DISTRIBUTED DEADLOCK DETECTION AND ITS APPLICATION TO REAL-TIME SYSTEMS

Chia-Shiang Shih

COINS Technical Report 92-60
September 1992

DISTRIBUTED DEADLOCK DETECTION AND ITS APPLICATION TO REAL-TIME SYSTEMS

A Dissertation Presented

by

CHIA-SHIANG SHIH

Submitted to the Graduate School of the
University of Massachusetts in partial fulfillment
of the requirements for the degree of

DOCTOR OF PHILOSOPHY

September 1992

Department of Electrical and Computer Engineering

© Copyright by Chia-Shiang Shih 1992
All Rights Reserved

This work was supported, in part, by the Charles Stark Draper Laboratory, Inc., and by NSF under grants IRI-8908693, CDA-8922572, and IRI-9114197.

DISTRIBUTED DEADLOCK DETECTION AND ITS APPLICATION TO REAL-TIME SYSTEMS

A Dissertation Presented

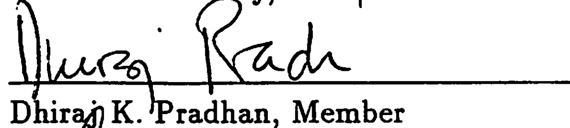
by

CHIA-SHIANG SHIH

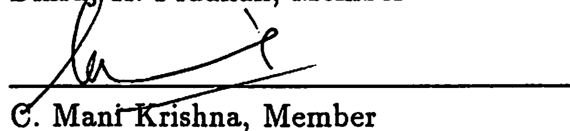
Approved as to style and content by:

John A. Stankovic, Chair

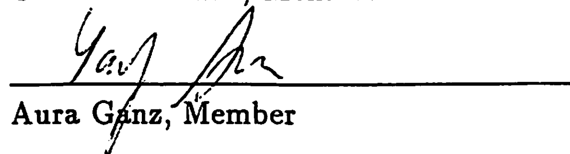
Donald F. Towsley, Member



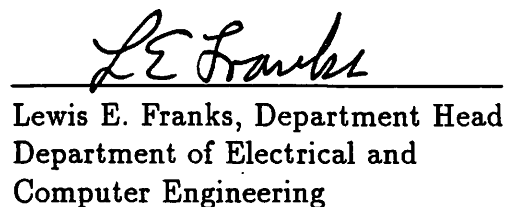
Dhiraj K. Pradhan, Member



C. Mani Krishna, Member



Aura Ganz, Member



Lewis E. Franks, Department Head
Department of Electrical and
Computer Engineering

ACKNOWLEDGMENTS

It has been my privilege and good fortune to work with my advisor Professor John A. Stankovic. He has been extraordinarily patient and supportive, having been always available for discussion and responding speedily to research reports. I would like to take this opportunity to thank him for his continuous support and guidance during the course of this research.

Thanks are also due to Professors Donald F. Towsley and Krithi Ramamritham for working together in carrying out this research and for their constructive comments and suggestions. Comments from my other committee members, Professors Dhiraj K. Pradhan, C. Mani Krishna, and Aura Ganz, have greatly benefited the dissertation. Also, my special thanks go to Dr. Walter H. Kohler for introducing me to the area of distributed computing and gave me an opportunity to do research in the CARAT group.

I would like to acknowledge my colleagues in the RT-CARAT research group (at one time or another) Dr. Asit Dan, Dr. Jiandong Huang, Bhaskar Purimetla, and Rajendran M. Sivasankaran for helpful discussions during this research.

Many thanks go to my officemates in the Distributed Computing Systems Laboratory for making the long hours of working pleasant. I greatly appreciate the assistance of the secretaries of the department, in particular of Betty Hardy, and the staff of RCF and ECS.

Finally, and most importantly, I would like to express my gratitude to my parents who have always believed in me and have been patiently waiting for me to complete this long study. Also, I would like to share this work with my wife, Yuan-Chuan, and my son, Kevin, who have suffered as much as I have while making this dissertation complete.

ABSTRACT

DISTRIBUTED DEADLOCK DETECTION AND ITS APPLICATION TO REAL-TIME SYSTEMS

SEPTEMBER 1992

CHIA-SHIANG SHIH

B.S., NATIONAL TAIWAN UNIVERSITY, TAIWAN, R.O.C.
M.S.E.C.E., UNIVERSITY OF MASSACHUSETTS
Ph.D., UNIVERSITY OF MASSACHUSETTS

Directed by: Professor John A. Stankovic

Deadlock is one of the most serious problems in multitasking concurrent programming systems. The problem is further complicated when the underlying system is distributed and when tasks have timing constraints. Although deadlock detection has been studied to some extent in database systems and timesharing operating systems, it has not been widely used in real-time systems. In this dissertation, we analyze, develop, and formally validate deadlock detection algorithms in distributed environments. The results are then extended to real-time applications by considering timing constraints in the algorithms. Also, attempts are made to apply these algorithms to real systems such as Ada environments. The Ada runtime environment is used as a system model to address problems and issues related to distributed deadlock detection in real-time applications.

One of the major achievements of this dissertation study is the development of a systematic method for the design and the verification of distributed deadlock detection algorithms. This novel methodology is applied to a variety of deadlock models ranging from simple (resource) deadlock models (e.g., the Single-Resource

request model) to complex (communication) deadlock models (e.g., the OR request model). The developed algorithms are then extended for real-time applications. For verification and performance evaluation, these algorithms are implemented in a distributed real-time database testbed called RT-CARAT (Real-Time Concurrency And Recovery Algorithm Testbed) where transaction timing constraints and soft real-time scheduling protocols are supported. Various experiments are conducted to study and evaluate these algorithms.

Our experimental results show that distributed deadlock detection can be very efficient. Compared to a baseline scheme "break deadlocks by deadline," any of these deadlock detection algorithms can significantly improve the overall system performance. Also, the deadlock detection/resolution approach outperforms another class of baseline schemes "timeout from waiting state and retry" when the deadlocks are simple and short (e.g., the simple resource deadlocks). However, the former approach performs worse than the latter when the average deadlock length is long (e.g., the complex communication deadlocks). The results also suggest that in real-time applications when the average wait-for path is long, an integrated solution is necessary to detect and resolve both the deadlocks and the long wait-for paths.

TABLE OF CONTENTS

Page

ACKNOWLEDGMENTS iv
ABSTRACT v
LIST OF TABLES xi
LIST OF FIGURES xii

Chapter

1. INTRODUCTION 1

1.1 Motivation 1
1.2 Contributions of the Dissertation 3
1.3 Plan of the Dissertation 7

2. THE DEADLOCK PROBLEM 9

2.1 Wait-For Graphs 9
2.2 Concepts from Graph Theory 13
2.3 Models of Distributed Concurrent Programming Systems 15

2.3.1 CSP – Communicating Sequential Processes 15
2.3.2 DP – Distributed Processes 16
2.3.3 Ada – A Distributed Concurrent Programming Environment 17

2.4 Deadlock Models 22

2.4.1 Resource and Communication Deadlock Models 23
2.4.2 General Resource System Model 25
2.4.3 A Hierarchy of Deadlock Models 27

2.4.3.1 Single-Resource Model 27
2.4.3.2 AND Model 28
2.4.3.3 OR Model 28
2.4.3.4 AND-OR Model 30
2.4.3.5 C(n,k) Model 30
2.4.3.6 Unrestricted Model 31

2.5 Deadlock Detection in Different Deadlock Models 32
2.6 Distributed Deadlock Detection and Resolution in Real-Time Systems 36

2.6.1 Deadlock Problems in Distributed Real-Time Systems 37

2.6.2 Criteria in Designing Distributed Deadlock Detection and Resolution Algorithms for Real-Time Systems 38

3. A SURVEY OF THE DISTRIBUTED DEADLOCK DETECTION AND RESOLUTION ALGORITHMS 44

3.1 Centralized Algorithms for Distributed Deadlock Detection 44
3.2 Distributed Algorithms for Distributed Deadlock Detection 47

3.2.1 Path-Pushing Algorithms 49
3.2.2 Edge-Chasing Algorithms 54
3.2.3 Diffusing Computations 56
3.2.4 Global State Detection 60

3.3 Hierarchical Algorithms for Distributed Deadlock Detection 62
3.4 Summary of the Distributed Deadlock Detection Algorithms 63

4. A METHODOLOGY FOR THE DEVELOPMENT OF DISTRIBUTED DEADLOCK DETECTION ALGORITHMS 67

4.1 Characterizations and Assumptions of Distributed Deadlock Detection 68
4.2 Overview of the Methodology 75
4.3 Distributed Deadlock Detection in Static Systems 78

4.3.1 Local View Definition for Distributed Deadlock Detection .. 79
4.3.2 Deadlock Core Set 80
4.3.3 Probe-Based Static Deadlock Detection Computation 81

4.4 Distributed Deadlock Detection in Dynamic Systems 83

4.4.1 From Global View 84

4.4.1.1 The Snapshot Approach 84
4.4.1.2 The Synchronization Mechanism Approach 88

4.4.2 From Local View 94

4.4.2.1 Safety Concern of Dynamic Algorithms 94
4.4.2.2 Derivation and Verification of Dynamic Algorithms 100
4.4.2.3 Progress Concern of Dynamic Algorithms 104

4.5 Concluding Remarks 108

5. CYCLE DETECTION 111

5.1 Assumptions of the Algorithms Concerning the Real-Time Constraints 112
5.2 The Principles of Cycle Detection 113
5.3 Algorithm for the Single-Resource Model 116

5.3.1 Static Algorithms for the Single-Resource Model 116
5.3.2 Dynamic Algorithm for the Single-Resource Model 117
5.3.3 Real-Time Constraints in the Single-Resource Algorithm 120
5.3.4 The Single-Resource Algorithm 120

5.4 Algorithm for the AND Model 125

5.4.1 Static Algorithm for the AND Model 125
5.4.2 Dynamic Algorithm for the AND Model 128
5.4.3 Real-Time Constraints in the AND Algorithm 130
5.4.4 The AND Algorithm 131

5.5 Concluding Remarks 137

6. KNOT DETECTION 139

6.1 Formal Definition of the OR Model Deadlocks 140
6.2 Knot Detection in Static Systems 142

6.2.1 The Principles of Knot Detection 143
6.2.2 Knot Detection in Finite Time 149
6.2.3 Single Detection of Knots 155

6.3 Knot Detection in Dynamic Systems 166

6.3.1 A Guaranteed Knot Detection Algorithm for Dynamic Systems 167
6.3.2 Single Detection of Knots in Dynamic Systems 176

6.4 Knot Detection with Timing Constraints 183
6.5 Application to Ada Environments 187

6.5.1 Ada Rendezvous and Task Termination 188
6.5.2 Design Assumptions Concerning the Ada Runtime Environments 189
6.5.3 Algorithm for the Detection of Ada Rendezvous Deadlocks and Task Terminations 191

6.6 Comparison to Other Work 200

6.6.1 The OR Deadlock and Knot Detection Algorithms 201

6.6.2 Deadlock Detection Algorithms in Ada Applications 202
6.7 Concluding Remarks 204
7. IMPLEMENTATION AND PERFORMANCE EVALUATION 206
7.1 The Real-Time Database Testbed RT-CARAT 206
7.2 The Schemes for the Experiments 211
7.3 Parameters and Performance Metrics 213
7.4 Experimental Results 215
7.4.1 Experiment 0: Algorithm Verification 217
7.4.2 Experiment 1: The AND Deadlock Model 218
7.4.3 Experiment 2: The CPU Scheduling and the Deadlock Detection 221
7.4.4 Experiment 3: The Single-Resource Deadlock Model 223
7.4.5 Experiment 4: The OR Deadlock Model 225
7.5 Summary of Results and Conclusions 228
8. SUMMARY AND FUTURE DIRECTIONS 255
8.1 Summary of Conclusions 255
8.2 Future Directions 261

APPENDICES

A. LIST OF ABBREVIATIONS 264
B. LIST OF NOTATIONS 265

BIBLIOGRAPHY 267

LIST OF TABLES

Table | Page

3.1 Performance of the Surveyed Algorithms 66

7.1 Experiment 1 Parameter Settings 219

7.2 Experiment 2 Parameter Settings 222

7.3 Experiment 3 Parameter Settings 224

7.4 Experiment 4 Parameter Settings 226

7.5 Mean Cyclic Wait Length 227

LIST OF FIGURES

Figure | Page

2.1 Examples of the system state graph 10

4.1 Example of a deadlock set 80

4.2 Example of the inconsistencies in a DGRG 98

5.1 Data structure for probes in the Single-Resource Algorithm 121

5.2 Data structure for tasks in the Single-Resource Algorithm 121

5.3 Data structure for resources in the Single-Resource Algorithm 122

5.4 Procedure for theprobe initiation in the Single-Resource Algorithm . 123

5.5 Procedure for tasks handling received probes in the Single-Resource Algorithm 124

5.6 Procedure for resources handling received probes in the Single-Resource Algorithm 124

5.7 Data structure for probes in the AND Algorithm 132

5.8 Data structure for tasks in the AND Algorithm 132

5.9 Data structure for resources in the AND Algorithm 133

5.10 Procedure for the probe initiation in the AND Algorithm 134

5.11 Procedure for tasks handling received probes in the AND Algorithm 135

5.12 Procedure for resources handling received probes in the AND Algorithm 136

6.1 An example of ties, stable ties, and knots 142

6.2 Example of an infinite loop in a knot 147

6.3 Data structure for probes in the OR Algorithm 192

6.4 Data structure for tasks in the OR Algorithm 193

6.5 Data structure for resources in the OR Algorithm 193

6.6 Procedure for the probe initiation in the OR Algorithm 195

6.7 Procedure for tasks handling received B-probes in the OR Algorithm 196

6.8 Procedure for tasks handling received F-probes in the OR Algorithm · · · · · · 197

6.9 Procedure for resources handling received B-probes in the OR Algorithm · · · · · · 198

6.10 Procedure for resources handling received F-probes in the OR Algorithm · · · · · · 199

7.1 RT-CARAT processes and message structure · · · · · · 207

7.2 Deadline Guarantee Ratio, AND Model System, DB Size 1667 Pages · 230

7.3 Deadline Guarantee Ratio, AND Model System, DL Window [40-160,12-48] · · · · · · 230

7.4 Record Throughput, AND Model System, DB Size 1667 Pages · · · · 231

7.5 Record Throughput, AND Model System, DL Window [40-160,12-48] · 231

7.6 Locking Statistics, AND Model System, DB Size 1667 Pages · · · · · · 232

7.7 Locking Statistics, AND Model System, DL Window [40-160,12-48] · 232

7.8 CPU Utilization, AND Model System, DB Size 1667 Pages · · · · · · · 233

7.9 CPU Utilization, AND Model System, DL Window [40-160,12-48] · · 233

7.10 Probe Statistics, AND Model System, DB Size 1667 Pages · · · · · · · 234

7.11 Probe Statistics, AND Model System, DL Window [40-160,12-48] · · 234

7.12 Deadline Guarantee Ratio, AND Model System, Non-Real-Time Scheduling, DB Size 1000 Pages · · · · · · 235

7.13 Deadline Guarantee Ratio, AND Model System, EDF Scheduling, DB Size 1000 Pages · · · · · · 235

7.14 Deadline Guarantee Ratio, AND Model System, Non-Real-Time Scheduling, DL Window [20-80,8-32] · · · · · · 236

7.15 Deadline Guarantee Ratio, AND Model System, EDF Scheduling, DL Window [20-80,8-32] · · · · · · 236

7.16 Record Throughput, AND Model System, Non-Real-Time Scheduling, DB Size 1000 Pages · · · · · · 237

7.17 Record Throughput, AND Model System, EDF Scheduling, DB Size 1000 Pages · · · · · · 237

7.18 Record Throughput, AND Model System, Non-Real-Time Scheduling, DL Window [20-80,8-32] · · · · · · 238

7.19 Record Throughput, AND Model System, EDF Scheduling, DL Window [20-80,8-32] 238

7.20 Probe Statistics, AND Model System, Non-Real-Time Scheduling, DB Size 1000 Pages 239

7.21 Probe Statistics, AND Model System, EDF Scheduling, DB Size 1000 Pages 239

7.22 Probe Statistics, AND Model System, Non-Real-Time Scheduling, DL Window [20-80,8-32] 240

7.23 Probe Statistics, AND Model System, EDF Scheduling, DL Window [20-80,8-32] 240

7.24 Deadline Guarantee Ratio, Single-Resource System, 2 Slave Sites, DB Size 1667 Pages 241

7.25 Deadline Guarantee Ratio, Single-Resource System, 1 Slave Site, DB Size 1667 Pages 241

7.26 Deadline Guarantee Ratio, Single-Resource System, 2 Slave Sites, DL Window [40-160,12-48] 242

7.27 Deadline Guarantee Ratio, Single-Resource System, 1 Slave Site, DL Window [40-160,12-48] 242

7.28 Record Throughput, Single-Resource System, 2 Slave Sites, DB Size 1667 Pages 243

7.29 Record Throughput, Single-Resource System, 1 Slave Site, DB Size 1667 Pages 243

7.30 Record Throughput, Single-Resource System, 2 Slave Sites, DL Window [40-160,12-48] 244

7.31 Record Throughput, Single-Resource System, 1 Slave Site, DL Window [40-160,12-48] 244

7.32 Locking Statistics, Single-Resource System, 2 Slave Sites, DB Size 1667 Pages 245

7.33 Locking Statistics, Single-Resource System, 1 Slave Site, DB Size 1667 Pages 245

7.34 Locking Statistics, Single-Resource System, 2 Slave Sites, DL Window [40-160,12-48] 246

7.35 Locking Statistics, Single-Resource System, 1 Slave Site, DL Window [40-160,12-48] 246

7.36 Probe Initiations, Single-Resource System, 2 Slave Sites, DB Size 1667 Pages 247

7.37 Probe Initiations, Single-Resource System, 1 Slave Site, DB Size 1667 Pages 247

7.38 Probe Initiations, Single-Resource System, 2 Slave Sites, DL Window [40-160,12-48] 248

7.39 Probe Initiations, Single-Resource System, 1 Slave Site, DL Window [40-160,12-48] 248

7.40 Probe Message Statistics, Single-Resource System, 2 Slave Sites, DB Size 1667 Pages 249

7.41 Probe Message Statistics, Single-Resource System, 1 Slave Site, DB Size 1667 Pages 249

7.42 Probe Message Statistics, Single-Resource System, 2 Slave Sites, DL Window [40-160,12-48] 250

7.43 Probe Message Statistics, Single-Resource System, 1 Slave Site, DL Window [40-160,12-48] 250

7.44 Deadline Guarantee Ratio, OR Model System, Random Accessing Pattern, DB Size 50 Pages 251

7.45 Deadline Guarantee Ratio, OR Model System, Contiguous Accessing Pattern, DB Size 50 Pages 251

7.46 Deadline Guarantee Ratio, OR Model System, Random Accessing Pattern, DL Window [40-160,12-48] 252

7.47 Deadline Guarantee Ratio, OR Model System, Contiguous Accessing Pattern, DL Window [40-160,12-48] 252

7.48 Record Throughput, OR Model System, Random Accessing Pattern, DB Size 50 Pages 253

7.49 Record Throughput, OR Model System, Contiguous Accessing Pattern, DB Size 50 Pages 253

7.50 Record Throughput, OR Model System, Random Accessing Pattern, DL Window [40-160,12-48] 254

7.51 Record Throughput, OR Model System, Contiguous Accessing Pattern, DL Window [40-160,12-48] 254

CHAPTER 1

INTRODUCTION

1.1 Motivation

Deadlock is one of the most serious problems encountered in multitasking concurrent programming systems. As early as the 1960's the deadlock problem was recognized and analyzed (Dijkstra [40] described it as the *problem of the deadly embrace*). Deadlock occurs when one or more tasks in a system are blocked by each other forever and their requirements can never be satisfied. A deadlock situation may arise if and only if the following four resource competition conditions hold in a system simultaneously: (1) mutual exclusion, (2) hold and wait, (3) no preemption, and (4) circular wait. To some degree the last condition implies the other three. However, it is quite useful to consider each condition separately in analyzing and designing a deadlock free system.

Principally, there are three strategies for dealing with the deadlock problem:

1. Deadlock Prevention – by ensuring that at lease one of the deadlock conditions cannot hold,
2. Deadlock Avoidance – by providing *a priori* information so that the system can predict and avoid deadlock situations, and
3. Deadlock Detection – by detecting and recovering from deadlock states.

The first two strategies ensure that the system will *never* enter a deadlock state. Deadlock prevention is commonly achieved either by guaranteeing that tasks do not have to hold and wait on resources (e.g., by forcing all tasks to acquire resources a priori), or by allowing preemption (e.g., a task that holds the needed resource might be preempted by another task with a higher priority). For deadlock avoidance, a task proceeds if the resulting global state is checked and proved to be safe from deadlock. These methods carry the following drawbacks [124]:

- They are usually inefficient when applied in complex distributed systems. Deadlock prevention is inefficient because it decreases system concurrency by restricting the execution of the tasks to avoid at least one of the deadlock conditions. For deadlock avoidance, checking for a safe state is computationally expensive and inefficient. This inefficiency is especially significant in a complex distributed system due to the large numbers of tasks and resources.
- They are apt to fail in complex distributed systems. For example, if the tasks are required to acquire resources a priori, a group of tasks may become deadlocked in the resource-acquiring phase due to lack of a perfect global synchronization mechanism. Similarly, in the deadlock avoidance case, due to inconsistent local views caused by the imperfect synchronization mechanism, different sites may all find the states safe and grant the requests concurrently, although the final global state may turn out to be deadlocked.
- The requirements for deadlock prevention or avoidance may not be fulfilled. For instance, in many systems future resource requests are unpredictable which makes "a priori resource acquiring" deadlock prevention impossible.

Alternatively, by applying a deadlock detection strategy, the system is allowed to enter a deadlock state which is then detected and recovered from. The detection of deadlocks requires the examination of the system state (principally, the task/resource interactions) for the presence of cyclic waits. Once a deadlock is formed, it persists until it is detected and broken (the so called *stable property* of the deadlock problem). The deadlock detection computation can be performed in parallel with the other normal system activities, therefore, it may not have a serious impact on the system performance. Also, since certain deadlock detection algorithms can be embedded in the underlying operating system, they are able to extend the fault tolerance of software design faults even if a deadlock prevention or avoidance approach is used in the user application.

Yet another potential benefit of the "detection" strategy for deadlocks is that it may be integrated with other related problems. For example, many problems

appear in multitasking systems, such as livelock (a.k.a. effective deadlock or starvation), task termination, and orphan tasks, which must be detected dynamically at runtime. Some of these problems, e.g. task termination and orphan task problems, carry the same stable property as the deadlock problem. The detection of these system faulty states requires the examination of task/resource interactions which is similar to certain techniques used in deadlock detection. Certain deadlock detection algorithms, therefore, can be tailored for the detection of these problems and vice versa, without too much additional effort and overhead.

Considering distributed deadlock detection as part of *global state detection* is another example where deadlock detection could be integrated in the resolution of a related problem. For instance, if a global state detection algorithm is adopted to facilitate applications such as distributed debugging and distributed system monitoring, it can be extended to the detection of distributed deadlocks with little additional overhead.

In real-time systems, deadlock prevention and avoidance methods have received most of the attention and are the current "best" strategies. However, because of the drawbacks pointed out above these strategies might work successfully in relatively simple systems, but may be inefficient and very difficult to design and verify in more complex systems such as multiprocessors or distributed systems. Distributed deadlock detection, which is the focus of this research, has been studied in distributed database systems and distributed timesharing operating systems, but has not been widely used in real-time systems. In the rest of this dissertation, we will survey the related research work in conventional non-real-time environments, and propose extensions for distributed real-time systems.

1.2 Contributions of the Dissertation

Deadlock detection requires knowledge of a system's global state. In distributed environments, the global system states are usually not maintained at a centralized

place and, hence, deadlock detection in a distributed environment is far more complicated than that in a centralized system. In this dissertation, we analyze, develop, and formally validate deadlock detection algorithms in distributed environments. The results are then extended to real-time systems by considering timing constraints in the algorithms. Also, we attempt to apply our algorithms to real systems such as Ada environments. We use the Ada runtime environment as a system model to address problems and issues related to distributed real-time deadlock detections. To evaluate our proposed algorithms, they are implemented in a real-time database testbed called RT-CARAT[1] where transaction timing constraints and soft real-time scheduling protocols are supported. Various experiments are conducted to study and evaluate these algorithms. The major contributions of this dissertation study are summarized below:

- **Identify the problems and the issues of the deadlock detection in real-time systems**

  In the non-real-time systems, the "stable property" is an important characteristic of the deadlock problem. This property means a deadlock situation persists once it is formed. Many algorithms proposed in the literature assumed and utilized this property. In real-time systems, timing constraints are attached to the tasks. A task may be timed out from a state in which it is waiting. Deadlocks may not be stable if timing constraints are considered. In this dissertation, we describe the deadlock problems for real-time systems in which timing constraints are considered. Three types of problems: "Stable Deadlock," "Temporal Deadlock," and "Non-deadlocked Blocking" are identified and discussed.

  We adopt Knapp's [77] hierarchy of deadlock models for the analysis of the problem complexity. Based on the sufficient conditions for deadlock detection, we roughly divide the problem into four levels of complexity: (1) the Single-Resource model, (2) the AND model, (3) the OR model, and (4) the AND-OR and the C(n,k) models. We address how the deadlock problems can be solved

[1] Real-Time Concurrency And Recovery Algorithm Testbed [67].

in each of the four levels of problem complexity and how the timing constraints are considered in the possible solutions.

- **A methodology for the development of distributed deadlock detection algorithms**

  Deadlock detection requires knowledge of a system's global state. Deadlock detection in a distributed environment (the global system states are composed of locally recorded system states at different sites across the network) is far more complicated than that in a centralized system (where a global system state can be kept in a centralized place) due to the lack of a physical common global clock[2] and unpredictable message delays. A good distributed deadlock detection algorithm should confine its knowledge about the global system state to local views that any individual site can manage to acquire and assumes no physical common global clock. Failure to recognize these limitations may account for the fact that many algorithms proposed in the literature are prone to error.

  The problems stated above can be divided into two parts, i.e., (1) how to correctly recognize a deadlock structure and (2) how to synchronize a deadlock detection computation with the underlying dynamically changing system. First, to deal with the problem (1), the underlying system is assumed to be static when a deadlock detection computation is performed. In such a assumed static environment, we can then focus on the principles of the deadlock detection based on the graph theory. Since it is difficult to have a global view in a distributed system, a "local view of a deadlock" is recommended for the definition of deadlocks and is suggested to be used in the development of "static" algorithms.

  To deal with problem (2), a systematic method is developed to guide the design of a "synchronization mechanism" which can be coupled with a known

[2]In distributed real-time systems, there might be well synchronized system clocks at different sites; however, as long as there is no physical common global clock, it is difficult to capture a global system state.

"static" deadlock detection algorithm to produce a "dynamic" algorithm. Two consistency criteria must be fulfilled when developing a synchronization mechanism for a dynamic algorithm. These criteria require that the synchronization mechanism of a derived dynamic algorithm should keep a deadlock detection computation in a dynamic system "meaningful" such that the state of the computation can correctly reflect the underlying system state. One interesting property of this meaningfulness requirement is that the meaningful part of a computation can be "projected" to a static system state. Such a projected computation should be equivalent to a computation which is directly generated in that system state by the dynamic algorithm. Also, in the development of a synchronization mechanism we need to ensure a certain level of "equivalence" between a static algorithm and its derived dynamic algorithm such that both algorithms utilize the same principles in deadlock detection and they should function the same when they are applied to a static system state. Consequently, if a dynamic algorithm is derived from a correct static algorithm following the guidelines described in the methodology, its correctness is guaranteed. The methodology can also be used "reversely" in the verification of a dynamic algorithm.

- **Algorithms based on cycle detection technique**

  A cycle in a wait-for graph is a necessary and sufficient condition for deadlocks in the Single-Resource and the AND deadlock models. Based on the definition of a local view of a cycle, principles of cycle detection are discussed. Static algorithms are developed for the Single-Resource and the AND deadlock models. Dynamic cycle detection algorithms are then derived from the static algorithms step by step following our methodology. The final algorithms are extended for real-time applications where timing constraints are considered.

- **Algorithms based on knot detection technique**

  A knot in a wait-for graph is a necessary and sufficient condition for deadlocks in the OR deadlock model. Based on a local definition of knots, simple static knot detection algorithms are developed. Dynamic knot detection algorithms

are then derived following our methodology. A final algorithm is developed for the detection of both the Ada rendezvous deadlocks and the task termination conditions in distributed environments.

- **Testbed implementation and performance evaluation**

  We have implemented three proposed algorithms for the Single-Resource, the AND, and the OR deadlock models in the RT-CARAT system where transaction timing constraints and soft real-time scheduling protocols are supported [67]. Together with a previously implemented algorithm (proposed by Chandy, Misra, and Haas [27]) for the AND deadlocks, a real-time scheduling protocol, and several timeout schemes, we conduct various experiments to study and evaluate the algorithms developed in this dissertation. Since these are real implementations, the actual protocol overheads can be measured. Our results show that distributed deadlock detection can be very efficient. From the results, we can also identify situations where simple timeouts are better than performing deadlock detections/resolutions while in other situations it is the other way around.

1.3 Plan of the Dissertation

The dissertation is organized as follows. Chapter 2 formally describes the deadlock problem in terms of graph theory. In this chapter, we also summarize the underlying system models and deadlock models. These deadlock models combined with the system model are used to serve both as a framework in the survey and as a basis for the analysis of the distributed deadlock detection and resolution algorithms given in the following chapters. Finally, this chapter ends with a discussion of the issues of deadlock detection and resolution in distributed real-time systems. In Chapter 3, we survey various approaches for the distributed deadlock detection and resolution algorithms proposed in the literature. Following the survey of the related work, we present our solutions in the next three chapters. In Chapter 4, a methodology is developed. Based on this methodology, we are able to systematically

design deadlock detection algorithms for the distributed environments as well as to formally prove their correctness. The algorithms based on cycle and knot detection are then presented in Chapters 5 and 6, respectively. One unique feature of our proposed algorithms is that they are able to address timing constraints in real-time applications. In Chapter 7, we report on a performance study of our proposed algorithms. Finally, a summary of this dissertation and directions for future research are found in Chapter 8. Throughout the dissertation, examples and algorithms are given in Ada-like syntax.

Chapter 2

THE DEADLOCK PROBLEM

In this chapter, we formally describe the deadlock problem in terms of wait-for graphs and concepts from graph theory. Also, we attempt to relate the formal properties of deadlock algorithms directly to the actual languages and systems such as Ada environments. Section 2.1 introduces the concept of wait-for graphs which have been used to model system states in describing deadlock related problems. Section 2.2 briefly summarizes a few terms and results from graph theory. Section 2.3 presents the underlying system model. In Section 2.4, we discuss several deadlock models. General strategies of finding solutions for different deadlock models are then discussed in Section 2.5. These deadlock models combine with the system model to serve both as a framework in the survey (Chapter 3) and as a basis for the analysis and development of the distributed deadlock detection and resolution algorithms in this dissertation. Section 2.6 describes some new concerns regarding deadlock in real-time systems, and shows how the deadlock problem can be divided into four levels of complexity. We also indicate that to fully support Ada semantics it is necessary to develop solutions for the most complex level.

2.1 Wait-For Graphs

A wait-for graph is a mathematical tool which has been used to model the system state in describing deadlock related problems. A wait-for graph is a *digraph* (directed graph). A digraph is a pair (V, E), where V is a nonempty set of *vertices* (which represent tasks or resources) and E is a set of *directed edges* (which represent "wait-for" dependencies). Each directed edge in E is an ordered pair (a, b) which means that a is waiting for b, where a and b are vertices in V. Also, the notation $\langle a \rightarrow b \rangle$ will be used to visualize a directed edge. If both task vertices and resource vertices coexist in a graph, a wait-for graph becomes a *bipartite* digraph. A bipartite

graph is one in which all the vertices in V are partitioned into two disjoint subsets (a subset of tasks T and a subset of resources R in the case of wait-for graphs) such that there are no edges connecting vertices from the same subset.

Figure 2.1-(a) illustrates a bipartite digraph. The set of vertices $V = \{a, b, c, d\}$ is partitioned into two subsets $\{a, d\}$ and $\{b, c\}$. The set of edges $E = \{(a, b), (a, b), (a, c), (c, d), (d, c)\}$ connects the two subsets of vertices. Vertex b is a sink, path "$c \rightarrow d \rightarrow c$" is a cycle, and set $\{c, d\}$ is a knot. Sink, path, cycle, knot and other concepts from graph theory will be formally defined in Section 2.2.

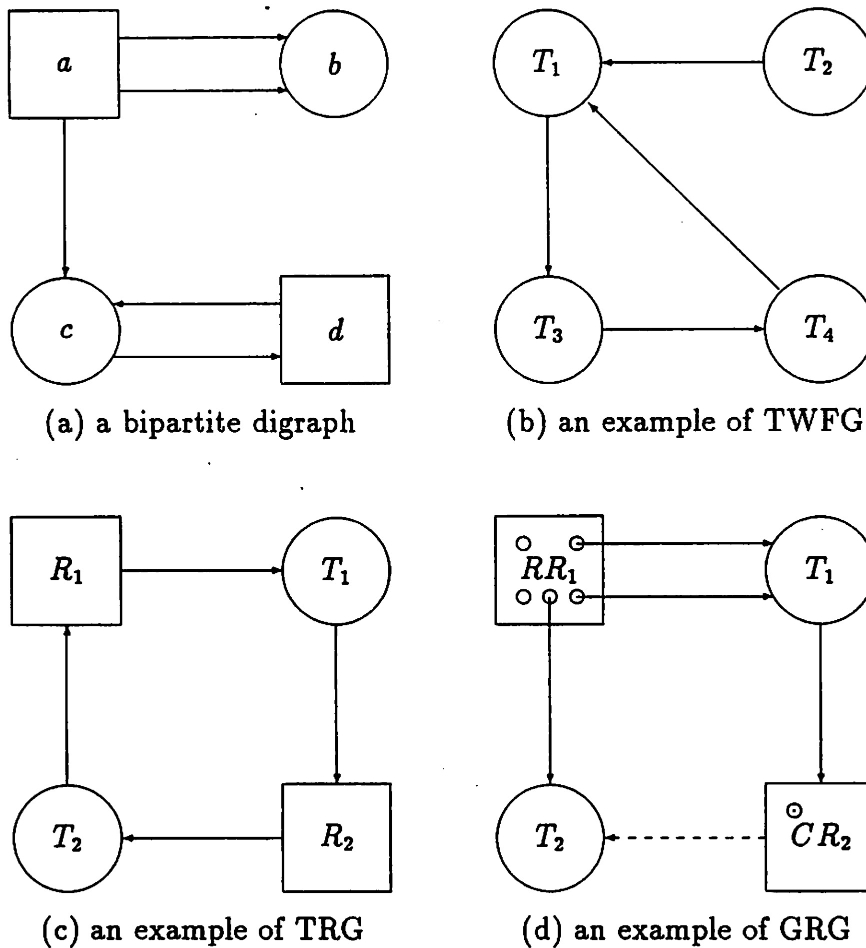


Figure 2.1: Examples of the system state graph

The state of a system is in general dynamic; that is, tasks continuously acquire and release resources and communicate with each other. Characterization of

deadlocks requires a representation of the system state in terms of task-task and/or task-resource interactions. Depending upon the complexity of the model, a system state can be depicted by one of the three types of wait-for graph:

TWFG: The TWFG (Task Wait-For Graph) is the simplest graph among the three types of graph. Only the task-task wait-for relations are depicted in the TWFG. A TWFG is a digraph in which vertices represent tasks. A directed edge (T_1, T_2) represents that a task T_1 is waiting for another task T_2 either due to a resource conflict or due to a synchronized communication attempt between them. In the database literature, a TWFG is referred to as a Transaction-Wait-For Graph in which vertices are transactions. A transaction can be viewed as a task that performs a sequence of database operations. In general, depending on the type of the underlying system, either tasks or transactions can represent the unit of computation in deadlock problems. The terms task and transaction are used interchangeably hereafter.

