayes, A.H., Brown, J.S., Reed, D.A., “Debugging and Performance Tuning for Parallel Computing Systems”, IEEE Computer Society Press, Los Alamitos, CA, USA, pp. 101-126 (1996).

[Helm 90] Helmbold, D.P., McDowell, C.E., Wang, J.-Z., “Detecting Data Races by Analyzing Sequential Traces”, Technical Report UCSC-CRL-90-57, Board of Studies in Computer and Information Sciences, University of California, Santa Cruz, CA, USA (Oct. 1990),
reprinted in: Proc. HICCS-24, Hawaii Intl. Conference on System Sciences, Vol. 2, Hawaii, USA, pp. 408-417 (January 1991).

[HePa 96] Hennessy, J.L., Patterson, D.A., “Computer Architecture: A Quantitative Approach”, 2nd Edition, Morgan Kaufmann Publishers, San Francisco, CA, USA (1996).

[HeLu 91] Herrarte, V., Lusk, E., “Studying parallel program behavior with Upshot”, Technical Report ANL-91/15, Argonne National Laboratory, IL, USA (August 1991).

[Hita 99] Hitachi, Ltd., “Super Technical Server Hitachi SR8000”, http://www.hitachi.co.jp/Prod/comp/hpc (1999).

[Hofm 87] Hofmann, R., Klar, R., Luttenberger, N., Mohr, B., “Zählmonitor 4: Ein Monitorsystem für das Hardware- und Hybridmonitoring von Multiprozessor- und Multicomputer-Systemen”, Proc. of the

4th GI/ITG-Fachtagung, “Messung, Modellierung und Bewertung von Rechnersystemen”, Springer-Verlag, Erlangen (September/October 1987) [in German]

[Hofm 93] Hofmann, R., “The Distributed Hardware Monitor ZM4 and Its Interface to MEMSY”, in: Bode, A., Cin, M.D., (Eds.), “Parallel Computer Architectures: Theory, Hardware, Software, Applications”, Springer-Verlag, Lecture Notes in Computer Science, Vol. 732, Berlin, pp. 66-79 (March 1993).

[Holl 59] Holland, J., “A Universal Computer Capable of Executing an Arbitrary Number of Subprograms Simultaneously”, Proc. East Joint Computer Conference, Vol. 16, pp. 108-113 (1959).

[Holl 94] Hollingsworth, J.K., Miller, B.P., Cargille, J., “Dynamic Program Instrumentation for Scalable Performance Tools”, Proc. SHPCC, 1994 Scalable High Performance Computing Conference, Knoxville, TN, USA, pp. 841-850 (May 1994).

[Holl 95] Hollingsworth, J.K., Miller, B.P., Lumpp, J.E. Jr., “Techniques for Performance Measurement of Parallel Programs”, in: Casavant, Th.L., Tvrdik, P., Plail, F., “Parallel Computers: Theory and Practices”, IEEE Computer Society Press (1995).

[HoMi 96] Hollingsworth, J.K., Miller, B.P., “An Adaptive Cost System for Parallel Program Instrumentation”, in: Bouge, L., Fraigniaud, P., Mignotte, A., Robert, Y., (Eds.), Proceedings of EuroPar ’96, Lecture Notes in Computer Science, Vol. 1123, Springer-Verlag Lyon, France, pp. 88-97 (August 1996).

[HoMi 99] Hollingsworth, J.K., Miller, B.P., “Instrumentation and Measurement”, in: Foster, I., Kesselman, C., (Eds.), “The Grid: Blueprint for a New Computing Infrastructure”, Morgan Kaufmann Publishers, San Francisco, California, USA, Chapter 14, pp. 339-365 (1999).

[Hond 95] Hondroudakis, A., “Performance Analysis Tools for Parallel Programs”, Version 1.0.1, Edinburgh Parallel Computing Centre, The University of Edinburgh, available at: http://www.epcc.ed.ac.uk/epcc-tec/documents.html (July 1995).

[Hood 96] Hood, R., “The p2d2 Project: Building a Portable Distributed Debugger”, Proc. SPDT’96, ACM SIGMETRICS Symp. on Parallel and Distr. Tools, Philadelphia, USA, pp. 127-136 (May 1996).

[Hood 90] Hood, R., Kennedy, K., Mellor-Crummey, J., “Parallel Program Debugging with On-the-fly Anomaly Detection”, Technical Report TR90-111, Dept. of Computer Science, Rice University (May 1990),
reprinted in: Proc. Supercomputing ‘90, New York, NY, USA, pp. 74-81 (Nov. 1990).

[HoCu 88] Hough, A.E., Cuny, J.E., “Initial Experiences with a Pattern-Oriented Parallel Debugger”, Proceedings of the ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging, May 5-6, 1988, University of Wisconsin, Madison, Wisconsin, USA, SIGPLAN Notices, Vol. 24, No. 1, pp. 195-205 (January 1989).

[HoCu 90] Hough, A.E., Cuny, J.E., “Perspective Views: A Technique for Enhancing Parallel Program Visualization”, Proc. ICPP ’90, 1990 International Conference on Parallel Processing, Vol. II, pp. 124-132 (August 1990).

[Hoss 99] Hossfeld, F., “Teraflops Computing: A Challenge to Parallel Numerics”, in: P. Zinterhof, M. Vajtersic, A. Uhl, (Eds.), “Parallel Computation”, Proc. 4th Intl. ACPC Conference, Lecture Notes in Computer Science, Vol. 1557, Springer-Verlag, Salzburg, Austria, pp. 1-12 (Feb. 1999).

[Hwan 93] Hwang, K., “Advanced Computer Architecture: Parallelism, Scalability, Programmability”, McGraw-Hill, Intl. (1993).

[HwXu 98] Hwang, K., and Xu, Z., “Scalable Parallel Computing - Technology, Architecture, Programming”, WCB McGraw-Hill, USA (1998).

[Hyat 99] Hyatt, M.S., “The Millennium Bug”, Broadway Books (April 1999).

[Hyrs 95] Hyrskykari, A., “Development of Program Visualization Systems”, Proc. VAMMS'93, 2nd Czech British Symposium of Visual Aspects of Man-Machine Systems, Praha (1993),
reprinted in: Technical Report A-1995-3, Department of Computer Science, University of Tampere, Finland, available at: ftp://ftp.cs.uta.fi/pub/reports/pdf/A-1995-3.pdf (1995).

[Jaki 95] Jakiela, H., “Performance Visualization of a Distributed System: A Case Study”, IEEE Computer, Vol. 28, No. 11, pp. 30-36 (November 1995).

[JeRe 92] Jensen, D.W., Reed, D.A., “A Performance Analysis Exemplar: Parallel Ray Tracing”, in: Concurrency - Practice and Experience, Vol. 4, No. 2, pp. 119-141 (April 1992).

[Jerd 97] Jerding, D., Stasko, J.T., Ball, T., “Visualizing Interactions in Program Executions”, Proc. ICSE-97, 1997 International Conference on Software Engineering, Boston, Massachusetts, USA, pp. 360-370 (May 1997).

[Joch 95] Joch, A., “How Software Doesn’t Work - Nine Ways to Make Your Code More Reliable”, BYTE, pp. 49-60 (Dec. 1995).

[Jone 96] Jones, C., “Software Estimating Rules of Thumb”, IEEE Computer, Vol. 29, No. 3, pp. 117-118 (March 1996).

[Joyc 87] Joyce, J., Lomov, G., Slind, K., Unger, B., “Monitoring Distributed Systems”, ACM Transactions on Computer Systems, Vol. 5, No. 2, pp. 121-150 (May 1987).

[JuMc 00] Jun, Y.-K., McDowell, Ch.E., “Scalable Monitoring Technique for Detecting Races in Parallel Programs”, in: Rolim, J., et al (Eds.), “Parallel and Distributed Processing”, Proc. HIPS 2000, 5th IEEE International Workshop on High-Level Parallel Programming Models and Supportive Environments, in conjunction with IPDPS 2000, Lecture Notes in Computer Science, Springer-Verlag, Vol. 1800, Cancun, Mexico, pp. 340-347 (May 2000).

[Kacs 97] Kacsuk, P., Cunha, J.C., Dozsa, G., Lourenco, J., Fadgyas, T., Antao, T., “A Graphical Development and Debugging Environment for Parallel Programs”, Journal of Parallel Computing, Haring, G., Kacsuk, P., Kotsis, G., (Eds.), “Distributed and Parallel Systems: Environments and Tools”, Elsevier Publisher, Vol. 22, No. 13, pp. 1699-1701 (1997).

[Kacs 98] Kacsuk, P., “Systematic Testing and Debugging of Parallel Programs by a Macrostep Debugger”, Proc. DAPSYS ’98, 1998 Workshop on Distributed and Parallel Systems, Budapest, Hungary, pp. 105-112 (1998).

[Kacs 99] Kacsuk, P., “Systematic Debugging of Parallel Programs Based on Collective Breakpoints”, Proc. of the International Symposium on Software Engineering for Parallel and Distributed Systems, Los Angeles, California, USA, pp. 83-96 (1999).

[Kacs 00] Kacsuk, P., “Systematic Macrostep Debugging of Message Passing Parallel Programs”, in: Kacsuk, P., Kotsis, G., “Distributed and Parallel Systems (DAPSYS’98)”, Future Generation Computer Systems, North-Holland, Vol. 16, No. 6, pp. 597-607 (April 2000).

[Kamk 93] Kamkar, M., Shahmehri, N., Fritzson, P., “Three Approaches to Interprocedural Slicing”, Proc. EUROMICRO ’93, Journal of Microprocessing and Microprogramming, North-Holland, Vol. 38, Nos. 1-5 (Sept. 1993).

[Kamk 96] Kamkar, M., Krajina, P., Fritzson, P., “Dynamic Slicing of Parallel Message-Passing Programs”, Proc. EUROMICRO PDP ’96, 4th EUROMICRO Workshop on Parallel and Distributed Processing, IEEE Computer Society Press, Braga, Portugal, pp. 170-177 (January 1996).

[Karp 87] Karp, A.H., “Programming for Parallelism”, IEEE Computer, Vol. 20, No. 5, pp. 43-57 (May 1987).

[KaRa 87] Karp, R.M., Rabin, M.O., “Efficient Randomized Pattern-Matching Algorithms”, IBM Journal of Research and Development, Vol. 31, No. 2, pp. 249-260 (1987).

[KeRi 88] Kernighan, B.W., Ritchie, D.M., “The C Programming Language, Second Edition, ANSI C”, Prentice-Hall Intl., Englewood Cliffs, NJ, USA (1988).

[Kess 96] Kessler, C.W., “Pattern-Driven Automatic Parallelization”, Journal Scientific Programming, Vol. 5, pp. 256-274 (1996).

[KiCh 97] Kilgore, R., Chase, C., “Re-execution of Distributed Programs to Detect Bugs Hidden by Racing Messages”, Proc. HICSS ’30, 30th Annual Hawaii Intl. Conference on System Sciences, Vol. 1, pp. 423-432 (January 1997).

[KiPa 93] Kim, J.L., Park, T., “An Efficient Protocol for Checkpointing Recovery in Distributed Systems”, IEEE Transactions on Parallel and Distributed Systems, Vol. 4 (8), S. 955 - 960 (Aug. 1993).

[Kime 95] Kimelman, D., Leban, B. Roth, T., Zernik, D., “Dynamic Graph Abstraction for Effective Software Visualization”, in: Eades, P., Zhang, K., “Software Visualization”, Special Issue, The Australian Computer Journal, Vol. 27, No. 4, pp. 129-137 (November 1995).

[Klar 93] Klar, R., “Event-Driven Monitoring of Parallel Systems”, in: Haring, G., Kotsis, G., (Eds.), “Performance Measurement and Visualization of Parallel Systems”, Proc. of the Workshop on Performance Measurement and Visualization Moravany, Czechoslovakia (October 1992), Elsevier Science Publishers, Amsterdam, NL, pp. 145-173 (1993)

[Klar 95] Klar, R., Dauphin, P., Hartleb, F., Hofmann, R., Mohr, B., Quick, A., Siegle, M., "Messung und Modellierung paralleler und verteilter Rechensysteme", B.G. Teubner, Stuttgart, Germany (1995) [in German].

[KoWi 93] Kohn, J., Williams, W., "ATExpert", Journal of Parallel and Distributed Computing, Vol. 18, No. 2, pp. 205-222 (June 1993).

[Kopp 97] Koppler, R., Grabner, S., Volkert, J., "Visualization of Distributed Data Structures for HPF-like Languages", in: Scientific Programming, Special Issue: "High Performance Fortran Comes of Age", Vol. 6, No. 1, pp. 115-126 (Spring 1997).

[KoLa 90] Korel, B., Laski, J., "Dynamic Slicing of Computer Programs", The Journal of Systems and Software (1990).

[KoFe 92] Korel, B., Fergusan, R., "Dynamic Slicing of Distributed Programs", Journal of Applied Mathematics and Computer Science, Vol. 2, No. 2, pp. 199-215 (1992).

[Knut 77] Knuth, D.E., Morris, J.H. Jr., Pratt, V.R., "Fast Pattern Matching in Strings", SIAM Journal of Computing, Vol. 6, No. 2, pp. 323-350 (1977).

[KrSt 93] Kraemer, E. and Stasko, J.T., "The Visualization of Parallel Systems: An Overview", Journal of Parallel and Distributed Computing, Vol. 18, pp. 105-117 (June 1993).

[KrSt 94] Kraemer, E., Stasko, J.T., "Toward Flexible Control of the Temporal Mapping from Concurrent Program Events to Animations", Proc. 8th Int. Parallel Processing Symposium (IPPS'94), Cancun, Mexico, pp. 902-908 (Apr. 1994).

[Kran 94] Kranzlmüller, D., Grabner, S., Volkert, J., "Combining the Benefits of Error-Debugging and Performance Evaluation", Proc. PCAT '94, Parallel Computing and Transputers Conference, Wollongong, Australia, pp. 79-88 (November 1994).

[Kran 95a] Kranzlmüller, D., Grabner, S., Volkert, J., "Race Condition Detection with the MAD Environment", Proc. of PART'95, Second Australasian Conference on Parallel and Real-Time Systems, Fremantle, Western Australia, pp. 160-166 (September 1995).

[Kran 95b] Kranzlmüller, D., Grabner, S., Volkert, J., "Message Passing Visualization with ATEMPT", in: D'Hollanger, E.H., Joubert, G.R., Peters, F.J., Trystram, D., "Parallel Computing: State-of-the-Art and Perspectives", Elsevier Science B.V., Proc. ParCo 95, Conference on Parallel Computing, Gent, Belgium, pp. 653-656 (September 1995).

[Kran 96a] Kranzlmüller, D., Grabner, S., J. Volkert, J., "Monitoring Strategies for Hypercube Systems", Proc. EUROMICRO PDP '96, 4th EUROMICRO Workshop on Parallel and Distributed Processing, IEEE Computer Society Press, Braga, Portugal, pp. 486-492 (January 1996).

[Kran 96b] Kranzlmüller, D., Grabner, S., Volkert, J., "Event Graph Visualization for Debugging Large Applications", Proc. SPDT'96, ACM SIGMETRICS Symposium on Parallel and Distributed Tools, Philadelphia, PA, USA, pp. 108-117 (May 1996).

[Kran 96c] Kranzlmüller, D., Koppler, R., Holzner, Ch., Grabner, S., Volkert, J., "Parallel Program Visualization with MUCH", Proc. ACPC '96, Conference of the Austrian Center for Parallel Computation, Springer-Verlag, Lecture Notes in Computer Science, Vol. 1127, Klagenfurt, Austria, pp. 148-160 (September 1996).