Figure 2.1-(b) is an example of TWFG with four tasks T_1, T_2, T_3, and T_4. Path $T_1 \to T_3 \to T_4 \to T_1$ is a cycle, and tasks T_1, T_2, T_3, and T_4 are deadlocked.

TRG: A TRG (Task-Resource Graph or, as referred to in the database literature, Transaction-Resource Graph) is a bipartite digraph in which vertices are partitioned into two subsets: a subset of all the task vertices in the system $T = \{T_1, T_2, ..., T_m\}$ and a subset of all the resource vertices in the system $R = \{R_1, R_2, ..., R_n\}$. Edges of a TRG depict assignments or pending resource requests. A pending request is represented by a *request edge* (T_i, R_j) directed from a requesting task T_i to the requested resource R_j. A resource assignment is represented by an *assignment edge* (R_k, T_l) directed from an assigned resource R_k to its holder task T_l.

Figure 2.1-(c) is an example of TRG with two tasks T_1 and T_2, and two resources R_1 and R_2. Notice that resources are depicted by squares "□" to distinguish resources from tasks which appear as circles "○". The edges $R_1 \to T_1$ and $R_2 \to T_2$

are assignment edges and edges $T_2 \rightarrow R_1$ and $T_1 \rightarrow R_2$ are request edges. Path $T_1 \rightarrow R_2 \rightarrow T_2 \rightarrow R_1 \rightarrow T_1$ is a cycle; T_1 and T_2 are deadlocked.

GRG: A GRG (General Resource Graph [66]) is a generalized TRG which is used to describe Holt's *General Resource System* [66]. The resource vertices in a GRG are partitioned into two disjoint subsets: a *reusable* resource subset $RR = \{RR_1, RR_2, ..., RR_x\}$ and a *consumable* resource subset $CR = \{CR_1, CR_2, ..., CR_y\}$. For each reusable resource RR_i, its *total units* is a strictly positive integer rt_i. For each consumable resource, its *producers* are a nonempty subset of the task set T. Edges are of three types:

Request edge: A request edge (T_i, RR_j) or (T_i, CR_k) is directed from a requesting task T_i to the requested resource RR_j or CR_k. It represents a pending request.

Assignment edge: An assignment edge (RR_i, T_j) is directed from a reusable resource RR_i to its assigned holder T_j. The total number of assignment edges directed from RR_i can not exceed rt_i.

Producer edge: A producer edge (CR_i, T_j) is directed from a consumable resource CR_i to one of its producers T_j. Edge (CR_i, T_j) exists if and only if T_j produces CR_i.

Figure 2.1-(d) is an example of GRG with two tasks T_1 and T_2, and two resources RR_1 and CR_2. Task T_1 holds two units of reusable resource RR_1 and requests one unit of consumable resource CR_2. Task T_2 also holds one unit of RR_1, and is the only producer of CR_2. The total inventory of RR_1 is $rt_1 = 5$. Notice that the graph is bipartite. Reusable resources are permanent and are depicted by small circles "$\circ$". Consumable resources are temporary and are depicted by dotted circles "$\odot$". Request and assignment edges may appear and disappear with state changes; they are depicted by solid arrows. Producer edges are permanent and are depicted by dashed arrows.

In the following discussions we use the concept of a GRG to merge the problems of the deadlocks due to both inter-task communication and resource competition.

The GRG was proposed by Holt to describe his *General Resource System* (GRS) [66]. More detailed discussions of the GRG and the GRS are given in Section 2.4.2.

2.2 Concepts from Graph Theory

In this section, some of the graph theory concepts are summarized and related to real systems. We then formally define the deadlock problem based on these graph theory concepts

In a GRG (V, E), a vertex v has a set of reachable neighbors $RN(v) = \{w \mid (v, w) \in E\}$. Also, v has a set of neighbors which can reach v, that is, $TN(v) = \{u \mid (u, v) \in E\}$. The state of a task in a system can be either "active" (i.e., executing) or "blocked" (i.e., idle and waiting for something). If a task vertex v in a GRG has outgoing edges, i.e., $|RN(v)| > 0$, its corresponding task is blocked; otherwise, it is active. The state of a resource in a system may be "available" (i.e., at least one unit is available if it is a reusable resource, or at least one unit has been produced if it is a consumable resource) and/or "held" (i.e., there are units of a reusable resource assigned to tasks, or a consumable resource to be produced is "held" by the producer). Similarly, if a resource vertex v has outgoing edges, i.e., $|RN(v)| > 0$, its corresponding resource is held. A held reusable resource might still be available if there are units not assigned.

A *directed path* is a sequence $(a, b, c, \ldots, y, z)$ of at least two vertices, where $(a, b), (b, c), \ldots, (y, z)$ are directed edges. Also, the notation $\langle a \rightarrow b \rightarrow \cdots \rightarrow y \rightarrow z \rangle$ will be used to visualize a directed path. A directed path represents the "wait-for" dependency relations among the vertices in the path. Each blocked task or held resource has a set of tasks and resources, for which it is "waiting," called its *dependent set* or termed as *reachable set* in graph theory. The reachability relations R defined on a GRG is a set of ordered vertex pairs (v, w), such that, $(v, w) \in R$ (i.e., v can reach w) if and only if there is a directed path from v to w. In other words, R is the transitive closure of E. For a vertex $v \in V$ its *reachable set* $RS(v)$ is the set of all vertices $w \in V$ such that there is a path directed from v to w.

Therefore, $RS(v)$ can be defined as the set $\{w \mid (v,w) \in R\}$. On the other hand, $TS(v)$ is the set of all vertices $u \in V$ such that there is a path directed from u to v. Likewise, $TS(v)$ can be defined as the set $\{u \mid (u,v) \in R\}$.

A *cycle* is a path with the same *start* and *end* vertex. A vertex v is in a cycle if and only if $v \in RS(v)$. Two cycles C_1 and C_2 are *nested* if $C_1 \cap C_2 \neq \emptyset$ and $C_1 \neq C_2$.

A *knot* K is a nonempty set of vertices such that the reachable set of each vertex v in K is exactly the knot K. That is, K is a knot if and only if $\forall v \in K$, the reachable set $RS(v) = K$.

The concept of a *strongly connected digraph*, which is similar to a knot, is also widely used in the deadlock detection literature. A digraph is said to be strongly connected if for every two vertices u and v in the graph there exists a directed path from u to v. A knot is a strongly connected digraph with no path which is directed from a vertex in the knot to a vertex outside of that knot.

A vertex i is an *isolated vertex* if it has no incoming edges (i.e., $|TN(i)| = 0$) and has no outgoing edges (i.e., $|RN(i)| = 0$). A non-isolated vertex s is a *sink* if it has no outgoing edges (i.e., $|RN(s)| = 0$ and $|TN(s)| > 0$) . If the end vertex of a directed path has no outgoing edges, it is a sink and represents an active task. Many systems use an "expedient" resource allocation strategy in which all requests for available units are granted immediately. In such systems, a resource vertex may have incoming edges (i.e., unavailable) only if it has outgoing edges (i.e., it has been held by some tasks). Therefore, we do not need to consider that a resource vertex may become a sink.

In Section 2.4 we will see that in the simple deadlock models, such as the Single-Resource model and the AND model, a system is in deadlock state if and only if there are cycles in the system state digraph. In the OR model, the existence of a knot in a system state graph is a necessary and sufficient condition for deadlock. However, in more complex models, such as the AND-OR model and the C(n,k) model, there is no simple construct of graph theory to describe the condition of

deadlock. Many deadlock detection algorithms are designed for simple models, therefore, their major functions are detecting cycles or knots.

2.3 Models of Distributed Concurrent Programming Systems

Deadlock can be formally studied in isolation by using graph theory. However, in this work we are interested in explicitly tying the formal properties of deadlock algorithms directly to the actual languages and systems that need to use the theory. In particular, we are interested in Ada and its run time system. We believe that there is an important gap that exists between the theory and its application that has not been addressed very well to date. To bridge this gap we must understand Ada's concurrency, synchronization, and resource allocation models and show how they relate to the theory. Here we identify Ada's concurrency model, but since it is based on notions of Hoare's "Communicating Sequential Processes" (CSP) [65], and Brinch Hansen's "Distributed Processes" (DP) [15], we first say a few words about CSP and DP.

2.3.1 CSP – Communicating Sequential Processes

Hoare's CSP recognized that synchronous input and output can be the basic primitives of concurrent programming. In CSP, tasks synchronize and communicate by means of *input* and *output* commands. An input/output command is delayed in one task until a matching output/input command is executed by another task. Messages are transferred using input and output commands, therefore, CSP is based on message-passing synchronization requiring no shared memory (variables). Messages are not buffered and their one-to-one transfers are synchronously performed.

The notation

$$[G_1 \rightarrow CL_1 \;\square\; G_2 \rightarrow CL_2 \;\square\; \cdots \;\square\; G_n \rightarrow CL_n]$$

defines an *alternative* command, where $G_i \rightarrow CL_i$ is a *guarded* command with the guard G_i and its corresponding command list CL_i. The alternative command is

executed by evaluating each guard G_i for *success* or *failure*. One of the command lists associated with a successful guard is chosen nondeterministically if more than one guard succeeds. If a guard contains an input/output command, it succeeds only when the preceding elements of the guard indicate success and the corresponding output/input command is executed.

The alternative command given above specifies the execution of exactly one of its constituent guarded command. An error results if no guard succeeds. On the other hand, a *repetitive* command

$$*[G_1 \rightarrow CL_1 \;\square\; G_2 \rightarrow CL_2 \;\square\; \cdots \;\square\; G_n \rightarrow CL_n]$$

specifies as many iterations as possible of its constituent alternative command. In a repetitive command the failure of all guards terminates the repetition instead of resulting in an error.

2.3.2 DP – Distributed Processes

Another mechanism based on synchronous message passing is Brinch Hansen's DP. In DP, a task performs two kind of *operations*: the *initial statement* defined within itself and the *external requests* made by other tasks. A task starts by executing its initial statement until the statement either terminates or waits for a condition to become true. Meanwhile, another pending operation (external request), if any, is started and continues until it either terminates or waits for a condition to become true. Therefore, this *interleaving* of the initial statement and the external requests is controlled by the program instead of the system clock interruption at the machine level.

The *common procedure* is the only form of task communication in DP. A task may call common procedures defined within itself or within other tasks. These procedures are the external requests to the called task. A task T_1 calls a procedure OP defined within another task T_2 by:

call $T_2.OP($*expressions, variables*$)$

Before the procedure OP is executed, the *expressions* are evaluated and their values

are assigned to the input parameters of OP. When OP completes, the values of its output parameters are assigned to the *variables* of the call. Parameter passing between tasks may be implemented either by shared memory or by one-to-one message passing without shared memory. In addition to CSP's guarded command, a *guarded region* in DP enables a task to wait until a choice among several guarded statements can be made.

2.3.3 Ada – A Distributed Concurrent Programming Environment

Ada's concurrent programming mechanisms are generalized from many aspects of CSP and DP. A *task* is the unit of computation in Ada environments. A task is a program module that is executed asynchronously. Tasks may communicate and synchronize their actions through:

- the *entry calls* and *accept statements*, which are a combination of procedure calls and message transfers, and
- the *select statement*, which is a non-deterministic control structure similar to the alternative guarded command in CSP and DP.

Entry declarations and calls are syntactically similar to procedure declarations and calls. Entry declarations can occur only in the specification of a task. The corresponding **accept** statements are given in the body of the task, which have the following form[1]:

accept *<entry−name>* [*parameters*]
    [**do** {*statement*} **end**];

The {*statement*} part of an **accept** statement can be executed only if another task invokes the *<entry−name>*. Invoking an *<entry−name>* (an entry call) is syntactically the same as a procedure call in DP. First, parameters are passed before the execution of the {*statement*}. After the execution reaches the **end** statement, parameters may be passed back. Both tasks are free to continue from this point.

[1]Square brackets [ ] denote an optional part, while braces { } denote a repetition of zero or more times.

The **accept** statement and the corresponding entry call are executed synchronously similar to the input and output commands in CSP. This synchronization performed between two tasks is the Ada's *rendezvous* concept. Thus the entry call and accept statement serve both as a communication mechanism and a synchronization tool.

Choices among several entry calls is accomplished by the **select** statement, which is similar to the guarded region in DP. There are three kinds of select statements: the *selective wait*, the *conditional entry call*, and the *timed entry call*:

The *selective wait* statement allows a task to **accept** an entry call from more than one task non-deterministically. A *selective wait* statement has the form

```
select
    select-alternative
{or
    select-alternative}
[else
    {statement}]
end select;
```

in which *select–alternative* is of the form

```
[when <boolean-expression> ⇒] selective-wait-alternative
```

and a *selective–wait–alternative* can be one of

```
  accept-statement [{statement}]
| delay-statement  [{statement}]
| terminate;
```

A *select–alternative* statement is said to be *open* if there is no prefixed guard (**when** clause) before it or if the <*boolean–expressions*> in the prefixed guard is true; otherwise, it is said to be *closed.* A *selective wait* statement can have at most one **terminate** alternative. The *delay–statement* and **terminate** alternatives cannot coexist in a *selective wait* statement. The **else** part is not allowed when either a *delay–statement* or a **terminate** alternative is present.

According to the *Ada Reference Manual* [86] the following rules define the execution of a *select–alternative* statement:

1. Determine all the open alternatives and start counting time for the open *delay−statements* (if any).

2. If there are open alternatives that can be selected, the execution follows the steps:

   (a) An open *accept−statement* alternative may be selected for execution only if a corresponding rendezvous is possible. The subsequent statements, if any, are then executed.

   (b) The subsequent statements following an open *delay−statement* will be selected for execution if no other alternative is selected before the specified delay duration has elapsed.

   (c) A **terminate** alternative may be selected if all the sibling tasks and their dependent tasks which belong to the same root creator have terminated or are waiting at a **terminate** alternative. A task terminates if it reaches the end of its code sequence or if a **terminate** alternative is selected. The termination of a task is subject to the condition that there are no calls pending to any entry of the task.

3. If the **else** part is present, it is executed under the condition that no open alternative can be selected immediately or all alternatives are closed. If no **else** part is present and open alternatives exist, the execution is delayed until an open alternative can be selected. If no **else** part is present and all alternatives are closed, an error exception is raised.

When attempting to perform an immediate rendezvous, a *conditional entry call* is used. A *conditional entry call* has the form

> **select**
>   *entry–call* [{*statement*}]
> **else**
>   [{*statement*}]
> **end select**;

If an immediate rendezvous is possible, then rendezvous takes place and the subsequent statements following the *entry–call* are executed; otherwise, the alternative sequence of statements specified in the **else** alternative is executed.

When attempting to establish a rendezvous within some specified time period, a *timed entry call* is used. A *timed entry call* has the form

> **select**
>   *entry–call* [{*statement*}]
> **or**
>   *delay–statement* [{*statement*}]
> **end select**;

If a rendezvous can be established within the specified period then rendezvous takes place and the statements following the *entry–call* are executed; otherwise, the statements following the specified *delay–statement* are executed.

As in most of the concurrent programming environments, deadlocks may occur due to tasks that are competing for shared resources in Ada's environment both at the underlying system level, and at the language level. For example, consider a program with two tasks T_1 and T_2 executing on a system with two disk drives. Each of T_1 and T_2 needs both disk drives together, say for copying a file from one disk to the other. Deadlock will occur if each task is holding the permission to use one disk drive and is waiting for the permission to use the other drive. Deadlocks may also occur due to tasks that are waiting for each other in Ada's rendezvous. For example, two tasks T_1 and T_2 each want to call the other task before accepting

an entry call from the other. Portions of the code of tasks T_1 and T_2 are illustrated as follows:

```
task body T1 is
⋮
begin
    ⋮
    T2.READ(···);
    accept READ ··· end READ;
    ⋮
end T1;
```

and

```
task body T2 is
⋮
begin
    ⋮
    T1.READ(···);
    accept READ ··· end READ;
    ⋮
end T2;
```

Tasks T_1 and T_2 are deadlocked at the entry calls $T_2.READ$ and $T_1.READ$, respectively. A more detailed discussion of deadlocks will be given in the Section 2.4.

Some aspects of real-time processing is supported by the **select** statement. The **else** alternative and the **delay** statement in the *selective wait* both provide ready escapes in the event that no open alternatives exist or that open alternatives are unduly delayed in their selection. Using the *conditional* or *timed* entry calls, the calling task can ensure that it will not be blocked due to the inability of the called task to complete a rendezvous. However, as discussed in Section 2.6.1, *temporal* deadlock is still a problem in such a real-time system.

Livelock occurs when interacting tasks cannot finish their work in a limited period of time. For example, if a task T_1 (T_2) is unable to rendezvous immediately with another task T_2 (T_1), it performs some secondary activity, so that it does not

waste time being blocked for a rendezvous, and then tries to rendezvous again. Livelock may occur if every time T_1 (T_2) attempts to rendezvous with T_2 (T_1), T_2 (T_1) is performing some secondary activity and is not ready for a rendezvous. Eventually T_1 and T_2 will miss their deadlines.

Also, task termination and orphan task problems may arise when using Ada's concurrent programming facilities in distributed environments:

- *Task termination* – In Ada, the termination of tasks, whether it is normal termination or abnormal termination, is well defined not to affect other executing tasks. It is not difficult to realize a task termination mechanism correctly in a single site system. However, task termination becomes complex and difficult when implemented in distributed environments due to intersite task interaction.
- *Orphan task* – In a distributed system, a task may create several subtasks at different sites. Due to site failure or network partitioning, a subtask might become an orphan which has lost connection to its parent task. Similarly, if site failure or network partitioning takes place when tasks from different sites are in a rendezvous, the tasks waiting for the rendezvous might become orphans.

The *stable* property of task termination and orphan tasks are similar to that of deadlock problem. These problems are all caused by task interaction and once they occur will remain until they are detected and resolved. Therefore, techniques such as *diffusing computations* and *global state detection* discussed in Chapter 3, can be extended and applied to solve these problems.

2.4 Deadlock Models

The sufficient conditions for detecting deadlock vary depending on the particular deadlock situation. Modelling deadlocks is a way of understanding the properties of deadlocks as well as a way of finding methods to detect and resolve the

deadlocks. In this section we discuss three deadlock model approaches: (1) the resources/communication deadlock model [27] used in most algorithms, (2) the general resource system model presented by R. C. Holt [66], and (3) a hierarchy of deadlock models presented by E. Knapp [77]. They are discussed in the following subsections.

2.4.1 Resource and Communication Deadlock Models

Many deadlock detection and resolution algorithms are used either in solving "resource" deadlocks [8, 23, 27, 35, 37, 44, 46, 54, 56, 57, 64, 76, 90, 93, 95, 100, 106, 114, 118, 126, 133], or used in communication based systems [27, 58, 97, 105], or both [14, 63]. However, the distinction between these two models is not always clear since the messages could be treated as implied resources and may not be distinguishable from physical or data resources.

In resource deadlocks, tasks access resources (for example, data items in database systems or memory buffers in operating systems). A task acquires a resource before accessing it and relinquishes it after using it. The accessed resource is held by the accessing tasks and edges are, therefore, formed in the TRG from the resource toward the accessing tasks. A task that requires resources which are not available (held by other tasks or not yet created) can not proceed until the requests are satisfied. The requesting task is waiting for the availability of the requested resources and, therefore, edges are formed in the TRG from the task toward the requested resources. A set of tasks is resource-deadlocked if and only if each task in the set requests at least one resource held by another task in the set.

In communication deadlocks, messages, tokens, or "synchronization between tasks" are the implied resources for which tasks wait. In Ada environments, tasks communicate via synchronization and communication points called *entries*. A synchronization or communication may involve two or more tasks. The waiting tasks can proceed only after all the tasks involved in the synchronization or communication are ready to synchronize or exchange information (i.e., the *rendezvous* concept).

An edge is formed in the TWFG from a waiting task toward its waited task. Each idle task has a set of tasks for which it is waiting, called its *dependent set.* A nonempty set of tasks S is communication-deadlocked [27] if and only if (1) all tasks in S are idle, (2) the dependent set of every task in S is a nonempty subset of S, and (3) there are no messages, tokens, or synchronization in transit between tasks in S.

There are several differences between the resource model and the communication model. One major difference is that different request patterns are assumed in the two models. In the resource model, a blocked task cannot proceed until it is granted *all* the requested resources. A cycle detection algorithm is usually used in the resource deadlock models. On the other hand, in a communication environment, such as CSP, DP, or Ada, a task's communication request may be satisfied by *at least one* of its corresponding tasks. For instance, in the Ada environment, the **accept** statement requires one of many potential calling tasks to be in rendezvous with the accepting task. A cycle is no longer a sufficient condition for communication deadlocks. Knot detection algorithms are usually used for the communication deadlock detection. A knot detection algorithm is more complicated than one which detects cycles. More detailed discussions are given in Section 2.4.3.

A second difference is that the agent of deadlock detection in the two environments is usually not the same. In the resource model, the wait-for dependency among tasks may not be known by the tasks directly. A "controller" (a "controller" could be a separate task or a shared data structure plus program code) at each site keeps track of its resources, and only the controllers can deduce that one task is waiting for another. On the other hand, in the early communication models, it is assumed that a task always knows the identities of the tasks it is waiting for. Thus, in the communication model the tasks have the necessary information to perform deadlock detection if they act collectively.

2.4.2 General Resource System Model

A generalized resource system model was proposed by R. C. Holt [66] which unifies the resource and communication deadlock models (see Section 2.4.1) into a *general resource system* (GRS) model. In Holt's GRS model, the term "resource" is used in a special sense to mean any object which may cause a task to become blocked. A resource is either reusable or consumable. "Reusable resources" are used to model competition for objects such as shared data and memory buffers. "Consumable resources" are used to model explicit interactions among tasks such as synchronization or exchange of signals or messages among tasks.

As an example of consumable resources, in Ada, synchronization between two tasks occurs when the task issuing an entry call and the task accepting an entry call are ready to establish a rendezvous. A rendezvous is a consumable resource. Either one of the calling or called tasks arriving at the rendezvous first will wait, and, hence, becomes the "consumer" of the "rendezvous." The second task which establishes a rendezvous is always the "producer" of the "rendezvous." After a rendezvous is established, the calling task becomes blocked while the called task is executing corresponding statements following the **accept** statement. The called task, therefore, is active and acts as the producer during the rendezvous period.

Both types of resources consist of a number of identical units which can be *requested* by tasks. The total number of units of a reusable resource is fixed, but it is unlimited for any of the consumable resources. A task requesting units is blocked until enough units are available to satisfy its request; then the task can *acquire* the requested unit. A task can *release* units only when it is not idle (waiting). The fundamental difference between reusable and consumable resources is that the units of a reusable resource are never created or destroyed, but only transferred (requested and acquired) from a pool of available units to a task and then transferred back (released) to the pool. On the contrary, units of a consumable resource are created ("produced," or released) and destroyed ("consumed," or requested and acquired).

The GRS model merges the communication deadlock model and the resource deadlock model by distinguishing the set of resources into two disjoint subsets, a set of reusable resources (resource deadlock model) and a set of consumable resources (communication deadlock model). A generalized TRG called the *general resource graph* (see GRG in Section 2.1) is used to depict the system state. A GRG is a TRG with each resource vertex explicitly labeled with a number of reusable/consumable resource units.

Holt suggested a *graph reduction* technique [66] to test if tasks are deadlocked. A GRG can be *reduced* by any task vertex T_i which is neither isolated nor blocked in the following manner:

- For each reusable resource vertex RR_j, delete all request edges (T_i, RR_j) and assignment edges (RR_j, T_i). Increment its total inventory units rt_j by the number of deleted assignment edges.
- For each consumable resource vertex CR_j, delete all request edges (T_i, CR_j) and producer edges (CR_j, T_i). Decrement its total inventory units ct_j by the number of deleted request edges (T_i, CR_j) . Set $ct_j = \infty$, if a producer edge (CR_j, T_i) was deleted.

A reduction of a GRG by a task vertex T_i may lead to the unblocking of another task T_j, therefore, the task T_j may be chosen as a candidate for the next reduction. A GRG is said to be *completely reducible* if there exists a sequence of graph reductions that deletes all edges in the GRG. A task T_i is not deadlocked in state S if and only if there exists a sequence of reductions in the corresponding GRG that leaves T_i unblocked. If a GRG is completely reducible, then the state it represents is not deadlocked.

Many systems use an "expedient" resource allocation strategy in which all requests for available units are granted immediately. In such systems, a new allocation of resources can take place only immediately following a resource request or a release. Holt [66] has proven that in a GRG, (1) a *cycle* is a necessary condition

for deadlock, and (2) if the graph is expedient, then a *knot* is a sufficient condition for deadlock.

2.4.3 A Hierarchy of Deadlock Models

Edgar Knapp [77] proposed a hierarchical set of deadlock models to describe the characteristics of deadlocks. Each model is characterized by the restrictions that are imposed upon the form resource requests can assume. For example, a task might need to acquire a combination of resources like (R_1 *and* R_2) *or* R_3. The hierarchical set of deadlock models ranges from very restricted request forms to models with no restrictions whatsoever. The hierarchy can expand the unified GRS model (see Section 2.4.2) to further explore the conditions for deadlock. This hierarchy can also be used to classify deadlock detection algorithms according to the complexity of the resource requests they permit.

2.4.3.1 Single-Resource Model

The simplest possible model is one in which a task can have at most one outstanding resource request at a time and none of the resources are sharable. Hence the maximum number of edges from a task or a resource in a GRG is 1. A *cycle* in a GRG is a necessary and sufficient condition for deadlock. A blocked task T is said to be deadlocked if T is in a cycle or T can only reach deadlocked tasks. Examples of this model can be found in database systems [12, 108]. The system can be formalized as a Single-Resource model in the Ada environment, if the resources are non-shareable and only one outstanding request is allowed, and no more than two tasks may be involved in either a rendezvous or task termination. The Mitchell-Merritt algorithm [100] is a very simple and elegant algorithm based on this Single-Resource model.

2.4.3.2 AND Model

In the AND model, tasks are permitted to request a set of resources and/or resources are sharable. A task is blocked until it is granted *all* the resources it has requested. A shared resource cannot be exclusively held until all of its shared lock holders have released the lock. The requests of this type, therefore, are called AND requests. This model was referred to as the resource deadlock model in the literature (see Section 2.4.1). Examples of the AND model system can be found in some distributed DBMS where subtransactions can be executed concurrently on different sites. In the Ada environment, the shared resources and the task termination mechanism can be formalized as the AND model. Again, a *cycle* in a GRG is the necessary and sufficient condition for the existence of deadlocks in the AND model. And a blocked task T is said to be deadlocked if T is either in a cycle or can only reach deadlocked tasks. The AND model is, therefore, strictly more general than the Single-Resource model.

A number of algorithms have been proposed based on this AND model: Chandy-Misra algorithm [23], Chandy-Misra-Haas algorithm [27], Gligor-Shattuck algorithm [54], Haas-Mohan algorithm [57], Menasce-Muntz algorithm [95], and the Obermarck algorithm [106].

2.4.3.3 OR Model

In contrast to the AND model, an alternative way for making resource requests is the OR model. In this model, a task is blocked until it is granted *any* of the resources it has requested. For example, in replicated distributed database systems, a read request for a replicated data item is satisfied by reading any copy of it. This model was first referred to as communication deadlock model (see Section 2.4.1) by Chandy et al. in [27]. In OR model, detecting a *cycle* in GRG is not a sufficient condition for deadlock. As pointed out by Holt [66], a *knot* is a sufficient condition for deadlock while a *cycle* is only a necessary condition. Hence, deadlock detection in the OR model can be reduced to finding knots in the GRG. However, a task

can be deadlocked without being in a knot. Therefore, a necessary and sufficient condition for a deadlocked task is: a blocked task T is deadlocked if T is in a knot or T can only reach deadlocked tasks.

There are similarities between the detection of an OR model deadlock and the detection of the termination of a group of cooperating tasks in a distributed computation [5, 6, 39, 41, 42, 48, 50, 98]. In a distributed system, tasks cooperate with each other in a computation by means of message exchange. A distributed computation is said to be globally terminated if it reaches a *final* state which, in turn, relies on its member tasks reaching their final states and being ready to terminate. The termination problem arises when tasks are ready to terminate locally, but they still agree to communicate with other cooperating tasks. The *global termination condition* is defined as the condition that each of the cooperating tasks in a distributed computation is either terminated or ready to terminate. A global termination condition is not satisfied if *any* of the cooperating tasks in a distributed computation is not waiting for termination. For example, in Ada environments a *selective wait* statement allows a task to **terminate** if all its sibling tasks and their dependent tasks which belong to the same root creator have terminated or are waiting at a **terminate** alternative. This is a pessimistic model of the termination problem in that it assumes all the active sibling tasks may want to make an entry call to the ones which are ready to terminate in a selective wait statement. The distributed termination problem can be viewed as a special case of an OR deadlock where all the cooperating tasks in a distributed computation are involved in a deadlock (waiting for others to terminate). Therefore, any knot detection algorithm for the OR deadlocks can also be tailored for solving the distributed termination problem and vice versa.

Francez [48] first brought the distributed termination problem into prominence by defining global termination conditions in CSP environments, and proposing a detection algorithm for them. Dijkstra and Scholten [42] introduced the notion of a *diffusing computation* (See Section 3.2.3) and suggested an algorithm to detect the termination of an arbitrary diffusing computation in any network environment.

Cohen and Lehmann [39] extended Francez's CSP termination model and solution to distributed systems where new cooperating tasks are created and terminated dynamically. Misra and Chandy [98] discussed how a termination computation algorithm can detect deadlock for the OR model. And then in [97] Misra and Chandy presented an algorithm for distributed knot detection. Chandy, Misra and Haas [27] presented an algorithm for the communication deadlock model which is an OR model in GRG. Natarajan's algorithm [105] is also based on the OR model. Their work was followed and improved on by Huang [68].

2.4.3.4 AND-OR Model

The AND-OR model is a generalization of the two previous models. A task in AND-OR model may specify resources in any combination of AND and OR requests. For example, a task may request resources R_1 *or* (R_2 *and* (R_3 *or* R_4)) where R_1, R_2, R_3, and R_4 may exist at different sites.

There is no simple construct of graph theory in terms of GRG to describe the deadlock condition in the AND-OR model. In principle, deadlocks in the AND-OR model can be detected by applying the test for OR model deadlock repeatedly, where each invocation operates on a subgraph of the AND part of the model. However, this strategy is not very efficient. Hermann and Chandy [63] proposed a more efficient algorithm for AND-OR deadlock. Their algorithm is based on a hierarchy of diffusing computations which they called a *tree* computation. The central idea of their algorithm is that when a diffusing computation reaches a blocked task: (1) the diffusing computation is propagated to its dependent set if it is an OR task, or (2) it initiates a separate tree computation if it is an AND task.

2.4.3.5 C(n,k) Model

The C(n,k) model, which was first formulated by Bracha and Toueg [13] as a k-out-of-n request model, is a generalization of the AND-OR model. Any AND-OR request can be directly converted into a C(n,k) request. On the other hand, a C(n,k)

request may be transformed into a set of AND and OR requests in the AND-OR model. The length of the corresponding AND-OR formula for a C(n,k) request is $k \cdot C(n, k)$. Both models can be used interchangeably. However, in most cases, the C(n,k) model allows one to express requests in a simpler form than that in the AND-OR model. Again, the algorithm presented by Bracha and Toueg [13] suffers from the same deficiencies as that of the AND-OR model. Although the AND-OR model can describe the interaction mechanisms among tasks defined in Ada, in general, we would like to categorize Ada runtime environment as the C(n,k) model because it is easier to formalize specific situations such as a task requests k pages of memory from a total of n pages.

2.4.3.6 Unrestricted Model

In the most general model, there is no assumed underlying structure for resource requests. The only assumption made is the stable property of deadlocks. Since Ada real-time applications have timing constraints which, if violated, may actually break a deadlock thereby breaking the stable property, great care should be taken in applying the techniques developed for this unrestricted model to the Ada environment.

The motivation of using the Unrestricted model is to separate the problem property (the stability of deadlocks) from the abstracted underlying system structures (e.g., the abstracted data structure and synchronization mechanisms provided in many high level languages such as Ada). Hence, a generalized solution can be developed. For example, in a general purpose system, one of the user applications may be a database system written in Pascal in which the user provided lock manager is of the complexity of the AND deadlock model. Yet another user written application may be an Ada concurrent program in which the rendezvous is the OR deadlock model. Both applications are written in high level languages with abstract data structures. Therefore, in this example, a system provided deadlock detection mechanism should be unrestricted and general enough to solve deadlocks of at least

the AND-OR model, regardless of what programming languages are used. However, the deadlock detection algorithms developed for the unrestricted model carry additional overhead which can be avoided in the algorithms designed for previously stated simpler models. This is due to the very fact that the underlying system computation structure is not fully utilized in helping to factor out some unnecessary conditions for deadlock. For example, an algorithm for the Unrestricted model usually needs to search the whole system to construct a GRG for deadlock detection. In contrast, if the only application running in the system is a database system as described above, an algorithm which sends probes to search part of the GRG for cycles is enough for the deadlock detection, and it is even beneficial if most of the deadlocks are two cycles.

All the algorithms designed for this Unrestricted model can be used to detect other stable properties (a.k.a. *quiescence* problem [26], examples are livelock or starvation, task termination detection, and orphan detection problems) as well. Also, the algorithms which use a global state detection technique for the detection of deadlocks can also be applied to distributed system monitoring, distributed debugging, etc.

Many algorithms related to this model have been studied theoretically such as: (1) stable properties detection by Chandy and Misra [26] and Hélary et al. [59], (2) global state detection by Chandy and Lamport [21], Spezialetti and Kearns [128], and Li et al. [87], (3) termination detection by Mattern [94], and (4) orphan detection by Shrivastava [122].

2.5 Deadlock Detection in Different Deadlock Models

In Section 2.4.3, a hierarchical set of six deadlock models was used to describe the characteristics of deadlocks. Except for the Unrestricted model, the problem complexity of the other five models can be roughly divided into four levels based on sufficient conditions for detecting deadlocks:

1. the Single-Resource model (contains only simple cycles; simple cycle detection is sufficient),
2. the AND model (contains nested cycles; nested cycle detection is sufficient),
3. the OR (cycle detection is not sufficient; knot detection is sufficient), and
4. the AND-OR and the C(n,k) models (both cycle and knot detections are not sufficient; cycle detection may detect false deadlocks whereas knot detection is not sufficient to detect all deadlocks; a correct detection algorithm requires the recognition of AND, OR, AND-OR, and C(n,k) requests).

In the Single-Resource model no nested deadlock cycles can occur. This property gives rise to an interesting solution. If deadlock detection probes are propagated in the opposite direction along the edges of the GRG, only the *in-cycle probes* initiated by the tasks in a cycle will detect deadlock. It is possible that only one task in a cycle will detect deadlock if a probe propagation rule is enforced. For example, in the algorithm developed by Mitchell and Merritt [100], each of the blocked tasks is assigned an unique identifier and a probe is propagated in the reverse direction only when its initiator identifier is larger than that of the destination task. This algorithm guarantees that only the probe with the largest initiator identifier is able to travel through the whole cycle to detect the deadlock. Such an algorithm simplifies the problem of resolution as well as guarantees that only genuine deadlocks will be detected in the absence of spontaneous time-outs and aborts.

In the AND deadlock model, since multiple outgoing edges as well as multiple incoming edges in the GRG are possible, nested cycles are expected. Each cycle in a group of nested cycles is stable, but the whole group of nested cycles is not stable because new cycles may be forming and attaching to the existing nested cycles. Since there are joint parts between any two nested cycles, the resolution of a deadlock may actually break more than one cycle at their common part of the graph. Therefore, a detected cycle may not exist if it was nested with another cycle which was detected and resolved earlier at a common part of these two cycles. To avoid false detection of deadlocks, we need to detect the whole group of the nested

cycles as well as to prevent any new cycle from attaching to it before the current ones are resolved. This requires synchronization between deadlock detection and other system activities, for example, to "freeze" the system while a deadlock detection and resolution in ongoing. Unfortunately, it is too costly to freeze a distributed system and, therefore, false detection of deadlocks is inevitable. Also, due to the existence of nested cycles, simple cycle detection algorithms, such as the ones used in the Single-Resource model, can no longer guarantee that only genuine deadlocks will be detected even in the absence of spontaneous time-outs and aborts. Situations concerning nested cycles have to be taken into consideration in order to minimize the detection of false deadlocks. In the probe based algorithms, *foreign probes*, which are initiated by tasks outside a cycle, may enter the cycle. A foreign probe may travel in the cycle more than once without detecting any deadlock if there is no mechanism to stop it. A foreign probe may interfere with the in-cycle probes and, hence, may cause the algorithm to fail to detect the deadlock. For example, similar to the probe propagation rule introduced in the Mitchell-Merritt algorithm, if the in-cycle probe with the largest label is expected to travel through the cycle to detect the deadlock, a foreign probe with a even larger label may enter the cycle and compete with the in-cycle probes. The algorithm may fail if such situations are not carefully considered.

In the OR model, a knot is a sufficient condition for deadlock while a cycle is only a necessary condition. Hence, deadlock detection in the OR model can be reduced to finding knots in the GRG. A task T_i in a GRG is in a knot if, for every task T_j reachable from T_i, T_i is reachable from T_j. To detect knots, probes are propagated in both forward (to search tasks which are reachable from T_i) and backward (to search tasks which can reach T_i) directions along the edges in GRG. After the GRG has been fully searched, the algorithm can determine whether a knot exists. Whenever a sink in the GRG is reached, a non-deadlock condition is found. A knot detection algorithm should be able to terminate if it detects either a knot or a non-deadlock condition. As discussed in Section 2.4.3.3, knot detection algorithms for OR model deadlocks can be tailored to resolve the distributed termination problem,

and vice versa. Many algorithms proposed in the literature [27, 68, 97, 105] are actually based on the notion of Dijkstra and Scholten's *diffusing computation* which was originally used for distributed termination detection. Similar to the previous two models, certain techniques can be used to reduce the number of probes traveling in the GRG.

The problem complexity of the remaining models — the AND-OR model and the C(n,k) model — are roughly the same. Many systems are neither solely the AND-OR model nor solely the C(n,k) model but a mixture of two. Such a mixed model system, however, can be mapped either to the AND-OR model or to the C(n,k) model. The mapping, in general, is easier toward the C(n,k) model than toward the AND-OR model. Since the AND-OR model is equivalent to the C(n,k) model, we will focus on the latter. As suggested by Bracha and Toueg [13], deadlocks in the C(n,k) model can be found by processing a global snapshot of the system. Once again, since AND deadlocks are embedded in the AND-OR model and the C(n,k) model, nested deadlocks, similar to the nested cycles in the AND model, may occur. Therefore, we face a similar false deadlock detection problem as found in the AND model.

Different deadlock models are assumed in the four levels of problem complexity categorized above. The applications of the algorithms developed for each of these deadlock models, therefore, have different restraints. For the Single-Resource model, resources must be non-sharable and must be distinguishable. Low level system provided task synchronization mechanisms can be allowed if no multiple outgoing edges in the GRG may result. For example, a *semaphore* is a synchronization tool provided in many systems. A semaphore S is an integer variable that can be accessed only through two *atomic* operations P and V. The atomic operation P decreases the integer S by 1 if S is greater than zero; otherwise, it waits. The atomic operation V, on the other hand, increases the integer S by 1. Such a semaphore is usually called a *counting* semaphore. A *binary* semaphore is a semaphore whose value is either 0 or 1. A counting semaphore may be "granted" to more than one task, which may cause multiple outgoing edges from the resource "semaphore,"

hence, is not permitted in this Single-Resource model. A binary semaphore may only be "granted" to at most one task, therefore, is allowed in the Single-Resource model. In the Ada environment, certain program restrictions must be enforced to ensure the single outgoing edge in GRG requirement of this model. For example, the Ada **accept** statement, which allows one of many potential calling tasks to be in rendezvous with the accepting task, must be programmed in a one to one fashion that limits the number of potential calling tasks to exactly one.

For the AND or OR model, some of the constraints of the previous Single-Resource model can be relaxed. In the Ada environment, once using an algorithm powerful enough to solve the AND model, then all deadlocks with the AND logic mechanisms involved, such as task termination, can be detected. Also, resources may be sharable in this model. For the OR model, all deadlocks with OR logic such as task interactions and synchronizations due to **accept** statements, can be detected. The counting semaphores can be used in the OR model.

The AND-OR and C(n,k) models are general enough that all the constraints of programming to conform to the previous two models can be removed. Resources may be indistinguishable or sharable. All the Ada task interaction and synchronization mechanisms are supported by the deadlock detection in this model.

2.6 Distributed Deadlock Detection and Resolution in Real-Time Systems

In this section, some of the design issues concerning deadlock detection problems in a distributed real-time environments are discussed. In the discussion, the problem is first defined. Then, design criteria for distributed deadlock detection and resolution algorithms for real-time systems are discussed.

2.6.1 Deadlock Problems in Distributed Real-Time Systems

In real-time systems, due to timing constraints attached to each task, time-outs and abnormal aborts may occur when a task is blocked. If a task T is blocked in a real-time environment, it may be involved in the following situations:

Stable Deadlock: This is the situation that the reachable set $RS(T)$ of the blocked task T in the GRG forms a deadlock (may be a cycle or a knot) and neither any time-out nor abnormal abort is expected. These deadlock conditions are stable in that once they are formed, they will remain until they are detected and resolved.

Temporal Deadlock: This is the situation where the reachable set $RS(T)$ of the blocked task T in the GRG forms a deadlock. However, due to timing constraints, a nonempty subset of tasks in a set of deadlocked tasks may be timed out or aborted from the blocked state. The deadlock situation, therefore, may not exist forever and, hence, is temporal.

Non-deadlocked Blocking: This is the situation in which a task is blocked, but it is not involved in any deadlock. The situation exists for a normal wait, or an abnormal condition such as being livelocked or being an orphan task. In a real-time setting, a task needs to make progress in a limited period of time. It is important that the waiting situation, for whatever reason, should be terminated in a timely manner to ensure the timing constraints.

The three situations stated above define three deadlock related problems in real-time systems. A stable deadlock in real-time systems is the same as a traditional deadlock in non-real-time systems. A temporal deadlock, on the other hand, is a special kind of deadlock which is not treated as a deadlock or is assumed not to exist in non-real-time systems. Such a deadlock is *temporal* and hence not *stable*. The *stable property* which is assumed in most of the traditional deadlock detection algorithms can no longer be used to detect temporal deadlocks in real-time systems. Timing constraints must be taken into consideration in detecting temporal deadlocks. The timing information collected for detecting temporal deadlocks can

also be applied to resolve many of the the problems associated with non-deadlocked blocking.

The detection and resolution of temporal deadlock is important for tasks with timing constraints in real-time systems. For example, in the Ada environment, task T may be in a temporal deadlock state if there is a cycle (or a knot) in its $RS(T)$ which contains tasks blocked by timed statements. If T carries the nearest deadline and the highest criticalness in the deadlocked task set, it is important that a timely detection and resolution is completed before a time in which it is still possible to meet the timing constraint of T. If no detection operation is attempted, task T may fail without knowing the existence of this temporal deadlock.

2.6.2 Criteria in Designing Distributed Deadlock Detection and Resolution Algorithms for Real-Time Systems

A deadlock detection algorithm is correct if and only if it satisfies the following criteria:

Safety Criterion: All the deadlocks detected by the algorithm must be genuine ones.

Progress Criterion: The algorithm must be able to detect any deadlock in the system in a finite time.

However, it is generally too expensive to completely achieve these two criteria when designing algorithms for distributed real-time systems. Some of the issues centered around the correctness criteria in distributed real-time systems are discussed next:

1. These criteria might be violated due to timing constraints in real-time systems. The timing constraints associated with tasks make deadlocks temporal (i.e., not stable, see Section 2.6.1). A temporal deadlock may disappear without ever being detected.

2. These criteria might be violated due to synchronization difficulties in distributed real-time systems. For example, if the deadlock detection computations are running concurrently with the other system activities, false deadlocks may be reported in the AND, AND-OR, and C(n,k) (see Section 2.5) deadlock models which violates Safety Criterion. To synchronize deadlock detection with the other system activities (e.g., to freeze the system while running a deadlock detection) is difficult and expensive, especially, in distributed real-time environments.

3. Another issue that arises is that sometimes one of the correctness criteria might be fulfilled at the sacrifice of the other one. Again, let's use the detection of AND model deadlocks as an example. It is required that the whole group of nested cycles should be detected together. However, most of the existing distributed deadlock detection algorithms for the AND model do not use such a complicated approach; instead, due to efficiency concerns, they simply detect and resolve individual deadlock cycles. When a cycle is detected, whether it is nested with other cycles is unknown. If so, it might have been broken at the intersecting part of the graph. Therefore, the limited information indicates only the potential existence of deadlocks and the algorithm is responsible for the decision of whether the situation is to be treated as a deadlock. Ignoring these potential deadlocks might violate the Progress Criterion. In contrast, treating these potential deadlocks as genuine ones might violate the Safety Criterion.

4. For soft real-time systems where violations will not cause any severe permanent faults, these two criteria can be relaxed to an acceptable level. For example, to cope with timing and synchronization problems described above, a compromise may be to speed up the algorithm so that the number of undetected and/or false deadlocks can be minimized. Also, only temporal deadlocks are allowed to go undetected since they will not remain in the system permanently. However, the effort of detecting temporal deadlocks before they disappear is still required in order to prevent a system from staying in

a deadlocked state for too long. As for the decision to be made regarding potential deadlocks, the Progress Criterion has a higher priority than Safety Criterion. The reason is that leaving the system in a deadlocked state is usually an uncontrollable fault whereas recovering from a false deadlock is a type of compensating action which impacts the system less severely. While this line of reasoning may not be true in some specific systems, our design of distributed deadlock detection algorithms for real-time systems will follow this criteria priority.

5. In many complex systems, (e.g., in distributed real-time systems,) structured (modular) and/or layered design approaches are used. Many deadlock detection or prevention algorithms only consider part of the system (i.e., a subset of the sites, the modules, and/or the layers of a system) as the problem domain and cannot detect or prevent deadlocks across related parts of the system. By related parts of a system we mean the sites, the modules, and/or the layers of a system which constitute the environment in which a group of interacting tasks execute. For example, suppose an algorithm is designed for detecting deadlocks at the application layer with the assumption that a prevention strategy is used in the underlying system to provide a "deadlock free" environment. The prevention strategy used in the underlying system has no knowledge of the user application layer, but simply prevents tasks from circular waiting upon system resources. Suppose we have two concurrent tasks T_1 and T_2 in the system. At the application layer, T_1 is waiting for T_2 in Ada's rendezvous while at the system layer T_2 is waiting for T_1 upon a system resource. Both waiting situations are allowed to occur separately in the different layers of the system. From a global viewpoint, the blockings across these two layers actually form a deadlock cycle. Such deadlocks cannot be detected or prevented unless a "complete" strategy is adopted.

In addition to the correctness criteria discussed above, we must consider a number of performance issues related to real-time requirements. One major question is whether deadlock detection is a feasible approach in a soft real-time system. After

all, if a task is blocked in a deadlock, then it is likely to miss its deadline unless the deadlock algorithm is invoked soon enough to not only detect and resolve the deadlock, but to also leave enough time for this task (and possible the aborted tasks) to complete (even in the presence of subsequent "normal" blocking conditions). Consequently, deadlock detection for a given task should begin as a function of its deadline, D, remaining execution time, E, and the execution time cost for deadlock detection and resolution, DR. In other words, deadlock detection for this task should start *no later than* $D - E - DR$. It would also be advantageous if the aborted task(s) were able to be restarted and also *still* make their deadlines. For many real-time systems we can assume that D and E are known (or at least we have good approximations for them). On the other hand, the execution time cost of deadlock detection and resolution will not be known and will vary considerably depending on the graph representing the "waiting" states of the distributed system, the cost and delays involved in sending messages, and the application processing the nodes are performing in addition to the deadlock algorithm itself. Fortunately, experience has shown that most deadlock cycles are short, so it may be possible to develop a reasonable estimate for DR for the common deadlock case. Note that when the estimates are wrong, one or more tasks involved in the deadlock will miss its deadline and abort, and thereby breaking the deadlock. Real-time deadlock detection will be successful when it finds deadlocks early enough, resolves them quickly; thus allows *more* and *higher value* tasks subsequently to make their deadlines than otherwise would have without deadlock detection (i.e., using schemes such as simply using timeout and abort, or using an "always abort" a lower value task if it blocks a high value task). Performance studies are required to determine which approach proves feasible in practice.

We must also consider resolution decisions based on real-time requirements. Sufficient information should be monitored and collected in order to make a good resolution decision to support real-time requirements. When collecting information to support real-time deadlock resolution, we need to consider: (1) the interdependency of the deadlocked tasks and their related tasks and (2) the timing

dependency among deadlocked tasks. In (1), by related tasks we mean the tasks (not necessary involved in the deadlock) which rely on the success of a deadlocked task. If a deadlocked task is chosen as the victim to be aborted, it may result a cascading abort of its related tasks. The reason for (2) is that a deadlock is a situation of cyclic wait and breaking a deadlock may result an acyclic wait which, in turn, results in a timing dependency among the surviving tasks. Depending on where a deadlock is broken, different timing dependencies might be formed. For example, a cycle of deadlocked tasks $T_1 \rightarrow T_2 \rightarrow T_3 \rightarrow T_1$ is detected. Each of these tasks may reside at different sites. Assume their criticalness order is $T_1 < T_3 < T_2$. A simple resolution strategy may be to abort the least critical task, which is task T_1 in this example. This resolution creates a timing dependency where T_2 waits for T_3 to satisfy its request. Suppose T_3 cannot complete in time for T_2 to make its deadline. Consequently, both T_1 and T_2 will fail in this resolution. If the timing dependency among these deadlocked tasks has been considered in the resolution, selecting T_3 as the victim to resolve the deadlock might be a better decision in that both T_1 and T_2 might succeed. Therefore, in addition to the criticalness of each task, the timing dependency information is needed to make a better resolution decision (in the sense that it allows more of the surviving tasks to meet their timing constraints).

Reliability of the distributed deadlock detection algorithm is another major concern. The underlying communication subsystem usually can be assumed to be reliable, and a major concern of the reliability in distributed systems is how to deal with site failures. Site failures in a distributed system usually change the system state which, in turn, cause the GRG collected at each site to become inconsistent. A deadlock detection algorithm is not reliable if it cannot quickly recover the GRG from a inconsistent state due to site failures. To ensure that deadlock detection computations function properly after site failures, the GRG inconsistency must be corrected in finite time. This basically can be achieved in one of the two ways: (1) inform the surviving sites of the failures to clean up inconsistent information or

(2) make the inconsistent information obsolete in the view of newly initiated deadlock computations so that the inconsistency may fade away. In any case, time is needed to recover and correct the inconsistency, and the deadlock computations execute concurrently with a site failure recovery may not be able to function properly. Therefore, the reliability criterion requires that the inconsistency be temporary and be corrected quickly so that only the deadlock detection computations in progress when the failures occur, may be affected.

Chapter 3

A SURVEY OF THE DISTRIBUTED DEADLOCK DETECTION AND RESOLUTION ALGORITHMS

Deadlock detection involves two basic tasks: (1) maintenance of the system state (in terms of TWFG, TRG, or GRG) and (2) search of the system state graph for the presence of deadlock conditions (cycles or knots). In distributed concurrent programming environments, a deadlock may involve several sites and the search for deadlocks greatly depends on the way the system state graph is maintained across the system.

Deadlock detection algorithms can be classified according to (1) the way the system state information is maintained or (2) the methodology used in searching for deadlock conditions. Singhal [124] classified the distributed deadlock detection algorithms based on the former notion into three types: *centralized*, *distributed*, and *hierarchical* approaches. In contrast, Knapp [77] adopted the latter notion to classify the distributed algorithms for the distributed deadlock detection into four classes: ***path-pushing algorithms***, ***edge-chasing algorithms***, ***diffusing computations***, and ***global state detection***. In this chapter, the distributed deadlock detection and resolution algorithms are summarized and categorized in the following sections according to both Singhal and Knapp's classifications.

3.1 Centralized Algorithms for Distributed Deadlock Detection

In the centralized algorithms for distributed deadlock detection, a designated site, often called the control site, has the responsibility of maintaining global system state information and searching it for deadlock conditions. The global system state information may be maintained continuously (either keeping track of the up-to-date system state information periodically, or whenever the system state changes) or

gathered from each site whenever the deadlock detection computation is needed. Since the control site can be viewed as a global system monitor, the kind of deadlock which can be detected is unrestricted. However, for simplicity, some algorithms place restrictions on the information gathered and on the algorithms used for searching for deadlocks (for example, searching for cycles instead of searching for knots), which, in turn, will restrict the kind of deadlock that can be detected. Since the control site has a global view of the whole system, deadlock resolution policy is relatively easy to carry out if the system was found in a deadlock state.

Centralized algorithms are conceptually simple and easy to implement, but they are highly inefficient and unreliable. The control site has to be informed by all the operations (resources requests, releases, etc.) across the system. Long response time for user requests, large communication overhead, and congestion of communication links near the control site are the problems which we can expect. Also, the communication and computation overheads of the deadlock detection are unrelated to the frequency of occurrence and the structure of deadlocks. Because the status of resource allocation is centralized at a single site, the centralized algorithms for distributed deadlock detection are subject to a single point of failure.

Some problems (such as long response time and congestion of communication links near the control site) can be diminished by having each site maintain its local resource status table for all its resources. For each resource, the status table keeps track of the tasks which have acquired the resource or are waiting for the resource. The control site, therefore, can collect these local resource status tables periodically for construction of the global system state. However, as pointed out by Ho and Ramamoorthy [64], the global system state may be inconsistent and false deadlocks may be detected due to the inherent communication delay and the lack of perfectly synchronized clocks in a distributed system.

For example, suppose two resources R_1 and R_2 are located at sites S_1 and S_2 respectively. Let the following two tasks T_1 and T_2 be started almost simultaneously at sites S_3 and S_4 respectively, and execute lock $L(\cdot)$ and unlock $U(\cdot)$ operations in the following total order sequence:

T_1 :	$L(R_1)$			$U(R_1)$			$L(R_2)$			$U(R_2)$
T_2 :		$L(R_1)$			$U(R_1)$	$L(R_2)$			$U(R_2)$	
time :	$\longrightarrow$		t_1					t_2		

The scenario is that both tasks T_1 and T_2 request resources R_1 and R_2 at sites S_1 and S_2, respectively. At time t_1, T_1 holds R_1 and T_2 is waiting for R_1 at site S_1. Both T_1 and T_2 finish accessing resource R_1 and start to request R_2 between time t_1 and t_2. At time t_2, T_2 holds R_2 and T_1 is waiting for R_2 at site S_2. Due to the communication delay and imperfect synchronization, site S_1 reports its local resource status table as $T_2 \rightarrow R_1 \rightarrow T_1$ at time t_1 and site S_2 reports its local resource status table as $T_1 \rightarrow R_2 \rightarrow T_2$ at time t_2. After constructing the global system state at time t_2, the control site reports a false deadlock $T_1 \rightarrow R_2 \rightarrow T_2 \rightarrow R_1 \rightarrow T_1$.

Ho-Ramamoorthy Algorithm: Based on the observations of the above problems, Ho and Ramamoorthy [64] proposed two algorithms called the *two-phase* and *one-phase* deadlock detection protocols. Their protocols were intended to solve the deadlock problem in AND models because only cycles are searched for in attempting to detect deadlocks. However, this restriction is not necessary. A more complex deadlock searching algorithm can be adopted in the designated control site to handle a more complex deadlock model.

In the two-phase deadlock detection protocol, each site maintains a status table for all the tasks that are initiated at that site. The status table of each task keeps track of the resources that the task has acquired and the resources that the task is waiting for. Periodically, a designated control site requests the status table report from all sites and constructs a global system state. If the global system state contains no cycle then there is no deadlock. Otherwise, the control site again requests the status table report from each site. Only the tasks common to both reports are used to construct a second phase global system state. The system is declared deadlocked if the same cycle is detected again in the second phase global system state.

Ho and Ramamoorthy claimed that a consistent view of the system can be guaranteed by using this two-phase deadlock detection protocol. However, Jagannathan and Vasudevan [72] have disproved the claim by a counterexample which shows false detection of deadlocks. Using two report phases reduces the probability of false detection but it does not eliminate it.

On the other hand, the one-phase deadlock detection protocol detects deadlocks in one communication phase. Two tables, the resource status table and the task status table, are maintained at each site and gathered by the designated control site periodically. Only the tasks that appear in both tables are used for the construction of the global system state. The system is deadlocked if and only if cycles can be found in the global system state.

The correctness of the one-phase deadlock detection protocol has been proven by Ho and Ramamoorthy. No false deadlocks are detected by the protocol because the inconsistency is eliminated by matching the information gathered from two status tables. In the bipartite GRG, the edges only appear between two disjoint groups of vertices, a group of tasks and a group of resources. An edge really exists if it is reported by both of its end vertices, that is, indicated both in a task status table and in a resource table. This pessimistic way of collecting global information might miss the detection of some real deadlocks formed while the status tables are being collected. However, all deadlocks will be detected eventually by the protocol due to the stable property of deadlocks.

The one-phase protocol is faster and requires fewer messages than the two-phase protocol. But it requires more storage for maintaining two status tables at each site as well as for transferring these two status tables in one communication phase.

3.2 Distributed Algorithms for Distributed Deadlock Detection

In distributed algorithms for deadlock detection, all sites cooperate to detect deadlocks in the system. Only partial knowledge of the global system state, which is enough for the detection of deadlocks, is learned at each site. Deadlock detection

computation can be initiated whenever there is an indication of a possible deadlock. In many of the distributed algorithms, the computation is initiated when a task suspects a deadlock[1], and the initiation site can be either the site where the task resides or the site where the awaited resource resides. The methodology used for searching for deadlock conditions can be roughly divided into four classes [77]: *path-pushing*, *edge-chasing*, *diffusing computations*, and *global state detection*, which are discussed in the following four subsections.

There are several reasons why distributed algorithms for deadlock detection are more attractive than centralized ones. First, unlike the centralized algorithms, the distributed algorithms are not subject to a single point of failure, and there is no single control site to be swamped with deadlock detection activities which might become a system bottleneck. Second, based on the observation of typical applications, the frequency of deadlocks (especially, the global ones) are low and the length of the deadlock cycles are short [55, 12]. In the distributed algorithms, deadlock detection can be initiated only if the system is suspected to be deadlocked and can be executed only at sites involved in the suspected deadlock, which makes it more efficient than the fixed cost centralized algorithms. Furthermore, distributed algorithms for deadlock detection could be used with the algorithms for deadlock prevention and deadlock avoidance to provide a better fault tolerance while keeping the total cost of the deadlock detection low. Third, a distributed algorithm is not necessarily more complex than a centralized one as most of the literature has pointed out. Both centralized and distributed algorithms have similar difficulties in the distributed environments due to the lack of global synchronized clocks and the inherent communication delay. A major factor that determines the complexity of a deadlock detection algorithm is what kind of deadlock model is being addressed by the algorithm. The Mitchell-Merritt algorithm [100] is a fully distributed Single-Resource model deadlock detection algorithm that has been claimed to be very simple and has been proved to be correct. According to the authors, their first

[1]In many cases, a waiting task sets a timer and if the time expires then it suspects deadlock and initiates the algorithm.

algorithm has been implemented in a database system in under an hour by one of them. One drawback of the distributed algorithms is the resolution of the detected deadlock is usually more difficult (except for the strategy that simply aborts the task which detects the deadlock) than that in the centralized ones. This is basically due to the lack of complete global information at each site and the same deadlock might be seen (or detected) at more than one site.

3.2.1 Path-Pushing Algorithms

Closely following the ideas of the centralized algorithms, the early distributed algorithms for deadlock detection maintained the notion of an explicit global system state. The basic idea is to construct a simplified form of global system state graph at each site which is sufficient to detect deadlocks. When a deadlock computation is initiated, each site sends its local state information to a number of neighboring sites. Upon receiving the local state information from neighboring sites, the local state graph is updated and then passed along. The procedure is repeated until some site has a sufficiently complete picture of the global system state graph to announce deadlock or to ensure that no deadlock exists. The name *path-pushing algorithms* come from the main feature of this scheme, that is, to transmit partial paths for the construction of the global system state graph.

This class of algorithms appeared around 1979-1982 when distributed algorithms for deadlock detection were first explored. Many algorithms in this class have been "proved" correct. However, they were subsequently found to be incorrect because they may fail to detect true deadlocks or they may detect false deadlocks. For example, Gligor and Shattuck [54] have illustrated that the distributed scheme presented by Menasce and Muntz [95] is incorrect because message could be out of order, and hence, may fail to detect genuine deadlocks or may detect false deadlocks. Obermarck's algorithm and proof [106] have been identified incorrect by himself as well as Elmagarmid [43] and Knapp [77] in the sense that false deadlock may be detected. A plausible reason for these algorithm failures is that the notion of consistent global state in the distributed environment was not well understood.

Consequently, most of the algorithms had to rely on freezing of the underlying system computation while the deadlock detection is proceeding.

Though these algorithms are incorrect, the Menasce-Muntz algorithm [95] and the Obermarck algorithm [106] are briefly summarized in order to describe the difficulties of this class of algorithms. The discussion is followed by the Badal algorithm [8] which is a correct extension of the Obermarck algorithm.

Menasce-Muntz algorithm: The Menasce-Muntz algorithm [95] was first to use a simplified form of the *transaction-wait-for graph* (see TWFG in Section 2.1). In their algorithm, only portions of the complete global TWFG are maintained at the transaction's original site for the detection of deadlocks. The underlying deadlock model is the AND model; hence, the algorithm searches for cycles in a subset of the global TWFG.

A transaction can be either *blocked* (with outgoing edges in the TWFG) or *non-blocked* (without outgoing edges in the TWFG). A *blocking-set(T)* of a transaction T is the set of all nonblocked transactions T' which can be reached from T in the TWFG via a directed path. When a transaction T is blocked, for each transaction T' in the *blocking-set(T)*, a blocking pair (T, T') is identified and this information is sent to the original site of T and T'. When the original site of T' receives the pair and it is not the original site of T, new blocking pairs (T, T'') are sent to the original site of T'' for each transaction T'' in the *blocking-set(T'')*. On receiving such pairs a site updates its TWFG and performs local deadlock detection in addition to transmitting new blocking pairs to other sites. A path (or cycle) thus constructed in a site's TWFG may be "condensed" in the sense that some intermediate transactions may be missing.

Gligor and Shattuck [54] have shown that this algorithm fails to detect genuine deadlocks or may detect false deadlocks. First, in the remote request case, the determination of whether a transaction is blocked or not is incorrect because the determination can not be made until the response come back from the remote site. Furthermore, when sufficient information is obtained to determine correctly whether

a transaction is blocked (i.e., when a blocking pair arrives), the protocol fails to use that information. In providing a remedy to these problems, Gligor and Shattuck suggested that every site should determine precisely whether a transaction is active, blocked, or waiting for the response of a remote request. Further, when a transaction is blocked it, in addition to identifying its blocking set, should identify a "potential blocking set" (i.e., the set of all waiting transactions reachable from a blocked transaction). As the authors point out, the modified algorithm is impractical because the implementation complexity arising primarily due to the condensation of the TWFG. For example, consider a waiting chain $T_1 \rightarrow T_2 \rightarrow \cdots \rightarrow T_n$ in which no two transactions reside in the same site. The chain is condensed into $T_1 \rightarrow T_n$ at the original site of T_n. When T_n releases its resources, there is no straightforward way to inform all the intermediate sites in this chain to update their TWFG's.

Obermarck algorithm: The Obermarck algorithm [106] combines Gray's centralized deadlock detection algorithm [56] and the Menasce-Muntz algorithm described above. The algorithm is developed for a distributed database system, in which transactions can have agents at multiple sites, but only have a single locus of control. The reliable underlying communication system assures that all messages are sent and received in the same sequence. The underlying deadlock model is the AND model; hence the algorithm searches for cycles to detect deadlocks.

In the algorithm, the local TWFG is built at each site by a deadlock controller, and both resource waits and communication waits are gathered from database and communication managers. The transactions are lexically ordered by their identifiers and the deadlock messages are only transmitted in one direction which reduces the overhead as well as assuring a single detection point of a deadlock cycle. A special vertex in a local TWFG called EX-ternal is used to represent intersite wait-for relations. The existence of the vertex EX in a local TWFG indicates potential global deadlocks. Only the portion(s) of the local TWFG related to potential global deadlocks are shipped as *String(s)* from one site to another to detect global deadlocks. Deadlock detection at each site is an iteration of the following steps executed by a deadlock controller. The deadlock controller:

1. waits for and receives deadlock related information (produced in the previous deadlock detection iteration) from other sites,
2. combines the received information with local TWFG and searches for cycles,
3. resolves local deadlocks by choosing a victim from each of the local cycles which does not have an EX vertex, and
4. for all cycles $EX \rightarrow T_1 \rightarrow \cdots \rightarrow T_n \rightarrow EX$ that contains vertex EX (those cycles constitute potential global deadlocks), sends the String $EX \rightarrow T_1 \rightarrow \cdots \rightarrow T_n$ to each site where the transaction T_n has an agent if the transaction identifiers has the order $T_1 > T_n$.

It is assumed in the correctness proof that the portion(s) of the local TWFG shipped as String(s) from one site to another will be frozen until it has been received and processed at some *final* site. A final site is either (1) the site where the TWFG information completes a cycle, or (2) the first site where a global deadlock can be proved not to exist. The assumption attempts to get a snapshot of part of the system for the deadlock detection algorithm, and Obermarck admits that it is not valid in real-world distributed systems. With this assumption, Obermarck points out that the local TWFG still may not be a true picture of the wait-for relations in the single site. False deadlocks, therefore, may be detected even with this assumption.

This algorithm has the primary advantage that the TWFG information is transmitted only when there is a potential global deadlock. However, the algorithm suffers from the detection of false deadlocks as pointed out above. Also, building the TWFG is expensive and the analysis for cycles is repeated every time a deadlock detection is initiated.

Badal algorithm: In an attempt to optimize the costly distributed deadlock detection algorithms in terms of detecting the most frequent deadlocks with maximum efficiency, Badal proposed a three level algorithm extension [8] of the Obermarck algorithm [106]. A deadlock cycle can have many different topologies. Badal partitioned the deadlocks into four types according to their topology. In

general, a deadlock can either be *local* (all the involved transactions and resources reside in a single site) or *global* (some of the involved transactions or resources might be remote). It is not necessary to use a costly algorithm designed to detect complex global deadlocks for detecting simple local ones. Also, the long deadlock cycles are less likely to occur and most of the deadlock cycles have only a length of two. This fact leads to the idea of simplifying the detection of global deadlocks by making the algorithm more efficient in detecting shorter global deadlocks. Based on these ideas, the performance is optimized in Badal algorithm by using three levels of deadlock detection to cope with different complexity classes of deadlocks. The activity at each level is more complex and costly than that at the preceding level.

The deadlock detection starts at the first level algorithm which detects global deadlocks of cycle length two. The level two algorithm starts whenever a requested resource is still not available after a period of time and no deadlock has been detected in level one. If all the involved activities are local, the level two algorithm can decide whether a deadlock occurs or not; otherwise, it will wait for another period of time after which it then either ships the information to the third level algorithm if the resource is still not available or proceeds if the resource becomes available. The third level of this algorithm is basically the same as the Obermarck algorithm [106] (with improvements which can reduce one intersite message) and, hence, suffers similar inefficiencies.

Badal has proved that the algorithm can detect the most frequent deadlocks with a minimum of intersite messages. However, he also pointed out that the algorithm has a constant overhead due to (1) the necessity to keep track of more information (in the lock tables) and (2) the frequent checking of deadlocks of length two. Consequently, this algorithm is most likely to be used in distributed environments where deadlocks occur frequently enough to justify the continuous overhead.

3.2.2 Edge-Chasing Algorithms

This class of deadlock detection algorithms received the name from the way they search for cycles in a distributed system state graph. A special message called a *probe* is initiated and propagated along the edges (called an edge-chasing algorithm by Moss [101]) of the system state graph. If the probe (or a matching one) is finally received by its initiator then a cycle is detected. Since this class of algorithms can only detect cycles, they can only be applied to solve deadlock models of complexity up to and including the AND model.

A nice feature of this approach in connection with deadlock detection is that the complexity and efficiency of an algorithm can vary according to the complexity of the underlying deadlock model. For example, Mitchell-Merritt algorithm [100] is a very simple and efficient algorithm based on the Single-Resource model while Chandy-Misra algorithm [23] is somewhat more complex but can be used in a more general AND deadlock model.