[Kran 96d] Kranzlmüller, D., Christanell, A., Grabner, S., Volkert, J., "Control and Data Flow Analysis for Parallel Program Debugging", in: Palma, J.M.L.M, Dongarra, J.J., (Eds.), "Vector and Parallel Processing - VECPAR'96 - Selected Papers", Springer Verlag, Lecture Notes in Computer Science, Vol. 1215, 2nd Intl. Conf. on Vector and Parallel Processing - Systems and Applications, Porto, Portugal, pp. 81-95 (September 1996).

[Kran 96e] Kranzlmüller, D., Grabner, S., Volkert, J., "Program Analysis Through Visualization", Eades, P., Zhang, K., (Eds.), "Software Visualization", Series on Software Engineering and Knowledge Engineering, Vol. 7, World Scientific Publishing, Singapore, pp. 183-202 (1996).

[Kran 97a] Kranzlmüller, D., Grabner, S., Volkert, J., "Debugging with the MAD Environment", Journal of Parallel Computing, Dongarra, J.J., Tourancheau, B., (Eds.), "Environments and Tools for Parallel Scientific Computing III", Elsevier Publisher, Vol. 23, No. 1-2, pp. 199-217 (April 1997).

[Kran 97b] Kranzlmüller, D., Grabner, S., Volkert, J., "Using Control and Data Flow Analysis for Race Evaluation", Lengauer, Ch., Griebl, M. Gorlatch, S., (Eds.), "Euro-Par'97 Parallel Processing", Proceed-

ings 3rd Intl. Euro-Par Conference, Lecture Notes in Computer Science, Vol. 1300, Springer-Verlag, Passau, Germany, pp. 102-109 (August 1999).

[Kran 97c] Kranzlmüller, D., Christanell, A., Volkert, J., "Supporting Parallel Program Debugging through Control and Data Flow Analysis", in: D'Hollander, E.H., Joubert, G.R., Peters, F.J., Trottenberg, U., "Parallel Computing: Fundamentals, Applications and New Directions", Proc. ParCo '97, Conference on Parallel Computing, Bonn, Germany, pp. 303-310 (September 1997).

[KrVo 98] Kranzlmüller, D., Volkert, J., "Debugging Point-To-Point Communication in MPI and PVM", Proc. EUROPVM/MPI 98 Intl. Conference, Liverpool, GB, Lecture Notes in Computer Science, Vol. 1497, Springer-Verlag, pp. 265-272 (Sept. 1998).

[Kran 99a] Kranzlmüller, D., Messner, J., Schaubschläger, Ch., "'Good Enough' Testing of Nondeterministic Parallel Programs", Proc. of PDPTA '99, Intl. Conference on Parallel And Distributed Processing Techniques and Applications (PDPTA '99), Arabnia, H.R. (Ed.), CSREA Press, Vol. 2, Las Vegas, Nevada, USA, pp. 641-645 (June 1999).

[Kran 99b] Kranzlmüller, D., Chassin de Kergommeaux, J., Schaubschläger, Ch., "Correction of Monitor Intrusion for Testing Nondeterministic MPI Programs", Amestoy, P., Berger, P., Dayde, M., Duff, I., Fraysse, V., Giraud, L., Ruiz, D., (Eds.), "Euro-Par'99 Parallel Processing", Proceedings 5th Intl. Euro-Par Conference, Lecture Notes in Computer Science, Vol. 1685, Springer-Verlag, Toulouse, France, pp. 154-158 (Aug./Sept. 1999).

[Kran 99c] Kranzlmüller, D., Reussner, R., Schaubschläger, Ch., "Monitor Overhead Measurement with SKaMPI", Proc. EUROPVM/MPI 99 Intl. Conference, Barcelona, Spain, Lecture Notes in Computer Science, Vol. 1697, Springer-Verlag, pp. 43-50 (September 1999).

[KrVo 99] Kranzlmüller, D., Volkert, J., "NOPE: A Nondeterministic Program Evaluator", in: Zinterhof, P., Vajtersic, M., Uhl, A. (Eds.), "Parallel Computation", Proc. of ACPC '99, 4th Intl. ACPC Conference, Lecture Notes in Computer Science, Vol. 1557, Springer, Salzburg, pp. 490-499 (Feb. 1999).

[KrWi 96] Krawczyk, H., Wiszniewski, B., "Interactive Testing Tool for Parallel Programs", in: Jelly, I., Gorton, I., Croll, P. (Eds.), "Software Engineering for Parallel and Distributed Systems", Chapman Hall, London, UK, pp. 98-109 (1996).

[KrWi 98] Krawczyk, H., Wiszniewski, B., "Analysis and Testing of Distributed Software Applications", in: Wilson, D.R., (Ed.), "C3 - Industrial Control, Computers, and Communication Series", Research Studies Press Ltd., Baldock, Hertfordshire, England (1998).

[Kraw 00] Krawczyk, H., Krysztop, B., Proficz, J., "Suitability of the Time Controlled Environment for Race Detection in Distributed Applications", in: Kacsuk, P., Kotsis, G., "Distributed and Parallel Systems (DAPSYS'98)", Future Generation Computer Systems, North-Holland, Vol. 16, No. 6, pp. 625-635 (April 2000).

[Kunz 94] Kunz, T., "Reverse Engineering Distributed Applications: An Event Abstraction Tool", International Journal of Software Engineering and Knowledge Engineering, Vol. 4, No. 3, pp. 303-323 (September 1994).

[Kunz 95] Kunz, T., "High-Level Views of Distributed Executions", Proc. AADEBUG '95, 2nd International Workshop on Automated and Algorithmic Debugging, Saint Malo, France, pp. 205-219 (May 1995).

[KuBl 94] Kunz, T., Black, J.P., "Understanding the Behaviour of Distributed Applications through Reverse-Engineering", Distributed Systems Engineering Journal, Vol. 1, No. 6, pp. 345-353 (December 1994).

[KuBl 95] Kunz, T., Black, J.P., "Using Automatic Process Clustering for Design Recovery and Distributed Debugging", IEEE Transactions on Software Engineering, Vol. 21, No. 6, pp. 515-527 (June 1995).

[Kunz 97] Kunz, T., Taylor, D.J., Black, J.P., "POET: Target-System-Independent Visualizations of Complex Distributed Executions", Proc. HICSS '30, 30th Hawaii International Conference on System Sciences, Hawaii, USA, Vol. I, pp. 452-461 (January 1997).

[Ladk 98] Ladkin, P., "Fallstricke auf dem Weg ins All", C'T, No. 19, pp. 158-159 (1998) [in German].

[Ladk 99] Ladkin, P., et al., "Computer-Related Incidents with Commercial Aircraft - A Compendium of Resource, Reports, Research, Discussion and Commentary", http://www.rvs.uni-bielefeld.de/publications/Incidents/ (1999).

[LaLe 95] Ladkin, P.B., Leue, S., “Interpreting Message Flow Graphs”, Formal Aspects of Computing, BCS, Vol. 7, No. 5, pp. 473-509 (1995).

[Lamp 78] Lamport, L., “Time, Clocks, and the Ordering of Events in a Distributed System”, Communications of the ACM, pp. 558 - 565 (July 1978).

[LaSc 96] Larus, J.R., Schnarr, E., “EEL: Machine-Independent Executable Editing”, Proc. PLDI 1995, ACM SIGPLAN’95 Conference on Programming Language Design and Implementation, ACM SIGPLAN Notices, Vol. 30, No. 6, La Jolla, California, USA, pp. 291-300 (June 1995).

[LaLe 97] Laudon, J., Lenoski, D., “The SGI Origin: A ccNUMA Highly Scalable Server”, Proc. of the 24th Intl. Symp. on Computer Architecture, pp. 241-251, http://www.sgi.com/origin/images/isca.pdf (June 1997).

[LeMe 87] LeBlanc, T.J., Mellor-Crummey, J.M., “Debugging Parallel Programs with Instant Replay”, IEEE Transactions on Computers, Vol. C-36, No. 4, pp. 471-481 (April 1987).

[LeBl 90] LeBlanc, T.J., Mellor-Crummey, J.M., Fowler, R.J., “Analyzing Parallel Program Executions Using Multiple Views”, Journal of Parallel and Distributed Computing, Vol. 9, No. 2, pp. 203-217 (June 1990).

[LeRo 85] LeBlanc, T.J., Robbins, A.D., “Event Driven Monitoring of Distributed Programs”, Proceedings 5th Intl. Conference on Distributed Computing Systems, IEEE Computer Society Press, Denver, CO, USA, pp. 515-522 (May 1985).

[LePa 85] LeDoux, C.H., Parker, D.S., Jr., “Saving Traces for Ada Debugging”, in: “Ada in Use”, Proc. of the ACM Ada International Conference, Cambridge University Press, Cambridge, UK, pp. 97-108 (1985).

[LeKu 99] Leinberger, W., Kumar, V., “Information Power Grid: The New Frontier in Parallel Computing?”, IEEE Concurrency, Vol. 7, No. 4, pp. 75-84 (October-December 1999).

[LeRu 93] Lerman, G., Rudolph, L., “Parallel Evolution of Parallel Processors”, Plenum Press, New York (1993).

[Leu 90] Leu, E., Schiper, A., and Zramdini, A., “Execution Replay on Distributed Memory Architectures”, Proc. 2nd IEEE Symposium on Parallel & Distributed Processing, Dallas, TX, USA, pp. 106-112 (Dec. 1990).

[LeSc 92] Leu, E., Schiper, A., “Execution Replay: A Mechanism for Integrating a Visualization Tool with a Symbolic Debugger”, in: Roberts, Y., Bouge, L., Cosnard, M., Trystram, D., (Eds.), Proc. CONPAR 92 - VAPP V, Lecture Notes in Computer Science, Vol. 634, Springer-Verlag (1992).

[LeSc 93] Leu, E., Schiper, A., “ParaRex: A Programming Environment Integrating Execution Replay and Visualization”, Dongarra, J.J., Tourancheau, B., (Eds.), “Environments and Tools for Parallel Scientific Computing”, North-Holland, Amsterdam, pp. 155-169 (1993).

[Levr 94] Levrouw, L.J., Audenaert, K.M., Van Campenhout, J.M., “A New Trace and Replay System for Shared Memory Programs Based on Lamport Clocks”, Proceedings EUROMICRO PDP ’94, 2nd EUROMICRO Workshop on Parallel and Distributed Processing, IEEE Computer Society Press, Malaga, Spain, pp. 471-478 (January 1994).

[LeAu 94] Levrouw, L.J., Audenaert, K.M., “Minimizing the Log Size for Execution Replay of Shared-Memory Programs”, in: Buchberger, B., Volkert, J. (Eds.), “Parallel Processing”, Proc. CONPAR 94 - VAPP VI, Third Joint International Conference on Vector and Parallel Processing, Springer Verlag, Lecture Notes in Computer Science, Vol. 854, Linz, Austria, pp. 76-87 (September 1994).

[Lewi 93] Lewis, T.G., “Foundations of Parallel Programming - A Machine-Independent Approach”, IEEE Computer Society Press, Los Alamitos, CA, USA (1993).

[Lewi 98] Lewis, T., “Joe Sixpack, Larry Lemming, and Ralph Nader”, IEEE Computer, Vol. 31, No. 7, pp. 107-109 (July 1998).

[Lewi 99] Lewis, T., “The Open Source Acid Test”, IEEE Computer, Vol. 32, No. 2, pp. 125-128 (Feb. 1999).

[Lieb 97] Lieberman, H., “The Debugging Scandal and What to Do About It”, Communications of the ACM, Vol. 40, No. 4, pp. 27-29 (April 1997).

[LiZh 95] Li, K.-C., Zhang, K., “Instrumenting Parallel Programs Based on a Logical Clock Approach”, in: D’Hollander, E.H., Joubert, G.R., Peters, F.J., Trystram, D., “Parallel Computing: State-of-the-Art

and Perspectives", Elsevier Science B.V., Proc. ParCo 95, Conference on Parallel Computing, Gent, Belgium, pp. 513-520 (September 1995).

[Lons 94] Lonsdale, G., Clinckemaillie, J., Vlachoutsis, S., Dobois, J., "Communication Requirements in Parallel Crashworthiness Simulation", in: Gentzsch, W., Harms, U., (Eds.), "High Performance Computing and Networking Europe 1994", Proc. HPCN Europe '94, Lecture Notes in Computer Science, Vol. 796, Springer-Verlag, Munich, Germany, pp. 55-61 (April 1994).

[Lons 95] Lonsdale, G., Elsner, B., Clinckemaillie, J., Vlachoutsis, S., de Bruyne, F., Holzner, M., "Experiences with Industrial Crashworthiness Simulation using the portable, message-passing PAM-CRASH Code", in: Hertzberger, G., Serazzi, G., (Eds.), "High Performance Computing and Networking Europe 1995", Proc. HPCN Europe '95, Lecture Notes in Computer Science, Vol. 919, Springer-Verlag, Milan, Italy, pp. 856-862 (May 1995).

[Lour 97] Lourenco, J., Cunha, J.C. Krawczyk, H., Kuzora, P., Neyman, M., Wiszniewski, B., "An Integrated Testing and Debugging Environment for Parallel and Distributed Programs", Proc. EUROMICRO '97, 23rd Euromicro Conference, Budapest, Hungary (September 1997).

[LuWi 97] Ludwig, T., Wismüller, R., "OMIS 2.0 -- A Universal Interface for Monitoring Systems", in: Bubak, M., Dongarra, J.J., Wasniewski, J., (Eds.) "Recent Advances in Parallel Virtual Machine and Message Passing Interface", Proc. 4th European PVM/MPI Users' Group Meeting, Springer-Verlag, Lecture Notes in Computer Science, Vol. 1332, Crakow, Poland, pp. 267-276 (November 1997).

[Ludw 96] Ludwig, T., Wismüller, R., Sunderam, V.S., Bode, A., "OMIS -- On-line Monitoring Interface Specification", Technical Report TUM-I9609, SFB-Bericht Nr. 342/05/96 A, Technical University Munich, Germany (February 1996)
latest revision: "OMIS -- On-line Monitoring Interface Specification (Version 2.0)", Technical Report TUM-I9733, SFB-Bericht Nr. 342/22/97 A, Technical University Munich, Germany, //http://wwwbode.informatik.tu-muenchen.de/~omis/ (July 1997).

[LuCu 96] Lumetta, S.S., Culler, D.E., "The Mantis Parallel Debugger", Proc. of SPDT'96: SIGMETRICS Symposium on Parallel and Distributed Tools, Philadelphia, PA, pp. 118-126 (May 1996).

[Lump 90] Lumpp, J.E. jr., Siegel, H.J., Marinescu, D.C., "Specification and Identification of Events for Debugging and Performance Monitoring of Distributed Multiprocessor Systems", Proc. 10th Intl. Conference on Distributed Systems, IEEE Computer Society Press, pp. 476-483 (June 1990).

[Mack 93] Mackey, M., "Program Replay in PVM", Technical Report, Concurrent Computing Department, Hewlett-Packard Laboratories (May 1993).

[Mail 95] Maillet, E., "Issues in Performance Tracing with Tape/PVM", Proceedings of EuroPVM'95, pp. 143-148 (1995).

[Mail 96] Maillet, E., "Le Traçage Logiciel d'Applications Parallèles: Conception et Ajustement de Qualité", PhD thesis, INPG, University of Grenoble, France (Sept. 1996) [in French].

[MaTr 95] Maillet, E., Tron, C., "On Efficiently Implementing Global Time for Performance Evaluation on Multiprocessor Systems", Journal of Parallel and Distributed Computing, Vol. 28, pp. 84-93 (July 1995).