A *priority* based probe algorithm is an extension of the edge-chasing algorithm. The priority is used to reduce the probe messages and provide deadlock resolution. Several priority based algorithms have been proposed: Mitchell-Merritt algorithm [100], Sinha-Natarajan algorithm [126] (remedied by Choudhary et al. [35, 36]), and Sanders-Heuberger algorithm [118]. They are all designed for the Single-Resource deadlock model.

Another variation of the probe based algorithm is called the SET-based algorithm proposed by Chandy and Misra [23], Haas and Mohan [57], and Choudhary et al. [34]. In the SET-based algorithm, a "SET" of the system state information is propagated along with the probe. According to the Choudhary's simulation results, the SET-based algorithm is more efficient than the probe based algorithms.

For this class of algorithms, we only present the Mitchell-Merritt algorithm [100] as an example due to its simplicity and elegance. Mitchell-Merritt algorithm has two versions (one without priority and the other one with priority) which capture most of the important features of the edge-chasing algorithms.

Mitchell-Merritt algorithm: The Mitchell-Merritt algorithm [100] is an edge-chasing algorithm in which probes are sent in the opposite directions on the edges of the TWFG. Each vertex of the TWFG has two labels: *private* and *public.* The private label of each vertex, though not constant, is unique and non-decreasing to the vertex all the times. The public label of each vertex can be read by other tasks and need not be unique. Initially, the labels at each vertex have the same value. The algorithm is executed in the following four steps:

1. *Block Step*: When a task begins to wait for another task upon some resources, it becomes *Blocked.* Both of the labels of the *Blocked* task are increased to a value greater than their previous values and greater than the public label of the blocking task.

2. *Active Step*: A task becomes *Active* when it gets a resource, times out or fails. Also, when a waiting task notices that the owner of a awaited resource changes, it must become *Active* and then *Blocked* again if it does not acquire the resource.

3. *Transmit Step*: When a *Blocked* task discovers that its public label is smaller than that of the blocking task, it updates its label to a value equal to that of the blocking task.

4. *Detect Step*: When a task receives its own public label back, a deadlock is detected.

When a task become *Blocked* it adds an edge to the TWFG by executing the *Block Step*. When a task becomes *Active*, it deletes an edge from the TWFG by executing the *Active Step*. The *Blocked* tasks execute the *Transmit Step* periodically. The effect is that the largest public label tends to migrate in the opposite direction along the edges of the TWFG. Only one task in a cycle will detect deadlock, which simplifies the problem of resolution. Only genuine deadlocks will be detected in the absence of spontaneous aborts. Spontaneous aborts are allowed, however, if false detection of deadlocks can be tolerated.

The simple algorithm given above can be easily extended to include priorities such that the lowest priority task in a deadlock cycle aborts itself. Each vertex in the TWFG now has two additional unique *priority numbers*, one private and one public. Initially, each task posts its unique private label and priority number. In the *Transmit Step*, vertices with equal public labels transfer the lower priority number. And in the *Detect Step*, a task detects a deadlock and aborts itself when it receives its own priority number together with its own public number. Since the task with the highest public label may be waiting for the lowest priority task, the low priority number may not be propagated around the cycle until after the public label has gone full-cycle. Thus, this algorithm can take up to twice as long to detect deadlock as the above first algorithm.

As pointed out by Knapp [77], the algorithm does not remain correct if public labels are propagated in the same direction as the directed edges instead of the other way around. The reason for this is that if the largest public label is owned by a deadlocked task which is not part of a cycle, this label might enter the cycle and circulate once without detecting the deadlock. After all the public labels in the cycle have been updated to a larger value than any of their labels, none of the tasks in the cycle would detect the deadlock. Also, for a similar reason, there seems to be no straightforward way to extend the algorithm to handle deadlocks in the AND model.

3.2.3 Diffusing Computations

Dijkstra and Scholten [42] introduced the notion of *diffusing computation* in a distributed system of tasks, and suggested an algorithm to detect the termination of an arbitrary diffusing computation in any network environment. The generality of the solution makes it is a candidate for a number of problems in the distributed systems. Using this algorithm, Misra and Chandy presented algorithms for distributed deadlock detection for CSP-like environments [97, 98]. Their work was followed and improved upon by Huang [68]. Also, Chandy, Misra and Haas [27] presented an algorithm for a communication deadlock model which is an OR model

in the GRS. Hermann and Chandy's algorithm [63] for the AND-OR model is a so called *tree* computation which is a hierarchy of diffusing computations.

First, we summarize the diffusing computation and its properties from [42]. In a distributed system, a diffusing computation can be mapped to a directed graph such that each vertex represents a task and each directed edge represents the relation between its two end tasks. If the graph contains an edge from vertex T_1 to vertex T_2, T_1 is called a *predecessor* of T_2 and T_2 is called a *successor* of T_1. It is assumed that there exists a root initiator vertex without incoming edges, from which a diffusing computation is initiated. The initiator is called *the environment* by Dijkstra and Scholten [42] due to its relation to the rest of the graph. Vertices different from the environment are called *internal* vertices. The *messages* are sent from a vertex to its successors, while the *signals* are propagated in the reverse direction along the edges. A diffusing computation grows by sending messages and shrinks by receiving signals. It is this feature that inspired the name *diffusing computations.*

Each vertex in a diffusing computation begins with a *neutral state.* The environment initiates the computation by sending out messages to its successors. In a diffusing computation, the first message sent to a vertex is called an *engaging message* while the last signal sent by a vertex, which is used to reply to its engaging message, is called an *engaging signal.* Upon reception of the first message (i.e., the engaging message), a vertex leaves its neutral state and becomes *engaged.* An internal engaged vertex (1) is free to propagate messages (including its engaging message) to its successors, and (2) is free to send signals to its predecessor for every message (except its engaging message) it receives. An internal engaged vertex is able to receive signals from its successors. An engaged vertex sends an engaging signal to its predecessor when it receives engaging signals from all its successors.

For each edge, its *deficit*, a non-negative value, is defined as the number of messages minus the number of signals transmitted over it. The neutral state of a vertex can now be redefined to be the state in which the deficits of all incoming and outgoing edges are zero. The diffusing computation terminates if the environment

returns to its neutral state because it implies that all the internal vertices are in the neutral state.

If an engaged vertex T_1 sent an engaging message to its neutral state successor T_2, the edge (T_1, T_2) is called an *engagement edge.* The engagement edge (T_1, T_2) will be finally canceled after the vertex T_2 sends an engaging signal to T_1 and returns to the neutral state. The lifetime of the engagement edge (T_1, T_2) is the same as the duration while T_2 is engaged. It follows that: (a) each engagement edge connects two engaged vertices, (b) engagement edges do not form cycles, and (c) each engaged internal vertex has exactly one incoming engagement edge. Consequently, the engagement edges form a rooted tree during a diffusing computation, and all engaged internal vertices are reachable from the environment via paths of these engagement edges.

The basic idea of using the diffusing computation for deadlock detection in the OR model is that the GRG can be implicitly reflected in the structure of the computation. When a task suspects a deadlock, it initiates a diffusing computation. Different from the original diffusing computation, the initiator is not necessarily the root of a tree in the GRG. The initiator may have incoming edges which are the hints of the deadlocks. During the computation, the engagement edges always form a *engagement tree* spanning over the set of tasks which are waiting for each other. The engagement tree contains no cycle. Such an engagement tree eventually vanishes if all the outgoing edges in the set of waiting tasks find a cycle. Therefore, if the computation finally terminates, the initiator declares a deadlock. If no deadlock exists, the initiator will remain idle in the engaged state until its resource requests are fulfilled, and the transmission of all the messages and signals will simply stop. The algorithms summarized below will further clarify how one uses diffusing computations for deadlock detection.

Chandy-Misra-Haas Algorithm: By using the paradigm of diffusing computations, Chandy, Misra, and Haas [27] presented a distributed deadlock detection algorithm for the communication (OR) model.

In this algorithm, messages are called *queries* and signals are called *replies*. A blocked task initiates a deadlock computation by sending queries to tasks in its dependent set. An executing task ignores all queries and replies. On the other hand, if the diffusing computation reaches a blocked task then that blocked task becomes *engaged* and participates in the computation. A query is answered whenever it reaches an engaged task. A query will be eventually answered if it travels along the edges of a cycle, and all the involved engagement edges will be canceled. It follows inductively that engagement edges do not form cycles. All the queries initiated find cycles if the diffusing computation terminates and the initiator returns to its neutral state. Consequently, the initiator is deadlocked if it returns to its neutral state.

As an example, a blocked task T_a initiates a deadlock detection by a diffusing computation and it requires resources B or C which are held by tasks T_b and T_c, respectively. Consider the situation that task T_b is executing and task T_c is waiting for T_a. The query received by task T_b will be ignored because it is not blocked. Upon receiving the query from T_a, task T_c becomes engaged and propagates the query to its dependent set which is $\{T_a\}$. Since T_a is in the engaged state, a reply will be sent back to T_c immediately. When T_c receives the reply from T_a, it sends back an engaging reply to T_a and returns to neutral state. No more queries and replies related to the computation initiated by task T_a will be send at this point. Task T_a remains idle and keeps waiting for the outcome of the executing task T_b. Consider another situation that both tasks T_b and T_c are waiting for the task T_a. Both engagement edges (T_a, T_b) and (T_a, T_c) will eventually be canceled. Finally task T_a terminates the computation and declares a deadlock after it receives replies from all the tasks in its dependent set $\{T_a, T_b\}$ and returns to its neutral state.

Hermann-Chandy Algorithm: Hermann and Chandy's algorithm [63] for the AND-OR model is a so called *tree* computation. A tree computation consists of a hierarchy of diffusing computations. First, the algorithm requires a general AND-OR request to be mapped to a regular form, such as disjunctive normal form. A task may have either an AND request or an OR request. An AND-OR request

issued by a task is mapped to a tree of tasks with only AND or OR requests. A blocked task T is deadlocked, if either

- T is an AND task and at least one of its requests will not be satisfied, or
- T is an OR task, and none of its requests will be satisfied.

The basic idea of the algorithm is that whenever a diffusing computation reaches a blocked OR task, the computation is propagated to its dependent set; if the engaged task is an AND task, it initiates a separate tree computation for each of its outgoing edges. Therefore, a tree computation consists of either a diffusing computation or a set of tree computations. Since an engaging reply (the one that replies to an engaging query) means an engaged blocked task has been reached, a cycle is found along the path on which the reply returns. An AND task will send back an engaging reply when it receives an engaging reply from one of its dependent set. An OR task will send back an engaging reply when it receives all the replies from its dependent set. If the initiator is an AND task, it terminates when it receives an engaging reply. If the initiator is an OR task, it terminates when it receives all the engaging replies. A tree computation terminates and declares deadlock whenever the initiator terminates.

3.2.4 Global State Detection

A key notion in distributed global state detection algorithms is that a *consistent global state* can be determined without freezing the underlying system computations. Chandy and Lamport [21] proposed the notion of *distributed snapshots* which is a mechanism to get information about the global state of a distributed computation without synchronization. Spezialetti and Kearns [128] modified this algorithm to gain more efficiency in some special application cases. The algorithm is intended to provide a general purpose framework which can be adapted to specific implementation requirements.

The Chandy and Lamport's global snapshot algorithm runs in two independent phases. During the first phase, each task records its own state, and the two tasks

connected by a communication channel cooperate in recording the channel state. Due to the lack of global clock, the recording procedure can only be carried out asynchronously, and the recorded system state is required to be "meaningful." The recorded task and channel states will then be collected and assembled in a second phase after the completion of the first phase.

The recorded task and channel states might be inconsistent in the global system state. For example, there are two tasks T_1 and T_2 connected by the channel C. Task T_1 sends a message along C to T_2. If T_1 records the state of channel C *before* sending the message and records its own state *after* sending the message, then an inconsistent global system state will erroneously show that a message is missing. Thus a "meaningful" global system state cannot be formed in this way.

Chandy and Lamport suggested that a special message, called a *marker*, being sent to avoid such inconsistent states. The outline of their algorithm is as follows:

Marker-Sending Rule for a Task T_1: For each outgoing channel C connected to T_1, a marker is sent along C after the state of T_1 is recorded and before any further message is sent along C.

Marker-Receiving Rule for a Task T_2: On receiving a marker along a channel C:

```
if T2 has not recorded its state then
    T2 records its own state;
    T2 records the state of C as empty;
else
    T2 records the state of C as the sequence of messages
    received along C between the event where it recorded its own
    state and the event where it received the marker;
end if;
```

The Distributed Snapshot algorithm is completely self-contained and makes no assumptions about the states it collects. The only assumption it utilizes is the *stable* system property. Therefore, this algorithm is suitable for a variety of stable system

property applications such as termination detection, deadlock detection, and global state monitoring.

In using the snapshot algorithm for distributed deadlock detection, the collected global system state is a consistent GRG. Let GRG_t denote the GRG at time t. Bracha and Toueg [14] proved that if the time t_1 is earlier than the time t_2, and a task T is deadlocked in the GRG_{t_1}, then the task T is deadlocked in the GRG_{t_2}. This result allows a global snapshot of the GRG to be analyzed for deadlocks in parallel with the continued operation of the system.

Bracha-Toueg Algorithm: Bracha and Toueg [14] proposed an algorithm to process the system snapshot GRG to find deadlocks in the C(n,k) model. Since there is no simple construct of graph theory in terms of GRG to describe the deadlock condition in the C(n,k) model, the graph reduction techniques suggested by Holt [66] are used to determine the existence of deadlocks. Each of the active tasks in the snapshot GRG can be scheduled to terminate and to release the resources it holds. The GRG thus can be *reduced* to a new state (see Section 2.4.2). A GRG is said to be *completely reducible* if there exists a sequence of graph reductions that reduces the GRG to a set of isolated vertices. A task T_i is not deadlocked in state S if and only if there exists a sequence of reductions in the corresponding GRG that leaves T_i unblocked. If a GRG is completely reducible, then the state it represents is not deadlocked.

3.3 Hierarchical Algorithms for Distributed Deadlock Detection

In hierarchical algorithms for distributed deadlock detection, sites are logically organized in a hierarchy. Each site is responsible for detecting deadlocks involving itself and its children sites. In most of the hierarchical arranged systems, the resource access pattern is localized to a cluster of sites belonging to a same parent site. The performance, therefore, can be optimized if a hierarchical deadlock detection algorithm reflects the hierarchical structure of its underlying system.

Menasce-Muntz Algorithm: In the hierarchical deadlock detection algorithm presented by Menasce and Muntz [95], the database (DB) is partitioned into a set of subdatabases (sub-DB's). The locking and deadlock controllers are arranged in a tree fashion. Each of the *leaf* controllers manage a sub-DB, while a *non-leaf* controller is responsible for distributed deadlock detection. A leaf controller maintains the part of the global GRG concerning the sub-DB managed at the controller. A non-leaf controller maintains the GRG including its children controllers and is responsible for detecting deadlocks involving only its children controllers. A non-leaf controller keeps track of the changes, such as the occurrences of allocation, wait, or release of the resources, in each of its children controllers. This job can be done continuously (that is, whenever a change occurs) or periodically. After each update of its own GRG, a non-leaf controller performs deadlock detection. This algorithm is developed for the AND model since it searches for cycles. However, the approach can be applied to a more complex deadlock model if an appropriate deadlock detection algorithm is used.

Ho-Ramamoorthy Algorithm: In the hierarchical deadlock detection algorithm presented by Ho and Ramamoorthy [64], sites which are close to each other are grouped into a cluster. A site in a cluster is chosen as the control site periodically. A control site collects status tables from all the sites in its cluster, and applies a one-phase deadlock detection protocol (see their algorithm in Section 3.1) to detect intra-cluster deadlocks. Also, a central control site is chosen dynamically which then collects the inter-cluster information and constructs a system wide GRG for inter-cluster deadlock detection.

3.4 Summary of the Distributed Deadlock Detection Algorithms

In this section, we have surveyed, though not completely, a variety of distributed deadlock detection algorithms. To summarize, we want to emphasize two issues: correctness and performance of these algorithms.

During the survey of the distributed deadlock detection algorithms, we have observed that correctness proofs are very difficult in this area. The basic reason is that these algorithms are developed for the distributed environments which are usually complex and non-deterministic in nature. The number of execution patterns of an algorithm can be extremely large which eliminates the possibility of exhaustively studying all possible situations. The lack of perfect global synchronization further complicates the correctness proof due to the timing sensitivity of the algorithms. Consequently, informal proof techniques based on intuitive arguments are widely used in most of the literature. These informal correctness proofs are relatively unreliable. Many algorithms were informally proved and claimed correct by their authors have been disproved later (many examples can be found in our survey). However, it is worth noting that the situation has been improved recently due to some important concepts, such as "global state consistency," "stability of the deadlocks," "complexity of the deadlock problems," and "diffusing computation," that have been introduced. These concepts makes the analysis of the problems and the resolutions more precise and, therefore, enhances the reliability of the algorithm correctness.

To evaluate the performance of a distributed deadlock detection algorithm is difficult because there are many related factors, such as the message traffic, the size of messages, the complexity of the problem to be addressed, the complexity of the algorithm, the frequency that the detection computation is invoked, the percentage of the invoked computations that detect deadlock, etc., which could affect the performance of an algorithm. Also, many algorithms were motivated by the performance improvement to a previously developed one. However, their achievements are usually hard to judge. For example, many authors of these improved algorithms argue that their algorithms could perform better because they reduce the deadlock message traffic. This argument, however, may not be true due to the fact that the complicated new algorithm may increase the overhead during the normal computations while they attempt to reduce the deadlock message traffic.

Therefore, a precise performance comparison among different algorithms is very difficult.

In Table 3.1, we have listed the performance of the surveyed algorithms from a variety of aspects. The algorithms of the hierarchical approach are excluded because they are the modified versions that take the workload access pattern into consideration and their performance upper bound is exactly the same as their original versions. The values are gathered from the original articles, Singhal's survey [124], and Knapp's survey [77]. The algorithms listed are developed for different kind of system environments which may have different problem complexities. Consequently, the values listed in the Table 3.1 can only serve as a reference and cannot be used to judge which algorithm performs better.

In general, Knapp [77] compares the performance of the distributed algorithms in terms of the problem complexity and the algorithm complexity. He concludes that the Single-Resource, the AND, and the OR models all have a worst-case algorithm complexity of $O(N^2)$ messages, in which N is the number of the vertices in a GRG. The AND-OR model requires at most $O(N^3)$ messages which doesn't include the complexity of the mapping used to transform a general request, say a C(n,k) request, to the regular AND-OR form. The C(n,k) model needs $O(N^2)$ messages to construct a consistent global snapshot in which the complexity of searching a snapshot for deadlocks is not included. Therefore, Knapp suggests that in selecting a deadlock algorithm for a particular application, the least general technique which is still general enough to solve the problem is advised.

Table 3.1: Performance of the Surveyed Algorithms

Algorithms	No. of Messages	Delay	Size of Messages	Problem Complexity
I. Centralized Approaches				
Ho-Ramamoorthy (two-phase)	$4N$	B: $4T$ W: $p+4NT$	V,L	AND
Ho-Ramamoorthy (one-phase)	$2N$	B: $2T$ W: $p+2NT$	V,L	AND
II. Distributed Approaches				
1. Path-Pushing Algorithms				
Menasce-Muntz	$m(n-1)$	nT	V,S	AND
Obermarck	$m(n-1)/2$	nT	V,M	AND
Badal (type I)	$\mathcal{M}-1$	$(\mathcal{M}-1)T$	V,M	AND
Badal (type II)	$n-1$	$(n-1)T$	V,M	AND
2. Edge-Chasing Algorithms				
Mitchell-Merritt (no priority)	$m(n-1)/2$	$(n-1)T/2$	C,S	Single-Resouce
Mitchell-Merritt (priority)	$m(n-1)$	$(n-1)T$	C,S	Single-Resouce
3. Diffusing Computation				
Chandy-Misra-Haas	W: $2m(n-1)$	$2dT$	C,S	OR
Hermann-Chandy	W: $2m^2(n-1)$	$2dT$	V,M	AND-OR
4. Global State Detection				
Bracha-Toueg	$4m(N-1)$	$4dT$	V,M	C(n,k)

N: number of sites; n: number of sites involved in deadlock;
m: number of tasks involved in deadlock; T: inter-site communication delay;
p: the period between two consequent GRG updates; d: diameter of GRG.

Message Sizes: V: variable; C: constant; L: large; M: medium; S: small.

B: best case; W: worst case.

Notes for Badal Algorithm:
type I: resource's intention lock can be determined before task migration;
type II: resource's intention lock can be determined after task migration;
for type I: $\mathcal{M} = \sum_{k=1}^{n}(n-k)$.

CHAPTER 4

A METHODOLOGY FOR THE DEVELOPMENT OF DISTRIBUTED DEADLOCK DETECTION ALGORITHMS

Deadlock detection requires knowledge of the global state of a system. In a distributed system the control mechanisms are not centralized and, hence, its global state is a collection of locally recorded system states. Also, typically, a distributed environment lacks a common physical global clock and the message delays are unpredictable. in such a distributed environment, to synchronously collect all the local states in the construction of a global system state is rather difficult. A good distributed deadlock detection algorithm should (1) confine its knowledge about the global system state to local views that any individual site can see and (2) use logical time instead of a real common global clock.

Until recently, little has been done on the subject of characterization and correctness of distributed deadlock detection. The lack of the understanding of the synchronization problems in a distributed environment (i.e., no physical common global clock and no centralized storage for system states) may account for many errors found in the literature. Both Kshemkalyani and Singhal [83] and Tay and Loke [131] have realized the problem and developed theories and solutions[1]. Their works are based on the Single-Resource and the AND deadlock models. The graph structure of the deadlocks considered in their theories and solutions restricts the application of their results to be directly used in problems such as OR deadlock detection. Although generalizations of these results can be made as claimed in the reports, they are not straightforward.

In this chapter, we develop a methodology for the design and verification of distributed deadlock detection algorithms which observes the above limitations.

[1]Although the motivations of these two related works are the same, they were performed independently and have taken different approaches.

The fundamental idea of this methodology is twofold. First, the principles of the detection of a certain deadlock structure (e.g., a cycle or a knot) can be developed by studying its graphical properties in static graphs and, hence, can be separated from the synchronization problems encountered in distributed systems. Second, the synchronization problems can be resolved by the stable property of deadlocks. The main focus of this chapter is the latter, i.e., to develop criteria to deal with the synchronization problems encountered in distributed environments.

This chapter is organized as follows. In Section 4.1, we first summarize characteristics and assumptions of deadlock detection in distributed environments. An overview of the methodology is given in Section 4.2. Issues concerning the design of distributed algorithms in static systems are discussed in Section 4.3. In Section 4.4, the deadlock detection computation in dynamic systems is characterized and a systematic method for the derivation of a dynamic algorithm from a static one is developed. Finally, concluding remarks are given in Section 4.5.

4.1 Characterizations and Assumptions of Distributed Deadlock Detection

A typical model for distributed systems is a network of sites connected by communication channels. There may be well synchronized system clocks (especially in real-time applications) at all sites, but no assumed physical common global clock. It is also generally assumed that there is no shared memory among the sites in a distributed system. Message passing is a commonly assumed mechanism for the data exchange and synchronization among different sites across the network. As surveyed in Chapter 3, the most efficient way for detecting deadlocks in such a distributed system is based on probe messages.

In Section 2.1, we introduced three types of wait-for graphs, i.e., TWFG (Task Wait-For Graph), TRG (Task-Resource Graph), and GRG (General Resource Graph), to represent the underlying system states for the purpose of deadlock detection. In a TWFG, only the task-task wait-for relations are depicted. Its

structure differs from a real system state in that the resources are omitted in the graph where resource competitions are the causes of the wait-for dependencies. A TWFG is usually maintained in a separate data structure where synchronizations between the graph and the underlying system activities are needed. Some of the proposed TWFG-based algorithms may be prone to error in real applications if this one extra level of synchronization is not considered carefully. On the other hand, the structure of a TRG can closely reflect a real system state when resources are involved in the wait-for relations. Usually, TRG's are used in database systems to deal with resource deadlocks where inter-task communications/synchronizations are not considered part of the deadlock problem (e.g., they may be solved by some sort of deadlock-free approaches). Unless it can be proven that the wait-for dependencies due to inter-task communications/synchronizations and those due to resource competitions are mutually exclusive, a cyclic wait may contain both types of wait-for relations. If such a deadlock occurs, both the communication sub-system and the resource manager may fail to recognize it. To avoid problems stated above, we choose GRG's to represent the underlying system states where both resource and communication deadlocks are modeled by the unified "general resources" (Section 2.4.2).

A GRG is a data structure which records the inter-dependent relations among tasks and resources in a system. As there is usually no centralized control mechanism in a distributed system, without loss of generality, we assume that a (global) GRG is a collection of local GRG's which are physically recorded at different sites across the network in the system. Since the structure of a GRG resembles a system state being recorded in the task and resource control tables, it can be integrated in a system with little extra overhead. In a running system, a GRG is dynamically updated to closely reflect the changes of the underlying system state. Such a dynamically changing GRG is called a DGRG (Dynamic GRG, to be defined later).

Since in a DGRG both task and resource vertices are included in a graph, for easy treatment in the algorithms, their states are defined uniformly as follows. In a dynamic system the DGRG is updated dynamically while the deadlock detection

computations (DDC's) are running. In such a DGRG, a vertex is in the "waiting" state if and only if there are outgoing edges; otherwise, it is "free." In a DGRG, a nonempty set of waiting vertices S are deadlocked if the reachable set of every $v \in S$ is a nonempty subset of S.

There are different types of vertices in a system and the concepts of "waiting" and "free" have different meanings when applied to a specific type of vertex.

Task Vertices: A task vertex is waiting if it is blocked (idle) and is waiting for its outstanding requests to be granted. A task vertex is free if it is actively running. In some systems, a task can be simultaneously waiting for requests to be granted as well as performing internal computations. Depending on the semantics of the task, such a task can be divided into two sub-tasks (or two threads), one is in waiting state and the other one is free, if the requests need to be granted in finite time; otherwise, the task can simply be treated as in a free state.

Resource Vertices: There are two kinds of resource vertices:

Consumable Resources: Consumable resources are used to model communication and synchronization requests among tasks. A consumable resource does not exist in a system until it is requested by a task. A consumable resource vertex starts waiting for a set of "producer" tasks to satisfy the request when it is initiated by a "consumer" task which issues the request. When one of the producers satisfies the request of a consumable resource, the resource becomes free and is "consumed" by its consumer.

Reusable Resources: In contrast to consumable resources, reusable resources are never created or deleted in a system. To determine the state of a reusable resource depends on the way the resource is managed in a system. A reusable resource may be shared by many tasks or it may only be exclusively assigned to a single task. Also, in the case of exclusive assignment, a reusable resource may have a fixed number of units and

each of them may only be exclusively allocated to a task. Let's first consider a reusable resource which can be shared by many tasks. Such a resource is free as long as no task is waiting for using it exclusively or is holding it exclusively; otherwise, the resource is waiting to be released. Suppose a resource r is shared by a nonempty set of tasks $\{v_i\}$. Although r is allocated to some tasks it is free until a task u wants to use it exclusively (i.e., u starts to wait for r and r starts to wait for all its holders $\{v_i\}$ to release it). After all the holders $\{v_i\}$ release r, r is then exclusively assigned to u and is waiting for u to release it. Next case to consider is exclusive assignments. Suppose a reusable resource has a fixed number of units for exclusive assignments. The resource is free if there are available units; otherwise, the resource is waiting for its holders to release any of its units. If such a resource contains only one unit, it starts waiting whenever it is allocated to a task.

An edge $\langle u \rightarrow v \rangle$ may be (1) a request edge, i.e., a task u has an outstanding request to a resource v or it is waiting for a message/synchronization from some other tasks (e.g., in Ada, task u is waiting for some other tasks in a rendezvous v), (2) an assignment edge, i.e., one instance of a resource u is assigned to a task v, or (3) a producer edge, i.e, u is a message request or a synchronization which may be fulfilled by a task v. For brevity, we say that the edge $\langle u \rightarrow v \rangle$ represents the situation in which u is "waiting" for v.

The notation $\langle \vec{u} \rightarrow v \rangle$ denotes that the edge $\langle u \rightarrow v \rangle$ is locally recorded at the vertex u. Similarly, $\langle u \rightarrow \vec{v} \rangle$ denotes that the edge $\langle u \rightarrow v \rangle$ is locally recorded at the vertex v.

When a vertex u starts to "wait" for a vertex v, an edge $\langle \vec{u} \rightarrow v \rangle$ is recorded at u. In a finite delay, the vertex v will learn that the vertex u is "waiting" for it and an edge $\langle u \rightarrow \vec{v} \rangle$ is recorded at v. The edge $\langle u \rightarrow v \rangle$ may be deleted if u and/or v initiates an edge deletion process. The initiator u (or v) of the edge deletion process removes the edge $\langle \vec{u} \rightarrow v \rangle$ (or $\langle u \rightarrow \vec{v} \rangle$) from its local records. When the

other end v (or u) knows that the edge $\langle u \rightarrow v \rangle$ is being deleted, it removes the record $\langle u \rightarrow \vec{v} \rangle$ (or $\langle \vec{u} \rightarrow v \rangle$) accordingly.

Edge records may be added to and deleted from a free vertex. Once a vertex becomes waiting, no more outgoing edges will be added to it. Also, a waiting vertex does not *initiate* the deletion of any of its incoming and outgoing edges. A waiting vertex u will become free if

- the requests for which it is waiting is fulfilled. If u is a task, it may proceed its computation after it becomes free. If u is a resource, it is available for allocation when it is free.
- it is selected as a victim to resolve a detected deadlock. Only tasks may be considered as the victims of the deadlock resolutions. When task u is chosen as a victim, it becomes free and starts to carry out the resolution by relinquishing its outstanding requests and releasing its holding resources.
- its waiting state terminates voluntarily. A vertex may terminate its waiting state (1) due to the program or system faults or (2) upon the expiration of a pre-programmed duration. The termination of a waiting state due to program or system faults is unpredictable and is assumed to rarely happen. On the other hand, the termination of a waiting state due to the expiration of a pre-programmed duration is very common in certain systems. For example, in real-time applications, tasks may leave waiting states according to their timing constraints. In such systems, the timing constraints can provide enough information to predict when a waiting state will terminate.

In this chapter, we are dealing with Stable deadlocks. We assume that vertices do not terminate their waiting states voluntarily and, hence, the deadlocks are stable until they are detected and resolved. This assumption is relaxed when the algorithms are extended for real-time applications. We ignore the voluntary termination of a waiting state due to program or system faults because such situations are rare to occur and they may only cause a system to recover from a false deadlock which will not damage the system severely.

Based on the above descriptions, a DGRG can be defined as follows.

Definition 4.1 (Dynamic System State GRG) A DGRG records the global state for a system dynamically. A DGRG contains vertices to denote tasks and resources in the system and edges to represent wait-for dependency among the tasks and resources. Each vertex in a DGRG is in one of the two states: free or waiting. Also, each vertex has a table of incoming and outgoing edges. A vertex is in the waiting state if and only if it has outgoing edges; otherwise, it is free. In a distributed system, each site maintains a local DGRG by keeping track of the state and the edge records for each of its local vertices. The global DGRG of such a distributed system is a collection of the local DGRG's recorded at every participating site.

A deadlock detection algorithm is based on the information available in such a distributed DGRG. The algorithm initiates computations to construct (but once constructed) static views of a DGRG and searches deadlock structures in the constructed views. In other words, a DGRG can be viewed as a shared data structure between the underlying system activities and the DDC's. The correctness of DDC's relies on the correct synchronization of accessing the DGRG.

Without losing generality, we assume that the synchronization of accessing the DGRG among the system activities and the DDC's is achieved through message passing[2]. Messages used in a system are of two types: (1) *control messages* for updating the structure of the DGRG (as described earlier) in a distributed environment and (2) *probe messages* for the DDC's. Suppose there are two vertices u and v in a DGRG which represent a task and a resource in a distributed system. When u propagates a probe message to v along $\langle u \rightarrow v \rangle$, this implies that the DDC is transferred[3] from a task (resource) represented by u to a resource (task) represented by v. The synchronization among system control messages and the

[2]In real implementations, for efficiency reasons, a local message may be realized as a direct modification to the DGRG.

[3]Likewise, in real implementations this may be done by directly updating the probe information in the local DGRG if both u and v are at the same site.

deadlock detection probes are based on the following assumptions which are typical in many message passing systems.

Assumption 4.1 A sequence of messages sent out from a vertex will be received and processed in the same order at the destination vertex.

Assumption 4.2 Messages may be processed in parallel at different vertices.

Assumption 4.3 If multiple messages are received and queued at a vertex, they will be processed one by one. In other words, there is no parallelism assumed within any vertex.

Assumption 4.4 The message propagation delay from a vertex to any of its neighbors is finite.

Assumption 4.5 The underlying communication subsystem is assumed to be reliable that no message will be lost or duplicated.

In the following chapters, we develop deadlock detection algorithms in a series of refinement steps. First, we assume the GRG is static when a DDC is running on it so that we can concentrate on the properties and principles of deadlock detection computations without worrying about the dynamic nature of the graph. Actually, many centralized deadlock detection algorithms are such static algorithms. Then, the static algorithm[4] is extended to deal with DGRG using the methodology developed in this chapter. And, finally, timing constraints are considered so that the algorithms may be used in real-time applications.

A static system GRG implies the absence of control messages in the system. Such a GRG can be created by either "freezing" the system activities or taking a "snapshot" [21] of the system state and then processing it "off-line." It is much easier to analyze and validate a DDC on a static GRG than on a DGRG. Basically, combined with protocols that either "freeze system" or "take snapshots," a static

[4]In the following discussions, an algorithm or a DDC is said to be static (or dynamic) if it is associated with a static (or a dynamic) system state GRG.

algorithm can be used in dynamic systems. However, there may exist better synchronization mechanisms for extending a static algorithm to a dynamic one. One of the goals of this chapter is to develop a systematic method of finding such a synchronization mechanism for static algorithms. Using the criteria suggested in our methodology, we may then develop dynamic algorithms from static algorithms and verify their correctness.

As a final step, we take into account the timing constraints in many real-time systems. When a task is waiting for a resource or is holding a resource, a corresponding edge is present in the DGRG. This waiting or holding situation may be terminated due to timing constraints associated with the task. We assume this timing information is available in the system so that we can predict how long a task could stay in a blocked waiting state and how long a task could be holding a resource. Therefore, each edge in a DGRG may have a lifetime deadline which reflects the timing constraints of its end vertices. With this available timing information in DGRG, we may then extend algorithms to real-time applications. Since these real-time extensions are specific to individual algorithms, they will be discussed in the next two chapters where algorithms are presented.

4.2 Overview of the Methodology

A deadlock detection algorithm is correct if and only if the following "progress" and "safety" concerns (Section 2.6.2) are satisfied:

Safety : A detected deadlock indicates an existing real deadlock in the underlying system.

Progress : An existing deadlock will be detected in finite time by some DDC.

In a dynamic system, a DDC constructs its own view of a GRG from the underlying DGRG. A deadlock is declared if a DDC finds a deadlock structure in its constructed GRG. The first concern requires that an algorithm should be able to (1) recognize a deadlock structure in a graph (i.e., in its constructed view of GRG),

and (2) assure that the detected deadlock structure does indicate a real deadlock in the underlying system (i.e., could be found in the DGRG). Therefore, we may consider an algorithm to consist of two parts: (a) a "quasi-static" procedure that recognizes the structure of deadlocks in GRG's and (b) a synchronization mechanism that deals with the dynamically changing nature of the underlying system. We call part (a) a "quasi-static" procedure because such a procedure searches for deadlock structures in a dynamically constructed (but once constructed) static view of the DGRG. In terms of finding deadlock structures in a graph it functions like a static algorithm. The idea of using a static algorithm in the design of a dynamic algorithm is that the procedure specified in part (a) could be similar to and, hence, could be based on a static algorithm. In the worst case (in terms of performance), the synchronization mechanism can be a "freezing" or a "snapshot taking" protocol and the so captured GRG's are then analyzed by a static algorithm.

In addition to the first concern, the second concern requires that an algorithm should be able to assure at least one DDC which satisfies the Safety concern is guaranteed to proceed to declare an existing deadlock. Usually a very loose condition is adopted to cope with the second concern; that is, a deadlock can be at least detected by a "latest" DDC which is usually initiated after the deadlock is formed. One of the optimizations frequently found in the literature is to reduce the number of DDC's detecting a deadlock to be one. Most of these optimizations rely on the "latest" DDC initiated "after" a deadlock is formed to be the one which leads to the declaration of the deadlock. In Section 4.4.2.3, a "tighter condition" under which an existing deadlock can be detected by a DDC (may be initiated "before"[5] the formation of the deadlock) is proposed with proof. This "tighter condition" may be used to further improve an algorithm to efficiently detect deadlocks *as early as possible.*

Based on the above analysis, the correctness criteria for a deadlock detection algorithm could be rephrased as follows:

[5]Since there is usually no common global clock in a distributed system, the concepts "latest," "after," and "before" are based on logical timestamps.

Recognition of deadlock structure : A DDC should be able to correctly recognize a deadlock structure in a graph (i.e., in a constructed view of a DGRG), and

Synchronization with the underlying system : A synchronization mechanism should fulfill the following two concerns:

1. Safety concern: The algorithm should be able to assure that a detected deadlock structure does indicate an existing deadlock state in the underlying system, and
2. Progress concern: The algorithm should be able to assure that at least one DDC which satisfies the Safety claim is guaranteed to proceed to declare an existing deadlock.

As mentioned earlier, to recognize a deadlock structure in a graph is a "quasi-static" procedure in a dynamic algorithm. How to fulfill the first criterion, therefore, can be studied in static algorithms. Issues concerning the design of such a static algorithm is discussed in Section 4.3.

To synchronize a DDC and the underlying system activities, we need to properly model their interactions. First, in Section 4.4.1 we adopt a global view in modeling the DDC and the underlying system (in terms of DGRG). According to this global view model, two consistency criteria (spatial and temporal) are proposed to verify if a DDC can be correctly executing in a DGRG. These consistency criteria are used to judge if a constructed view of a DGRG is "meaningful." We use the concept of meaningfulness to indicate if a constructed view of a DGRG can faithfully reflect the real situation of the underlying system state. An interesting result in meeting the meaningfulness requirement is that a dynamic DDC can be "projected" onto a static GRG and is equivalent to a "static" computation. This also implies that a dynamic algorithm can be derived from a static one.

In Section 4.4.2, we then discuss how the meaningfulness of a DDC can be maintained according to the information which is local to each of the DDC's participating vertices. The notion of meaningfulness and the deadlock's stable property

also led to the derivation of conditions for a synchronization mechanism to satisfy the Safety and the Progress concerns. These conditions only rely on information that is local to each vertex. Consequently, these conditions provide a feasible way of realizing a synchronization mechanism.

From the above descriptions, the methodology can be used to derive a dynamic algorithm from a static one. This methodology is also useful in the verification of the correctness of dynamic deadlock detection algorithms. A summary of these two usages can be found in Section 4.4.2.2.

4.3 Distributed Deadlock Detection in Static Systems

A static algorithm describes deadlock detection computations executing on a static GRG. Since it is generally assumed that there are no centralized control mechanisms and no common shared storage in a distributed system, the global system state GRG is a collection of local GRG's recorded at different sites across the network. The algorithms considered in such distributed systems are based on the probe computations.

The derivation of deadlock detection algorithms are usually based on the way the deadlock problems are defined. There are several guidelines for the deadlock definitions so that they can properly lead to the derivation of probe-based algorithms for distributed applications. First, the definition should be confined to the information which can be easily gathered at each vertex in the system. Such a definition is called a "local view definition" (Section 4.3.1). Second, the definition should be based on a well defined graphic structure (e.g., a cycle or a knot) which satisfies both the necessary and the sufficient conditions of the deadlocks in a system. Such a set of deadlocked vertices is called a "deadlock core set" (Section 4.3.2). Finally, in Section 4.3.3, we formally model a probe-based DDC in a static GRG.

4.3.1 Local View Definition for Distributed Deadlock Detection

The conventional definitions for deadlock structures are usually derived from the concepts of the system states. The main concern of such definitions is whether a system is in a deadlock state. The definition usually requires a global view of the whole system state GRG. For example, a deadlock cycle may be defined as a group of waiting vertices which form a path in a GRG with the same *start* and *end* vertex (Section 2.2). Such a definition suggests an algorithm to collect the whole system GRG in order to find cycles in it. However, this is not appropriate for the deadlock detection in distributed environments. To remedy the problem, we need to use a "local view definition" to describe the deadlock situations in a distributed system.

In a "local view definition" a deadlock is defined in a form that any single vertex can "see" it. This definition consists of (1) functions which can be invoked at any vertex and (2) deadlock conditions which are based on the functions' return values. Any function used in a local view definition must return the necessary value to the vertex where the function is invoked. For example, we may re-define the above cycle deadlock as: a vertex v is in a cycle if and only if $v \in RS(v)$ (Section 2.2). The reachable set function $RS(v)$ can be realized by means of forward probe propagations. When a vertex v wants to know if it is involved in a cycle, a probe is initiated and propagated along the edges in a GRG to search $RS(v)$. Since the size of a GRG is finite and the message delay is finite (Assumption 4.4), the probe will eventually reach every member of the reachable set $RS(v)$. The deadlock condition is defined as $v \in RS(v)$. The probe may come back to v if and only if $v \in RS(v)$. Therefore, a cycle is detected if the initiator v receives its probe back; otherwise, this probe computation will terminate normally with no cycle detected. There is no global system state GRG need to be constructed in this local view definition and it is relatively easy (compared to the first cycle definition) to be realized in the distributed environments.

4.3.2 Deadlock Core Set

In the literature, deadlock is sometimes defined in such a general way that there is no information concerning the structure of the deadlocks. For instance, many algorithms are based on the concept of a *deadlock set.* A typical definition of a *deadlock set* is a nonempty set of blocked vertices S where the reachable set of every $v \in S$ is a nonempty subset of S. One of the major problems with this definition is that even if such a deadlock set can be identified, it is difficult to have a proper resolution to break it due to the lack of deadlock structure information. For example, a deadlock set is shown in Figure 4.1. This deadlock situation may be resolved only if both cycles A and B are identified and broken. Without knowing the two cycle structures in the deadlock set, the deadlock may not be resolved properly.

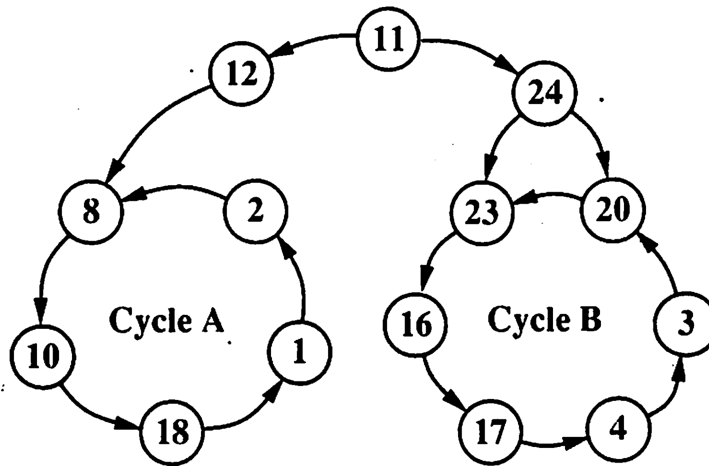


Figure 4.1: Example of a deadlock set

Instead of detecting deadlock sets in a system, we need to identify and resolve *deadlock core sets.* A deadlock core set satisfies both necessary and sufficient conditions for the existence of deadlocks in a certain system. Usually a deadlock core set has a well defined graphic structure such as cycle and knot. As discussed in Chapter 2, cycles are the deadlock core sets in the Single-Resource and the AND model deadlock systems whereas knots are the deadlock core sets for the OR model systems. A deadlock set is only a sufficient condition for deadlocks. Many

non-core vertices in a deadlock set, such as vertices 11, 12, and 24 in Figure 4.1, are transitively waiting for some other vertices in the deadlock core sets. A deadlock state may persist without these non-core vertices in a deadlock set. Consequently, it is important that an algorithm should be able to identify deadlock core sets. For brevity, we use deadlock to denote the deadlock core set in the following discussions.

4.3.3 Probe-Based Static Deadlock Detection Computation

The algorithms discussed in this chapter are based on the probe computations. The local view definition requires that a deadlock is always declared by a DDC at the vertex where the computation was initiated. Without lost of generality, we further require that a DDC which leads to the declaration of a deadlock be initiated at a core vertex in the detected deadlock. Although this is not a necessary requirement in terms of the correctness, this requirement eliminates many complicated situations and, hence, simplifies the development and verification of an algorithm. This requirement can always be achieved because (1) the initiator of a DDC is known and (2) a deadlock core set is the target to be identified. If a DDC finds that its initiator is not in the detected deadlock core set, it terminates without declaring the deadlock. The efficiency may be improved if a mechanism is developed to eliminate DDC's which are initiated at non-core vertices earlier.

A probe-based static DDC can be pictured as a set of operations executed on a static GRG. A DDC starts from an initiation operation when a vertex invokes a deadlock detection algorithm. The initiation operation then leads to a set of subsequent operations to carry out the computation defined by the algorithm. Each operation has a condition to trigger its execution. When an operation is complete, its result is in effect (i.e., lead to the subsequent operations or return values to the initiator vertex). An operation can be formally defined as follows.

Definition 4.2 (Deadlock Detection Operation) Let OP_k be an operation in a DDC which is performed by the execution of a deadlock detection algorithm. The operation OP_k consists of three steps: (1) the evaluation of a

trigger predicate tr_k, (2) the execution of the operation procedure op_k, and (3) the result predicate re_k. If the operation op_k is executed, tr_k should have held prior to the beginning of op_k, and re_k should hold upon the termination of the operation.

For example, consider an operation OP_k which forwards a probe from u to v. A predicate tr_k asserts that the probe being forwarded must be at vertex $u \in GRG$ and $\langle u \rightarrow v \rangle \in GRG$, then the procedure op_k can be performed. Within finite time, upon the termination of op_k, re_k asserts that the probe must be received at vertex v.

A DDC is a computation that uses probes to search and explore vertices and edges in a GRG. A probe may come back to the initiator to report evidence of the existence of cyclic waiting. Each operation (except for the initiation operation) in a DDC is responsible for exploring one edge in a GRG. The result of an operation may "lead to" the triggering of another operation (or a set of operations). The notation "$\leadsto$"(reads "leads to") is used to describe such a precedence relation between two predicates. For example, $re_k \leadsto tr_l$ means that the result of an operation OP_k leads to the triggering of another operation OP_l. A static DDC, therefore, can be modeled as a partially ordered sequence of operations as follows.

Definition 4.3 (Static DDC) A Deadlock Detection Computation generated by a static algorithm is a partially ordered sequence of operations $\{OP_k | k = 0, 1, \ldots, n\}$ where OP_0 is the initialization operation and OP_n is one of the most recent operations of the computation. Also, $\forall$: $1 < l < n$, $\exists$: $k < l$ such that $re_k \leadsto tr_l$.

The structure of such a DDC is a tree of operations starting from the root OP_0 (i.e., the initialization operation of the DDC). The result predicate re_k of an operation OP_k may lead to a set of trigger predicates $\{tr_l\}$ which, in turn, trigger a set of operations $\{OP_l\}$. The set $\{OP_l\}$ may be empty if the operation OP_k reaches the initiator of the DDC or leads to no valid trigger predicates (e.g., the probe reaches a leaf vertex and finds no path to continue its propagation). Operations

which repeatedly propagate a probe in the same cycle should be properly avoided. Since the size of a GRG is finite, within finite time the DDC should terminate (when the DDC tree cannot grow).

The Safety concern is easy to be verified if a static algorithm may only generate such well defined DDC's. To verify the Progress concern is subject to each individual algorithm. However, comparing to a dynamic algorithm, it is relatively easy to analyze and verify a static algorithm.

The tree structure of a DDC represents a partial ordering which reflects the inter-dependencies among operations. The time order in which a DDC's operations are actually executed is not important. The wait-for dependencies in the underlying system are reflected in the structure of a DDC. Therefore, the equivalence of two DDC's can be defined as follows.

Definition 4.4 (Equivalence of DDC's) Two DDC's are said to be equivalent if they both apply to the same GRG and form the same tree structure.

In Sectionder-ver-dyn-algo, we describe how the concept of the DDC equivalence can be used in the derivation of a dynamic algorithm from a static one.

4.4 Distributed Deadlock Detection in Dynamic Systems

In this section, we model DDC's in dynamic systems and develop a systematic method for the derivation of dynamic algorithms from static ones. For easy understanding, in Section 4.4.1, a global view is assumed. Then in Section 4.4.2, a practical approach is developed based on the information which is local to each vertex.

4.4.1 From Global View

Suppose we have a global monitor in a distributed system which is capable of keeping track of the global system state. With this global monitor, we can verify an algorithm which is executing in a DGRG as we have characterized in Section 4.1.

The easiest way of applying a static algorithm to a dynamic system is to "freeze" the system state or to take a "snapshot" off the system [21] and then apply this static algorithm in the "frozen" state or in the "snapshot" GRG. However, we believe that there exist a more efficient mechanism to synchronize a DDC and the underlying dynamically changing system. In the following subsections, the snapshot approach is first summarized and based on that our synchronization mechanism approach is then developed.

4.4.1.1 The Snapshot Approach

To detect deadlocks in a distributed system, we may first "freeze" the system to determine a global system state and then apply a static algorithm on the frozen global system state. However, in a distributed system, it is very difficult and costly to simultaneously freeze the whole system to get a global state. Instead, Chandy and Lamport [21] suggest taking a consistent "snapshot" of the system state. A "snapshot" in a distributed system is a *consistent view* of a global system state which can be constructed dynamically without "freezing" the underlying system computations (see also Section 3.2.4).

A vertex (with the help of the underlying system) can keep track of its own state and the messages it sends and receives. To determine a global system state, a vertex v must inform all the connected (through all possible communication/synchronization channels) vertices to record their own states and the states of the communication channels they are directly connected to. The global snapshot is the collection of all the locally recorded states.

Each vertex state change or each message send/receive is assumed to be an atomic event. There are many types of system activities in a dynamic system. For example, task (or resource) vertices may be created, deleted, migrated to another place, split, or joined together, and communication channels may be created or removed. For the purpose of deadlock detection, we are mostly interested in the events that are related to the DGRG. These events include: (1) the state of a vertex is changed from (or to) a waiting state to (or from) a free state, (2) the receiving of a message which causes the creation or the deletion of an edge record, and (3) the sending of a message which is due to the creation or the deletion of an edge record.

Due to the lack of a global clock and a common storage, the snapshot taking procedure can only be carried out asynchronously at each vertex across the network, and the recorded system state is required to have no message/synchronization in transition and the events recorded in a snapshot must satisfy the causal relation [83, 85] (i.e., a logical order, Lamport first put it as "happened before" partial ordering [85]). For example, if the deletion of an edge record $\langle u \rightarrow \vec{v} \rangle$ is happened before the creation of another edge record $\langle \vec{v} \rightarrow w \rangle$, a snapshot which implies the coexistence of the two edges $\langle u \rightarrow \vec{v} \rangle$ and $\langle \vec{v} \rightarrow w \rangle$ is inconsistent. The causal relation can be formally defined as follows.

Definition 4.5 (Event Causal Relation) The relation "$\prec$" on a set of events in a system is the smallest transitive relation satisfying the following conditions: (1) If a and b are events that happened in the same vertex (task or resource), and a comes before b, then $a \prec b$. (2) If a is the initiation of a message/synchronization and b is the corresponding event (receive the message or response to the synchronization), then $a \prec b$. The relation "$\prec$" is transitive and two events a and b are said to be *concurrent* if $a \not\prec b$ and $b \not\prec a$.

Chandy and Lamport have suggested that such a global state snapshot is "meaningful" in a sense that it could be used to solve problems with *stable property*. For example, each of the consistent snapshots can be processed "off-line" by a static deadlock detection algorithm. If a deadlock appears in a snapshot, it indeed exists in the underlying system. Also, if a deadlock is formed in the underlying system,

it persists and can be found in a global system snapshot later. Therefore, if an algorithm checks system snapshots periodically (or whenever there is evidence of the existence of deadlocks), an existing deadlock will be detected eventually.

Chandy and Lamport suggested that a special message, called a *marker*, being sent to record the consistent states (see Section 3.2.4). However, this snapshot taking algorithm makes no assumptions about the states it collects. For deadlock detection, we may restrict a snapshot taking to collect information that is required for the construction of a GRG. The types of events which need to be recorded to construct GRG's differ from system to system. However, it is not difficult to identify these necessary events.

Usually, an edge in a recorded GRG implies a sequence of events. For example, in a system, an edge recorded in a GRG may have gone through the following events: (1) vertex u sends a request to vertex $v \neq u$ and records $\langle \vec{u} \rightarrow v \rangle$, (2) v receives the request but does not grant it immediately, (3) v then sends u a request pending message and records $\langle u \rightarrow \vec{v} \rangle$, and, finally (4) u receives the request pending message from v and the edge $\langle \vec{u} \rightarrow v \rangle$ is confirmed. Between the events (1) and (2), the state of the edge $\langle u \rightarrow v \rangle$ is inconsistently recorded at the vertices u and v. Only after the event (4) completes, the edge $\langle u \rightarrow v \rangle$ is confirmed at its both ends, thereby, can be consistently recorded in a GRG. Therefore, in addition to the requirement that no message/synchronization could be in transition in a snapshot, a consistent GRG further requires that no edge creation/deletion could be in transition. A consistent GRG can be defined as follows.

Definition 4.6 (Consistent GRG) In a *Consistent* GRG, (1) the edges are consistently recorded at its two end vertices and (2) the events which form the GRG (e.g., the vertex state changes and the creation and deletion of the edge records) must satisfy the causal relation.

The marker algorithm does not directly generate a consistent GRG. Since the creation or deletion of an edge is composed of a sequence of events, a snapshot may

inconsistently record edges which are being created or deleted. A consistent GRG can be constructed from a snapshot using one of the following approaches.

1. All the edges which are in the transition of being created or deleted are eliminated in the construction of a consistent GRG. This is a rather simple approach. The edges which are being deleted will eventually disappear from the system. The edges which are being created will hopefully appear in a consistent GRG in the future. Bracha and Toueg [14] have used and proven this approach in their deadlock detection algorithm for the C(n,k) model. This approach has been regarded as incorrect by Knapp [77] because certain deadlocks may be ignored in a consistent GRG. However, this is not a severe problem if an algorithm can guarantee that the ignored deadlock will appear and be detected in a future consistent GRG.

2. All the edges which are in the transition of being deleted are eliminated in the construction of a consistent GRG. All the edges which are in the transition of being created are recorded after their existence are confirmed. This approach, although manifestly correct, requires a complicated extension to the original marker algorithm to confirm and record the forming edges.

3. All the edges which are in the transition of being deleted are eliminated in the construction of a consistent GRG. All the edges which are in the transition of being created are assumed to exist in the constructed consistent GRG. In Section 4.4.2.1, we will prove that the forming edges can be counted as the stable edges if they are found in a deadlock structure and will not cause the detection of false deadlocks. Therefore, this approach is appropriate for the detection of deadlock structures in a constructed consistent GRG.

To take a snapshot off a system requires the marker messages to be propagated through every possible communication/synchronization channels to inform every vertex to record its local states. However, in most of the cases, the wait-for edges in a constructed consistent GRG may only be mapped to a small portion of the communication channels in the system where marker messages are propagated.

Consequently, the process of taking snapshots incurs large unnecessary overheads to the system.

4.4.1.2 The Synchronization Mechanism Approach

Instead of performing deadlock detection in the GRG's which are built on top of the snapshots, a deadlock detection algorithm which is equipped with a properly designed synchronization mechanism can be directly applied to a DGRG. Suppose we have a deadlock detection algorithm which is applied to a DGRG thus generating a dynamic DDC. The only difference between a dynamic DDC and a static one (characterized in Section 4.1) is that the former is performed in a dynamically changing GRG. In this section, we analyze dynamic DDC's and suggest consistency criteria for the development of synchronization mechanisms.

Again, probes are used in a dynamic DDC to search and explore vertices and edges in a dynamic system. From time to time the probes are searching at different places in the system to explore the bridging edges (i.e., dependency relations) among vertices to build a view of a consistent system state in terms of a static GRG. Each operation in a DDC is responsible for exploring one edge in a DGRG. A probe may come back to the initiator to report evidence of the existence of a cyclic waiting. However, the information gathered may include an unstable part of the DGRG and, hence, the constructed local view of DGRG may be inconsistent. Consequently, a declared deadlock may not exist or an existing deadlock may never be declared.

Suppose in a system, a snapshot is *quickly* taken immediately after each state change and after the evaluation of each predicate of a DDC. The snapshot taking procedure is invoked at the place where the state change occurs or where the predicate is evaluated. The snapshot taking procedure is performed so quickly that no further system state changes or predicate evaluations can be made before the completion of the snapshot taking. To facilitate the analysis and verification of a dynamic DDC, the GRG constructed by the DDC is examined in these snapshots. A subscript is used to specify the situation a snapshot is taken. For example,

the notation ST_k^{tr} denotes a snapshot which is taken after the evaluation of the trigger predicate tr_k of an operation OP_k. Also, $tr_k(ST_k^{re})$ means that the trigger predicate tr_k is examined (i.e., re-evaluated) in the snapshot ST_k^{re}. Since global knowledge is assumed in the analysis, the time orders $\prec$ and $\succ$, i.e., happened before and happened after, respectively, between two snapshots are assumed to be known. Superscripts are used to number a sequence of snapshots in time order, e.g., $ST^1 \prec ST^2 \prec \cdots \prec ST^n$.

In Definition 4.6, a consistent GRG built on top of a snapshot requires that

- no edge creation/deletion is in transition. This requirement is "spatial" in the sense that it specifies the consistency within a snapshot where a GRG is constructed.
- the events in terms of the vertex state changes and the creation and deletion of the edge records must satisfy the causal relation. This requirement is "temporal" because it specifies the consistency regarding the temporal logic of a snapshot taking procedure.

Similarly, the sources of inconsistency in a dynamic DDC can be classified into two types: "spatial" and "temporal." A spatial inconsistency is an intra-snapshot inconsistency while a temporal inconsistency is an inter-snapshot inconsistency.

Specifically, a spatial inconsistency results when an edge in a DGRG is temporarily recorded at its two ends inconsistently. Again, without global knowledge of a distributed system, the predicates of the operations are usually evaluated according to local information at a certain vertex. If a predicate is evaluated to be true while the corresponding edge is in a transition state, it must hold after the transition state; otherwise, the condition that the predicate asserts may never exist. For instance, when the probe arrives at vertex u, the edge record $\langle u \rightarrow \vec{v} \rangle$ is being deleted by v. Before the vertex u notices that the edge is being deleted, u may have triggered an operation OP_k to send a probe through a vanishing edge $\langle u \rightarrow v \rangle$. Such a situation is allowed in a snapshot but should be avoided in the construction of a consistent GRG.

A temporal inconsistency, on the other hand, results when a view of DGRG is inconsistently constructed based on the information collected across different time slots. In a DDC, a view of a DGRG is constructed at the initiator based on the edges explored over a period of time when its probes were traveling in the system. If some of the probes ever reached the unstable part of the system, the explored edges may exist at different time slots. When putting two never coexisting edges together, the constructed view of DGRG may inconsistently reflect the dependency relations among the involved vertices. For example, the path $\langle d \rightarrow u \rightarrow v \rangle$ exists in a snapshot ST^1. A probe initiated at d reaches v and the snapshot $ST^2 \succ ST^1$ is taken. The edge $\langle \vec{u} \rightarrow v \rangle$ is then removed by u in a snapshot $ST^3 \succ ST^2$ (e.g., due to u's request being satisfied by another vertex). The edge $\langle \vec{v} \rightarrow d \rangle$ is added by v in a snapshot $ST^4 \succ ST^3$. Eventually, the probe may come back to the initiator d and a non-existing cycle $\langle d \rightarrow u \rightarrow v \rightarrow d \rangle$ may be reported. The temporal consistency requires that although the cycle $\langle d \rightarrow u \rightarrow v \rightarrow d \rangle$ combines information from different snapshots, all the edges must coexist in a single snapshot.

To deal with spatial inconsistency, an operation in a DDC must satisfy the following Spatial Consistency criterion.

Criterion 4.1 (Spatial Consistency) *An operation is spatially consistent if and only if both its trigger and its result predicates are evaluated valid in a snapshot.*

In a DGRG, although the wait-for dependency between two vertices may come and go many times, each incarnation is treated as a different edge. When an operation explores an edge, it is limited to one incarnation of the wait-for dependency between the two vertices. When a GRG is constructed from the information provided by a set of operations, all the edges must coexist in a certain snapshot; otherwise, the constructed GRG is temporally inconsistent. An operation is meaningful in a snapshot in the sense that it reflects a real edge in the underlying system DGRG. The meaningfulness of the operations is defined as follows.

Definition 4.7 (Meaningful Operation) An operation is meaningful in a snapshot if and only if the edge it explored exists in the snapshot.

Obviously, a meaningful operation must be spatially consistent in the snapshot. Based on meaningful operations, Temporal Consistency can be defined as follows.

Criterion 4.2 (Temporal Consistency) *A temporally consistent GRG is constructed by a set of (spatially consistent) meaningful operations in a snapshot.*

At this point, it is clear that the Spatial Consistency criterion is a necessary condition for the Temporal Consistency criterion. If a DDC satisfies the Temporal Consistency criterion, it satisfies the Spatial Consistency criterion as well. The most important thing is to verify the meaningfulness of the operations when a GRG is constructed.

In a dynamic DDC, although an operation is performed over a period of time and across two connected vertices, the evaluation of its predicates can be assumed to be atomic. To analyze the temporal consistency problem, the DDC modeled in Section 4.3.3 is expanded to a tree of predicates and a snapshot is associated with every predicate. The time order of the snapshots is the same as their associated predicates. The "previous" snapshot means the snapshot which is associated with the parent predicate in the DDC tree. To verify the meaningfulness of the operations in a DDC can be broken down to the verification of the meaningfulness of the predicates as follows.

1. For the root predicate tr_0 of a DDC tree:

   (a) tr_0 is evaluated valid and meaningful in the initial snapshot ST_0^{tr}; and

   (b) tr_0 is meaningful in a snapshot $ST^i \succ ST_0^{tr}$ if and only if it was meaningful in the previous snapshot ST^{i-1} and can be re-evaluated valid in the current snapshot ST^i.

2. For a non-root predicate p_k in a DDC tree which is first evaluated valid in a snapshot ST_k^p:

(a) p_k is meaningful in the snapshot ST_k^p if its parent predicate is meaningful in ST_k^p; and

(b) p_k is meaningful in a snapshot $ST^i \succ ST_k^p$ if and only if it was meaningful in the previous snapshot ST^{i-1} and can be re-evaluated valid in the current snapshot ST^i.

An operation OP_k is meaningful in a snapshot ST^i if both its trigger predicate tr_k and result predicate re_k are meaningful in the snapshot ST^i. In the following discussions, an operation is said to be meaningful at time t means that it is examined meaningful in a snapshot taken at time t.

Since the evaluation of a predicate is timely based on the evaluation of its parent predicate, we cannot simply re-evaluate a predicate in a later snapshot independently. The re-evaluation of a predicate has to involve the re-evaluation of its parent predicate. Consequently, the re-evaluation of a predicate requires a "replay"[6] of the DDC from the root predicate tr_0. And a non-root predicate is re-evaluated meaningful in a snapshot implies all its ancestor predicates in the same DDC tree are meaningful in that specific snapshot.

A synchronization mechanism should be able to distinguish the meaningful operations from the meaningless ones when a view of the underlying DGRG is constructed. In Section 4.3, we characterize a DDC as a computation which is initiated at a vertex and all the necessary value are returned to the initiator. Although a GRG is not necessary to be physically built at the initiator of a DDC, the returned information reflects certain structure of the DGRG where the probes have searched. It is equivalent to say a GRG is built at the initiator and the same graph structure is found. For example, when a probe is received at its initiator u, u is in a cycle. We may ask the probe to report the path it has traveled and make the same conclusion. When the initiator of a DDC tries to put together a GRG to determine the existence of a deadlock structure, it takes a final system snapshot ST^f. The

[6]Replay is for reasoning only, algorithms will be shown to automatically satisfy the requirements without expensive replay.

synchronization mechanism should guarantee that all the operations which may affect the construction of the GRG are meaningful in the final snapshot ST^f. A practical approach for the development of the synchronization mechanisms is given in Section 4.4.2.

To summarize, each of the operations in a DDC is spatially consistent if it satisfies the Spatial Consistency criterion. When a spatially consistent operation explores an edge, the edge may exist at the time it is explored. In addition to the requirements for meeting the Spatial Consistency criterion, the Temporal Consistency criterion further asserts that when a DDC constructs a view of a DGRG only the results of its meaningful operations may be in effect. The meaningless operations which do not show up in the replay of a DDC should be ignored when constructing a view of a DGRG. Consequently, a DDC which satisfies both Spatial Consistency and Temporal Consistency criteria can be "projected" to any static snapshot of a DGRG by "replaying" its meaningful operations in that specific snapshot. The "execution" of a correct dynamic DDC up to a certain time when a snapshot ST^f is taken is equivalent to the "replay" of a "static version" of the DDC in ST^f which consists of all the meaningful operations in the ST^f. To be more specific, suppose a dynamic DDC, say $DDDC_A$, can be projected to a snapshot ST^f as a static DDC, say $SDDC_A$. $DDDC_A$ must satisfy both Spatial and Temporal consistency criteria. Obviously, $DDDC_A$ may declare a deadlock if and only if its projected $SDDC_A$ declares it.

One property of a projected static DDC is that it preserves inter-dependency of the meaningful operations in its corresponding dynamic DDC. The preserved inter-dependency of the meaningful operations may correctly reflect certain graph structures in a final snapshot where a GRG is constructed. However, such a projected static DDC may not be equivalent to a real static DDC which is directly generated in that final system state. For example, an algorithm (e.g., Algorithm 6.3 in Section 6.3.1) requires that forward probes be merged together when received at a free vertex. They may be propagated as a single probe if the vertex becomes waiting later. In a projected $SDDC_A$, we may observe merge operations in just the same

way as they occurred in the $DDDC_A$. Suppose the whole system becomes static in ST^f and a static DDC, say $SDDC_B$, is directly performed in that snapshot. It is impossible to find merged probes propagating in $SDDC_B$ because there is no vertex that could become waiting from free state. Both $SDDC_A$ and $SDDC_B$ may reflect the same graph structure in the underlying system but they are not equivalent in terms of their DDC structure. This example shows that the "re-evaluation" or the "replay" of $DDDC_A$ has to take into account that the underlying DGRG is changing. A "re-evaluation" or a "replay" does not mean that a real static $SDDC_B$ is directly "executed" in the final state.

4.4.2 From Local View

So far we have developed the notion of meaningfulness for the operations in a DDC from a global view. To examine the meaningfulness of the operations in a DDC requires the knowledge of the history (i.e., be able to replay a DDC) of the DDC. In this section, we develop conditions for the Safety and the Progress concerns, which are based on the meaningfulness of the operations and the stable property of the deadlocks. To examine these conditions only requires information that is local to each vertex.

4.4.2.1 Safety Concern of Dynamic Algorithms

The meaningfulness of a DDC is sufficient to satisfy the Safety concern. As long as the operations are meaningful, no false deadlocks may be declared. The meaningfulness as defined requires a knowledge of the whole computation (we need to "replay" the whole computation to decide if an operation is meaningful). In this section we develop a local method to judge if an operation in a detected deadlock structure is meaningful and, hence, could be used to resolve the Safety concern of the DDC.

When searching a graph for deadlock structures, we are looking for the existence of the edges and the relationship among them. Without a global view,

at each vertex, only the connectivity between pairs of adjacent operations might be known. Two operations OP_j and OP_k are said to be adjacent if and only if $re_j(ST_j^{re}) \rightsquigarrow tr_k(ST_k^{tr})$ at a vertex v. The connectivity of these two adjacent operations relies on both the result predicate re_j and the trigger predicate tr_k being evaluated to be valid simultaneously (or in a snapshot) at v. At least, if $re_j(ST_k^{tr})$ is valid, i.e., the result predicate re_j remains valid when the trigger predicate tr_k is evaluated, the two adjacent operations OP_j and OP_k are connected at v.

Definition 4.8 (Connectivity of Adjacent Operations) Two adjacent operations OP_j and OP_k are connected at a vertex v if and only if (1) $re_j(ST_j^{re}) \rightsquigarrow tr_k(ST_k^{tr})$ at v and (2) $re_j(ST_k^{tr})$ is valid.

A set of operations are meaningful in a snapshot ST^f, if they satisfy both Spatial and Temporal consistency criteria. For Spatial consistency, a vertex can only examine the validity of an operation's trigger or result predicate which is locally evaluated at the vertex. For Temporal consistency, a vertex can only evaluate the connectivity of the adjacent operations. Now the question is how the Safety claim can be made with these limitations in the system.

First, let's paint each edge in a DGRG with colors to reflect its status. We borrow Chandy and Misra's method [23] to paint the edges as follows. An edge $\langle u \rightarrow v \rangle$ is painted

gray if u has sent a request message to v to add the edge $\langle \vec{u} \rightarrow v \rangle$ in the DGRG but v has not yet received the message;

black if v has received a request message from u and the request is not yet granted by v (i.e., the edge $\langle u \rightarrow \vec{v} \rangle$ has been added to DGRG);

white if v (or u) has deleted the edge $\langle u \rightarrow \vec{v} \rangle$ (or $\langle \vec{u} \rightarrow v \rangle$) and is sending a message to u (or v) to remove the edge $\langle \vec{u} \rightarrow v \rangle$ (or $\langle u \rightarrow \vec{v} \rangle$) from DGRG but u (or v) has not yet received it.