[Malo 89] Malony, A.D., "Multiprocessor Instrumentation: Approaches for Cedar", in: Simmons, M., Koskela, R., Bucher, I., "Instrumentation for Future Parallel Computing Systems", Frontier Series, ACM Press, Addison-Wesley, Chapter 1, pp. 1-33 (1989).

[Malo 91a] Malony, A.D., "Event Based Performance Perturbation: A Case Study", Proc. of the 3rd ACM SIGPLAN Symposium on Principles & Practise of Parallel Programming PPOPP, Williamsburg, Virginia, USA (Apr. 1991),
reprinted in: ACM SIGPLAN Notices, Vol. 26, No. 7, pp. 201-212 (July 1991).

[Malo 91b] Malony, A.D., Hammerslag, D.H., Jablonowski, D.J., "TraceView: A Trace Visualization Tool", IEEE Software, Vol. 8, No. 5, pp. 19-28 (Sept. 1991).

[MaRe 89] Malony, A.D., Reed, D.A., "Visualizing Parallel Computer System Performance", in: Simmons, M., Koskela, R., Bucher, I., "Instrumentation for Future Parallel Computing Systems", Frontier Series, ACM Press, Addison-Wesley, Chapter 4, pp. 59-90 (1989).

[MaRe 90] Malony, A.D., Reed, D.A., "A Hardware-Based Performance Monitor for the Intel iPSC/2 Hypercube", ACM Computer Architecture News, Vol. 18, No. 3, pp. 213-226 (September 1990).

[MaRe 91] Malony, A.D., Reed, D.A., “Models for performance perturbation analysis”, Proc. Workshop Parallel Distributed Debugging, ACM SIGPLAN/SIGOPS and Office of Naval Research (May 1991).

[Malo 92] Malony, A.D., Reed, D.A., Wijshoff, H., “Performance Measurement Intrusion and Perturbation Analysis”, IEEE Transactions on Parallel And Distributed Systems, Vol. 3, No. 4, pp. 433-450 (July 1992).

[MaWi 93] Malony, A.D., Wilson, G.V., “Future Directions in Parallel Performance Environments”, in: Haring, G., Kotsis, G., (Eds.), “Performance Measurement and Visualization of Parallel Systems”, Proc. of the Workshop on Performance Measurement and Visualization Moravany, Czechoslovakia (October 1992), Elsevier Science Publishers, Amsterdam, NL, pp. 331-351 (1993).

[Mari 90] Marinescu, D.C., Lumpp, J.E. jr., Casavant, T.L., “Models for Monitoring and Debugging Tools for Parallel and Distributed Software”, Journal of Parallel and Distributed Computing 9, No. 2, pp. 171-184 (June 1990)

[MaJa 97] Martin, J.M.R., Jassim, S.A., “A Tool for Providing Deadlock Freedom”, in: “Parallel Programming and Java”, Proceedings of the 20th World Occam and Transputer User Group Technical Meeting, IOS Press (1997).

[Matt 88] Mattern, F., “Virtual Time and Global States of Distributed Systems”, in: Parallel and Distributed Algorithms, North-Holland, pp. 215-226 (October 1988).

[Matt 92] Mattern, F., “On the Relativistic Structure of Logical Time in Distributed System”, in: Bigre 78 - Sommaire, C3: Theme “Systemes et Environnements”, pp. 3-20 (March 1992).

[MaBe 93] May, J., Berman, F., “Panorama: A Portable, Extensible Parallel Debugger”, Proc. 3rd ACM/ONR Workshop on Parallel and Distributed Debugging, San Diego, CA, USA (May 1993),
reprinted in: ACM SIGPLAN Notices, Vol. 28, No. 12, pp. 96-106 (Dec. 1993).

[McHe 89] McDowell, Ch.E., Helmbold, D.P., “Debugging Concurrent Programs”, ACM Computing Surveys, Vol. 21, No. 4, pp. 593-622 (December 1989).

[MiTa 95] Miei, T., Takahashi, N., “XdbxR/pvm: A PVM Program Debugger with Demand-Driven Replay”, Dongarra, J.J., Gengler, M., Tourancheau, B., Vigouroux, X., (Eds.), Proceedings EuroPVM’95 (1995).

[Mill 93] Miller, B.P., “What to Draw? When to Draw? - An Essay on Parallel Program Visualization”, Technical Report CS-TR-92-1103, Computer Science Department, University of Wisconsin, Madison, USA (July 1992)
reprinted in: Journal of Parallel and Distributed Computing, Vol. 18, No. 2, pp. 265-269 (June 1993).

[MiCh 88] Miller, B.P., Choi, J.-D., “A Mechanism for Efficient Debugging of Parallel Programs”, Proc. PLDI’88, ACM SIGPLAN’88 Conference on Programming Language Design and Implementation, Atlanta, Georgia, pp. 135-153 (June 1988)
reprinted in: ACM SIGPLAN Notices, Vol. 23, No. 7 (July 1988).

[Mill 86] Miller, B.P., Macrander, C., Sechrest, S., “A Distributed Programs Monitor for Berkeley UNIX”, Software - Practice and Experience, Vol. 16, No. 2, pp. 183-200 (February 1986).

[Mill 90a] Miller, B.P., Clark, M., Hollingsworth, J.K., Kierstead, St., Lim, S.S., Torzewski, T., “IPS-2: The Second Generation of a Parallel Program Measurement System”, IEEE Transactions on Parallel and Distributed Systems, Vol. 1, No. 2, pp. 206-217 (April 1990).

[Mill 90b] Miller, B.P., Fredrikson, L., So, B., “An Empirical Study of the Reliability of UNIX Utilities”, Communications of the ACM, Vol. 33, No. 12, pp. 32-44 (December 1990).

[Mill 94] Miller, B.P., Hollingsworth, J.K., Callaghan, M.D., “The Paradyn Parallel Performance Measurement Tools and PVM”, in: Dongarra, J.J., Tourancheau, B., (Eds.), “Environments and Tools for Parallel Scientific Computing II”, SIAM Press (1994).

[Mill 95a] Miller, B.P., Callaghan, M.D., Cargille, J.M., Hollingsworth, J.K., Irvin, R.B., Karavanic, K.L., Kunchithapadam, K., Newhall, T., “The Paradyn Parallel Performance Measurement Tool”, IEEE Computer, Vol. 28, No. 11, pp. 37-46 (November 1995).

[Mill 95b] Miller, B.P., Koski, D., Lee, C.P., Maganty, V., Murthy, R., Natarajan, A., Steidl, J., “Fuzz Revisited: A Re-examination of the Reliability of UNIX Utilities and Services”, Technical Report CS-TR-95-1268, Computer Science Department, University of Wisconsin, Madison, USA (April 1995).

[Mohr 92] Mohr, B., “Ereignisbasierte Rechneranalysesysteme zur Bewertung paralleler und verteilter Systeme”, PhD Thesis, University of Erlangen-Nürnberg, VDI-Verlag, Fortschritt-Berichte, Vol. 10, No. 221 (1992) [in German].

[Mohr 93] Mohr, B., “Standardization of Event Traces Considered Harmful or Is an Implementation of Object-Independent Event Trace Monitoring and Analysis System Possible?”, in: Dongarra, J.J., Tourancheau, B., (Eds.), “Environments and Tools for Parallel Scientific Computing”, Elsevier Science Publishers, pp. 103-124 (1993).

[Moor 96] Moore, G.E., “Can Moore’s Law Continue Indefinitely?”, Computerworld (July 1996).

[Morr 92] Morris, Ch., “Academic Press Dictionary of Science and Technology”, Academic Press (August 1992).

[MPI 95] Message Passing Interface Forum, “MPI: A Message-Passing Interface Standard - Version 1.1”, http://www.mcs.anl.gov/mpi/ (June 1995).

[Mucc 99] Mucci, P.J., Browne, S., Ho, G., Deane, C., “PerfAPI - Performance Data Standard and API”, http://icl.cs.utk.edu/projects/papi/ (1999).

[Müll 99] Müller, K.H., Purgathofer, P., Vymazal, R., “Chaos 2000 - Das globale Zeitbeben”, Döcker Verlag, Wien (1999) [in German].

[MüSc 99] Müller-Schloer, Ch., Schallenberger, B., (Eds.), “Vom Arbeitsplatzrechner zum ubiquitären Computer - Technik und Anwendungen”, VDE-Verlag, Berlin, Offenbach (1999) [in German].

[Nage 96] Nagel, W.E., Arnold, A., Weber, M., Hoppe, H.-C., Solchenbach, K., “VAMPIR: Visualization and Analysis of MPI Resources”, Supercomputer 63, Volume XII, Number 1, pp. 69-80 (Jan. 96).

[NASA 97] NASA High Performance Computing and Communication (HPCC) Program, “PetaFLOPS Enabling Technologies & Applications”, http://www.hq.nasa.gov/hpcc/petaflops (June 1997).

[Nata 86] Natarajan, N., “A Distributed Scheme for Detecting Communication Deadlocks”, IEEE Transactions on Software Engineering, Vol. SE-12, No. 4, pp. 531-537 (April 1986).

[nCUB 90] nCUBE Corporation, “nCUBE 2 Programmer’s Guide”, Release 2.0, Beaverton, OR, USA (December 1990).

[nCUB 92] nCUBE Corporation, “nCUBE 2 Programmer’s Reference Manual”, Release 3.0, Foster City, CA, USA (1992).

[Need 95] Needleman, R., (Ed.), “Notorious Bugs”, BYTE, 20 Years Special Issue, pp. 125-128 (Sept. 1995).

[Netz 93] Netzer, R.H.B., “Optimal Tracing and Replay for Debugging Shared-Memory Parallel Programs”, Technical Report CS-93-15, Dept. of Computer Science, Brown University, Providence, Rhode Island, USA (Sept. 1993).
reprinted in: Proc. of the 3rd ACM/ONR Workshop on Parallel and Distributed Debugging, San Diego, CA, USA (May 1993),
reprinted in: ACM SIGPLAN Notices, Vol. 28, No. 12, pp. 1-11 (Dec. 1993).

[Netz 96] Netzer, R.H.B., Brennan, T.W., Damodaran-Kamal, S.K., “Debugging Race Conditions in Message-Passing Programs”, Proc. SPDT’96, ACM SIGMETRICS Symposium on Parallel and Distributed Tools, Philadelphia, PA, USA, pp. 31-40 (May 1996).

[NeDa 94] Netzer, R.H.B., Damodaran-Kamal, S.K., “Accurate Race Condition Detection for Message-Passing Programs”, Technical Paper, Dept. of Computer Science, Brown University (April 1994) [preprint].

[NeMi 91] Netzer, R.H.B., Miller, B.P., “Improving the Accuracy of Data Race Detection”, Technical Report TR-995, Computer Science Department, Univ. of Wisconsin, Madison, Wisconsin (Jan. 1991),
reprinted in: ACM Press, Proc. of the 3rd ACM SIGPLAN Symposium on Principles & Practise of Parallel Programming PPOPP, Williamsburg, Virginia, USA (Apr. 1991),
reprinted in: ACM SIGPLAN Notices, Vol. 26, No. 7, pp. 133-144 (July 1991).

[NeMi 92a] Netzer, R.H.B., Miller, B.P., “What are Race Conditions? - Some Issues and Formalizations”, ACM Letters on Programming Languages and Systems, Vol. 1, No. 1, pp. 74-88 (March 1992).

[NeMi 92b] Netzer, R.H.B., Miller, B.P., “Optimal Tracing and Replay for Debugging Message-Passing Parallel Program”, Proc. Supercomputing 92, Minneapolis, MN, USA, pp. 502-511 (November 1992).
reprinted in: The Journal of Supercomputing, Vol. 8, No. 4, pp. 371-388 (1994).

[Neym 99] Neyman, M., Bukowski, M., Kuzora, P., “Efficient Replay of PVM programs”, Proc. EUROPVM/MPI 99 Intl. Conference, Barcelona, Spain, Lecture Notes in Computer Science, Vol. 1697, Springer-Verlag, pp. 83-90 (September 1999).

[Nobl 95] Noble, J., Groves, L., Biddle, R., “Object Oriented Program Visualization in Tarraingim”, in: Eades, P., Zhang, K., “Software Visualization”, Special Issue, The Australian Computer Journal, Vol. 27, No. 4, pp. 138-149 (November 1995).

[Nohr 94] Nohr, M.L., (Ed.), “Unix System V: Understanding ELF Object Files and Debugging Tools”, Programmer Collection, PTR Prentice Hall, Englewood Cliffs, NJ, USA (1994).

[Ober 95] Oberhuber, M., “Elimination of Nondeterminacy for Testing and Debugging Parallel Programs”, Proc. AADEBUG ’95, 2nd International Workshop on Automated and Algorithmic Debugging, Saint Malo, France, pp. 315-316 (May 1995).

[Ober 98] Oberhuber, M., “Methoden zur kontrollierten Ausführung nichtdeterministischer kummunizierender Programme”, Dissertation, LRR-TUM, Lehrstuhl für Rechnertechnik und Rechnerorganisation, Technical University Munich, Germany (Dec. 1998) [in German].

[Ouds 96] Oudshoorn, M.J., Widjaja, H., Ellershaw, S.K., “Aspects and Taxonomy of Program Visualization”, in: Eades, P., Zhang, K., (Eds.), “Software Visualization”, Series on Software Engineering and Knowledge Engineering, Vol. 7, World Scientific Publishing, Singapore, pp. 3-26 (1996).

[PaLi 88] Pan, D.Z., Linton, M.A., “Supporting Reverse Execution of Parallel Programs”, Proceedings of the ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging, May 5-6, 1988, University of Wisconsin, Madison, Wisconsin, USA, SIGPLAN Notices, Vol. 24, No. 1, pp. 124-129 (January 1989).

[Panc 94] Pancake, C.M., Cook, C., “What Users Need in Parallel Tool Support: Survey Results and Analysis”, Proceedings of the 1994 Scalable High Performance Computing Conference, Knoxville, TN, USA, pp. 40-47 (May 1994).

[Panc 96a] Pancake, C.M., “Visualization Techniques for Parallel Debugging and Performance-Tuning Tools”, in: Zomaya, A.Y., “Parallel Computing: Paradigms and Applications”, Intl. Thomson Computer Press, pp. 376-393 (1996).

[Panc 96b] Pancake, C.M., “Collaborative Efforts to Develop User-Oriented Parallel Tools”, in: Simmons, M.L., Hayes, A.H., Brown, J.S., Reed, D.A., “Debugging and Performance Tuning for Parallel Computing Systems”, IEEE Computer Society Press, Los Alamitos, CA, USA, pp. 355-366 (1996).

[Panc 99] Pancake, C.M., “Applying Human Factors to the Design of Performance Tools”, Amestoy, P., Berger, P., Dayde, M., Duff, I., Fraysse, V., Giraud, L., Ruiz, D., (Eds.), “Euro-Par’99 Parallel Processing”, Proceedings 5th Intl. Euro-Par Conference, Lecture Notes in Computer Science, Vol. 1685, Springer-Verlag, Toulouse, France, pp. 44-60 (Aug./Sept. 1999).

[PaNe 93] Pancake, C.M., Netzer, R.H.B., “A Bibliography of Parallel Debuggers, 1993 Edition”, Proc. of the 3rd ACM/ONR Workshop on Parallel and Distributed Debugging, San Diego, CA, USA (May 1993),
reprinted in: ACM SIGPLAN Notices, Vol. 28, No. 12, pp. 169-186 (Dec. 1993).

[Panc 95] Pancake, C.M., Simmons, M.L., Yan, J.C., (Eds.), “Guest Editor’s Introduction: Performance Evaluation Tools for Parallel and Distributed Systems”, IEEE Computer, Vol. 28, No. 11, pp. 16-19 (November 1995),
and: IEEE Parallel & Distributed Technology, Vol. 3, No. 4, pp. 14-20 (Winter 1995).

[PaUt 89] Pancake, C.M., Utter, S., “Models for Visualization in Parallel Debuggers”, Proc. Supercomputing ’89, Reno, Nevada, USA, pp. 627-637 (November 1989).

[Paul 99] Paul, J., “Software Engineering - General Testing and Debugging Guidelines”, http://www.jody-paul.com/SWE/TD/TestDebug.html (1999).

[Pfis 97] Pfister, G.F., “In Search of Clusters”, 2nd Edition, Prentice-Hall PTR, ECS Professional, NJ, 1997.

[Pine 99] Pineo, P.P., “Debugging Parallelized Code”, in: Buyya, R., (Ed.), “High Performance Cluster Computing - Programming and Applications”, Vol. 2, Prentice-Hall PTR, New Jersey, USA, Chapter 10, pp. 197-224 (1999).

[Plan 95] Plank, J.S., Xu, J., Netzer, R.H.B., “Compressed Differences: An Algorithm for Fast Incremental Checkpointing”, Technical Report CS-95-302, Computer Science Department, University of Tennessee (August 1995).

[Port 98] Porter, P., “Projects Gone Bad”, Editor’s Letter, Software Magazine, http://www.softwaremag.com/Apr98/sm048ed.htm, p. 8 (April 1998).

[Pric 92] Price, B.A., Small, I.S., Baecker, R.M., “A Taxonomy on Software Visualization”, Proc. HICSS ’25, 25th Hawaii International Conference on System Sciences, Hawaii, USA, Vol. II, pp. 597-606 (January 1992).

[Pric 93] Price, B.A., Baecker, R.M., Small, I.S., “A Principled Taxonomy of Software Visualization”, Journal of Visual Languages and Computing, Vol. 4, No. 3, pp. 211-266 (September 1993).

[Pric 98] Price, B.A., Baecker, R.M., Small, I.S., “An Introduction to Software Visualization”, in: Stasko, J.T., Domingue, J.,Brown, M.H., Price, B.A., (Eds.) “Software Visualization - Programming as a Multimedia Experience”, MIT Press, Cambridge, Massachusetts, USA, Section 1, Chapter 1, pp. 3-27 (1998).

[Pryl 99] Prylli, L., Tourancheau, B., Westrelin, R., “The Design of a High-Performance MPI Implementation on the Myrinet Network”, Proc. EUROPVM/MPI 99 Intl. Conference, Barcelona, Spain, Lecture Notes in Computer Science, Vol. 1697, Springer-Verlag, pp. 223-230 (Sept. 1999).

[Quin 87] Quinn, M.J., “Designing Efficient Algorithms for Parallel Computers”, McGraw-Hill, Singapore (1987).

[Rabe 99] Rabenseifner, R., “Automatic Profiling of MPI Applications with Hardware Performance Counters”, in: Dongarra, J.J., Luque, E., Margalef, T., (Eds.), “Recent Advances in Parallel Virtual Machine and Message Passing Interface”, Proc. EUROPVM/MPI 99, 6th European PVM/MPI Users’ Group Meeting, Barcelona, Spain, Lecture Notes in Computer Science, Vol. 1697, Springer-Verlag, pp. 35-42 (September 1999).

[RaSa 88] Ranka, S., Sahni, S., “Image Template Matching on SIMD Hypercube Multicomputers”, in: Bailey, D.H., (Ed.), Proc. ICPP ’88, 1988 International Conference on Parallel Processing, Vol. III - Algorithms and Applications, The Pennsylvania State University, USA, pp. 84-91 (August 1988).

[Rayn 88] Raynal, M., “Distributed Algorithms and Protocols”, John Wiley & Sons (1988).

[Rayn 92] Raynal, M., “About Logical Clocks for Distributed Systems”, ACM Operating System Review, Vol. 26, No. 1, pp. 41-48 (1992).

[Rayn 00] Raynal, M., “Logical Instantaneity and Causal Order: Two ‘First Class’ Communication Modes for Parallel Computing”, in: Bode, A., Ludwig, Th., Karl W., Wismüller, R., (Eds.), “Euro-Par 2000 Parallel Processing”, 6th International Euro-Par Conference, Lecture Notes in Computer Science, Springer-Verlag, Vol. 1900, Munich, Germany, pp. 35-42 (August/September 2000).

[RaSi 96] Raynal, M., Singhal, M., “Logical Clocks: Capturing Causality in Distributed Systems”, IEEE Computer, Vol. 29, No. 2, pp. 49-56 (February 1996).

[Reed 93] Reed, D.A, Aydt, R.A., Noe, R.J., Roth, P.C., Shields, K.A., Schwartz, B., Tavera, L.F., “Scalable Performance Analysis: The Pablo Performance Analysis Environment”, Skjellum, A. (Ed.), “Proceedings of the Scalable Parallel Libraries Conference”, IEEE Computer Society, pp. 104-113 (October 1993).

[Reed 95] Reed, D.A., Shields, K.A., Scullin, W.H., Tavera, L.F., Elford, C.L., “Virtual Reality and Parallel Systems Performance Analysis”, IEEE Computer, Vol. 28, No. 11, pp. 57-67 (November 1995).

[Reed 97] Reed, D.A., Nickolayev, O.Y., Roth, P.C., “Real-Time Statistical Clustering for Event Trace Reduction”, Journal of Supercomputing Applications and High-Performance Computing, Vol. 11, No. 2, pp. 144-159 (Summer 1997).

[ReRi 99] Reed, D.A., Ribler, R., “Performance Analysis and Visualization”, in: Foster, I., Kesselman, C., (Eds.), “The Grid: Blueprint for a New Computing Infrastructure”, Morgan Kaufmann Publishers, San Francisco, California, USA, Chapter 15, pp. 367-393 (1999).

[Reus 98] Reussner, R., Sanders, P., Prechelt, L., Müller, M., “SKaMPI: A Detailed, Accurate MPI Benchmark”, Proc. EUROPVM/MPI 98 Intl. Conference, Liverpool, GB, Lecture Notes in Computer Science, Vol. 1497, Springer-Verlag, pp. 52-59 (September 1998).

[Ribl 98] Ribler, R.L., Vetter, J.S., Simitci, H., Reed, D.A., “Autopilot: Adaptive Control of Distributed Applications”, Proc. HPDC ’98, 7th IEEE Symposium on High-Performance Distributed Computing, Chicago, IL, USA (July 1998).

[RiLa 98] Richards, B., Larus, J.R., “Protocol-Based Data-Race Detection”, Proc. SPDT’98, ACM SIGMETRICS Symposium on Parallel and Distributed Tools, The Resort at the Mountain, Welches, OR, USA, pp. 40-47 (August 1998)

[RiTo 92] van Rick, M., Tourancheau, B., “The Design of the General Parallel Monitoring System”, Programming Environments for Parallel Computing, IFIP, North Holland, pp. 127-137 (1992).

[RoCo 92] Roman, G., Cox, K.C., “Program Visualization: The Art of Mapping Programs to Pictures”, Proceedings of the 14th Intl. Conference on Software Engineering, Melbourne, Australia, pp. 412-420 (May 1992).

[Rons 95] Ronsse, M.A., Levrouw, L.J., “Efficient Coding of Execution-Traces of Parallel Programs”, in: Veen, J.P., (Ed.), Proceedings of the ProRISC / IEEE Benelux Workshop on Circuits, Systems and Signal Processing”, STW, Utrecht, pp. 251-258 (March 1995).

[RoKr 98] Ronsse, M.A., Kranzlmüller, D., “RoltMP - Replay of Lamport Timestamps for Message-Passing Parallel Systems”, Proceedings EUROMICRO PDP ’98, 6th EUROMICRO Workshop on Parallel and Distributed Processing, IEEE Computer Society Press, University of Madrid, Spain, pp. 87-93 (January 1998).

[RoLe 95] Ronsse, M.A., Levrouw, L.J., “An Experimental Evaluation of a Replay Method for Shared Memory Programs”, in: D’Hollander, E.H., Joubert, G.R., Peters, F.J., Trystram. D., “Parallel Computing: State-of-the-Art and Perspectives”, Proceedings of ParCo ‘95, Advances in Parallel Computing, Vol. 11, Elsevier, North-Holland, Gent, Belgium (September 1995).

[RoLe 96] Ronsse, M.A., Levrouw, L.J., “On the Implementation of a Replay Mechanism”, in: Bouge, L., Fraigniaud, P., Mignotte, A., Robert, Y., (Eds.), Proceedings of EuroPar ’96, Lecture Notes in Computer Science, Vol. 1123, Springer-Verlag Lyon, France, pp. 70-73 (August 1996).

[RoDe 98] Ronsse, M.A., De Bosschere, K., “JiTI: Tracing Memory References for Data Race Detection”, in: D’Hollander, E., Joubert, F., Trottenberg, U., (Eds.), “Parallel Computing: Fundamentals, Applications and New Directions”, Proc. ParCo ’95, Advances in Parallel Computing, Vol. 12, Bonn, Germany, pp. 327-334 (February 1998).

[Rons 00] Ronsse, M.A., De Bosschere, K., Chassin de Kergommeaux, J., “Execution Replay and Debugging”, Proc. AADEBUG 2000, 4th Intl. Workshop on Automated Debugging, Munich, Germany, pp. 5-18 (August 2000).

[Rose 96] Rosenberg, J.B., “How Debuggers Work: Algorithms, Data Structures, and Architecture”, John Wiley & Sons, New York (1996).

[RoWr 93] Rover, D.T., Wright, Ch.T., Jr., “Visualizing the Performance of SPMD and Data-Parallel Programs”, Journal of Parallel and Distributed Computing, Vol. 18, pp. 129-146 (1993).

[SaLy 87] Sarin, S., Lynch, L., “Discarding Obsolete Information in a Replicated Data Base System”, IEEE Transactions on Software Engineering, Vol. SE-13, No. 1, pp. 39-47 (January 1987).

[SaMa 93] Sarukkai, S.R., Malony, A.D., “Perturbation Analysis of High Level Instrumentation for SPMD Programs”, Proc. of PPOPP ’93, 4th ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming, pp. 44-53 (1993).

[Scha 99] Schaubschläger, Ch., Kranzlmüller, D., Volkert, J., “A Complete Monitor Overhead Removal Strategy”, Proc. of PDPTA ‘99, Intl. Conference on Parallel And Distributed Processing Techniques and Applications, Arabnia, H.R. (Ed.), CSREA Press, Vol. 4, Las Vegas, Nevada, USA, pp. 1894-1897 (June 1999).

[Schn 91] Schneider, H.-J., (Ed.), “Lexikon der Informatik und Datenverarbeitung”, 3. Edition, R. Oldenbourg Verlag, München (1991) [in German].

[Seur 96] Seuren, M.F.H., “Design and Implementation of an Automatic Event Abstraction Tool”, Masters thesis, University of Waterloo, Ontario, Canada (1996).

[Shen 96] Shende, S., Cuny, J., Hansen, L., Kundu, J., McLaugry, S., Wolf, O., “Event and State-Based Debugging in TAU: A Prototype”, Proc. SPDT’96, ACM SIGMETRICS Symposium on Parallel and Distributed Tools, Philadelphia, PA, USA, pp. 21-30 (May 1996).

[Sieg 92] Siegel, H.J., et al, “Report of the Purdue Workshop on Grand Challenges in Computer Architecture for the Support of High Performance Computing”, Journal of Parallel and Distributed Computing, Vol. 16, No. 3, pp. 199-211 (1992).

[SiRa 97] Sienkiewicz, J., Radhakrishnan, T., “DDB: A Distributed Debugger Based on Replay”, Journal of High Performance Computing, National University of Singapore, Vol. 4, No. 1, pp. 37- 45 (December 1997).

[SiKo 90] Simmons, M., Koskela, R., (Eds.), “Performance Instrumentation and Visualization”, Frontier Series, ACM Press, Addison-Wesley (1990).

[Simo 99] Simon, H.D., Kramer, W.T.C., Lucas, R.F., “Building the Teraflops/Petabytes Production Supercomputing Center”, Amestoy, P., Berger, P., Dayde, M., Duff, I., Fraysse, V., Giraud, L., Ruiz, D., (Eds.), “Euro-Par’99 Parallel Processing”, Proceedings 5th Intl. Euro-Par Conference, Lecture Notes in Computer Science, Vol. 1685, Springer-Verlag, Toulouse, France, pp. 61-77 (Aug./ Sept. 1999).

[Sing 89] Singhal, M., “Deadlock Detection in Distributed Systems”, IEEE Computer, Vol. 18, No. 11, pp. 37-48 (November 1989).

[Sist 96] Sistare, S., Allen, D., Bowker, R., Jourdenais, K., Simons, J., Title, R., “A Scalable Debugger for Massively Parallel Message-Passing Programs”, in: Hayes, A.,H., Simmons, M.L., Brown, J.S., Reed, D.A., “Debugging and Performance Tuning for Parallel Computing Systems”, IEEE Computer Society Press, Los Alamitos, California, pp. 145-160 (1996).

[Smit 84] Smith, E.T., “Debugging Tools for Message-Based, Communicating Processes”, Proceedings 4th Intl. Conference on Distributed Computing Systems, San Francisco, CA, pp. 303-310 (May 1984).

[Smit 97] Smith, R.B., Wolczko, M., Ungar, D., “From Kansas to Oz - Collaborative Debugging When a Shared World Breaks”, Communications of the ACM, Vol. 40, No. 4, pp. 72-78 (April 1997).

[SnHo 88] Snelling, D.F. and Hoffmann, G.-R., “A Comparative Study of Libraries for Parallel Processing”, Proc. Intl. Conf. on Vector and Parallel Processors, Computational Science III, Parallel Computing, Vol. 8 (1-3), pp. 255-266 (1988).

[Snir 98] Snir, M., Otto, S., Huss-Lederman, S., Walker, D., Dongarra, J.J., “MPI - The Complete Reference”, 2nd Edition, MIT Press, Cambridge, Massachusetts (1998).

[Snod 88] Snodgrass, R., “A Relational Approach to Monitoring Complex Systems”, ACM Transactions on Computer Systems, Vol. 6, No. 2, pp. 157-196 (May 1988).

[Spaf 89] Spafford, E.H., “The Internet Worm: Crisis and Aftermath”, Communications of the ACM, Vol. 32, No. 6, pp. 678-687 (June 1989).

[Spie 93] Spiers, B.T., “Reordering with Hindsight”, Technical Report MIT/LCS/TR-585, Massachusetts Institute of Technology, Laboratory for Computer Science, Cambridge Massachusetts, USA (November 1993).

[SrEu 94] Srivastava, A., Eustace, A., “ATOM: A System for Building Customized Program Analysis Tools”, Technical Report WRL 94/2, Western Research Laboratory, Compaq (Digital Equipment), Palo Alto, California, USA (1994)
reprinted in: Proc. PLDI ’94, ACM SIGPLAN’94 Conference on Programming Language Design and Implementation, SIGPLAN Notices, Vol. 29, No. 6, Orlando, Florida, USA, pp. 196-205 (June 1994).

[Stal 93] Stallman, R.M., “GDB Manual: The GNU Source Level Debugger”, Free Software Foundation (1993).

[Stan 98] Stankovic, N., Zhang, K., Kranzlmüller, D., “Visper: Parallel Processing and Java”, Proc. PDPTA ’98, Intl. Conference on Parallel and Distributed Processing Techniques and Applications, Las Vegas, NV, USA, pp. 349-354 (July 1998).

[Stas 98] Stasko, J.T., Domingue, J.,Brown, M.H., Price, B.A., (Eds.) “Software Visualization - Programming as a Multimedia Experience”, MIT Press, Cambridge, Massachusetts, USA (1998).