Both gray and black edges are called "dark edges." If a deadlock contains only dark edges, it is a "dark deadlock."

Chandy and Misra [23] used colored edges to address a similar problem in dynamic systems and proved that "dark" cycles (i.e., the AND model deadlocks) detected by their proposed algorithm are the genuine deadlocks. Similar techniques are used in the literature (e.g., the AND deadlock detection in [27] which is derived from [23], the AND-OR deadlock detection in [63], and the C(n,k) deadlock detection in [14]) to prove the safety concern of the proposed algorithms, i.e, false detection of deadlocks are eliminated.

The colored edges describe the dynamically changing nature of the underlying system. The purpose of a dynamic DDC is to construct a view of such a dynamic system (in terms of a constructed GRG). Since each operation in a DDC is related to an edge in the underlying DGRG, a color may be assigned to the operation to reflect the status of the edge when it is detected. A constructed GRG, therefore, could be described by the colored operations/edges. An operation is "black" if it is performed over a black edge. An operation is "gray" if it discovers a gray edge that is forming. An operation is "white" if it explores a vanishing white edge.

Suppose in the underlying system, an edge $\langle u \rightarrow v \rangle$ is formed and then deleted as follows:

1. at time t_1, u sends a request message to v,
2. at time t_2, v sends a request message to w,
3. at time t_3, v receives u's request message,
4. at time t_4, v receives w's grant message.
5. at time t_5, v sends a grant message to u, and
6. at time t_6, u receives v's grant message.

The edge $\langle u \rightarrow v \rangle$ is gray between t_1 and t_2, black between t_2 and t_5, and white between t_5 and t_6. Also, suppose an operation OP_k is performed to explore the edge $\langle u \rightarrow v \rangle$. *The operation OP_k is "black" if it is performed over a black edge*, i.e., $t_2 \preceq ST_k^{tr} \preceq t_4$ and $t_3 \preceq ST_k^{re} \preceq t_4$. *The operation OP_k is "gray" if it discovers a forming gray edge*, i.e., $t_1 \preceq ST_k^{tr} \preceq t_3$ and $t_3 \preceq ST_k^{re} \preceq t_4$. *The operation OP_k is "white" if it*

explores a vanishing white edge, i.e., $t_1 \preceq ST_k^{tr} \preceq t_6$ and $t_5 \preceq ST_k^{re}$. Please note that, an operation may be incomplete if its result predicate has never been evaluated. Once an operation's result predicate is evaluated, its color is determined. As indicated in this example, both black and gray operations may disappear later but not whitening.

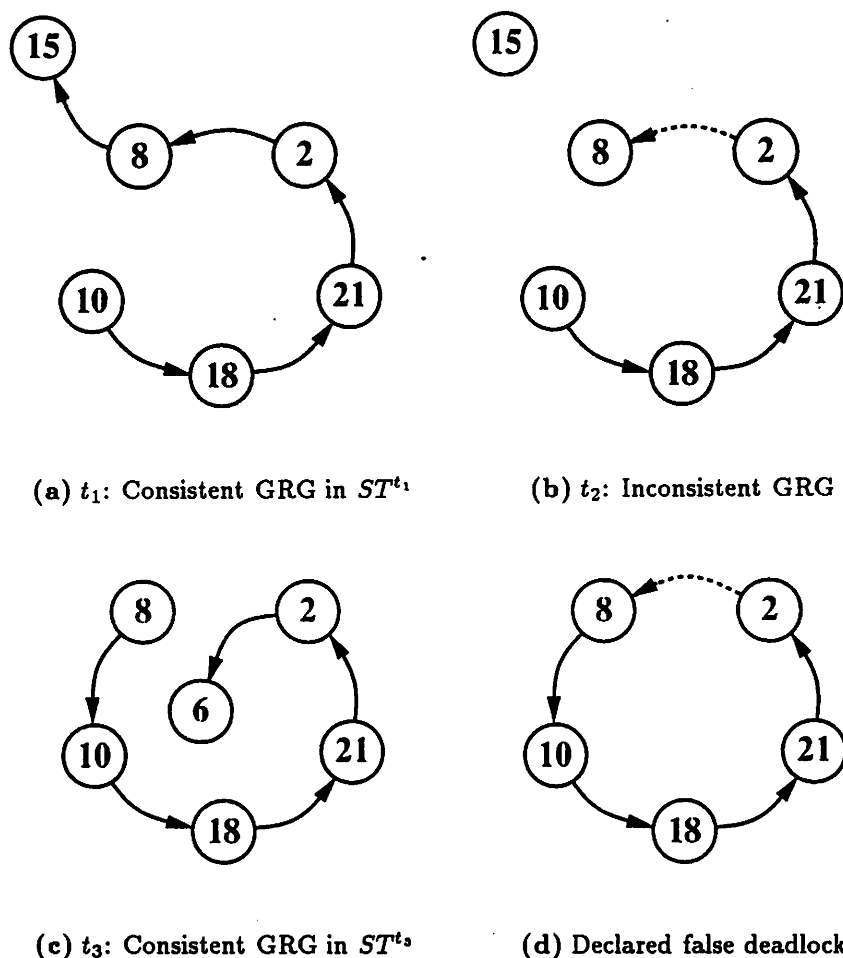
The above example shows how the color of an operation indicates the condition when it explores an edge in a dynamic system. When a view of a DGRG is constructed, an color is assigned to each of the edges in the constructed GRG. We use the same color for the edge in a constructed GRG and the operation which explores it.

Figure 4.2 shows how the inconsistencies may cause the failure of an algorithm. In this example, suppose a simple cycle detection algorithm is used to detect deadlocks in a dynamic system and the inconsistencies which might happen in dynamic systems are not considered carefully. The claimed deadlock in a constructed GRG (i.e., Figure 4.2-(d)) is a false one because it contains a vanishing white edge and three dark edges that are not meaningful in a final snapshot.

The difference between a gray operation and a black operation basically depends on the time when the operation is triggered. If an operation is triggered before the edge is known at the other end, e.g., $t_1 \preceq ST_k^{tr} \preceq t_3$ in the above example, it is a gray operation. If an operation is triggered when the edge is known by its both ends, it is a black edge. It is difficult to distinguish a gray operation from a black operation when a vertex evaluates a result predicate. On the other hand, a white operation can always be identified by checking the existence of the edge from which the probe of the operation was received. In a constructed GRG, the following lemma proves that with respect to Spatial Consistency, we only need to deal with white edges.

Lemma 4.1 If white edges are eliminated in a constructed GRG, all the operations satisfy Spatial Consistency criterion.

Proof: In a constructed GRG, black edges represent black operations which satisfy the Spatial Consistency criterion. Both gray and white edges represent



(a) t_1: Consistent GRG in ST^{t_1}

(b) t_2: Inconsistent GRG

(c) t_3: Consistent GRG in ST^{t_3}

(d) Declared false deadlock

Figure 4.2: Example of the inconsistencies in a DGRG.
(a) At time t_1, a system state is consistently recorded as ST^{t_1}. In ST^{t_1}, 21 initiates a DDC and its probe has reached 2. **(b)** At time $t_2 \succ t_1$, the system state is inconsistently recorded. Between t_1 and t_2, 15 has granted 8's request and deleted the edges $\langle 8 \rightarrow 15 \rangle$. At time t_2, 8 is deleting the edge $\langle 2 \rightarrow \vec{8} \rangle$ by granting 2's request. Almost simultaneously, 2 is sending out a probe along the white edge $\langle 2 \rightarrow 8 \rangle$ without knowing that 8's grant message is coming the other way. **(c)** At time $t_3 \succ t_2$, a system state is consistently recorded as ST^{t_3}. In ST^{t_3}, 8 starts to wait for 10 and the probe from 2 is received. The edge $\langle \vec{8} \rightarrow 10 \rangle$ may be a gray one. Also, after receiving 8's grant message, 2 starts to wait for 6. **(d)** Eventually, the probe may come back to 21 and a false deadlock (with a deleted white edge $\langle 2 \rightarrow 8 \rangle$ and meaningless dark edges $\langle 8 \rightarrow 10 \rangle$, $\langle 10 \rightarrow 18 \rangle$, and $\langle 18 \rightarrow 21 \rangle$) may be declared.

operations involving transient inconsistent states but only white edges are the violators of Spatial Consistency criterion. This is because a gray edge is one which is forming. Both trigger and result predicates of a gray operation will be valid in a snapshot taken upon the completion of the operation. On the other hand, a white operation explores a vanishing edge and both its predicates will become invalid eventually. Consequently, all the dark operations in the constructed GRG satisfy Spatial Consistency criterion. ■

In searching for a deadlock structure in a graph, an algorithm should be able to correctly identify the existence of edges and the connectivity between each pair of adjacent edges. According to Lemma 4.1, the dark edges (i.e., black and gray operations) may be treated as real edges (i.e., meaningful operations) when constructing a view of a DGRG. In the following lemma, we further prove that Temporal Consistency criterion can be met if (1) a constructed deadlock structure is a dark one and (2) the connectivity between adjacent operations are well maintained at each vertex in the deadlock.

<u>**Lemma 4.2**</u> In a DDC, if the connectivity between adjacent operations are well maintained, the operations (edges) in a constructed dark deadlock fulfill the Temporal Consistency criterion.

Proof: Suppose a dark deadlock D is constructed by a DDC, by Lemma 4.1, each operations in D satisfies the Spatial Consistency criterion.

In the underlying DGRG, a gray edge should have turned black when its corresponding operation terminates. Therefore, when a dark deadlock D is constructed by the DDC, all the detected edges in the underlying DGRG have turned black, i.e., consistently recorded at its two ends.

When two adjacent operations are connected at a vertex v, one of them must be an outgoing edge. Therefore, v must be waiting when two operations are connected through it. The state of the vertex v and the edges which connect through it should remain unchanged until the DDC constructs a GRG which contains the deadlock D. This stable property should be true for

all the vertices and the edges discovered by the operations in the constructed dark deadlock D. This property can be proven by contradiction as follows.

Suppose an edge $\langle u \rightarrow v \rangle \in D$ is removed after it is explored by an operation OP_k. This means that eventually, u will become free and delete $\langle \vec{u} \rightarrow v \rangle$ from its records. Since u was waiting when the tr_k was evaluated, to become free u must be able to reach a free vertex which is not in the deadlock set D. It is contradictory to the definition of deadlock that the reachable set of every vertex $u \in D$ is a nonempty *subset* of D. Therefore, D is not a deadlock; otherwise, $\langle u \rightarrow v \rangle \notin D$.

All the operations in D are stable from the time they are performed until a GRG which contains the deadlock D is constructed. Consequently, we can conclude that all the operations in D fulfill the Temporal Consistency criterion. ■

By Lemmas 4.1 and 4.2, we can conclude that

Theorem 4.1 In a DDC, if the connectivity between adjacent operations are well maintained, the dark deadlock structure in the constructed GRG is a genuine one.

Proof: By Lemmas 4.1 and 4.2, the operations in such a constructed dark deadlock fulfills both Spatial and Temporal consistency criteria. All operations in D are meaningful when D is detected and, hence, do indicate the existing edges in the underlying DGRG. Consequently, the detected deadlock D does indicate an existing deadlock in the underlying system. ■

4.4.2.2 Derivation and Verification of Dynamic Algorithms

The methodology developed in this chapter can be applied in two ways. First, a dynamic algorithm can be derived from an existing static algorithm based on this methodology. The derived dynamic algorithm must satisfy consistency criteria. Improvements may then be made on the derived dynamic algorithm. Second, the

theorems derived in this chapter could be used to verify a dynamic algorithm. For example, to verify the above improved dynamic algorithm, we may need to prove that the algorithm satisfies consistency criteria and all its projected static DDC's can be simulated by another correct static algorithm. In this section, the techniques for the derivation and the verification of dynamic algorithms are discussed.

First, suppose we have an algorithm, say $ALGO_S$, which is used in static systems and we want to develop another algorithm, say $ALGO_D$, to perform equivalent functions, e.g., the detection of cycles/knots, in dynamic systems. $ALGO_D$ could be derived from $ALGO_S$ with Spatial and Temporal consistency concerns. The Spatial and Temporal consistency criteria are used to deal with the dynamically changing nature of the underlying system. Furthermore, $ALGO_D$ is derived from $ALGO_S$ such that each projected static DDC[7] generated by $ALGO_D$ should be equivalent to a DDC due to $ALGO_S$ (see Definition 4.4 for the equivalence of DDC's). Since $ALGO_S$ can correctly detect deadlocks in any static GRG, so can the projected DDC's due to the derived $ALGO_D$.

As another example, suppose we have a dynamic algorithm $ALGO_D$ which is to be verified. We could relate $ALGO_D$ to some static algorithm, say $ALGO'_S$, as follows. First, $ALGO_D$ must be proven to satisfy both Spatial and Temporal consistency criteria which, in turn, means every DDC it generates could be projected to a static DDC. Such a projected DDC is equivalent to an execution of $ALGO_D$. Suppose a static algorithm $ALGO'_S$ is derived to simulate $ALGO_D$'s projected DDC's in any static state (i.e., to generate equivalent static DDC's). If $ALGO'_S$ is proven to have the property that each of its DDC's can correctly recognize a deadlock structure, the same property also holds for the projected DDC's by $ALGO_D$. Consequently, $ALGO_D$ can be proven correct if it satisfies both Spatial and Temporal consistency criteria and can only generate projected DDC's which are equivalent to the executions of the correctly proven static algorithm $ALGO'_S$.

[7]Here, the discussion is focused on any single DDC, i.e., except for necessary interventions from the DGRG, no other DDC's may affect the execution of a certain DDC.

Theoretically, there exists at least one $ALGO'_S$ that randomly generates any possible DDC pattern (i.e., propagate probes randomly) in a static state including all the possible projected DDC's due to $ALGO_D$. However, such an $ALGO'_S$ is not useful in that it does not have the property we need to verify $ALGO_D$. The purpose of executing $ALGO_D$ is to correctly detect deadlock structures in the DGRG. Its net effect (i.e., a projected DDC in a static GRG) should be equivalent to the detection of the same deadlock by a static algorithm $ALGO'_S$. The goal of finding an $ALGO'_S$ is, therefore, twofold; i.e., (1) to simulate $ALGO_D$'s projected DDC's in static states and (2) to preserve the properties of $ALGO_D$ which could then be verified through $ALGO_S$. In general, such a $ALGO'_S$ can be constructed systematically as follows by taking the advantage of that $ALGO_D$ must satisfy both Spatial and Temporal consistency criteria.

A dynamic algorithm specifies how the operations in a computation are invoked in distributed environments. An operation may be (1) triggered by the state changes of the underlying system (e.g., when a vertex becomes waiting it may initiate a DDC), (2) triggered by some other operations (e.g., a probe may continue its propagation when received by a waiting vertex), or (3) triggered by mixed situations of (1) and (2) (e.g., a previously received probe may continue its propagation when a vertex becomes waiting). A dynamic DDC contains all three types of operation triggering while a static DDC (i.e., by executing an algorithm in a static system state) has no operations due to state changes. A projected DDC is a static version of a dynamic DDC in which all three types of operation triggering may be found. As we have discussed in Section 4.4.1.2, this is a major difference between a "replayed" projected DDC and a "real" static DDC.

A projected DDC contains meaningful operations which were triggered by the underlying system state changes and/or some preceding meaningful operations. There are two types of system state changes that may affect a dynamic DDC, i.e., (1) when a vertex becomes waiting and (2) when a vertex becomes free. The second type of state changes may only cause the operations to become meaningless. This type of state change can be ignored since it does not have any explicit

effect that could be found in a projected DDC. On the other hand, the first type of state changes may have effect on the projected DDC. A meaningful operation requires that its triggering vertex, once becoming waiting, should stay in the waiting state until a GRG is finally constructed. In order to "tightly simulate" projected DDC's in static states (i.e., generating equivalent static DDC's while preserving the properties of $ALGO_D$), $ALGO'_S$ should be able to simulate any operation triggering situations that may be found in the projected DDC's. Consequently, $ALGO'_S$ should be able to simulate every meaningful operation that is triggered either by preceding operations or when a vertex becomes waiting or a mixture of both.

Suppose a set of rules is used in $ALGO_D$ to specify how to trigger operations in different situations. The rules can be classified into three types: (1) rules that are used to specify how to trigger operations from preceding operations, (2) rules that are used to specify how to trigger operations when a vertex becomes waiting, and (3) rules that are used to specify how to trigger operations when a vertex becomes free. These rules may be applied individually or mixed in $ALGO_D$. The meaningless operations are ignored in the projected DDC's, therefore, the third group of rules can be ignored when constructing $ALGO'_S$. The inter-dependency between any two meaningful operations are specified by the first group of rules. The inter-dependency is preserved in the projected DDC's and can be simulated by directly applying group (1) rules in $ALGO'_S$. The situations that vertices may change into waiting state do not happen in static states. However, these situations could be simulated in $ALGO'_S$ as follows. Each of the vertices in a static state is arbitrarily assigned a finite delay time. When a static algorithm is applied to a static state, a timer is started. Upon the expiration of the assigned delay time, a vertex may then initiate new DDC's and/or participate in the propagations of the received probes as that may happen in a dynamic system when the vertex becomes waiting.

Consistency criteria may help to preserve the properties of $ALGO_D$ as much as possible in the process of constructing $ALGO'_S$. It is up to the specific $ALGO_D$

and $ALGO'_S$ pair to determine if the latter can be used to verify the former. We will demonstrate this technique in the development of Algorithm 6.3.

4.4.2.3 Progress Concern of Dynamic Algorithms

The progress concern of a dynamic algorithm requires that an existing deadlock is guaranteed to be detected by some DDC's in finite time. To make such a claim for the developed algorithm, we need to find out the circumstances under which an existing deadlock will be detected by a DDC in finite time. In the following discussions, a "tight condition" under which an existing deadlock may be detected by a certain ongoing DDC is proposed and proven based on the consistency criteria we have developed.

Whether (1) a dynamic algorithm $ALGO_D$ is derived from a static algorithm $ALGO_S$ with consistency concerns or (2) $ALGO_S$ is constructed from $ALGO_D$ to verify the latter, both algorithms should handle probe propagations in the same way in a *real* static GRG. By *real* static GRG we mean that there are no state changes in the underlying DGRG for $ALGO_D$ and the simulation of state changes in $ALGO_S$ is turned off by setting the delay time to zero for all vertices. So far we have only discussed the "DDC structure" equivalence of the two algorithms; i.e., every projected DDC due to $ALGO_D$ is equivalent to a DDC performed by $ALGO_S$. Another important characteristic of the two algorithms is that they tend to continue their DDC's with equivalent operations if they are not complete in a certain system state. This "progress" equivalence of the DDC's from the two algorithms is discussed in details in the following.

Suppose at time t_i a global system state can be determined by a snapshot ST^i quickly taken at t_i. For brevity, we say it is the system state ST^i. Suppose $ALGO_D$ generates a DDC, say $DDDC_D(ST^i, ST^f)$, in a DGRG from an initial system state ST^i until the final current state ST^f. The meaningful operations of $DDDC_D(ST^i, ST^f)$ can be projected to ST^f to form a static DDC $SDDC_D(ST^f)$.

The $SDDC_D(ST^f)$ should be equivalent to a DDC performed by $ALGO_S$ in the static state ST^f, say $SDDC_S(ST^f)$.

In a static state, a DDC is said to be "complete" if no more operations may be performed; otherwise, it is said to be "incomplete" and may continue new operations in that state until complete or state change. If in a final state ST^f, $SDDC_D(ST^f)$ is complete, so is its equivalent $SDDC_S(ST^f)$. Both $SDDC_D(ST^f)$ and $SDDC_S(ST^f)$ may become incomplete (they may also become meaningless) in a new state ST^{f+1} and continue equivalent operations until complete or next state change. The continued equivalent operations are performed on a "real" static state and the implication behind this equivalence is twofold: (1) whatever operations appended to $SDDC_D(ST^f)$ due to $ALGO_D$, equivalent operations should also be performed by $ALGO_S$ and (2) whatever operations $ALGO_S$ may continue on $SDDC_S(ST^f)$ can be used to predict the growth trend of $SDDC_D(ST^f)$ due to $ALGO_D$ in that static state.

Suppose an incomplete $SDDC_S(ST^f)$ may continue its execution of $ALGO_S$ until complete in ST^f. Depending on the characteristic of $ALGO_S$, we may get a set of possible complete DDC's $\{CSDDC_S(ST^f)\}$. That is, if $ALGO_S$ is a verified algorithm such that it can correctly detect an existing deadlock in finite time, the whole set of DDC's $\{CSDDC_S(ST^f)\}$ will agree on whether there is a deadlock is detected. In other words, if one of $\{CSDDC_S(ST^f)\}$ may detect a deadlock, all of them will declare the same deadlock; otherwise, none of $\{CSDDC_S(ST^f)\}$ may declare any deadlock. This property is proven in the next lemma (Lemma 4.3).

Lemma 4.3 Suppose $ALGO_S$ is a correct static deadlock detection algorithm. Starting from an incomplete DDC $SDDC_S(ST^f)$, all the possible complete DDC's $\{CSDDC_S(ST^f)\}$ due to $ALGO_S$ will declare the same deadlock, if any, or none of them may declare any deadlock.

Proof: Suppose a DDC is initiated at a vertex d and will terminate at d if a deadlock is found. The minimum possible incomplete DDC is the initiation

operation at d and, from the initiator d, a universal set of all possible complete DDC's can be derived in ST^f. A $\{CSDDC_S(ST^f)\}$ derived from an incomplete $SDDC_S(ST^f)$ is a subset of this universal complete DDC set. The lemma could, therefore, be proven if all possible complete DDC's starting from a vertex will agree on whether a deadlock is detected. That is if d is in a deadlock D, any possible DDC initiated at d will lead to the declaration of the deadlock D; otherwise, none of them will detect any deadlock.

Suppose $ALGO_S$ is a verified static algorithm such that it could correctly detect an existing deadlock in a state ST^f in finite time. There is no repeated invocation of $ALGO_S$ at the same initiator d in the same state ST^f which, in turn, means although there may be more than one possible DDC's starting from d, at most one of them may be executed. Suppose d is in a deadlock D but not every possible DDC initiated at d will lead to the declaration of the deadlock D. Vertex d may fail to declare the deadlock D if the executed DDC does not find it. Since this kind of failure may occur at every vertex in D, there is no guarantee that deadlock D may be detected in finite time. This result contradicts the progress claim of $ALGO_S$ that an existing deadlock can be detected in finite time. Therefore, in order to support the progress claim of $ALGO_S$, it is required that every possible DDC starting from an initiator d should lead to the declaration of a deadlock D if $d \in D$. Also, if d is not in any deadlock, none of its possible complete DDC could find a deadlock. This is because an existing deadlock may only be declared by its member vertices (Section 4.3.3). Consequently, the lemma is proven because all the possible DDC's initiated at a vertex will agree on the same result, so do a subset of those DDC's. ■

As discussed earlier a complete DDC due to $ALGO_S$ in a state ST^f can be used to predict the growth trend of $SDDC_D(ST^f)$ due to $ALGO_D$. This results in the following theorem.

Theorem 4.2 Suppose there is a deadlock $D \in ST^f$ and a dynamic $DDDC_D(ST^i, ST^f)$ due to $ALGO_D$ is initiated in a state ST^i no latter than the current state ST^f. In state ST^f, the projected $SDDC_D(ST^f)$ is an incomplete DDC and is equivalent to $SDDC_S(ST^f)$ which is also an incomplete DDC performed by $ALGO_S$. Deadlock D could eventually be detected by $DDDC_D$ if D could be declared by a complete DDC which is continued from $SDDC_S(ST^f)$ due to the execution of $ALGO_S$ in the state ST^f.

Proof: In the system state ST^f, if $SDDC_S(ST^f)$ can lead to the detection of a deadlock D, it must be initiated at a vertex $d \in D$ and it can eventually detect the deadlock D by means of its operations in D. Also, a DDC can detect at most one deadlock, we can treat each of the deadlocks in ST^f independently and ignore the meaningful operations taking place outside of any deadlock.

In a dynamic system, a deadlock is a stable property such that it will not change once it is formed, i.e., $D \in ST^g$, $\forall ST^g \succ ST^f$, since $D \in ST^f$. Therefore, we can ignore meaningful operations of $DDDC_D$ executed outside of D and treat D as a separate *real* static subgraph after ST^f. In other words, within the scope of D, both $SDDC_D(ST^f)$ and $SDDC_S(ST^f)$ could continue equivalent operations regardless the state changes outside of the deadlock D. If $SDDC_S(ST^f)$ may lead to the declaration of the deadlock D, by Lemma 4.3, every possible complete DDC derived from $SDDC_S(ST^f)$ within the scope of D will agree on that result. Since based on $ALGO_D$, $DDDC_D$ will only develop equivalent DDC's as those derived from $SDDC_S(ST^f)$ due to $ALGO_S$, eventually, $DDDC_D$ will terminate and declare the deadlock D just the same way as $SDDC_S(ST^f)$ does. ■

4.5 Concluding Remarks

In this chapter, a methodology for the design and evaluation of distributed deadlock detection algorithms is developed. The correctness concerns of the algorithm lie on two issues: first, if it can correctly recognize deadlock structures and second, if it can properly synchronize with the underlying dynamically changing system. The Safety and the Progress concerns are addressed in each of the two issues. The reorganized correctness concerns suggest that the principles of the detection of deadlock structure can be developed by studying its graphical properties in static graphs and, hence, can be separated from the synchronization problems encountered in the distributed systems. The main focus of this chapter is the development of synchronization mechanisms for the dynamic algorithms.

A dynamic DDC requires that all of its operations must be meaningful when a view of the underlying DGRG is constructed. The Spatial and the Temporal consistency criteria are used to satisfy this requirement. The meaningful operations of a dynamic DDC can be "projected" to a static system state GRG, which can then be simulated by a static algorithm. Both algorithms share some common properties (e.g., using the same set of principles for the detection of a certain deadlock structure in GRG's) such that the dynamic algorithm is correct if its corresponding static algorithm can be proven correct.

Theorem 4.1 suggests that to construct a meaningful view of a deadlock in a dynamic system, it is required that (1) the connectivity between adjacent operations are well maintained and (2) the detection of white edges are avoided. Both requirements can be realized using information local to each vertex. This theorem points out a feasible way to satisfy the Safety concern.

Theorem 4.2 shows when an existing deadlock can be detected by a certain deadlock detection computation. When a deadlock is formed in a dynamic system, Theorem 4.2 proves that the meaningful operations of a DDC initiated in that deadlock can detect the deadlock in finite time provided that the DDC is allowed to continue successfully. This is a relatively tight condition in comparison with the

condition usually found in the literature that a deadlock could at least be detected by a "latest" DDC (usually means a DDC initiated after the formation of the deadlock). Many algorithms (include the algorithms proposed in this dissertation) are optimized such that each deadlock can be declared exactly once. This optimization introduces competitions among DDC's. When we develop DDC competition rules for an algorithm, this tight condition could help us to push the algorithm to allow a deadlock to be detected as early as possible.

Based on our methodology, distributed deadlock detection algorithms can be developed in refinement steps as follows.

1. Static Algorithm: Develop principles for the deadlock detection in a static graph.
2. Dynamic Algorithm:
   - Analyze and avoid the situations where white edges may be detected.
   - Analyze and maintain the connectivity between adjacent probe operations.
   - Examine the situations when a new computation should be initiated to ensure the Progress concern.
3. Real-Time Applications:
   - Associate a deadline to each of the edges.
   - Analyze the meaningfulness of a probe concerning the deadlines associated with the edges.
4. Maintain the principles developed in the first step throughout the rest of the design procedure.

In the following chapters we derive dynamic deadlock detection algorithms based on the static ones which have been proven correct. The derived dynamic algorithm needs to fulfill both Spatial and Temporal consistency criteria when constructing a view of a deadlock. Theorem 4.1 suggests that this can be done at each

vertex locally by eliminating the detection of white edges as well as maintaining the connectivity between any pair of adjacent operations. Following this theorem, rules are designed to handle dynamic situations. In addition, Theorem 4.2 suggests that meaningful DDC's should be preserved as much as possible under these dynamic system rules. This is not only to guarantee that an existing deadlock can at least be detected once (i.e., the progress claim of the correctness concern) but also to push the declaration of an existing deadlock to be as early as possible. Theorem 4.1 also guarantees that such a derived dynamic algorithm may only declare genuine deadlocks (i.e., the Safety claim of the correctness concern). Improvements may then be made to the derived dynamic algorithm. To verify the new improved algorithm, it could be related to some static algorithm as described in Section 4.4.2.2. Since both algorithms share certain common properties, the improved dynamic algorithm can be verified by proving that the corresponding static algorithm can correctly detect deadlocks in static GRG's.

CHAPTER 5

CYCLE DETECTION

Cycle detection is the basis for the detection of the Single-Resource and the AND deadlocks. In this chapter we develop two algorithms: one for the detection of Single-Resource deadlocks (to be referred to as the Single-Resource Algorithm) and the other one for the detection of AND deadlocks (to be referred to as the AND Algorithm).

In these two algorithms optimizations are made due to efficiency concerns. The optimizations can reduce the probe overhead in terms of reducing the number of probe messages passed around as well as eliminating the possibility of repeated detection of a cycle (which means single point of detection for each deadlock). The deadlock resolution can also benefit from this single point of detection feature since no synchronization is necessary when resolving a deadlock.

Also, timing constraints are considered in these two algorithms. In real-time applications, when a task is waiting, it is usually assigned a deadline in order to time out from a waiting state. These two algorithms could utilize tasks' waiting deadlines for deadlock detection in real-time systems.

Following the guidelines suggested in Chapter 4, these two cycle detection algorithms are developed in three refinement steps. Before we start the development of our algorithms, in Section 5.1, some assumptions concerning the real-time constraints are summarized. In Section 5.2 the principles of cycle detection are analyzed in terms of a static GRG. The synchronization mechanisms and the deadline computations are directly developed for each of the algorithms (in Sections 5.3 and 5.4). Concluding remarks are found in Section 5.5. In this chapter, we will give clear reasoning before the presentation of each algorithm but skip formal proofs of the resulting algorithms.

5.1 Assumptions of the Algorithms Concerning the Real-Time Constraints

The two deadlock detection algorithms developed in this chapter are our first attempt at dealing with timing constraints in distributed deadlock detection. Since there are complicated issues involved in the determination of an intermediate deadline whenever a task is in a waiting state, we assume that when a task becomes blocked, there is a known deadline for that waiting state. A task's waiting state terminates when its waiting deadline expires. Our algorithms only utilize the waiting deadline information for deadlock detection but do not specify how such a deadline is assigned. Also, since there are complicated issues involved in the resolution of deadlocks in distributed real-time systems, we simply assume the resolution of a detected deadlock is done by choosing the task which declares the deadlock as the victim.

Due to waiting deadlines, spontaneous time-outs and aborts may occur and may cause false detection of temporal deadlocks. False detection of temporal deadlocks could be minimized by carrying waiting deadline information in each of the probes. If all the system clocks are perfectly synchronized and all the deadlines carried by the probes are absolutely accurate, the false detection of temporal deadlocks is eliminated. Also, a temporal deadlock may not be detected if the timing constraint in a cycle is so tight that none of the existing probes can finish traveling through the cycle in time. An undetected temporal deadlock is resolved automatically when a task in the cycle times out or aborts. This spontaneous time-out or abort, however, may not be the best resolution of a temporal deadlock.

As stated in Section 2.6.2 it is very difficult in a distributed real-time system for a deadlock detection algorithm to fulfill the two correctness criteria. The two algorithms presented in this chapter only "attempt" to detect temporal deadlocks. It is assumed that most of the deadlocks are simple cycles. The two algorithms are designed to be efficient especially when detecting simple cycles. Therefore, the undetected temporal deadlocks can be minimized.

5.2 The Principles of Cycle Detection

We begin our development of the cycle detection algorithms by analyzing cycle detection principles in static systems. First, as discussed in Section 4.3.1, a local view of a cycle can be defined as follows.

Definition 5.1 (Local View of a Cycle) In a GRG, a vertex is in a cycle if and only if it can find a directed path to itself.

In Definition 5.1, the found path is a cycle. If we search a GRG along the directed edges, a vertex v is in a cycle if and only if $v \in RS(v)$. On the other hand, if we search a GRG in the reverse direction of the edges, a vertex v is in a cycle if and only if $v \in TS(v)$. If a vertex v wants to know if it is in a cycle, it may send out a probe message to search the GRG. The probe may be propagated in either direction, i.e., forward or backward, but should be consistent throughout the whole probe computation. If v receives its probe back, it is in a cycle.

In a GRG, a cycle may be connected with vertices which don't belong to the cycle. A probe is called a "foreign" probe if it is propagated in a cycle but was initiated at a vertex outside of that cycle. One of the problems with probe-based cycle detection is that foreign probes may be propagated in a cycle forever without declaring the cycle. There are three strategies which may be used to resolve this problem:

1. **Prevent foreign probes from entering into any cycle.**

   This is the best strategy among the three but it can only be applied to the detection of Single-Resource model deadlocks. This strategy is based on the fact that in a Single-Resource model system, there may be incoming edges into cycles but there are no outgoing edges from any cycle. If probes are propagated backward, they may be propagated out from cycles but they may not go into any cycle. Mitchell and Merritt's algorithm [100] is a very good example of the application of this strategy. In Section 5.3, we develop a Single-Resource Algorithm based on this strategy.

2. **At each vertex, record the received probes and stop a probe's propagation when it is received the second time.**

   This is the most popular strategy used in the algorithms for AND deadlocks (e.g., [27], [35], and [126]). First, Chandy, Misra, and Haas use the notion of *dependent set* in their resource deadlock detection algorithm [27]. In their algorithm, each probe carries its initiator's ID as the probe ID. Each vertex collects a dependent set which contains the received probe ID's. Whenever a probe is processed at a vertex it is checked against the vertex's dependent set before further propagation. A probe will not be propagated if it is found in a vertex's dependent set. This algorithm has two problems: (1) it may fail to detect certain cycles and (2) it may declare a cycle multiple times.

   Problem (1) is due to the fact that a vertex, say v, may need to initiate probe computation many times whenever it becomes blocked. If vertices u and v are in the same cycle and v is already in u's dependent set (due to a previous probe computation initiated at v), v's probe will be stopped at u and won't declare the cycle. If v is the very one which could detect the cycle, the cycle will not be detected. For example, a cycle $\langle a \rightarrow b \rightarrow c \rightarrow a \rangle$ is formed when two edges $\langle b \rightarrow c \rangle$ and $\langle c \rightarrow a \rangle$ are created almost simultaneously. Each of b and c initiates a probe computation about the same time. However, c receives its probe back earlier than b and decides to abort itself to break the detected cycle. When b's request is granted due to the abortion of c, its probe has just arrived at a. Now, we have $\langle a \rightarrow b \rangle$ and b is in a's dependent set. Suppose b starts to wait for a later, b's new probe will be stopped at a and the cycle $\langle a \rightarrow b \rightarrow a \rangle$ will not be detected. An easy remedy to this problem is to always generate a new ID for the initiated probe whenever a vertex becomes blocked.