[StKr 93] Stasko, J.T., Kraemer, E., “A Methodology for Building Application-Specific Visualizations of Parallel Programs”, Journal of Parallel and Distributed Computing, Vol. 18, pp. 258-264 (1993).

[Stei 98] Stein, R.M., “Does Determinism Dictate Dignity?”, IEEE Computer, Vol. 31, No. 3, p. 124 (March 1998).

[StBo 89] Stephenson, M., Boudillet, O., “GECKO: A Graphical Tool for the Modelling and Manipulation of Occam Software and Transputer Hardware Topologies”, in Askey, Ch., (Ed.), “Occam and the Transputer, Research and Applications”, IOS Press, pp. 139-144 (Sept. 1989).

[Ster 92] Sterling, B., “The Hacker Crackdown - Law and Disorder on the Electronic Frontier”, Bantam Books, U.S. (1992).

[Stit 92] Stitt, M., “Debugging - Creative Techniques and Tools for Software Repair”, Wiley Professional Computing Series, John Wiley & Sons, New York (1992).

[Ston 88a] Stone, J.M., “A Graphical Representation of Concurrent Processes”, Proc. of the ACM SIGPLAN/ SIGOPS Workshop on Parallel & Distributed Debugging, Madison, Wisconsin, USA, pp. 226-235 (May 1988).

[Ston 88b] Stone, J., “Debugging Concurrent Processes: A Case Study”, Proc. PLDI’88, ACM SIGPLAN’88 Conference on Programming Language Design and Implementation, Atlanta, Georgia, pp. 145-153 (June 1988)
reprinted in: ACM SIGPLAN Notices, Vol. 23, No. 7 (July 1988).

[Stro 91] Stroustrup, B., “The C++ Programming Language”, 2nd Edition, Addison-Wesley Publishing Company, Reading, MA, USA (1991).

[Sund 90] Sunderam, V.S., “PVM: A Framework for Parallel Distributed Computing”, in: Concurrency - Practice & Experience, Vol. 2, No. 4, pp. 315-339 (1990).

[Tai 95] Tai, J.C., “Race Detection for Message-Passing Programs”, Technical Report, TR-96-04, Computer Science Department, North Carolina State University, USA (1995).

[TaCa 91] Tai, K.C., Carver, R.H., Obaid, E.E., “Debugging Concurrent Ada Programs by Deterministic Execution”, IEEE Transactions on Software Engineering, Vol. 17, No. 1, pp. 45-63 (January 1991).

[TaCa 96] Tai, K.C., Carver, R.H., “Testing Distributed Programs”, in: Zomaya, A.Y., (Ed.), “Parallel and Distributed Computing Handbook”, McGraw-Hill, New York, Chapter 33 (1996).

[Tayl 92] Taylor, D.J., “A Prototype Debugger for Hermes”, Proc. of the 1992 CAS Conference, IBM Canada Ltd. Laboratory, Centre for Advanced Studies, Toronto, Ontario, Canada, pp. 29-42 (November 1992).

[Tayl 93] Taylor, D.J., “The Use of Process Clustering in Distributed System Event Displays”, Proc. of the 1993 CAS Conference, IBM Canada Ltd. Laboratory, Centre for Advanced Studies, Toronto, Ontario, Canada, pp. 505-512 (January 1993).

[TeCh 97] Teodorescu, F., Chassin de Kergommeaux, J., “On Correcting the Intrusion of Tracing Non-deterministic Programs by Software”, Proc. EUROPAR’97 Parallel Processing, 3rd Intl. Euro-Par Conference, Springer, Lecture Notes in Computer Science, Vol. 1300, Springer-Verlag, Passau, Germany, pp. 94-101 (Aug. 1997).

[Thro 99] Throop, J., “OpenMP: Shared-Memory Parallelism From The Ashes”, IEEE Computer, Vol. 32, No. 5, pp. 108-109 (May 1999).

[Topo 95] Topol, B., “Visualization/Animation in Cluster Environments - PVaniM”, Graphics Visualization & Usability Center, Available via World Wide Web at the following URL: http://www.cc.gatech.edu/ gvu/softviz/parviz/pvanim/pvanim.html (1995).

[TsYa 95] Tsai, J.J.P., Yang, S.J.H., “Monitoring and Debugging of Distributed Real-Time Systems”, IEEE Computer Society Press, Los Alamitos, California (1995).

[TuCa 93] Turner, S.J., Cai, W., “The ‘Logical Clocks’ Approach to the Visualization of Parallel Programs”, in: Haring, G., Kotsis, G., (Eds.), “Performance Measurement and Visualization of Parallel Systems”, Proc. of the Workshop on Performance Measurement and Visualization, Moravany, Czechoslovakia (October 1992), Elsevier Science Publishers, Amsterdam, NL, pp. 45-66 (October 1993).

[Vaut 87] Vautherin, J., “Parallel Systems Specifications with Colored Petri Nets and Algebraic Abstraction Data Types”, in: Rozenbery, G., (Ed.), “Advances in Petri Nets”, Springer-Verlag, Berlin, pp. 291-308 (1987).

[ViSa 94] Vion-Dury, J.Y., Santana, M., “Virtual Images: Interactive Visualization of Distributed Object-Oriented Programming Systems”, Proc. OOPSLA ’94, Conference on Object-Oriented Systems, Languages, and Applications, in: ACM SIGPLAN Notices, Vol. 29, No. 10, pp. 65-84 (October 1994).

[Volk 97] Volkert, J., “Rechnerarchitekturen”, in: Rechenberg, P., Pomberger, G., “Informatik-Handbuch”, Carl Hanser Verlag, München, Wien, pp. 275-292 (1997).

[WaRo 95] Waheed, A., Rover, D.T., “A Structured Approach to Instrumentation System Development and Evaluation”, Proc. SC ’95, Supercomputing 95 Conference, ACM Press, San Diego, California, USA (December 1995).

[Wald 95] Waldschmidt, K., (Ed.), “Parallelrechner: Architekturen - Systeme - Werkzeuge”, Leitfäden der Informatik, Teubner, Stuttgart, Germany (1995) [in German].

[Walt 95] Walton, G.H., Poore, J.H., Trammell, C.J., “Statistical Testing of Software Based on a Usage Model”, in: Software - Practice and Experience, Vol. 25, No. 1, pp. 97-108 (January 1995).

[Wang 99] Wang, F., Zheng, Q., An, H., Chen, G., “A Parallel and Distributed Debugger Implemented with Java”, in: Chen, J., Lu, J., Meyer B., (Eds), Proc. TOOLS ’31, 31th International Conference on Technology of Object Oriented Languages and Systems, IEEE Computer Society Press, Nanjing, China, pp. 342-346. (September 1999).

[Wang 95] Wang, Y.-M., Chung, P.-Y., Lin, I.-J., Fuchs, W.K., “Checkpoint Space Reclamation for Uncoordinated Checkpointing in Message-Passing Systems”, IEEE Transactions on Parallel and Distributed Systems, Vol. 6, No. 5, pp. 546 - 554 (May 1995).

[WaBl 97] Wasserman, H., Blum, M., “Software Reliability via Run-Time Result-Checking”, Journal of the ACM, Vol. 44, No. 6, pp. 826-849 (November 1997).

[Wein 88] Weinand, A., Gamma, E., Marty, R., “ET++ - An Object Oriented Application Framework in C++”, OOPSLA’88 Conference Proceedings, SIGPLAN Notices, Vol. 23, No. 11, San Diego, pp. 168-182 (September 1988).

[WeKu 98] Weinreich, R., Kurschl, W., “Dynamic Analysis of Distributed Object-Oriented Applications”, in: Proc. HICSS ’30, 30th Hawaii International Conference on System Sciences, Kona, Hawaii, USA (January 1997).
reprinted as: Technical Report TR-SE 98.01, CD-Laboratory for Software Engineering, Johannes Kepler University Linz, Austria [1998].

[Weis 82] Weiser, M., “Programmers Use Slicing When Debugging”, Communications of the ACM, Vol. 25, No. 7, pp. 446-452 (July 1982).

[Weis 84] Weiser, M., “Program Slicing”, IEEE Transaction on Software Engineering, Vol. 10, No. 4, pp. 352-357 (July 1984).

[Will 90] Williams, S.A., “Programming Models for Parallel Systems”, John Wiley & Sons, England, GB (1990).

[Will 92] Williams, K.J., Andersland, M.S., Gannon, J.A., Lumpp, J.E., Casavant, T.L., “Necessary Conditions for Tracking Timing Perturbations in Timed Petri Nets”, Proc. 13th Allerton Conference on Communication, Control, and Computing (1992).

[WiAl 99] Wilkinson, B., and Allen, M., “Parallel Programming - Techniques and Applications Using Networked Workstations and Parallel Computers”, Prentice Hall, New Jersey, USA (1999).

[Wirt 94] Wirtz, G., “Modularization and Process Replication in a Visual Parallel Programming Language”, Proc. of the IEEE International Symposium on Visual Programming, St. Louis, USA, pp. 72-79 (1994).

[Wism 97a] Wismüller, R., “Debugging Message Passing Programs Using Invisible Message Tags”, in: Bubak, M., Dongarra, J., Wasniewski, J., (Eds.), “Recent Advances in Parallel Virtual Machine and Messag Passing Interface”, Proc. 4th European PVM/MPI Users' Group Meeting, Lecture Notes in Computer Science, Springer Verlag, Vol. 1332, Crakow, Poland, pp. 295-302 (November 1997).

[Wism 97b] Wismüller, R., Ludwig, T., Bode, A., Borgeest, R., Lamberts, S., Oberhuber, M., Röder, C., Stellner, G., “The TOOL-SET Project: Towards an Integrated Tool Environment for Parallel Programs”, in: De, X., Großpietsch, K.-E., Steigner, C., (Eds.), Proc. APPT ’97, 2nd Sino-German Workshop on Advanced Parallel Processing Technologies, Koblenz, Germany, pp. 9-16 (Sept. 1997).

[Wism 98a] Wismüller, R., Trinitis, J., Ludwig, T., “OCM --- A Monitoring System for Interoperable Tools”, Proc. SPDT ’98, 2nd ACM SIGMETRICS Symposium on Parallel and Distributed Tools, ACM Press, Welches, OR, USA, pp. 1-9 (August 1998).

[Wism 98b] Wismüller, R., “On-Line Monitoring Support in PVM and MPI”, Proc. EuroPVM/MPI’98, Lecture Notes in Computer Science, Vol. 1497, Springer-Verlag, Liverpool, UK, pp. 312-319 (September 1998).

[WoMo 98] Wolf, F., Mohr, B., “EARL - A Programmable and Extensible Toolkit for Analyzing Event Traces of Message Passing Programs”, Technical Report FZJ-ZAM-IB-9803, http://www.kfa-juelich.de/zam/docs/printable/ib/ib-98/ib-9803.ps, Forschungszentrum Jülich, Zentralinstitut für Angewandte Mathematik, Germany (April 1998).

[Worl 92] Worley, P.H., “A New PICL Trace File Format”, Technical Report ORNL/TM-12125, Oak Ridge National Laboratory, Tennesee, USA (Sept. 1992).

[Wu 96] Wu, W., Spezialetti, M., Gupta, R., “On-line Avoidance of the Intrusive Effects of Monitoring on Runtime Scheduling Decisions”, Proc. 16th IEEE Computer Society Intl. Conference on Distributed Computing Systems, Hong Kong, pp. 216-223 (May 1996).

[Wu 98] Wu, W., Gupta, R., Spezialetti, M., “Experimental Evaluation of On-line Techniques for Removing Monitoring Intrusion”, Proc. of SPDT’98, SIGMETRICS Symposium on Parallel and Distributed Tools, Welches, Oregon, pp. 30-39 (Aug. 1998).

[WuBe 84] Wuu, G., Bernstein, A., “Efficient Solutions to the Replicated Log and Dictionary Problems”, Proc. 3rd ACM Symposium on Principles of Distributed Computing, Vancouver, B.C., Canada, pp. 233-242 (1984).

[Xion 96] Xiong, J., Wang, D., Zheng, W., Shen, M., “Buster: An Integrated Debugger for PVM”, Proc. ICAPP ’96, 2nd Intl. Conference on Algorithms and Architectures for Parallel Processing, pp. 124-129 (June 1996).

[Yan 95] Yan, J.C., Sarukkai, S., Mekra, P., “Performance Measurement, Visualization and Modelling of Parallel and Distributed Programs using the AIMS Toolkit”, Software - Practice and Experience, Vol. 25, No. 4, pp. 429-461 (April 1995).

[YaSa 96] Yan, J.C., Sarukkai, S.R., “Analyzing Parallel Program Performance Using Normalized Performance Indices and Trace Transformation Techniques”, Parallel Computing, Vol. 22, No. 9, pp. 1215-1237 (November 1996).

[Yan 98] Yan, J.C., Jin, H.H., Schmidt, M.A., “Performance Data Gathering and Representation from Fixed-Size Statistical Data”, Technical Report NAS-98-003, http://www.nas.nasa.gov/Research/Reports/Techreports/1998/nas-98-003.pdf, NAS Systems Division, NASA Ames Research Center, Moffet Field, California, USA (February 1998).

[YoYo 99] Yourdon, E., Yourdon, J., “Time Bomb 2000: What the Year 2000 Computer Crisis Means to You!”, Yourdon Press Computing Series, Prentice Hall, 2nd Edition (February 1999).

[Zaki 99] Zaki, O., Lusk, E., Gropp, W., Swider, D., “Toward Scalable Performance Visualization with Jumpshot”, International Journal of High-Performance Computing Applications, Vol. 13, No. 3, pp. 277-288 (Fall 1999).

[Zeme 99] Zemerly, M.J., Delaitre, T., Justo, G.R., Winter, S.C., “EDPEPPS: A Graphical Environment for the Parallel Software Development Life-Cycle”, Proc. of PDPTA ‘99, Intl. Conference on Parallel And Distributed Processing Techniques and Applications (PDPTA ‘99), Arabnia, H.R. (Ed.), CSREA Press, Las Vegas, Nevada, USA, pp. 1737-1743 (June 1999).

[Zern 92] Zernik, D., Snir, M., Malki, D., “Using Visualization Tools to Understand Concurrency”, IEEE Software, Vol. 9, No. 3, pp. 87-92 (May 92).

[ZhMa 94] Zhang, K. and Ma, W., “Graphical Assistance in Parallel Program Development”, Proc. VL’94 - 10th IEEE International Symposium on Visual Languages, St. Louis, USA, pp. 168-170 (Oct. 1994).

[Zhan 98] Zhang, K., Sun, S., Li, K.-C., “Dynamically Instrumenting Message-Passing Programs Using Virtual Clocks”, Proc. HPDC7, 7th IEEE Symposium on High Performance Distributed Computing, Chicago, IL, USA (July, 1998).

[Zhan 99] Zhang, K., Ma, X., Hintz, T., “The Role of Graphics in Parallel Program Development”, Journal of Visual Languages and Computing, Academic Press, Vol.10, No.3, pp. 215-243 (June 1999).

[ZiCh 90] Zima, H., Chapman, B., “Supercompilers for Vector and Parallel Computers”, ACM Press, Frontier Series, Addison-Wesley Publishing Company, New York, NY, US (1990).

[Zuse 69] Zuse, K., “Rechnender Raum”, in: Schriften zur Datenverarbeitung, Vol. 1, Friedrich Vieweg & Sohn, Braunschweig (1969) [in German].

[Zuse 76] Zuse, K., “Some Remarks on the History of Computing in Germany”, Lecture at the International Research Conference on the History of Computing, University of California, Los Alamos, available at http://www.zib.de/zuse/ (June 1976).

[Zuse 86] Zuse, K., “Der Computer - Mein Lebenswerk”, 2nd Edition, Springer-Verlag (1986) [in German].

People

A

Abrams, M. 80
Agrawal, H. 259
Aiken, H. 38
Audenaert, K.M. 96

B

Babbage, Ch. 38
Bach, J. 58, 59, 244
Baecker, R.M. 114
Balzer, R.M. 82, 127
Bates, P. 118, 128, 261
Bemmerl, T. 104
Bernstein, A.J. 49
Blum, M. 78, 97
Boehm, B.W. 58, 60
Boyer, R.S. 119
Brennan, T.W. 93
Britcher, R.N. 75
Butler, R.M. 52

C

Cai, W. 100, 118
Chassin de Kergommeaux, J. 96, 100, 198
Choi, J.-D. 260
Conway, M.E. 38
Cuny, J.E. 119, 142
Curtis, R.S. 94
Cypher, R. 93

D

Damodaran-Kamal, S.K. 93
Dekel, E. 231, 270
Dongarra, J.J. 39
Dubois, P.F. 58

E

Eckert, J.P. 38

F

Fan, J.-J. 119
Feynman, R.P. 116
Fidge, C.J. 133, 134, 137, 190
Flynn, M.J. 39, 43
Foster, I.T. 47, 49
Francioni, J.M. 93, 113
Fromentin, E. 119

G

Geist, G.A. 52, 101, 258
Gill, S. 38
Grabner, S. 98, 122, 174, 185, 188, 191, 228, 249, 260

H

Händler, W. 43
Heath, M.T. 52
Helmbold, D.P. 91
Holland, J. 38
Hollingsworth, J.K. 100, 160
Hopper, G.M. 67
Hossfeld, F. 36
Hough, A.E. 119
Hwang, K. 43, 47, 49

J

Jones, C. 74, 75

K

Kacsuk, P. 99, 188
Kamkar, M. 260
Karp, R.M. 119
Kasparov, G. 46
Knuth, D.E. 119
Kohn, J. 118
Kunz, T. 119

L

Ladkin, P. 68
Lamport, L. 116, 134, 136, 208
LeBlanc, T.J. 95, 147
Leibniz, G. 38
Leu, E. 93, 94, 100, 213
Levrouw, L.J. 96, 244
Lewis, T.G. 74
Lieberman, H. 61
Lourenco, J. 115
Ludwig, T. 112
Lusk, E.L. 52

M

Maillet, E. 100, 101
Malony, A.D. 100, 111
Mattern, F. 134, 138
Mauchly, J.W. 38
McDowell, C.E. 91
Mellor-Crummey, J.M. 95
Meuer, H.W. 39
Miller, B.P. 74, 92, 93, 95, 100, 114, 118, 160, 244
Mohr, B. 108, 109, 111
Moore, G. 42
Moore, J.S. 119
Morris, R.T. 70

N

Netzer, R.H.B. 92, 93, 95, 244

O

Oberhuber, M. 93, 97

P

Pancake, C.M. 113
Pascal, B. 38
Peyton, B.W. 52
Porter, P. 67
Price, B.A. 114

Q

Quinn, M.J. 46

R

Rabin, M.O. 119
Raynal, M. 135, 139
Reed, D.A. 100, 111, 112
Reussner, R. 101
Ronsse, M.A. 96, 244
Rosenberg, J.B. 83, 104
Rudd, K.W. 39

S

Sarukkai, S.R. 118
Schickhard, W. 38
Schiper, A. 100
Seuren, M.F.H. 119
Small, I.S. 114
Strohmeier, E. 39
Su, K.-Y. 119
Sunderam, V.S. 52, 107

T

Tai, J.C. 93
Teodorescu, F. 100
Tourancheau, B. 128
Turner, S.J. 100

V

van Rick, M. 128
Volkert, J. 98, 228, 249, 276
von Neumann, J. 38, 47

W

Wasserman, H. 78, 97
Weiser, M. 185, 259
Wijshoff, H. 100
Wileden, J.C. 118, 128, 261
Williams, W. 118
Wirth, N. 258
Wismüller, R. 107
Wittie, L.D. 94
Worley, P.H. 52

X

Xu, Z. 43, 47, 49

Y

Yan, J.C. 118

Z

Zhang, K. 100
Zuse, K. 38

Program Analysis Tools

A

AIMS 116
Annai 120
Ariadne 119
ATEMPT 121
ATExpert 118
ATOM 160
Autopilot 112

B

Belvedere 119
Buster 95

C

CDFA 121
CHITRA 80
CODE 117
CXdb 105
CXTrace 116

D

DCBD 105
DDB 95
DDBG 115, 163
Deadlock Checker 87
DeHiFo 120
DyninstAPI 160

E

EARL 111, 119
EBBA 118
EDPEPPS 116
EEL 160
EMU 111, 112, 121
EXDAMS 82

F

Falcon 117

G

gdb 105, 110
GDDT 120
GRADE 115
Guard 79

H

HeNCE 114
Hence 117
Hindsight 98

I

IGOR 82
ipd 105
IPS-2 118
IVD 95

J

JiTI 160
Jumpshot 117

K

Kansas 108

M

MAD 120
Mantis 117
MPL* 96, 198, 241
MPP Apprentice 116
MUCH 121

N

ndb 105
Node Prism 120
NOPE 112, 121, 215

O

OCM 112, 163
OrWell 117

P

P2D2 ..... 105, 110, 117
Pablo 112
Paje 117
Panorama 95, 105
PARADE 117
Paradyn 117, 160, 180
Paragraph 112, 116, 180
ParAide 116
PARASIT 112, 121
PatternTool 268
PDBG 105, 110, 115, 116
PDT 95, 105, 110, 120
PICL 111, 159
POET 119
Poet 119
PPD 82
Prism 116, 119
PROVE 116
PV 116
PVaniM 117

R

RDS 98

S

SIMPLE 110, 119
SPiDER 120
Spyder 82
STEPS 115
svPablo 112

T

Tape/PVM 116
TAU 119
TCE 98
TOOL-SET 115
TOPSYS 117
Totalview 105, 110
TraceView 118
TRAPPER 116
Trapper 117

U

Upshot 117

V

VAMPIR 117
Vampir 180
Virtue 112
Visper 114
VISTOP 117

X

XPVM 117

Z

ZM4 110

Index

A

Abacus 37
absence of ordering..... 139
abstract event classes 151
abstraction..... 106, 185, 260
    horizontal..... 264
    vertical 265
acceptance test 57
accepted parameters..... 164
accidental nondeterminism 89
actual races 92
Airbus A320 68
airplane crashes 68
airplanes
    Airbus A320 68
    Boeing B757..... 69
all-to-one communication..... 229, 250
Alpha 160
American Airlines 69
amount
    of data 127
    of debugging data 84, 105, 106
    of monitor overhead 180
    of testing 245
analysis
    automatic 172
    automatic nondeterminism 238
    component 108, 109
    control and data flow..... 185, 246
    flowback 260
    in-depth..... 186
    on-line..... 162
    post-mortem..... 162
    race condition 185
    tools layers..... 108
Analytical Engine 38
Animation Choreographer 117
anomalous effects 84, 105
ANSI X3H5 52
apparent races 92
application scale 106
application-specific visualization 120
arbitrary orderings 133
Argonne National Laboratory..... 52
Ariane-5 rocket 69
array
    inspection..... 151, 156, 169, 182
    three-dimensional 184
artificial
    events..... 156, 182
    program execution..... 196, 225
    re-execution..... 217, 229
    replay..... 98, 235
ASCI..... 29, 37, 52
    Blue Mountain 40
    Option Red 40
    Option White..... 29
ASCI - Advanced Strategic
        Computing Initiative 73
assertions 119
AT&T 70
ATEMPT - A Tool for Event ManiPulaTion..... 121
ATM 44
atomic weapon laboratory 71
auralization 115
auspicious cuts 191
automatic
    abstraction 262
    analysis..... 172
    checking 106
    event manipulation..... 98
    nondeterminism analysis..... 238
    pattern matching..... 268
    testing 97
automatization 106, 238

B

backtracking
    variables 260
    with re-execution 80
backwards
    debugging..... 80
    slicing 260
bank
    accounts..... 71
    transfers..... 69
base debuggers 105
basic patterns 269
behavior..... 126
behavioral
    abstraction 128
    characteristics..... 172
    properties..... 178
benchmarks
    Linpack 40
    NAS..... 42
    SKaMPI..... 42
    SPEC 42
Berkeley NOW..... 44
Bernstein’s Theorem 49

bidirectional communication 144
binary
tree 269
wrapping 158, 160
Black & Decker 69
blocking
in the network 250
receive 151, 197, 216
receive event 152
Boeing B757 69
break-down 172
breakpointing 62, 80, 185–192
characteristics 187
event 189, 220
breakpoints 62, 85, 104
global 188
local 188
location 81
message 188
strongly complete collective 188
types of 188
broadcasts 165
buffer overrun 175
bugs 75
classification 63
definition 66
first 67
lethal 68
Millennium 29, 71
notorious 67–72
Pentium 73
potential 73
Burroughs 38
butterfly networks 269
bystander effect 87
Byte magazine 68

C

C++ exception handling 173
cache
conflicts 86
contents 86, 127
Cache Coherent - Non-Uniform
Memory Access 252
candidates
race condition 206, 217, 230, 243, 244
CAPSE - Computer Aided
Parallel Software Engineering 268
cardiac control systems 71
Carnegie Mellon University 95
causal
certainty 87
order 131
relationship 131
causality 140
graph 117
CAVE - CAVE Automatic
Virtual Environment 146
CC-NUMA architecture 252
CDFA - Control- and
Data-Flow Analyzer 121
center of interest 118
chaotic computations 89
characteristics of breakpointing
mechanisms 187
checking
automatic 106
simple 78
checkpoint and rollback recovery 274
checkpointing 82, 122, 274
checkpoints
consistent 192
CHIMP 53
CISC CPU 251
classification
of abstraction mechanisms 262, 276
of bugs 63
clock
counter 136, 165
global 135
local 135
logical 164
matrix 139
partially ordered 135
real-time 164, 168
synchronization 180, 250
totally ordered 135
updates 167
vector 138, 165, 167
clocks
Lamport 136, 165, 244
partially ordered logical 137
piggybacked 241
totally ordered logical 136
types of 134
unsynchronized 146
vector 190
clustering 118
CM-5 50
COFF - Common Object File Format 160
collaborative debugging 108
collapsing 263
collective communication 53
communication
bidirectional 144
broadcasts 165
chart 117
context 153

events 151
function 158
graph 119, 122, 145
nonblocking
receive 169
patterns 232
statements 174
time 180
universe 153
complete
cut 188
testing 230, 235
completeness problem 87, 96, 102, 228
completion 129, 153
complexity 84, 105
reduction of 261
compression 112, 244
computation time 180
computational errors 172
Computational Grids 44
computers
ASCI Blue Mountain 40, 252
ASCI Red 40
ASCI White 29
CONVEX SPP 1000 42
Cray C-90 43
Cray T3D 43, 44, 45
Cray T3E 40, 43, 45, 111
Cray T3E 1200 40
Cray T3E 900 40
Cray T-90 43
DEC Alpha Server 8400 44
DEC TruCluster 44
Deep Blue 29
ENIAC 38
first supercomputer 38
Goodyear MPP 43
Harvard Mark I 38
Hitachi SR 2201 42
Hitachi SR 8000 ..... 40, 43, 46
HP CONVEX Exemplar 1200 121
IBM R50 44
IBM SP Silver 40, 46
IBM SP2 44
Intel iPSC/2 111
Intel iPSC/860 ..... 95, 105
Intel Paragon 45
Metacomputer 44
nCUBE 95
nCUBE 2 ..... 105, 111, 120, 154, 159, 208, 251
NEC SX-4 43
petaflop 37
Sequent Symmetry 82
SGI Origin 2000 ..... 40, 46, 111, 121, 153, 252
SGI Power Challenge 44
SNI VP200 41
Stanford DASH 44
teraflop 36–37
TM CM-5 ..... 42, 50
Transputer 110
ubiquitous 29
Zuse Z1, Z2, Z3 38
computing
power 275
ubiquitous 47
concurrency map 117
concurrent
events 139, 141
order 140
order relation 144
races 92
conferences
Mannheim Supercomputer 40
Supercomputing 40
consistency 67
consistent checkpoints 192
contention 86
context 83
control and data flow
analysis 98, 185, 246
characteristics 245
controlled execution 93, 97
control-parallel 48
CONVEX 105
Convex
SPP 1000 42
correction 60
correctness 146
program 59, 200
COTS - Commodity Off-The-Shelf 37
COW - Clusters of Workstations 43
C-preprocessor 157
CPU
CISC 251
DEC Alpha 21164 45
MIPS 46
state 127
crashtest simulations 89
Cray
C-90 43
Message Passing Toolkit MPT 153
T3D 43, 44, 45
T-90 43
credit-card 71
critical path 116
cumulative data 127
cuts 122, 188
auspicious 191
exchange 219
inauspicious 191

meaningful partial 188
mechanism for checkpointing 274
partial 188, 220
simple 188
unfavorable 191
cycles
debugging 78, 80–83, 150, 243
nondeterminism testing 228
testing 78, 78–80, 150
traditional 79
cyclic
debugging 94, 118
models 56

D

data access displays 116
data circulation 231
data races 92
data-driven monitoring 276
data-machine-processor views 116
data-parallel model 48
DCBD - Dawning Cluster DeBugger 105
DDB - Distributed DeBugger 95
deadlock 87
analysis 116
Debis Munich 42
debugger 61, 62
architecture 104
requirements 105
debugging 60, 61, 67
backwards 80
cycle 78, 80–83, 94, 102, 150, 243
cyclic 118
definition 61
focus 106
history of 67
on-line 95
post-mortem 95
tool 61
key issues 83
DEC
Alpha 21164 45
Alpha server 8400 44
TruCluster 44
Deep Blue 29
defect
potential 74
removal rule 75
definitions
bug 66
debugging 61
error 63
event graph 143
probe effect 99
supercomputer 39
degrees of freedom 88
Demand-Driven Replay 96
Denver airport 69
descendants 245
direct and indirect 230
destination process 152
"details are the jungle ..." 258
determinism 87
definition 88
deterministic
execution 94
programs 133
routing 208
Deterministic Execution 96
Difference Engine 38
Digital UNIX 160
dimension exchange pattern 271
direct descendants 230
displays
Animation Choreographer 117
application-specific 120
causality graph 117
communication chart 117
communication graph 122
concurrency map 117
critical path 116
data access 116
data-machine-processor views 116
event graph 117
Feynman 116
Gantt chart 116
Hasse diagram 116
heat-diagram 183
histogram-bargraphs 116
Lamport timeline view 117
logical clocks 146
memory access patterns 116
Petri Nets 116
process grid 117
program graph 116
space-time diagram 116, 185
state-time animation 117
static call graph 116
three-dimensional 116
time
histogram 117
Virtual Reality 116
xy-contour plots 116
distance doubling 210, 235, 241
distributed
snapshots 274
systems 135

division by zero 173, 267
down-scaling..... 102, 258
Dragonair 69
DSM - Distributed Shared-Memory Machines 43
dynamic
analysis 59
instrumentation..... 158, 160
slicing 260