   Sinha and Natarajian [126] use priority based probes to solve the problem (2) so that a cycle may be declared exactly once. Since the information of a certain computation are distributed at different places where the probes of the

computation are received, Sinha and Natarajian's algorithm requires a post-resolution computation to clean up the out of date dependency information. Their algorithm becomes very complicated and incorrect. Choudhary et al. made an attempt to fix the problem found in Sinha and Natarajian's priority based algorithm, but it resulted in an even more complicated algorithm and which was incorrect when first published. All of these complications are due to the fact that the states of a probe computation are distributed at different places and out of date information needs a post-resolution clean up computation.

3. **In each probe, record the visited vertices and stop further propagation when a vertex is visited a second time.**

   This is an alternative way of recognizing whether a probe has visited a vertex twice. The AND Algorithm developed in Section 5.4 is based on this strategy. There are several interesting aspects of this strategy: (a) Part of a GRG is carried with the probe and when new vertex are visited, new information can be appended to the sub-GRG collected by the probe. For example, a cycle can be declared when a probe finds it has visited a vertex twice. (b) When a vertex puts itself into a probe's sub-GRG, it can be treated as a merge of probe computations. For example, suppose a vertex v becomes blocked and a new probe computation is needed. Vertex v may put itself in a received probe and when the probe comes back, v could find itself in the probe and declare a cycle. If the second strategy is used, a separate probe computation is needed in this case. (c) In addition to the declaration of a cycle, it is necessary to identify the members of the detected cycle. If so, this strategy may be the best choice.

   One disadvantage of this strategy is that the probes are of variable length and are longer than the probes required by the previous strategies. In spite of this disadvantage, the strategy may still be justified by the advantages due to the aspects discussed above.

In the algorithms developed in the following sections, cycle detection computations (CDC's) are only invoked by the tasks. This is because that a GRG is a bipartite graph such that each edge connects a task and a resource. In a static GRG, it is sufficient that each edge can be reached by a cycle detection computation. In a dynamic system, whenever a new edge is created it is due to either (1) a task is waiting for a resource or (2) a resource is acquired by a task. If (1) is the case, the resource must be held by another task and a CDC may be invoked by the waiting task. On the other hand, if (2) is the case, the task must be active and it is not necessary to invoke any cycle detection. Therefore, it is sufficient that CDC's are only invoked at the tasks in a system.

5.3 Algorithm for the Single-Resource Model

In this section a deadlock detection algorithm is developed for the Single-Resource model systems. Following the methodology developed in Chapter 4, the Single-Resource algorithms are developed in three refinement steps, i.e., (1) simple cycle detection in static systems (Section 5.3.1), (2) simple cycle detection in dynamic systems (Section 5.3.2), and (3) extension for real-time applications (Section 5.3.3). Finally, an integrated Single-Resource algorithm for distributed real-time applications is presented in Section 5.3.4.

5.3.1 Static Algorithms for the Single-Resource Model

The following algorithm for the Single-Resource deadlock detection in static systems is based on the first strategy in Section 5.2.

Algorithm 5.1

> Initiate a probe for each task in a GRG. Probes are propagated backward along the edges of a GRG. If a probe comes back to its initiator, a deadlock is found. ◇

Algorithm 5.1, although it guarantees the detection of all deadlocks, allows all tasks in the cycle to detect the deadlock. It has two drawbacks: (1) it is inefficient in terms of message overhead and (2) it is complicated to recover a deadlock due to the multiple detections of the cycle. By using a technique similar to that used in the Mitchell and Merritt's algorithm [100], the following algorithm can reduce the number of probe messages and achieve a single point of detection of every deadlock cycle.

Algorithm 5.2

> Initiate a probe for each task in a GRG. Each probe, when it is initiated, is assigned a unique ID. Each vertex in the GRG memorizes the largest probe ID it has propagated. Only probes that have larger ID's than previous probes are allowed to pass through a vertex. Probes are propagated backward along the edges of the GRG. If a probe comes back to its initiator, a deadlock is found. ◇

Since probes are propagated backward, they may be propagated out from cycles but they may not go into any cycle. Therefore, we don't have to consider that any foreign probe may appear in a cycle. Since ID's are unique, there exists a probe with the largest ID in a cycle which can pass through every vertex in the cycle and declare the cycle. Since this largest probe is initiated in the cycle, its initiator must remember its ID. Therefore, no other probe could pass through the initiator of the largest probe because they have smaller ID's. Consequently, a cycle will be declared exactly once by the largest probe initiated in the cycle.

5.3.2 Dynamic Algorithm for the Single-Resource Model

Backward probe propagation in a Single-Resource model system ensures that probes are always received at waiting vertices (i.e., blocked task and held resources). Since a waiting vertex does not voluntarily delete its only outgoing edge, we don't have to worry about the white edges in the Single-Resource dynamic algorithm. Also, since probes are always received at waiting vertices via its only outgoing edge,

they can be propagated to any incoming edge immediately. Also, if a new incoming edge is connected to a waiting vertex, Theorem 4.2 suggests that all the probes received at this vertex while it is waiting can continue their propagation through the new incoming edge. There is no special concern of the connectivity between the adjacent edges since probes are always received from the very outgoing edge while a vertex is waiting. By Theorem 4.1, the safety concern is satisfied.

In a dynamic system, a DGRG may be updated only when a task or a resource changes its state. As discussed earlier, resources do not initiate probe computations when they are held by tasks. The rest of the situations are described in the synchronization mechanism in Algorithm 5.1 as follows.

Algorithm 5.3

Synchronization Mechanism:

- Each of the tasks/resources keeps track of the probes received while it is blocked/held.
- Whenever a task/resource becomes active/free, it deletes all the probes received while it was blocked/held.
- Whenever a task is blocked by a pending request, it resumes the propagation of all the probes stored at the waited resource as well as initiates a probe for itself.

Description of a CDC:

Probes are propagated backward along the edges of a GRG. If a probe comes back to its initiator, a deadlock is found. ◇

Since backward probe propagation in a Single-Resource model system ensures that probes are always propagated in the stable part of a DGRG, both the Spatial and the Temporal consistency criteria are automatically fulfilled when it applies to dynamic systems. Therefore, this algorithm guarantees that in dynamic systems, all cycles and only genuine cycles will be detected in finite time.

We now want to derive a dynamic algorithm based on Algorithm 5.2 which could detect every cycle exactly once. Similar to Algorithm 5.3, safety concern can be fulfilled without dealing with the white edges and the connectivity problems in the following dynamic algorithm.

To generate a unique ID for each invocation of CDC we need to consider the following factors. First, when a vertex becomes blocked, it needs to override all its previously involved now meaningless probe computations. Also, according to Theorem 4.2, it is not necessary to always initiate a new probe computation to override an old meaningful one whenever a vertex becomes blocked. Therefore, when a vertex v becomes blocked, it acquires a probe from the resource for which it is waiting. If this probe is larger than v's previous probes and is still valid (meaningful at v), no new probe should be generated. Otherwise, a new probe computation is initiated with a logical timestamp which is a number greater than the largest timestamp that a task and its waiting resource have ever seen. To guarantee the uniqueness of each probe timestamp, it could be combined with its initiator's ID which is unique across the whole system.

Algorithm 5.4

Synchronization Mechanism:

- Each of the tasks/resources keeps track of the largest probe received while it is blocked/held.
- Whenever a task/resource becomes active/free, it marks the stored largest probe as invalid.
- Whenever a task is blocked by a pending request, it acquires the probe stored at the waited resource. If this probe is is still valid and is larger than the local one, no new probe is generated; otherwise, a new probe is initiated with a logical timestamp which is a number greater than the largest timestamp the task has ever seen.

Description of a CDC:

The probes which are allowed to pass through a vertex are in increasing

timestamp order. Probes are propagated backward along the edges of a GRG. If a probe comes back to its initiator, a deadlock is found. ◇

We could imagine that a smaller timestamp is generated in the case that no new probe is needed and the correctness argument of this algorithm is then similar to that of Algorithm 5.2 in terms of meaningful projected CDC's.

5.3.3 Real-Time Constraints in the Single-Resource Algorithm

Now, let's consider how tasks' waiting deadlines can be incorporated in a probe computation. Since probes are propagated backward to search vertices which can reach their initiator, probes may become meaningless if one of its visited vertices leaves the waiting state due to the expiration of its deadline. Therefore, each probe may be associated with a deadline which is the earliest task waiting deadline that a probe has ever seen. A probe misses its deadline if at least one of the tasks it visited misses the deadline. Therefore, a probe is discarded immediately if it is found to miss its deadline.

5.3.4 The Single-Resource Algorithm

In this section, we present a simple probe algorithm that deals with the Single-Resource deadlock model in distributed systems. This algorithm is based on Algorithm 5.4 along with a deadline checking mechanism as described in Section 5.3.3. This algorithm is able to detect all stable deadlocks and attempts to detect temporal deadlocks.

The data structures for the probes, tasks, and resources are defined in Figures 5.1, 5.2, and 5.3, respectively. In Figure 5.1 the fields *probe_id* and *initr_id* give each probe a unique logical timestamp (identification). A probe is said to be larger than another one if it carries a larger *probe_id*. The larger *initr_id* is used to distinguish between two probes with the same *probe_id*. The deadline of a probe is defined by the field *probe_dl*. Figure 5.2 shows the data structure for tasks, which

may be part of a *task control block* or may be a separate data structure dedicated for deadlock detection. Two buffers are prepared for storing probes for each task: *probe_init* stores its own initiated probe and *probe_buf* stores the largest probe ever received. Figure 5.3 defines the data structure for resources. Only one probe buffer *probe_buf* is prepared for storing the largest probe ever received for each resource since no probe may be initiated by resources.

```
type PROBE_ID_TYPE is range 0..INTEGER'LAST;
type TASK_ID_TYPE is range 0..INTEGER'LAST;
type PROBE_TYPE is
   record
      probe_id : PROBE_ID_TYPE := 0;   -- probe id
      initr_id : TASK_ID_TYPE := 0;    -- the task_id of the probe initiator
      probe_dl : DURATION := 0.0;      -- probe deadline is determined by the earliest tim-
                                       -- ing constraint in its travelling path
   end record;
```

Figure 5.1: Data structure for probes in the Single-Resource Algorithm.

```
type TASK_STATE_TYPE is (ACTIVE, BLOCKED);
type TASK_TYPE is
   record
      task_id : TASK_ID_TYPE := 0;
      task_state : TASK_STATE_TYPE := ACTIVE;
      task_dl : DURATION := 0.0;  -- deadline of task's request
      probe_init : PROBE_TYPE;    -- probe initiated
      probe_buf : PROBE_TYPE;     -- probe buffered
      holding_table : RES_TABLE_TYPE;  -- resources held by the task
      pending_table : RES_TABLE_TYPE;  -- pending requests of the task
   end record;
```

Figure 5.2: Data structure for tasks in the Single-Resource Algorithm.

When a task becomes *BLOCKED* from the *ACTIVE* state, it checks the status of the requested resource. If a valid probe is found at the requested resource, the probe may continue its propagation if it has a larger ID (i.e., compared to the one in task's *probe_buf*). Otherwise, a new probe may be initiated after a delay time Δt which is chosen as a function of a task's deadline and/or the average blocking time of a request. The logical timestamp (i.e., the *probe_id*) of the newly created probe is the

```
type RESOURCE_ID_TYPE is range 0..INTEGER'LAST;
type RESOURCE_STATE_TYPE is (FREE, HELD);
   -- For a consumable resource, it is FREE if it is produced but is not consumed yet; on the
   -- other hand, it is HELD by its producer if it is requested but is not produced yet.
type RESOURCE_TYPE is
   record
      resource_id : RESOURCE_ID_TYPE := 0;
      resource_state : RESOURCE_STATE_TYPE := FREE;
      probe_buf : PROBE_TYPE;       -- probe buffered
      waiting_queue : QUE_TYPE;     -- waiting queue for the resource
      grant_queue : QUE_TYPE;       -- granted tasks of the resource
   end record;
```

Figure 5.3: Data structure for resources in the Single-Resource Algorithm.

increment of the largest *probe_id* ever received by its initiator. The deadline of the new probe is initially set according to the initiator's timing constraints. The newly created probe is, then, treated as the largest probe ever received and is propagated accordingly. The **procedure** *TASK_INIT_PROBE* depicted in Figure 5.4 describes how a probe is initiated.

The **procedure** *TASK_RCV_PROBE* described in Figure 5.5. is invoked when a task receives a probe. This algorithm guarantees that only tasks in the *BLOCKED* state may receive probes since probes are propagated in the reverse direction along the edges in GRG. The received probe is, first, checked to see if it has missed its deadline. If so, it is discarded immediately because at least one task in the path that the probe traveled has timed out or was aborted at the time the probe is received. If the probe is still valid, it is checked whether it is initiated by the receiving task. If so, a deadlock (which may either be a stable deadlock or a temporal deadlock) is found. Otherwise, the probe is checked to see if it is the largest probe ever received. If so, the deadline of the probe is updated, if necessary, and then the probe is propagated to all the resources held by the task.

The **procedure** *RESOURCE_RCV_PROBE* shown in Figure 5.6 is invoked when a resource receives a probe. This algorithm guarantees that only *HELD* resources may receive probes since probes are propagated in the reverse direction from a

```
procedure TASK_INIT_PROBE (T: in out TASK_TYPE,
                           R: in RESOURCE_TYPE) is
   -- This procedure is invoked when a task T requested a resource R which is not FREE. The
   -- task T is in transition from ACTIVE state into BLOCKED state. It requests the probe
   -- from the waited resource R.probe_buf. A period of waiting time Δt which is chosen as
   -- a function of task T's deadline and/or the average blocking time of a request might be
   -- inserted before the initiation of a new probe.
   R_ID : RESOURCE_ID_TYPE;
begin
   if ((R.probe_buf.probe_dl <= current_time) or else
      (R.probe_buf.probe_id <= T.probe_buf.probe_id)) then
      -- prepare a new probe after a delay time Δt
      T.probe_init.probe_id := MAX(R.probe_buf.probe_id, T.probe_buf.probe_id) + 1;
         -- function MAX(a,b) returns the maximum value of a and b
      T.probe_init.initr_id := T.task_id;
      T.probe_init.probe_dl := T.task_dl;
      T.probe_buf := T.probe_init;
   else
      T.probe_buf := R.probe_buf.probe_init;
   end if;
   -- propagate the new probe to all the resources it holds
   for R_ID in T.holding_table loop
      SEND (T.probe_buf, R_ID);
   end loop;
end TASK_INIT_PROBE;
```

Figure 5.4: Procedure for the probe initiation in the Single-Resource Algorithm.

BLOCKED task to all its *HELD* resources. In the Single-Resource model, a resource can only be held exclusively by one task and does not initiate any probes. It is not necessary to detect deadlocks at a resource vertex. The probe at the resource vertex, therefore, is only checked to see if it has missed its deadline. If so, the probe is discarded; otherwise, it is propagated to all the tasks waiting for that resource.

Initially, all tasks are *ACTIVE* when created, all reusable resources are *FREE*, and all consumable resources are *HELD* by the producers when requested. When a task is in transition from the *ACTIVE* state to the *BLOCKED* state, it adds an edge to the GRG and executes the **procedure** *TASK_INIT_PROBE* to initiate a deadlock detection probe. When a task becomes *ACTIVE* from the *BLOCKED* state, it deletes the corresponding edges from the GRG. Resources are the passive entities in the GRG which will not initiate deadlock computation. If resources are eliminated and a TWFG is considered, the correctness of this algorithm still holds.

```
procedure TASK_RCV_PROBE (T: in out TASK_TYPE; P: in PROBE_TYPE) is
   -- This procedure is invoked whenever a task T receives a probe P.
   R_ID : RESOURCE_ID_TYPE;
begin
   if ((P.probe_dl <= current_time) or else
      (P.probe_id < T.probe_buf.probe_id) or else
      ((P.probe_id = T.probe_buf.probe_id) and then
       (P.initr_id < T.probe_buf.initr_id))) then
      null; -- discard the received probe
   elsif ((P.probe_id = T.probe_init.probe_id) and then
         (P.initr_id = T.task_id) ) then
       a deadlock is found;
   else
      -- update its deadline if necessary and put it in probe_buf and propagate it
      if (P.probe_dl > T.task_dl) then
         P.probe_dl := T.task_dl;
      end if;
      T.probe_buf := P;
      for R_ID in T.holding_table loop
         SEND (P, R_ID);
      end loop;
   end if;
end TASK_RCV_PROBE;
```

Figure 5.5: Procedure for tasks handling received probes in the Single-Resource Algorithm.

```
procedure RESOURCE_RCV_PROBE (R: in out RESOURCE_TYPE;
                               P: in PROBE_TYPE) is
   -- This procedure is invoked whenever a resource R receives a probe P.
   T_ID : TASK_ID_TYPE;
begin
   if ((P.probe_dl <= current_time) or else
      (P.probe_id < R.probe_buf.probe_id) or else
      ((P.probe_id = R.probe_buf.probe_id) and then
       (P.initr_id < R.probe_buf.initr_id))) then
      null; -- discard the received probe
   else
      -- put it in probe_buf and propagate it
      R.probe_buf := P;
      for T_ID in R.waiting_queue loop
         SEND (P,T_ID);
      end loop;
   end if;
end RESOURCE_RCV_PROBE;
```

Figure 5.6: Procedure for resources handling received probes in the Single-Resource Algorithm.

Whenever a task becomes *ACTIVE* or a resource becomes *FREE*, the probe stored at the *probe_buf* is invalidated by resetting *probe_dl* := *0*.

A GRG can be implemented as a two dimensional matrix. One dimension represents tasks, and the other dimension represents resources. Each of the elements in a GRG matrix represents one of the following three states: (1) the task is waiting for the resource, (2) the resource is held by the task, or (3) there is no relationship between them. Another possible implementation of GRG is to store the information in each of the task tables and resource tables, for example, the *holding_table* in *TASK_TYPE* and the task *waiting_queue* in *RESOURCE_TYPE* used in our algorithm data structure.

An agent may be assumed to handle the deadlock detection activities at each site. The probe *SEND* procedure, which is not described explicitly in the algorithm, is assumed to be handled by the agent. A simple copy operation can accomplish a local *SEND* operation, while a real message will be sent out for an inter-site *SEND* operation. The procedures *TASK_INIT_PROBE*, *TASK_RCV_PROBE*, and *RESOURCE_RCV_PROBE* are designed to be executed by the agent on behalf of each task or resource.

5.4 Algorithm for the AND Model

In this section a deadlock detection algorithm is developed for the AND model systems. Again, the algorithms are developed in three refinement steps in Sections 5.4.1, 5.4.2, and 5.4.3, and an integrated AND algorithm for distributed real-time applications is presented in Section 5.4.4.

5.4.1 Static Algorithm for the AND Model

Suppose Algorithm 5.2 is modified for the AND model as follows.

Algorithm 5.5

Initiate a probe for each task in a GRG. Each probe, when it is initiated, is

assigned a unique ID. Each vertex in the GRG memorizes the largest probe ID it has propagated. Only probes that have larger ID's than previous probes are allowed to pass through a vertex. Probes are propagated either forward or backward along the edges of a GRG. If a probe revisits a task, a deadlock is found. ◇

A GRG in the AND model is symmetric in the sense that multiple incoming and outgoing edges of a vertex are allowed. We may find both incoming edges as well as outgoing edges in an AND model cycle. Therefore, in terms of the graph structure, it makes no difference whether probes are propagated in either the forward or the backward direction. Since the foreign probes may enter a cycle in the AND model and may interfere with the in-cycle probes, it is required that every probe (either in-cycle probes or foreign probes) be able to detect deadlocks. Instead of declaring a cycle at a probe's initiator, Algorithm 5.5 declares a cycle whenever a probe revisits a task. This algorithm guarantees that all cycles in a GRG will be detected in finite time.

Since in Algorithm 5.5, a foreign probe is required to declare a cycle when it revisits a task, carrying the initiator's ID with each of the probes is not enough for this purpose. A set of tasks/resources which contains part of a GRG may be included in each of the probes in order for foreign probes to determine if cycles are found. Such a probe is referred to as a "set-based" probe.

The notion of set-based probes was first proposed by Chandy and Misra [23] and subsequently developed by Haas and Mohan [57]. In Haas and Mohan's algorithm, a probe carries a *set* of permanent blocking edges that has been known to the probe. The probes are propagated in the forward direction along the edges of a GRG. Upon receiving a probe, each task searches for cycles that involve itself and deletes the edges related to the detected cycles from the set. If the remaining set is not empty, the task will append itself to the set and propagate it to the tasks it is waiting for. The set grows as it reaches more and more tasks and shrinks when cycles are detected.

In the AND Algorithm proposed here, each probe includes a set of edges which only contains the path traveled by the probe. A set is a one-dimensional chain in our algorithm as opposed to a tree-like structure sub-GRG in the algorithms previously proposed in the literature. The original motivation for propagating a tree-like set in each of the probes is to discover all cycles that involve a deadlocked task which can then act as a deadlock resolver. If deadlock resolution is taken into consideration, some of the detected cycles might have been broken (false deadlocks) due to the fact that cycles may be nested in the AND model. When a deadlocked task knows all cycles that it is involved with, this only reduces the false detection of deadlocks. On the contrary, if only chain-like set probes are propagated in detecting cycles, each deadlocked task will detect at most one cycle at a time. The remaining cycles, if they exist, will be detected as soon as all the involved tasks are searched by a probe. Unlike a tree-like set algorithm, in which a deadlock resolution is delayed until a task can determine it has detected all the cycles it is involved in, our algorithm attempts to resolve deadlocks as soon as it is detected. The probability of related false detection of deadlocks will be reduced since the detected deadlocks are resolved as soon as possible. Also, processing and propagation of the tree-like set probes are more costly compared to the chain-like set probes. Therefore, our algorithm can avoid some false detection of deadlocks comparable to the previous algorithms, while providing better efficiency. Consequently, in real-time applications where timing constraints are important, a chain-like set probe algorithm is more attractive than a tree-like set probe algorithm.

Since the chain set based probe computation could declare a cycle whenever a task finds itself in the chain of a previously received probe, each of the task in a probe chain could be treated as a co-initiator of the probe. Therefore, a probe can be treated as a merged multiple probes which share the same probe timestamp. Also, the chain set represents the wait-for dependencies of the tasks/resources found in the chain. The direction of the probe propagation may affect the management of the chain which, in turn may affect the correctness of the algorithm. We will discuss

this issue in Section 5.4.3 after a brief description of how real-time constraints are incorporated in the computations.

5.4.2 Dynamic Algorithm for the AND Model

We now develop a dynamic algorithm based on Algorithm 5.5. Algorithm 5.5 allows probes to be propagated either forward or backward. If the probes are propagated backward, similar to the reasoning of Algorithm 5.4, both the Spatial and Temporal consistency criteria are automatically fulfilled. On the other hand, if the probes are propagated forward, the detection of white edges need to be avoided. Also, probes may be received by the active task if forward propagation is used. Since Theorem 4.2 suggests that the meaningful CDC's should be preserved as much as possible, those probes received while a task is active may continue their propagations if necessary. However, care should be taken concerning the connectivity between two adjacent operations if a probe is received at a active task. If a probe received at an active task is saved for later use, the existence of the incoming edge from which the probe was received must be checked when it triggers subsequent probe propagations after the task becomes blocked. This mechanism requires a complex data structure to keep track of the information concerning where each probe was received and it incurs constant overhead when a task is active. Instead, a probe P which is supposed to be sent to an active task T from a resource R could be kept at R. The probe P may be propagated to T later after T becomes blocked and R is still held by T. Since the tasks which are waiting for the resource R do not withdraw their requests voluntarily, the connectivity requirement at resource R is fulfilled.

For a reason to be discussed in Section 5.4.3, we use forward propagation in the following dynamic AND algorithm. Again, resources do not initiate probe computations when they are held by tasks. In the AND model system, the state of a task may be changed when (1) it is blocked by a set of pending request, (2) it is granted one of its pending requests (but is still blocked by other outstanding requests), or (3) it becomes active after all its requests are granted. We can imagine

that in the case (2), a task T is temporarily active and then becomes blocked by a new set of pending requests. While T is temporarily active, it acquires a new resource which dramatically changes its surrounding graph structure. Consequently, the occurrence of (1) or (2) may trigger a new probe computation.

Algorithm 5.6

Synchronization Mechanism:

- Each of the tasks/resources keeps track of the largest probe it has ever received.
- The probes received at active tasks will be ignored.
- Whenever a task/resource becomes active/free, it marks the stored largest probe as invalid.
- When a task is (1) blocked by a set of pending request or (2) granted one of its pending requests but remains blocked, it acquires the probes stored at its holding resources. If the largest probe acquired is still valid and is larger than the one which is previously stored at the vertex, no new probe is generated; otherwise, a new probe is initiated with a logical timestamp which is a number greater than the largest timestamp that the task has ever seen.

Description of a CDC:

The probes which are allowed to pass through a vertex are in increasing timestamp order. Probes are forward propagated along the edges of a GRG. If a probe revisits a task, a deadlock is found. ◇

Again, each probe in Algorithm 5.6 carries a chain of task/resource ID's which records its propagation path. Every cycle will be declared exactly once by the largest probe propagated in the cycle at a task which first appears twice in the probe chain.

5.4.3 Real-Time Constraints in the AND Algorithm

Unlike the Single-Resource Algorithm, only part of a probe's trace may form a cycle. A single timing constraint can no longer be associated with each probe. A timing constraint for each task in a probe chain should be included. When searching for cycles in the chain set, the timing constraints attached to each of the task ID's are also evaluated. A deadlock is found if none of the tasks which form the cycle has missed its deadline. A chain may be broken at a task in the chain if the task is found to have missed its deadline. Also, the task ID's in the chain which are waiting for the one that missed its deadline should be discarded. This is because the wait-for relations may have been changed when the task which missed deadline aborted and released its resources.

We now explain why backward propagation cannot be used if timing constraints are considered. If a probe is propagated backward, the chain grows by adding new dependents to the chain. A new dependent is impossible to add to the chain if the chain is broken. Therefore, a backward propagated probe should be discarded if its chain is broken. For example, consider a chain $T_i \rightarrow R_j \rightarrow T_k \rightarrow \cdots \rightarrow T_m \rightarrow \cdots \rightarrow R_x \rightarrow T_y$ which is propagated along with a probe. If the task T_m is found to miss its deadline, the chain is broken at T_m and its left hand side of the chain $T_i \rightarrow R_j \rightarrow \cdots \rightarrow T_m$ is discarded. Since the probe is propagated backward, a new entry (a dependent of T_i) should be added to the left hand side of T_i which no longer exists in the chain; the whole probe, therefore, should be thrown away.

One problem with the backward propagation of the probe chain is that a meaningful part of a probe chain may be discarded. Such an algorithm may fail to detect certain deadlocks which are supposed to be searched and declared by the discarded probes. Following the previous example, suppose at the time the probe is discarded, a cycle $T_k \rightarrow R_l \rightarrow T_i \rightarrow R_j \rightarrow T_k$ exists and all the other probes are eliminated due to their smaller timestamps. This deadlock may not be detected if no new probes with a larger timestamp could possibly reach this cycle. Consequently, backward

probes cannot be used in real-time applications because the probe chain may break due to timing constraints.

In contrast, if the probe is propagated forward along the directed edges, the new entries are added to the right hand side of T_y and the probe, after discarding its invalidated part of the chain, can continue to search the GRG until it reaches an end vertex (an active task) or finds a cycle. In other words, the forward propagated probe avoids the error that the backward probe exhibits.

The subset of the resource vertices may be eliminated in the GRG by replacing an assignment edge (or a producer edge) and a request edge pair attached to a resource vertex with a single directed edge between two tasks. For example, $T_i \rightarrow R_j \rightarrow T_k$ can be simplified to $T_i \rightarrow T_k$ if the resource vertex R_j is not necessary in the graph. If all the resource vertices are eliminated, it becomes a task-wait-for graph (TWFG). In the AND Algorithm presented in the following, we ignore the resource vertices in the probe chain since we are not interested in detecting deadlocks at the resource vertices in a GRG. This simplification can reduce the size of the probe messages.

5.4.4 The AND Algorithm

In this section, based on Algorithm 5.6 and the deadline computation described in Section 5.4.3, we propose a set-based probe algorithm that deals with the AND deadlock model in distributed real-time systems. This algorithm is able to detect all the stable deadlocks and attempts to detect temporal deadlocks. Spontaneous time-outs and aborts are allowed if they are needed for timing constraints.

The data structures for the probes, tasks, and resources are defined in Figures 5.7, 5.8, and 5.9, respectively. In Figure 5.7, the fields *probe_id* and *initr_id* are defined in the same way as those in the algorithm for the Single-Resource model. A set of *task_id*'s which record the path of a probe are chained together in the probe. The field *chain_head* points to the head of such a path. Figure 5.8 shows the data structure for tasks. Two buffers are prepared for storing probes for

each task: the *probe_init* stores its own initiated probe and the *probe_buf* stores the largest probe ever received. Also, the data structure for chained task is defined as *CHAINED_TASK*. In the *CHAINED_TASK*, each *task_id* is attached with a *task_dl* (task deadline). Figure 5.9 defines data structure for the resources. Only one probe buffer is prepared for storing the largest probe ever received for each resource since no probe may be initiated by resources.

```
type PROBE_ID_TYPE is range 0..INTEGER'LAST;
type TASK_ID_TYPE is range 0..INTEGER'LAST;
type TASK_PTR; -- point to a task in a chain
type PROBE_TYPE is
record
    probe_id : PROBE_ID_TYPE := 0;  -- probe identification
    initr_id : TASK_ID_TYPE := 0;      -- the task_id of the probe initiator
    chain_head : TASK_PTR := null;
        -- a chain of tasks which records the path of the probe;
        -- the chain_head points to the head of the path
end record;
```

Figure 5.7: Data structure for probes in the AND Algorithm.

```
type TASK_STATE_TYPE is (ACTIVE, BLOCKED);
type TASK_TYPE is
    record
        task_id : TASK_ID_TYPE := 0;
        task_state : TASK_STATE_TYPE := ACTIVE;
        task_dl : DURATION := 0.0;  -- deadline of task's request
        probe_init : PROBE_TYPE;     -- probe initiated
        probe_buf : PROBE_TYPE;      -- probe buffered
        holding_table : RES_TABLE_TYPE;  -- resources held by the task
        pending_table : RES_TABLE_TYPE;  -- pending requests of the task
    end record;
type CHAINED_TASK; -- a task in a chain
type TASK_PTR is access CHAINED_TASK;
type CHAINED_TASK is
    record
        task_id : TASK_ID_TYPE := 0;  -- task id
        task_dl : DURATION := 0.0;      -- task deadline
        next : TASK_PTR;  -- pointer link to the next task in chain
    end record;
```

Figure 5.8: Data structure for tasks in the AND Algorithm.

```
type RESOURCE_ID_TYPE is range 0..INTEGER'LAST;
type RESOURCE_STATE_TYPE is (FREE, HELD);
   -- For a consumable resource, it is FREE if it is produced but is not consumed yet; on the
   -- other hand, it is HELD by its producer if it is requested but is not produced yet.
type RESOURCE_TYPE is
   record
      resource_id: RESOURCE_ID_TYPE:=0;
      resource_state : RESOURCE_STATE_TYPE := FREE;
      probe_buf : PROBE_TYPE;      -- probe buffered
      waiting_queue : QUE_TYPE;    -- waiting queue for the resource
      grant_queue : QUE_TYPE;  -- granted tasks of the resource
   end record;
```

Figure 5.9: Data structure for resources in the AND Algorithm.

Probes may be initiated by the tasks when one becomes *BLOCKED* from the *ACTIVE* state or when a *BLOCKED* task is granted one of its pending requests and remains *BLOCKED*. A period of time Δt which is chosen as a function of a task's deadline and/or the average blocking time of a request may be inserted right before the initiation of a new probe. Again, by Theorem 4.2, a new probe will be initiated only if the previously received one is meaningless (with an empty chain in this case) or it is with a smaller timestamp. The **procedure** *TASK_INIT_PROBE* depicted in Figure 5.10 describes how a probe is initiated. A probe chain is created and is accessed through the *chain_head* in the new probe. The new probe is stored both in *probe_init* and *probe_buf*, and is propagated to each resource in its *pending_table* (pending request table).

The **procedure** *TASK_RCV_PROBE* depicted in Figure 5.11 describes how a probe is received and processed at a task. When a task receives a probe, it first checks the validity of the probe including (1) if the received carries a larger timestamp and (2) whether it was received from a non-white edge. If a probe is received by an *ACTIVE* task, it is discarded immediately. On the other hand, a *BLOCKED* task will process a received validated probe as follows. In the received probe, the head of the chain is treated separately because if it misses its deadline, the whole chain is thrown away and the probe will not be propagated. The cycle detection is done by search the current *task_id* in the chain starting from the head of the chain.

```
procedure TASK_INIT_PROBE (T: in out TASK_TYPE) is
    -- This procedure is invoked when an ACTIVE task T requests resources which are not all
    -- FREE, or when a BLOCKED task is granted one of its pending requests but remains
    -- BLOCKED. A period of waiting time Δt which is chosen as a function of task T's deadline
    -- and/or the average blocking time of a request might be inserted before the initiation of a
    -- new probe.
    ptr : TASK_PTR;     R_ID : RESOURCE_ID_TYPE;     P, FP : PROBE_TYPE
begin
    P.probe_id := T.probe_buf.probe_id
    P.chain_head := null;  -- it is an INVALID probe
    for R_ID in T.holding_table loop     -- poll probes from holding_table
        RECEIVE (FP, R_ID);  -- FP is the probe from R_ID
        if ((FP.probe_id < P.probe_id) or else
           ((FP.probe_id = P.probe_id) and then
            (FP.initr_id < P.initr_id)) or else
           (FP.chain_head.task_dl <= current_time)) then
           null; -- discard the received probe
        else
           P := FP;
        end if;
    end loop;
    ptr := P.chain_head;
    if ((ptr = null) or else       -- empty chain
       (ptr.task_dl <= current_time)) then  -- the head missed its deadline
       -- prepare a new probe after a delay time Δt
       T.probe_init.probe_id := T.probe_buf.probe_id + 1;
       T.probe_init.initr_id := T.task_id;
       T.probe_init.chain_head := new TASK_PTR (T.task_id, T.task_dl, null);
       T.probe_buf := T.probe_init;  -- put the new probe in probe_buf
    else
       -- append the task T to the head of the chain
       T.probe_buf.chain_head := new TASK_PTR (T.task_id, T.task_dl, ptr);
    end if;
    -- propagate the new probe to all the resources it is waiting for
    for R_ID in T.pending_table loop
       SEND (T.probe_buf, R_ID);
    end loop;
end TASK_INIT_PROBE;
```

Figure 5.10: Procedure for the probe initiation in the AND Algorithm.

```
procedure TASK_RCV_PROBE (T: in out TASK_TYPE;
                          P: in out PROBE_TYPE) is
   -- This procedure is invoked when a task T receives a probe P.
   ptr : TASK_PTR;    found : BOOLEAN := FALSE;
   R_ID : RESOURCE_ID_TYPE;
begin
   ptr := P.chain_head;
   if ((not IN_TAB(R, T.holding_table)) or else -- from a white edge
       (T.task_state = ACTIVE) or else
       (P.probe_id < T.probe_buf.probe_id) or else
       ((P.probe_id = T.probe_buf.probe_id) and then
        (P.initr_id < T.probe_buf.initr_id)) or else
       (ptr.task_dl <= current_time)) then
       null; -- discard the received probe
   elsif (ptr.task_id = T.task_id) then
        a deadlock is found;
   else
       -- search the current task in the chain
       while not found and then ptr.next /= null loop
          if (ptr.next.task_dl <= current_time) then
             ptr.next := null; -- discard the rest of the chain
          else
             ptr := ptr.next;
             if (ptr.task_id = T.task_id) then
                found := TRUE;
             end if;
          end if;
       end loop;
       if found then
           a deadlock is found;
       else
          -- append the task T to the head of the chain
          P.chain_head := new TASK_PTR (T.task_id, T.task_dl, P.chain_head);
          T.probe_buf := P; -- put it in probe_buf
          for R_ID in T.pending_table loop
             SEND (P, R_ID);  -- propagate the probe to every
          end loop;           -- resource in T's pending_table
       end if;
   end if;
end TASK_RCV_PROBE;
```

Figure 5.11: Procedure for tasks handling received probes in the AND Algorithm.