E

EARL - Event Analysis and Recognition Language 111
EBBA - Event Based Behavioral Abstraction..... 118
ECS - Erlanger Classification Scheme 43
e-cube routing 208
Edinburgh Parallel Computing Center 53
EDL - Event Definition Language 118
effects
anomalous..... 84, 105
bystander 87
completeness problem 87, 228
deadlock..... 87
flip-flop..... 223
irreproducibility..... 87, 93, 103, 187, 228
livelock 87
maze..... 85, 93
observable..... 126
probe 87, 99, 102, 250
stampede 87
ELF - Executable and Linking Format 160
Emory University, Atlanta..... 107
EMU - Event Monitoring Utility..... 111, 112, 121
enforced causal ordering..... 139
ENIAC 38
entry point..... 129
environmental parameters 86
epsilon-interval 249
equivalent execution 94, 212, 240
errors
at communication edges 173
at event vertices 173
at sequential edges..... 173
buffer overrun..... 175
computational 172
definition 63
division by zero 173, 267
extraction 267
follow-up 122
forever blocked..... 175
Heisenbugs 99
highlighting 178
incorrect communication pairs..... 165
incorrect computation 185
incorrect perceived order 165
isolation..... 267
location..... 60
of origin..... 141
ordering 98
overlapping of columns..... 185
root 122
subsequent..... 141
suspicious..... 201
ET++ application framework 121
Ethernet 44
event 94, 128
attributes..... 130, 182, 240
accepted parameters..... 164
supposed parameters..... 164
breakpointing 189
causal relationship..... 131
causality 137
initial 144
level..... 130
location..... 130, 165
manipulation 98, 122, 217, 228, 235
occurrence 154
ordering 131–143, 274
properties..... 155
relations..... 118
scope 130, 165
sequential order..... 131
terminal 144
time zones 141
traces 119
type..... 130
event graph 117, 122
definition 143
isomorphic..... 240
model..... 150, 172
operations 261
patterns 245, 254
visualization 145
event-driven monitoring..... 276
events
array inspection..... 169
artificial 156, 182
blocking receive 152
breakpointing 220
concurrent 139, 141
first 209
hiding of 266
high level..... 118
independent 141
intended nondeterministic 200
isolated receive event..... 174
isolated send event 174

last ... 189
level of generation ... 165
low level ... 118
multiple streams ... 165
nonblocking receive event ... 152
nonblocking send ... 151
nondeterministic ... 197, 238
nondeterministic receive ... 241
ordering ... 132
primitive ... 118
random number generation ... 169
responsible ... 201
terminal ... 173
uncritical nondeterministic ... 246
unintended nondeterministic ... 200
unordered ... 139
user defined ... 129
variable inspection ... 169
with different message length ... 174
EVL - Electronic Visualization Laboratory ... 146
evolutionary testing ... 79
examples
all-to-one-communication ... 229
distance doubling ... 210
exception ... 172
handler ... 177
exchange
cut ... 219
sequence ... 231
exclusive-or operation ... 271
EXDAMS - EXtendable Debugging and Monitoring System ... 82
execution
backtracking ... 82
failure ... 129
initial ... 235, 245
profile ... 127
statistics ... 180
time ... 258
exit-part ... 161
external services ... 82
external sources ... 91
extraction ... 106, 261

F

failure ... 65, 75, 129, 172, 175
density ... 75
detection ... 274
program ... 161
runtime ... 267
fault ... 65, 75
tolerance ... 53, 82
fault-tolerant computing ... 274
FDDI ... 44
FDL - Filter Description Language ... 119
feasible races ... 92
"feel the execution" ... 146
Feynman diagram ... 116
FIFO - First In First Out ... 88, 200
FIFO queue ... 212, 274
file pointers ... 81
filename ... 182
filtering ... 106, 118, 261, 267
final results ... 196
finger-print ... 240, 242
finite automata ... 106
finite element solver ... 258
first event ... 209
fish eye view ... 118
flip-flop effect ... 223
flooding the user ... 85
flowback analysis ... 80, 260
Flynn's classification ... 43
follow-up errors ... 122
FORMAN language ... 118
forward
slicing ... 260
FPU - Floating Point Unit ... 251
functionality testing ... 56

G

Gantt chart ... 116
Gaussian elimination ... 258
gdb - GNU debugger ... 105, 110
GDDT - Graphical Data Distribution Tool ... 120
general races ... 92
generation of finger-prints ... 243
generic debugging system ... 104
global
breakpoints ... 188
clock ... 135
exchange pattern ... 271
operations ... 199
real-time clock ... 135
viewpoint ... 133, 134
Globus project ... 44
GNU C compiler ... 74
GNU public license ... 107
Good Enough testing ... 58, 99, 244
Goodyear Massively Parallel Processor ... 43
GRADE - Graphical Application Development Environment ... 115
Grand Challenges ... 29, 34–35, 39, 73
granularity ... 127

graph
communication 145
of executions..... 241
GRAPNEL language 99, 115
Gray code scheme..... 270
Griffith University, Australia 79
GUP Linz..... 120

H

hackers..... 70
Hamming distance 252, 270
happened before relation 140, 165
hardware
block-diagram..... 180
counters 110
independence..... 159
instrumentation..... 160
monitor 160
oriented analysis techniques..... 180
hardware, hybrid, software
monitors..... 110
Harvard Mark I 38
Harvard University 67
Hasse diagram 116
HCI - Human Computer Interface..... 113
heat-diagram 183
Heisenberg (Uncertainty) Principle..... 83, 99, 250
Heisenbugs 83, 99
heterogeneity 53, 105
hide events 266
hide processes..... 266
hierarchy 118
high level events 118
High Performance Fortran..... 50, 95
highlighting errors 178
histogram-bargraphs 116
history
of debugging..... 67
of parallel computing..... 37
history tape 82, 127
Hitachi 40
SR 2201 42
SR 8000 ..... 43, 46
horizontal abstraction 264
hospitals..... 71
hot spots..... 127
HP CONVEX SPP Exemplar 1200 121
HPD standard..... 108
HPDF - High Performance
Debugging Forum 108
HPF - High Performance Fortran 50, 120
HPF-like 50
data distribution..... 183
distribution scheme 157
hybrid
monitor..... 160
hypercube architecture 251
hypercube data circulation 232, 270

I

I/O
activities 129
bandwidth..... 275
descriptors 81
parallel..... 165, 242
state 127
IBM
AIX 160
R50 44
SP Silver..... 40, 46
SP2 44
IBM - International Business Machines..... 38, 40
ILLIAC IV 38
immersion..... 146
implicit parallelism..... 49
inauspicious cuts 191
inclusion relation..... 119
incorrect
communication pairs..... 165
computation..... 185
ordering 133
overlapping 185
parameters 174
perceived order..... 165
program states 173
increased complexity..... 84
incremental
checkpointing..... 274
tracing 82, 243
independent events 141
in-depth analysis..... 186, 212
indeterminacy 89
indirect descendants 230
individual memory accesses..... 244
Information Power Grid 44
initial
event..... 144
execution 235, 238, 245
user feedback 245
init-part..... 161
input data..... 238
input testing cycle 102
inspection 62, 104
array 151, 156, 182
interactive..... 172
variable..... 151, 156, 182

installation test 57
Instant Replay 94, 95, 96
instrumentation 157–160, 215
  binary wrapping 158
  dynamic 158, 160
  object code 158
  operating system 160
  parsing 158
  textual substitution 158
integrated debugging and performance analysis . 108
integration test 57
Intel 40, 73, 105
  iPSC/860 95
  Paragon 45
  x86 160
intended
  nondeterminism 245, 246
  nondeterministic events 200
  race conditions 217
interactive
  inspection 172, 181
intergroup communication 265
intermediate results 196
intermittence 67
internal sources 91
Internet 70
Internet Worm 70
interrupt service routines 91, 159
interval for message arrival 249
intragroup communication 265
intrusive replay 243
IPD - Intel Paragon Debugger 105
iPSC/2-Hypercube 111
iPSC/860 105
irreproducibility effect 87, 93, 103, 187, 228
isolated events
  isolated receive event 174
  isolated send event 174
isolation 106, 261
isomorphic event graphs 240
ISVL - Internet Software Visualization Laboratory 108
IVD - Interactive Visualization Debugger 95

J

jail sentences 71
Java Development Kit 74
JiTI - Just in Time Instrumenter 160

K

Kansas multiuser virtual world 108
key enabling technologies 34

L

LAM 53
Lamport
  clocks 136, 165, 244
  ordering mechanism 136
  scalar time 147
  timeline view 117
languages
  EARL 111
  FORMAN 118
  GRAPNEL 99, 115
  HPF 95
last event 189
latency 86
Lawrence Livermore National Laboratory 37
learning rate 75
lethal errors 68
level
  of abstraction 118, 260
  of event generation 130, 165
  of granularity 244
  of monitoring
    minimum, medium, maximum 176
life-cycle 56, 60
life-cycle models
  cyclic models 56
  linear phases model 56
  phases model 56
  spiral model 56
  waterfall model 56
linear phases model 56
line-number 182
Linpack 40
livelock 87
load imbalance 86
LOC - Lines Of Code 75
local
  breakpoints 188
  clock 135
  real-time clock 135
location of event 130
logging 128
logical
  clock 179, 182
    approach 100
    counter 164, 165
    display 146
  time display 181

timestamps 135
long-lasting programs 82
loop detection 263
Los Alamos National Laboratory 37, 40
low level communication-library 155
low level events 118
LRR - Lehrstuhl für Rechnertechnik
und Rechnerorganisaton 107
Lufthansa 68

M

Mach operating system 95
machine level 131
macro nodes 269
macros 157, 159
macrostep 98
MAD - Monitoring And Debugging
environment 120
Make 79
Mannheim Supercomputer conference 40
manual abstraction 262
Mark II 67
massively parallel 36, 37, 39,
45–46, 258, 274
master-slave 260
master-worker 88
matrix clocks 139
matrix time 139
maze effect 85, 93
meaningful partial cuts 188
memory access patterns 86, 116
memory consumption 240
MEMSY - Modular Expandable
Multiprocessor System 110
message
arrival 245
arrival order 248
breakpoints 188
contents identifier 152
flow graph 116
order 207
origin 152
overtaking 203, 207, 214, 241
queue inspection 129
races 92
type 152
message-passing 50, 52–53
messages
pending 191, 207
racing 205, 209, 219, 255
winning 205, 209
Metacomputer 44
Metasystems 44
methods for event ordering 134
Microsoft 73
Millennium Baby 71
Millennium Bug 29, 71
MIMD 48
MIMD multiprocessor 251
MIPS 46
MIPS R10000 processor 252
Mississippi State University 53
modelling reality 88
module test 57
monitor 109, 155
buffer 177
intrusion 100
operations 160, 244
overhead 159
amount of 180
overhead removal 250
monitoring 60, 110, 128, 132
component 108
delay 162
hardware, software, hybrid 122
hybrid 111
isolated receives 176
software 111
Moore’s law 41–42
moth 67
MPI 52, 192, 202, 207,
231, 232, 234, 242, 244
communicator 153
Forum 53
profiling interface 159
standard 150, 152
MPI - Message Passing Interface 50
MPICH 53
MPMD - Multiple Program Multiple Data 48
MPMD programming model 260
MPP - Massively Parallel Processors 43
MTBF - Mean Time Between Failure 73
MTBI - Mean Time Between Interrupt 72
MUCH - MUltiprocessor Class Hierarchy 121
multiple
event streams 165
observers 133
sequential bugs 85
mutual exclusion 92
Myrinet 44, 53

N

NAS benchmark 42
Naval Oceanographic Office 41
NCU - Network Communication Unit 159
nCUBE 52, 95, 105

nCUBE 2 multiprocessor 111, 120, 154, 159, 208, 251
NEC
SX-4 43
neighboring processes 271
network
communication unit 159
conflicts 86
throughput 86
networks
ATM 44
Ethernet 44
FDDI 44
Myrinet 44
Nines Problem 72
nonartifact races 93
nonblocking
receive 151, 169, 197, 216
receive event 152
send 151
send event 151
tests 198
nondeterminacy 89
nondeterminism 87, 196
accidental 89
intended 245
on the interconnection network 86
sources of 91
testing cycle 102, 228
unintended 89
nondeterministic
events 197, 238
parallel programs 87–101, 150
programs 133
receive events 241
receives 244
non-overtaking 234
message transmission 208
systems 244
non-repeatability 93
NOPE - NOndeterministic Program Evaluator 112, 121, 215
not happened before relation 141
notorious errors 67–72
NP-complete problems 88
nuclear stockpile 37
number of possible executions 229
number of processes 258
instrumentation 158
object file formats 160
objects of interest 127
observability problem 96, 131, 132, 140
observable effects 126
observation 215
component 108, 109
overhead 103
observer notification 134
occasional errors 67
OCM - OMIS Compliant Monitor 112, 163
Ohio Supercomputing Center 53
OMIS 112
OMIS - Online Monitoring Interface Specification 107
omission races 92
omnipotent viewpoint 133
on-line 109, 132, 262
analysis 162
debugging 95
visualization 168
on-the-fly race detection 93
On-the-Fly Replay 96
OpenMP 52
operating system 155
instrumentation 160
operational arguments 164
optimal storage of trace data 244
order
concurrent 140
message arrival 248
of messages 207
sequential 131, 140, 241
ordering
absence of 139
arbitrary 133
enforced causal 139
errors 98
incorrect 133
methods for 134
of events 274
partial 140, 165
Origin 2000 40
output checking 60
overtaking
messages 203
of messages 207, 214, 234, 241
owner-compute 51

O

Oak Ridge National Laboratory 52, 111
object code
format 160

P

p4 - Portable Programs for Parallel Processors 52
PAM-CRASH 89

pan 184
parallel
finite element solver 264
I/O..... 165, 242
massively 36
slicing 260
trivially 47
parameters
accepted 164
supposed 164
PARASIT - PARAllel SImulation Tool 112, 121
parser 159
parsing 158
partial
cut 188, 220
order..... 118
ordering 140, 165
partially ordered
clocks..... 135
logical clocks..... 137
path creation 208
Patriot missiles..... 68
pattern
database 273
matching 119, 263, 267
patterns 118
communication 232
dimension exchange 271
event graph 245, 254
global exchange..... 271
in object oriented programs..... 118
simple exchange 269
PCL - Performance Counter Library 111
PDT - Parallel Debugging Tool..... 95
pending messages 191, 207
Pentium Bug 73
PerfAPI/PAPI - Performance
Application Programming Interface..... 111
performance
bottlenecks..... 180
hot spots..... 127
tuning..... 56
perturbation..... 101
PetaFlops project 37
Petri Nets 116
phase time..... 147
phases model 56
phreaks..... 70
physical time view..... 147
PICL 52
PICL - Portable Instrumented
Communication Library 52, 111, 159
piggybacked
clock 136, 241
sequence number 208
pipeline status 127
Poet - Partial Order Event Tracer..... 119
POET - Problem-Oriented
Event Trace interface function library .. 119
point-of-exchange 219
point-to-point communication..... 53, 154, 207
Poisson’s equation..... 147, 178, 185, 192
portability 51, 105, 159
POSIX Threads 52
possible executions..... 229
post/wait syncronization..... 92
post-mortem 109, 262
analysis..... 162
debugging..... 95
potential of reduction 245
PowerPC..... 160
power-plants..... 71
PPD - Parallel Program Debugger 82
precedence condition..... 139
preparation part 162
primitive events..... 118
print statement..... 113
probabilities..... 249
Probe Effect..... 83, 99, 250
definition 99
probe effect..... 87, 102
problems
completeness 96, 102
observability..... 96
vector clock..... 167
process
grid 117
grouping and hiding 265
groups..... 260
history 81, 274
isolation..... 275
lock-up 175
start..... 129
termination 129
processes
hiding of 266
neighboring 271
numbers of 258
processing
throughput 47
processor speed 86
profiling..... 127
interface..... 159
program
behavior..... 126
complexity..... 261
comprehension 114
context information..... 83
correctness..... 59, 200
counter..... 127
failure 161, 172
graph 116

re-execution 187
slicing 122, 185, 259
start 129
states 126
termination 78, 129
understanding 246, 261
programmer's laziness 89, 200, 232
promiscuous receives 198
propagation delay 134
protocol-based race detection 92
Ptools - Parallel Tools Consortium 107, 111
PVM 52, 107, 207, 244
PVM - Parallel Virtual Machine 50, 52, 154
PVP - Parallel Vector Processors 43

Q

quality assurance 59
quantum mechanics 83, 116

R

race condition candidates 98, 196, 206, 217, 230, 243, 244
race conditions 91–93
analysis 185
formal definition 91
intended 217
message-passing 92
nonartifact 93
shared variable 92
types of 92
unintended 217
racing messages 205, 209, 219, 255
random number generation 91, 129, 151, 156, 169, 182, 197, 216, 218
range of reduction 245
ray tracing, parallel 88
RCS 79
RDS - Race Detection Testing Strategies 98
real-time
clocks 164, 168, 179, 181
timestamps 135
real-time global clock 135
real-time local clock 135
real-world time 147
receives
blocking 216
nonblocking 169, 198, 216
nondeterministic 244
promiscuous 198
wild card 164
record&replay 94, 112, 213, 218, 274
control-driven 94
data-driven 94
optimal 95
recording 128
Redmond 73
reduction 106, 261
methods 259
re-execution 80, 94, 98, 172, 187, 192, 196
iterative 85
on demand 82
register contents 127
regular grammars 106
relation
concurrent events 141
concurrent order 140, 144
happened before 140, 165, 189
not happened before 141
on patterns 270
sequential order 140, 143
relational operations 269
reliability 59, 72
removal
monitor overhead 250
rendezvous communication 100
replay
buffer 274
intrusive 243
required test-and-fix iterations 76
requirements
debugger 105
parallel computing 36
Research Center Jülich 111
responsible events 201
restricted
race condition candidates 210
set of racing messages 210
results 196
reverse execution 80
rollback recovery 122
ROLT method 244
ROLT^{MP} 244
root errors 122
rotation 184
round robin 269
run-time observation 61
runtime-failure 267

S

S^{2}MP - Scalable Shared-memory MultiProcessing 252
Sandia National Laboratories 37, 40

satellite Clementine 69
scalability 53
scalable patterns 268
scalar time 139
scaling 267
SCCS 79
scheduling decisions 86
scope
  of attention 106
  of event 130
  of test reduction 245
scrolling 181
SDDF - Self-Defining Data Format 112
SDI - Strategic Defense Initiative 72
selection operations 269
semantic context 172
semi-automatic abstraction 262
Sequent Symmetry 82
sequential
  happened before relation 189
  order 140, 241
  order relation 143
sequential order of events 131
SGI 40
  ASCI Blue Mountain 252
  Cray T3E 45, 111
  IRIX 252
  Origin 2000 ..... 46, 111, 121, 153, 252
  Power Challenge 44
shared-variable 50
  paradigm 145
  programs 151, 244
  race conditions 92
SIMD 48
simple
  checking 78, 97
  cut 188
  exchange pattern 269
SIMPLE - Source related Integrated
    Multiprocessor and -computer
    Performance evaluation,
    visualization and modeLing
    Environment 110
simplified finger-print mechanism 242
simultaneous events
  events
    simultaneous 139
single-stepping 85
SISD, SIMD, MISD, MIMD 43
SKaMPI 42
slices 259
  time 274
slicing 185, 259
  backward 260
  forward 260
  parallel 260
slot machines 71
SMP - Symmetric Multiprocessors 43
SNI
  VP200 41
software
  life cycle 56
  quality 59
  reliability 59
  stable 59
  visualization 114
Solaris 52
source code 104
  connection 158, 164
  level 131
  reference 182
sources of nondeterminism 91, 196
SP 40
SP - Scalable Powerparallel 46
space-time diagram 116, 145, 168, 178, 185
SPARC 160
SPEC benchmark 42
spiral model 56
SPMD - Single Program Multiple Data 48, 255
SPMD programming model 260
SPPS - Serial Program Parallel System 47
spy satellite 71
SR 8000 40
stability 72
stable software 59
stampede effect 87
standard pattern library 273
standards
  HPD 108
  OMIS 107
Stanford DASH 44
state changes 126–131
state-time animation 117
static
  analysis 59, 60
  call graph 116
  object code 160
  slicing 259
statistical testing 79, 106
statistics 180
steering 215, 275
STEPS - Structural TEsting of
    Parallel Software 115
storage part 162
strongly complete collective breakpoints 188
subsequent errors 141
Sun Microsystems 74
SUN Solaris 160
supercomputer 38, 39, 40
  definition 39
  first 38
  ILLIAC IV 38

Supercomputing conference 40
supposed parameters 164
suspected error location 82
suspicious errors 201
system
  calls 91
  interrupts 129
  memory 127

T

T3E 40, 43
T3E 1200 40
T3E 900 40
task-parallel 48
tax computer 69
TCE - Time Controlled Environment 98
TDL - Trace Description Language 119
Technical University Munich 107
telephone network 70
temporal distance 147
Teraflop computing 36–37
terminal event 144, 173
termination 88
test
  completion criterion 102
  nonblocking 198
  reduction
    potential 245
    range 245
    scope 245
testing
  amount of 245
  automatic 97
  black-box 57
  complete 230, 235
  cycle 78, 80, 150
    input 102
    nondeterministic 102
  evolutionary 79
  functionality 56
  Good Enough 58, 99, 244
  large projects 80
  problems 57
  process 60
  statistical 79, 106
  white-box 57
testing and debugging cycles 133
testing and tuning 56–57
textual substitution 158
Therac-25 68
Thinking Machines 42
three-dimensional data arrays 184
throughput processing 47
tightly coupled systems 135
time
  communication 180
  computation 180
  histogram 117
  matrix 139
  reduction 118
  scalar 139
  scale 179, 181
  slices 274
  vector 139
  virtual 100
  zones
    PAST 189
    PAST, FUTURE, PARALLEL 142
Time Tunnel Virtual Reality system 146
timestamping 132
  mechanisms 135
timestamps
  logical 135
  real-time 135
timing information 164
TM
  CM-5 42
Top 500 ..... 39, 45, 53, 252
topology 251
totally ordered
  clocks 135
  logical clocks 136
Totalview 105
trace file flushing 177, 179
trace-driven simulation 94
tracing 62, 128
  incremental 243
traffic violations 69
transformation 106, 261
Transputer 110
tree 269
  of executions 235, 241
trivially parallel 47
truthful debugging 83
type
  of clocks 134
  of event 130

U

ubiquitous
  computer 29
  computing 47
UK Centre for Science 41
UK Meteorological Office 40
unbalanced hardware-software relations 180
uncritical nondeterministic events 246

unfavorable cuts 191
UNICOS 45
UNIFY 53
unintended
    nondeterminism 89, 93, 246
    nondeterministic events 200
    race conditions 217
unique execution identifier 240
University Erlangen-Nürnberg 110
University of Tennessee 40
University of Tokyo 40
UNIX 74
    sockets 162
unordered
    events 139
    races 92
unpredictable changes 197
unsynchronized clocks 146
uranium depot 71
usability 105
user
    defined event 129
    feedback 245
    knowledge 268

V

validation 57
validity and correctness
    of input 78
    of results 79
variable
    backtracking 260
    inspection 151, 156, 169, 182
vector
    clock 138, 165, 182, 190
    time 139, 147
verification 57
vertical abstraction 265
view kernel VK 116
virtual memory systems 82
Virtual Reality 116, 146
virtual time 100
visual
    attributes 184
    programming 113
    representation 261
visualization
    component 108
    on-line 168
    taxonomies 115

W

waterfall model 56
wild card 164, 182, 202, 210, 234
Windows 73, 74, 160
winning message 205, 209, 211
World Wide Web 108

X

xy-contour plots 116

Y

Y2k 29, 71
year 2000 71

Z

ZM4 - Zählmonitor 4 110
zoom 181, 184, 267
Zuse
    Z1-Z3 computers 38

Appendices

Appendix A

This small example code represents a simple parallel program based on MPI. It was used in Chapter 6 to describe the basic monitoring approach, with a visualization as a space-time diagram in Section 6.6:

```
/* perform some arbitrary computation */
void do_something();

/* prepare the data array and the info structure */
void do_array(double data[][], trcArrayInfo *info);

/* simple test program for MPI */
int main(int argc, char **argv)
{
    char           proc;
    int            myrank, mysize;
    int            result;
    trcArrayInfo   info;
    double         data[500][500];

    MPI_Status     stat;
    MPI_Request    req1, req2;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
    MPI_Comm_size(MPI_COMM_WORLD, &mysize);

    /* round robin - nonblocking send, blocking receive */
    if(myrank==0) {
            proc=random()%(mysize-1);
            MPI_Isend(&proc,1,MPI_CHAR,
                    (myrank+1)%(mysize-1),1,
                    MPI_COMM_WORLD,&req1);
            MPI_Recv (&proc,1,MPI_CHAR,
                    (myrank+mysize-2)%(mysize-1),1,
                    MPI_COMM_WORLD,&stat);
    }
    else if(myrank < mysize - 1) {
            MPI_Recv (&proc,1,MPI_CHAR,
                    (myrank+mysize-2)%(mysize-1),1,
                    MPI_COMM_WORLD,&stat);
            MPI_Isend(&proc,1,MPI_CHAR,(myrank+1)%(mysize-1),1,
                    MPI_COMM_WORLD,&req1);
    }

    /* nonblocking receive */
    else {
            MPI_Irecv(&proc,1,MPI_CHAR,MPI_ANY_SOURCE,1,
                    MPI_COMM_WORLD,&req1);
            do {
```

```
                        do_something();
                        MPI_Test(&req1,&result,&stat);
                } while(!result);
        }

        if(myrank==proc)
                MPI_Isend(&proc,1,MPI_CHAR,mysize-1,1,
                        MPI_COMM_WORLD,&req2);
        else
                monVARTRACE(&proc,MON_CHAR,MON_CHAR_SIZE,1);

        do_array(data,&info);
        monARRAYTRACE(data,&info);

        MPI_Finalize();
}
```

Appendix B

This example code represents a nondeterministic parallel program based on MPI, which tries to determine a global characteristic (i.e. maximum or minimum) of some locally distributed data items by distance doubling. This means, that the communication partners are determined by doubling the communication distance during each iteration. Afterwards the resulting global characteristic as available on all processes is printed. This example was used in Chapter 8 to describe visualization of nondeterminism analysis:

```
int main(int argc, char *argv[])
{
    int myrank, mysize;
    int mymax, max;
    int dist = 1;
    MPI_Status stat;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
    MPI_Comm_size(MPI_COMM_WORLD, &mysize);

    srand ((int) time(NULL));
    mymax = myrank*rand()%255;

    while (dist < mysize) {
            MPI_Send(&mymax, 1, MPI_INT,
                     ((myrank+dist)%mysize), 3,
                     MPI_COMM_WORLD);

            MPI_Recv(&max, 1, MPI_INT,
                     MPI_ANY_SOURCE, 3,
                     MPI_COMM_WORLD, &stat);

            if (max > mymax) mymax = max;
            dist += dist;
    }

    printf("%d: %d\n",myrank, mymax);

    MPI_Finalize();
}
```

Appendix C

This example code represents a simple nondeterministic parallel program based on MPI, where all slave processes are sending some data to a master process. The printed result of this program is the order of incoming messages, which are identified by their origin process. It was used in Chapter 9 to describe automatic nondeterminism analysis:

```
/* do some random waiting */
void delay();

int main(int argc, char *argv[])
{
    int i,j, rank,size,msg;
    MPI_Status status;
    MPI_Init(&argc,&argv);

    MPI_Comm_rank(MPI_COMM_WORLD,&rank);
    MPI_Comm_size(MPI_COMM_WORLD,&size);

    if(rank==0) {
            for(i=1;i<size;i++) {
                    MPI_Recv(&msg,1,MPI_INT,MPI_ANY_SOURCE,
                             0,MPI_COMM_WORLD,&status);
                    printf("%d",msg);
            }
            printf("\n");
    }
    else {
            delay();
            MPI_Send(&rank,1,MPI_INT,0,0,MPI_COMM_WORLD);
    }

    MPI_Finalize();
}
```

Appendix D

This example code represents a simple parallel program based on MPI that is often used to perform data circulation in hypercube architectures. The result of this code on each process is the summation of each value that passes by. Due to a mistake, the program exhibits nondeterministic behavior and may deliver incorrect results during some executions. It was used in Chapter 9 to describe automatic nondeterminism analysis:

```
/* compute the next communication partner
   for data circulation */
int f(int i)
{
    int n=0;
    while (i%2==0) { i/=2; n++; }
    return n;
}

int main (int argc, char **argv)
{
    int rank, size, i, sum, msg=0, target, pNum;
    MPI_Status status;

    MPI_Init (&argc, &argv);
    MPI_Comm_rank (MPI_COMM_WORLD, &rank);
    MPI_Comm_size (MPI_COMM_WORLD, &size);

    msg = rank;
    sum = rank;

    for (i=1; i<size; i++) {
            target = rank^(1<<f(i));
            MPI_Send (&msg, 1, MPI_INT, target, 1,
                    MPI_COMM_WORLD);
            MPI_Recv (&msg, 1, MPI_INT, MPI_ANY_SOURCE, 1,
                    MPI_COMM_WORLD, &status);
            sum+=msg;
    }
    printf ("%d: %d\n", rank, sum);

    MPI_Finalize ();
    return 0;
}
```

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Dissertation selbständig und ohne fremde Hilfe verfasst habe. Ich habe dazu keine weiteren als die angeführten Hilfsmittel benutzt und die aus anderen Quellen entnommenen Stellen als solche gekennzeichnet.

Linz, September 2000 Dipl.-Ing. Dieter Kranzlmüller

Curriculum Vitae

Persönliche Daten:

Name:	Dipl.-Ing. Dieter August Kranzlmüller
Familienstand:	verheiratet seit 14. März 1997 mit Judith Ingrid
Kinder:	Miriam, geboren 07.12.1998
Staatsbürgerschaft:	Österreich
Wohnadresse:	A-4040 Linz, Landgutstraße 11
20.04.1969	geboren in Schwertberg OÖ als Sohn der kfm. Angestellten Ursula Kranzlmüller und des Karosseurs August Kranzlmüller

Ausbildung/Beschäftigung:

1975 - 1979	Volksschule Schwertberg
1979 - 1983	Hauptschule Schwertberg
1983 - 1987	Bundes-Oberstufenrealgymnasium Perg
1987 - 1992	Studium der Informatik an der Johannes Kepler Universität Linz
01.07.1992	Sponsion zum Diplom-Ingenieur der Informatik
1989 - 1992	Angestellter, Technische EDV, Firma E. Eisenbeiss, Enns
1993	Leiter, Integrierte Informationsverarbeitung, Firma E. Eisenbeiss, Enns
01.04.1993-31.03.1995	Vertragsassistent, Institut für Informatik, Abteilung für Graphische und Parallele Datenverarbeitung Joh. Kepler Universität Linz
01.04.1995-30.11.1999	Universitätsassistent, Institut für Technische Informatik und Telematik, Abteilung für Graphische und Parallele Datenverarbeitung, Joh. Kepler Universität Linz
seit 01.12.2000	Wissenschaftlicher Mitarbeiter, Institut für Technische Informatik und Telematik, Abteilung für Graphische und Parallele Datenverarbeitung, Joh. Kepler Universität Linz

Veröffentlichungen:

Diplomarbeit:	Kranzlmüller, D., "*CIM-Konzept in der Getriebefertigung*" Institut für Systemwissenschaften, Abteilung für Systemtechnik und Automation, Joh. Kepler Universität Linz, 1992
Sonstiges:	57 Artikel in wissenschaftlichen Journalen, Büchern, Konferenzbänden, und technischen Berichten.