The deadlines are also checked for each task in the chain. If an expired task is found, the un-searched part of the chain is disconnected. If no cycle is found, the current task is appended to the head of the chain, and the probe is save in the *probe_buf* and propagated to all the resources which the task is waiting for.

The **procedure** *RESOURCE_RCV_PROBE* shown in Figure 5.12 is invoked when a resource receives a probe. The resource first checks the validity of the received probe. If the probe is valid and is the largest probe ever received, it is saved in the *probe_buf* and propagated to all the tasks in the resource's *grant_queue*.

```
procedure RESOURCE_RCV_PROBE (R: in out RESOURCE_TYPE;
                              P: in PROBE_TYPE) is
   -- This procedure is invoked when a resource R receives a probe P.
   T_ID : TASK_ID_TYPE;
begin
   if ((not IN_QUE(T, R.waiting_queue)) or else  -- from a white edge
      (P.probe_id < R.probe_buf.probe_id) or else
      ((P.probe_id = R.probe_buf.probe_id) and then
       (P.initr_id < R.probe_buf.initr_id))) then
      null; -- ignore the received probe
   else
      R.probe_buf := P;  -- put it in probe_buf
      for T_ID in R.grant_queue loop
         SEND (P, T_ID);
      end loop;
   end if;
end RESOURCE_RCV_PROBE;
```

Figure 5.12: Procedure for resources handling received probes in the AND Algorithm.

In addition to the *SEND* procedure mentioned in the Single-Resource Algorithm, two functions, *IN_TAB* and *IN_QUE*, are used to check if a probe is received from a white edge. The **function** *IN_TAB* searches a task's *holding_table* to see if the probe is received from a resource held by it. Similarly, **function** *IN_QUE* searches a resource's *waiting_queue* to check if the probe is received from a task which is waiting for it.

There are two weak points of this algorithm. First, the detection of a deadlock may not be done in a limited period of time. This is because foreign probes with

increasing timestamps may keep on interfering with each other until one eventually travels through the whole cycle. The detection of an existing deadlock, therefore, may be indefinitely delayed. This situation is more likely to happen in a complicated GRG where a infinite number of new tasks are dynamically created. However, most of the systems, although complicated, do have an upper limit for the maximum number of tasks allowed in the system. Also, the statistical analyses, such as the one done by Gray et al. [55], have shown that most of the deadlocks are simple cycles with a length of two to three vertices involved. This implies that the chance of infinitely delay of a deadlock detection is rare, and in many systems it may be justifiable to live with this rare occurrence in order to take advantage of the optimization made with the probe timestamps. Secondly, the resolution of a detected deadlock is limited to the abortion of the task which declares the deadlock. If more information is carried with each of the probes, such as the priorities of the tasks in the chain, the algorithm may be able to declare the deadlock at the task which is going to be chosen as the victim for the resolution. The priority may be defined to reflect any combination of the *criticalness, timing constraint, task processing time, laxity, amount of I/O completed*, etc. Also, if the priority considers the degree of forward and backward dependency of the task in GRG, the false detection of deadlocks due to the existence of nested cycles may be further minimized. However, more overhead in terms of the size and the number of the probe messages is required.

5.5 Concluding Remarks

In this chapter we developed cycle detection algorithms for the Single-resource and the AND deadlock models. These algorithms are able to detect stable deadlocks and attempt to detect temporal deadlocks. One unique aspect of these algorithms is their ability to address timing constraints of tasks. Both algorithms are based on probes to detect deadlock cycles. Probe message overheads are optimized by carefully choosing a probe propagation direction and imposing a probe propagation rule. Also, based on Theorem 4.2, these algorithms are optimized to detect cycles as

early as possible (most of the algorithms in the literature tend to allow the "latest" computation to declare a cycle).

In the Single-Resource Algorithm, we have shown that the backward probe propagation is the best choice. In the AND Algorithm, forward probe propagation is chosen because the chain carried by a probe may break due to task's waiting deadlines. In both algorithms probes are assigned timestamps. The probes which are allowed to pass through a vertex are in increasing timestamp order. This probe propagation rule can greatly reduce the probe overhead and ensure the single detection of deadlock cycles. Also, we point out that imposing this rule may cause an unlimited delay of the detection of certain deadlocks. However, this situation may be so rare that it is still justifiable to take advantage of the optimization made with the probe timestamps.

CHAPTER 6

KNOT DETECTION

The OR deadlock model was first referred to as Communication Deadlock model by Chandy et al. In this model, a task is blocked until it is satisfied by *any* of the resources it has requested or *any* of the messages or synchronizations it has been waiting for. Most of the OR deadlock detection algorithms proposed in the literature are basically derived from Dijkstra and Scholten's notion of "diffusing computation." However, the structure of a diffusing computation does not directly define a *knot* which is the necessary and sufficient condition for the existence of an OR deadlock. Therefore, many efforts have been made to adapt the diffusing computations to detect knots which cause OR deadlocks. Instead of using diffusing computations, we take a new approach to directly develop efficient knot detection algorithms in refinement steps. Since our approach can effectively recognize the properties of knots, we are able to develop more efficient and practical algorithms than the previous ones.

The first part of this chapter is the development of knot detection algorithms and their extensions to real-time applications. First in Section 6.1, a formal definition of the OR model deadlocks is given. Then, knot detection algorithms are developed in three refinement steps, i.e., (1) knot detection in static systems (Section 6.2), (2) knot detection in dynamic systems (Section 6.3), and (3) their extensions to real-time applications (Section 6.4).

In this dissertation study we are also interested in applying the algorithms to distributed real-time systems such as Ada environments. In Ada, the rendezvous mechanism may cause OR model deadlocks. Also, timing constraints may be involved in Ada rendezvous through **delay** statements. The detection of Ada rendezvous deadlocks is a good application of the algorithms developed in the first part of this chapter. Moreover, there are similarities between the detection of an

OR model deadlock and the detection of the termination of a group of cooperating tasks in a distributed computation. In Ada, the **terminate** statements may be used in association with rendezvous. This suggest that we could resolve two problems in an integrated fashion. Therefore, in Section 6.5, we demonstrate how our real-time knot detection algorithms are used in the detection of Ada rendezvous deadlocks and task terminations.

In Section 6.6, we briefly summarize related work. And, finally, concluding remarks are found in Section 6.7.

6.1 Formal Definition of the OR Model Deadlocks

In Chapter 2, we have briefly defined knots in terms of graph theory concepts (Section 2.2) and used it in the analysis of the OR deadlocks. In this section, we will formally study knots and the OR deadlocks in depth.

According to Chandy et al. [27], a nonempty set of tasks S is deadlocked in the OR model (i.e., is communication-deadlocked) if and only if (a) all tasks in S are blocked, (b) the dependent set of every task in S is a nonempty subset of S, and (c) there are no messages, tokens, or synchronizations in transit between tasks in S. Condition (c) is required when a consistent snapshot of the system state is taken to construct a GRG. Suppose we have such a consistent GRG. The OR deadlock condition can, therefore, be defined as follows.

Definition 6.1 (OR Deadlock) In the OR model, a nonempty set of vertices(tasks/resources) S is deadlocked if and only if $\forall v \in S$, (1) $RS(v) \subseteq S$ and (2) $|RN(v)| > 0$.

Condition (1) means v can only reach vertices in S and condition (2) means the non-isolated vertex v is not a sink. Obviously, any vertex which is in a knot is deadlocked. However, according to Definition 6.1, a deadlocked vertex is not necessarily in a knot. Actually, as we discussed in Section 4.3.2, knots are the deadlock core sets

for the OR deadlocks. It is a misconception to define a deadlock set S as a knot[1]. Instead of using knots, Huang [68] has introduced the notion of a ***stable tie*** to describe a OR deadlock set. The concepts of *tie* and ***stable tie*** can be defined as follows.

Definition 6.2 (Tie) A *tie* T is a nonempty set of vertices such that the reachable set of each vertex v in T is T or a subset of T. That is T is a tie if $\forall v \in T$, the reachable set $RS(v) \subseteq T$.

Definition 6.3 (Stable Tie) A ***stable tie*** S is a tie in which there is no sink, i.e., $\forall v \in S, |RN(v)| > 0$.

It is clear that in addition to the condition (1) of Definition 6.1, which is ensured by Definition 6.2, Definition 6.3 ensures condition (2). From the definitions stated above, it is easy to see that the following properties hold:

Property 6.1 A stable tie K is a knot if there exists no $S \subset K$ that is a tie.

Property 6.2 A stable tie contains at least one knot.

Figure 6.1 shows an example of knots, ties, and stable ties. There are two knots A and B shown in the figure. Also, both knot A and knot B are stable ties which contain no other ties. Stable tie C contains knot B. Stable tie D is the largest stable tie in the picture and it contains all the stable ties and knots in the graph including knot A, knot B, and stable tie C. In addition to stable tie D, the whole picture is a tie.

From Properties 6.1 and 6.2 we can see that an OR model deadlock (which is described by a stable tie) is broken only if all the knots it contains are detected and resolved. Also, it is easy to see that a digraph is free of OR model deadlocks (i.e., free of stable ties) if and only if it is free of knots. Therefore, the detection and resolution of OR deadlocks can be reduced to finding and breaking all knots in a GRG.

---

[1]Liu [89] has pointed out this misconception found in Natarajan's paper [105].

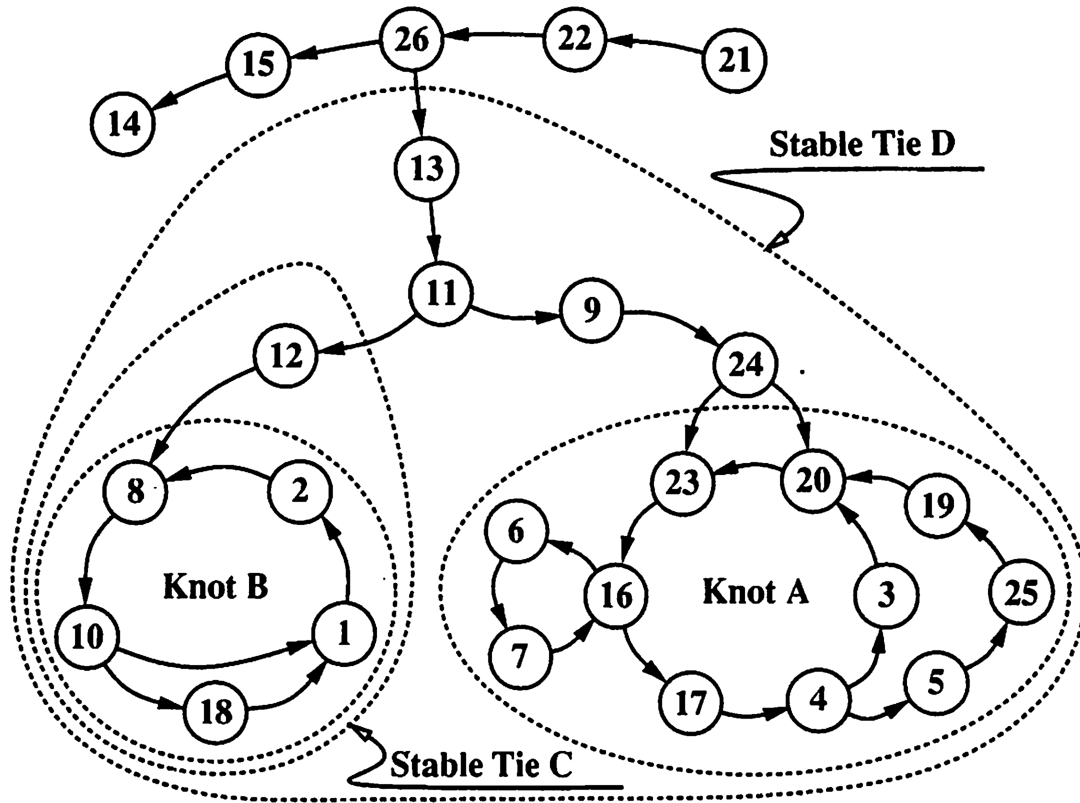


Figure 6.1: An example of ties, stable ties, and knots

6.2 Knot Detection in Static Systems

We begin our development of knot detection algorithms from static systems. In static systems, we assume there are no control messages in the underlying system, which means a knot detection computation (KDC) is running on a static GRG. The purpose of studying static knot detection is to concentrate on the properties and principles of KDC's.

In the following sections, we first develop a basic knot detection algorithm (Algorithm 6.0) for static systems. Algorithm 6.0 is not practical because it may not terminate in finite time. However, Invariant 6.1 proven in this basic algorithm serves as a principle of the knot detection of the subsequent algorithms. We then begin the development of the first real algorithm (Algorithm 6.1) by focusing on individual KDC's. Algorithm 6.1 treats each KDC independently and guarantees a

knot will be detected in finite time. Then, Algorithm 6.2 is developed to take into account that multiple KDC's may be executing concurrently. In Algorithm 6.2 a "voting" mechanism is developed to allow a knot to be declared exactly once.

6.2.1 The Principles of Knot Detection

In this section, we investigate some of the principles of knot detections. A basic knot detection algorithm (Algorithm 6.0) is derived from the definitions and properties of knots in a static GRG. The primary concern here is how to find a knot structure in a distributed GRG. Algorithm 6.0 is neither efficient nor practical because it does not care if an existing knot could be detected in finite time (i.e., it does not satisfy the progress concern). The only purpose of Algorithm 6.0 is to allow us to present some fundamental principles for the development of the subsequent algorithms.

In a distributed system, it is usually difficult to have a global view of the whole system. The knot definition in Section 2.2, unfortunately, suggests the use of such a global view which needs to examine all vertices in a system in order to find out every knot in the system. However, a knot can be defined from the view point of a vertex in a GRG. This is called the "local view definition" of a knot.

Definition 6.4 (Local View of a Knot) For a vertex d in a GRG, its reachable set $RS(d)$ is a knot if and only if $RS(d) \subseteq TS(d)$.

By using Definition 6.4 a knot may be identified from a vertex within the knot. This can be accomplished by initiating a probe based KDC from a vertex d (called d-computation)[2] if we want to determine if d is in a knot. The probes are propagated along the directed edges of GRG. If a vertex $v \neq d$ receives probes initiated by d, it is in $RS(d)$. The vertex v can then participate in d's computation by propagating the received probes to its neighbors $RN(v)$. If u appears in a probe propagation thread

[2]Since such probe computations are concurrently initiated at different vertices, to distinguish them from each other we need to identify each of them uniquely. Here, we use initiator's identification to name such a computation. This naming convention might be changed as the algorithms get more complicated.

which comes back to d eventually, u can reach d (i.e., $u \in TS(d)$). If the GRG is fully searched by the probes (i.e., every member of $RS(d)$ has been reached) and all threads of probe propagations have come back to d (i.e., $RS(d) \subseteq TS(d)$), d is in the knot $RS(d)$.

Again, it is difficult to keep track of all the probes propagated in a distributed environment. Huang [68] suggested a *probe strength computation* which allows a vertex to initiate a probe computation and keep track of it locally. In Huang's algorithm, each probe carries a probe strength. A probe strength may split if its carrier is propagated to several vertices. A probe strength may join at the initiator if its carrier is propagated back. If the initiator sees the original probe strength propagated back, all probes have come back. This idea leads to the following algorithm.

Algorithm 6.0

Convention: A probe (d, v, s) carries three parameters. The first parameter d is the ID of the probe initiator; the second parameter v indicates the destination vertex; the third parameter s shows the current strength of the probe. When a d-computation is initiated the total probe strength is set to one. At the initiator d, a variable S_d is used to accumulate the returned back probe strength.

In a static GRG, the following procedures can be used to test if a vertex d is in a knot.

1. From d, a KDC (d-computation) is initiated as follows:

   S_d := *0*;
   Send a probe $(d, u, 1/|RN(d)|)$ to every vertex $u \in RN(d)$.

2. For each vertex $u \neq d$ which receives a d-computation probe (d, u, s), it does:

   Send a probe $(d, v, s/|RN(u)|)$ to every vertex $v \in RN(u)$.

3. After initiating a probe computation, the initiator d starts collecting returned probes and accumulating the returned probe strength in S_d. If the accumulated probe strength S_d reaches one, a knot is found. ◇

The total probe strength is originally set to one and never enhanced (e.g., duplicated probes may create extra strength) or destroyed (e.g., discarded probes may destroy the strength they carry)[3]. Throughout a d-computation, probes may be (1) split and propagated to several vertices or (2) propagated back and joined at the initiator. For the convenience of discussion, the operations in which a probe strength splits or joins in S_d are assumed to be atomic in the sense that none of these operations may be in transition whenever we check the probe strengths in a GRG. Suppose at time t_i, a snapshot of a d-computation is taken and a set of probes $\{P_j{}^{t_i}\}$ is found. The strength of the probe $P_j{}^{t_i}$ is $S_{P_j}{}^{t_i}$. Also, in the same snapshot, the strength at the initiator is $S_d{}^{t_i}$. The following invariant holds.

<u>**Invariant 6.1**</u> Throughout a d-computation, the sum of all its probe strengths in the system is always one. That is $S_d{}^{t_i} + \sum_{\forall j} S_{P_j}{}^{t_i} = 1$.

Proof: Although the probes may be processed in parallel at different vertices, the assumption that these probes are processed in an arbitrary total order does not affect the correctness of this invariant. With this assumption, we can divide a d-computation into steps which are the snapshots of the probes in the system after each probe is processed. The invariant can be proven by induction as follows.

Step 0: Initially, $S_d{}^{t_0} = 0$ and the probes sent to $RN(d)$ are with strength $1/|RN(d)|$. Therefore, $S_d{}^{t_0} + \sum_{\forall j} S_{P_j}{}^{t_0} = 0 + |RN(d)|/|RN(d)| = 1$.

Step i: Suppose at time t_i, $S_d{}^{t_i} + \sum_{\forall j} S_{P_j}{}^{t_i} = 1$.

Step $i+1$: From Step i to $i+1$, a probe $P_k{}^{t_i}$ may be

[3]Here, we are only talking about characteristics of Algorithm 6.0. The underlying communication subsystem is assumed reliable that no message might be lost or duplicated.

(1) sent to the initiator d:

$$\begin{aligned} S_d{}^{t_{i+1}} &= S_d{}^{t_i} + S_{P_k}{}^{t_i} \\ \sum_{\forall j} S_{P_j}{}^{t_{i+1}} &= \sum_{\forall j} S_{P_j}{}^{t_i} - S_{P_k}{}^{t_i} \\ \Rightarrow S_d{}^{t_{i+1}} + \sum_{\forall j} S_{P_j}{}^{t_{i+1}} &= S_d{}^{t_i} + \sum_{\forall j} S_{P_j}{}^{t_i} = 1 \end{aligned}$$

or (2) sent to a vertex $u \neq d$:

$$\begin{aligned} S_d{}^{t_{i+1}} &= S_d{}^{t_i} \\ \sum_{\forall j} S_{P_j}{}^{t_{i+1}} &= \sum_{\forall j \neq k} S_{P_j}{}^{t_i} + RN(u) \times (S_{P_k}{}^{t_i}/RN(u)) \\ \Rightarrow S_d{}^{t_{i+1}} + \sum_{\forall j} S_{P_j}{}^{t_{i+1}} &= S_d{}^{t_i} + \sum_{\forall j} S_{P_j}{}^{t_i} = 1 \end{aligned}$$

Consequently, $S_d{}^{t_i} + \sum_{\forall j} S_{P_j}{}^{t_i} = 1$ at any time t_i in a d-computation. ∎

Suppose probes are grouped and propagated in steps as follows. At each step i:

1. Take a snapshot of the d-computation. The probes are found at a set of vertices V_i.
2. Propagate every probe found in V_i to its reachable neighbors. That is, $V_{i+1} = RN(V_i)$, where $RN(V_i) = \{w \mid v \in V_i, (v,w) \in E\}$.
3. Repeat for the next step $i+1$.

At each step i, the explored reachable set $RS^i(d)$ is the set of vertices which have been visited by at least one probe from the d-computation.

Lemma 6.1 New reachable vertices $(RS^{i+1}(d) - RS^i(d))$ may only be explored by the probes sent out from the vertices in $(RS^i(d) - RS^{i-1}(d))$. That is

$$(RS^{i+1}(d) - RS^i(d)) \subseteq RN(RS^i(d) - RS^{i-1}(d))$$

Proof: At each step i, there is a set of newly explored reachable vertices $(RS^i(d) - RS^{i-1}(d)) \subseteq V_i$. At next step $i+1$, $(RS^{i+1}(d) - RS^i(d)) \subseteq V_{i+1} = RN(V_i)$,

which, in turn, means the newly explored set, i.e., $(RS^{i+1}(d) - RS^{i}(d))$, is included in the set of vertices which can be reached from V_i.

Suppose a vertex $v \in (RS^{i+1}(d) - RS^{i}(d))$ is reachable from $u \in (V_i - (RS^{i}(d) - RS^{i-1}(d)))$. Since u is not newly explored at step i, all its reachable neighbors should have been visited at least once by the time of step i. This implies $v \notin (RS^{i+1}(d) - RS^{i}(d))$ which is contradictory to the assumption. Therefore, there is no vertex $v \in (RS^{i+1}(d) - RS^{i}(d))$ which is reachable from $(V_i - (RS^{i}(d) - RS^{i-1}(d)))$.

Consequently, every member of $(RS^{i+1}(d) - RS^{i}(d))$ is a reachable neighbor of $(RS^{i}(d) - RS^{i-1}(d))$. In other words, $(RS^{i+1}(d) - RS^{i}(d)) \subseteq RN(RS^{i}(d) - RS^{i-1}(d))$. ■

One problem with Algorithm 6.0 is that it may not terminate in finite time. For example, Figure 6.2 shows a knot in a GRG. If Algorithm 6.0 is initiated at the vertex 1, 2, 8, or 10, the detection computation will terminate in finite time. However, if the computation is initiated at the vertex 18, the probe strength will always split at vertex 10 and a propagation thread will involve an infinite loop.

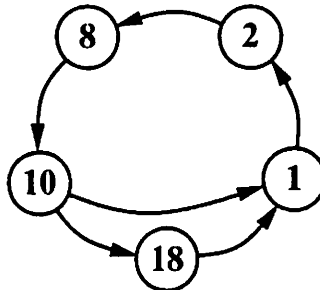


Figure 6.2: Example of an infinite loop in a knot

We will discuss the remedies of this problem in Section 6.2.2. Although a d-computation may be executing perpetually due to infinite loops, based on Lemma 6.1 and the fact that the size of a GRG is finite the complete reachable set $RS(d)$ could be explored (i.e., visited at least once by d-computation probes) in finite time.

Lemma 6.2 By forward propagation of the probes initiated from d, every member of the $RS(d)$ is guaranteed to be reached in finite time.

Proof: The construction of the reachable set $RS(d)$ by Algorithm 6.0 can be recursively described as follows.

Step 0: Starting from $RS^0(d) = \emptyset$.

Step 1: Probes are sent to d's neighbors $RN(d)$, that is

$$RS^1(d) = RN(d) = \{u \mid (d, u) \in E\}$$

Step i: Suppose probes have visited the set $RS^i(d)$.

Step $i+1$: By Lemma 6.1,

$$RS^{i+1}(d) = RS^i(d) \cup RN(RS^i(d) - RS^{i-1}(d))$$

Step ∞: Following these steps, $RS(d) = RS^\infty(d)$.

According to the algorithm, no vertex may be removed from the visited reachable set, that is, $\forall i, RS^i(d) \subseteq RS^{i+1}(d)$. Also, the size of $RS^i(d)$ is upper bounded by the number of vertices in GRG (i.e., $|V|$). Therefore, it is not true that, $\forall i, RS^i(d) \subset RS^{i+1}(d)$. Consequently, within a finite steps, say k steps, the set may stop growing, that is,

$$\cdots \subset RS^{k-1}(d) \subset RS^k(d) = RS^{k+1}(d)$$

Since $RS^{k+1}(d) - RS^k(d) = \emptyset$, $RS^{k+2}(d) = RS^{k+1}(d) \cup RN(\emptyset) = RS^{k+1}(d)$. By induction, $RS(d) = RS^\infty(d) = RS^k(d)$. Therefore, $RS(d)$ will eventually be constructed in a finite number, say k, steps. Assumption 4.4 assures that these finite number of steps can be done in finite time. ■

Based on Invariant 6.1 and Lemma 6.2, we can prove following theorem which provides a local condition at the initiator d of a KDC when a knot is found by the computation.

Theorem 6.1 A knot $RS(d)$ is found if a d-computation observes $S_d = 1$.

Proof: By Invariant 6.1, $S_d = 1$ means $\sum_{\forall j} S_{P_j} = 0$. Since each probe should carry a positive valued strength, $\sum_{\forall j} S_{P_j} = 0$ implies $\{P_j\} = \emptyset$, which, in turn, means all probes are propagated back to the initiator and the computation has come to the end. Also, by Lemma 6.2, all vertices in $RS(d)$ are guaranteed to be visited at least once by the probes. Consequently, by the time $S_d = 1$, all the vertices in $RS(d)$ are visited at least once and all the probes are propagated back to the initiator, which, in turn, implies $RS(d) \subseteq TS(d)$. According to Definition 6.4, a knot $RS(d)$ is found since $RS(d) \subseteq TS(d)$. ■

Although Algorithm 6.0 is not practical due to the possible infinite loops in knots, the idea of declaring a knot as that described in Theorem 6.1 serves as the foundation of the algorithms developed in the following sections.

6.2.2 Knot Detection in Finite Time

There are at least two strategies to remedy the infinite loop problem from which Algorithm 6.0 suffers:

1. To remember the history of the probe propagation threads so that any loop could only appear once in a propagation thread.
2. To send probes backward to mark the vertices which belong to the $TS(d)$ so that a vertex $u \in TS(d)$ may send a forward probe which has been repeatedly revisiting u directly back to the initiator d to break the infinite loops.

To realize the first strategy, each probe has to keep track of its traveling thread history. The traveling thread is analyzed at each vertex when a probe is received. If there are loops in a probe's history, the information can be used to avoid the repetition of the same loop. For example, a probe which has traveled $\langle d \rightarrow \cdots \rightarrow a \rightarrow b \rightarrow c \rightarrow a \rightarrow b \rightarrow c\rangle$ will not be forwarded to vertex a again to avoid the repetition of the loop $\langle a \rightarrow b \rightarrow c \rightarrow a\rangle$.

The second strategy is basically inspired by Lemmas 6.1 and 6.2. Although a d-computation may not terminate in finite time, the $RS(d)$ could be completely reached by the forward probes in finite time. For the convenience of discussion, let's classify probes into two categories "primary" and "non-primary." In a d-computation, the first forward (or backward, as we are going to use them in this algorithm) probe received at a vertex u is recognized as a *primary* forward (or backward) probe by u; otherwise, it is a *non-primary* one. The vertices in $(RS^i(d) - RS^{i-1}(d))$ are newly explored at step i which, in turn, means they have just received the first probe from d-computation. Therefore, the probes found in $(RS^i(d) - RS^{i-1}(d))$ are the primary forward probes. As described in Lemmas 6.1 and 6.2, all vertices in $RS(d)$ could be reached by the primary forward probes in finite time. Similarly, in finite time, all vertices in $TS(d)$ could be reached by the primary backward probes. Therefore, in terms of exploring the sets $RS(d)$ and $TS(d)$, the propagation of non-primary probes is not necessary. The idea behind the second strategy is that a repeatedly received probe is a kind of non-primary probe which can be eliminated from further propagation in order to break the endless looping. Such a non-primary forward probe may be directly sent back to the initiator to carry out the probe strength computation if necessary.

Both strategies have their advantages and disadvantages. We adopt the second strategy for the following reasons. First, the delay of declaring a knot is in general longer in the first strategy than in the second one. This is because the second strategy allows a detection computation to execute in opposite directions concurrently. For example, if a probe computation is initiated in Figure 6.2 at vertex 18, the longest propagation thread by applying the first strategy is $\langle 18 \rightarrow 1 \rightarrow 2 \rightarrow 8 \rightarrow 10 \rightarrow 1 \rightarrow 2 \rightarrow 8 \rightarrow 10 \rightarrow 18 \rangle$. The vertex 18 may declare the knot after 9 propagation delays. On the other hand, if the second strategy is applied, the longest thread will be $\langle 18 \rightarrow 1 \rightarrow 2 \rightarrow 8 \rightarrow 10 \rightarrow 1 \rightarrow 18 \rangle$. It takes 5 hops to revisit vertex 1 and one more hop to directly go back to the initiator 18. Since the backward probe takes 4 hops to reach vertex 1, it is very likely that when a forward probe revisits vertex 1, 1 has already been marked as a member of $TS(18)$;

therefore, the revisit probe (a non-primary probe) can be immediately forwarded back to 1. Consequently, vertex 18 may declare a knot as soon as 6 propagation delays. Another reason to adopt the second strategy is that the backward probes can be used to reduce the number of vertices which report the existence of a knot to be exactly one. We will discuss this in detail later in Section 6.2.3. The improved algorithm which detects knots in finite time is as follows.

Algorithm 6.1

Convention: A probe (d, v, s, dir) carries four parameters. The first three parameters d, v, and s are the same as those in Algorithm 6.0. The fourth parameter dir defines the propagation direction of the probe. A dir can be either forward (F) or backward (B). The forward propagation F-probes are the same as those probes used in Algorithm 6.0. The backward propagation B-probes are used to mark the $TS(d)$ of the initiator d. The probe strength computation is carried out by the F-probes and is ignored (marked "x" which means don't care) in the B-probes. At each vertex u the probe information is recorded separately for each d-computation as follows. Two flags F_u^d and B_u^d are used to indicate if u has received the F- and B-probes from d-computation, respectively. A variable S_u^d at u is used for the temporary accumulation of the received F-probe strength from d-computation. Initially when a GRG is constructed, $\forall d, u \in V$, $F_u^d := FALSE$, $B_u^d := FALSE$, and $S_u^d := 0$.

In a GRG, a KDC (d-computation) is initiated at each vertex $d \in V$ as follows.

1. Initialization phase of a d-computation at vertex d:

   Send a B-probe (d, u, x, B) to every vertex $u \in TN(d)$;
   Send an F-probe $(d, v, 1/|RN(d)|, F)$ to every vertex $v \in RN(d)$.

2. For each vertex $v \neq d$, if v receives an F-probe (d, v, s, F) it does:

   if $(F_v^d = FALSE)$ **then** -- it is a primary F-probe
      $F_v^d := TRUE$;
      Send an F-probe $(d, w, s/|RN(v)|, F)$ to vertices $w \in RN(v)$;

```
else  -- it is not a primary F-probe
    if (B_v^d = TRUE) then
        -- a primary B-probe has been received
        Send an F-probe (d, d, s, F) to the initiator d;
    else  -- no B-probe has yet been received
        -- Accumulate the strength until a B-probe is received.
        S_v^d := S_v^d + s;
    end if;
end if.
```

3. For each vertex $u \neq d$, if u receives a B-probe (d, u, x, B) it does:

```
if (B_u^d = FALSE) then  -- it is a primary B-probe
    B_u^d := TRUE;
    Send a B-probe (d, w, x, B) to every vertex w ∈ TN(u);
    if (S_u^d > 0) then
        -- non-primary F-probes have been received
        Send an F-probe (d, d, S_v^d, F) to the initiator d;
        S_v^d := 0;
    end if;
else  -- it is not a primary B-probe
    null;  -- discard the received probe
end if.
```

4. After initiating a KDC, the initiator d starts collecting returned d-computation F-probes and accumulating the returned probe strength in S_d^d. If the accumulated probe strength $S_d^d = 1$, a knot is found. ◇

In Algorithm 6.1, the primary probes are distinguished from the other probes, and only primary probes are fully propagated in a d-computation. In the proof of Lemma 6.2, $(RS^i(d) - RS^{i-1}(d))$ defines a set of vertices which are newly explored by the first received forward probes. In other words, an F-probe may be found in $(RS^i(d) - RS^{i-1}(d))$ if and only if it is a primary one. Since the proof of Lemma 6.2 is based on the propagation of primary forward probes, it will still hold if all the non-primary forward probes are ignored when received. Thus, Lemma 6.2 can be rephrased for Algorithm 6.1 as follows.

Lemma 6.3 In a d-computation, every member of the $RS(d)$ is guaranteed to be reached by a primary F-probe in finite time.

Similarly, the following lemma holds for the B-probes.

Lemma 6.4 In a d-computation, every member of the $TS(d)$ is guaranteed to be reached by a primary B-probe in finite time.

Since non-primary probes are no longer allowed to continue propagating in the GRG, infinite loops can be eliminated. Since the size of a static GRG is finite, with Assumption 4.4 the following lemma holds.

Lemma 6.5 In a static GRG, any d-computation as described by Algorithm 6.1 will terminate (i.e., no more probe propagation activity) in finite time.

Proof: By Algorithm 6.1, in a d-computation, each vertex $u \neq d$ in GRG could appear at most once in the propagation thread of a primary F-probe and at most twice in a non-primary F-probe thread. Similarly, each vertex $u \neq d$ in GRG could appear at most once in a primary B-probe thread and at most twice in a non-primary F-probe thread. Since the size of a static GRG is finite, the length of any probe (primary or not) thread will be finite. The lemma is therefore proven with the assumption that each propagation delay is finite. ∎

In the worst case, the longest probe propagation thread would be a probe which has traveled through a directed Hamiltonian path, if such a path exists in GRG, revisits a vertex, and then is sent back to the initiator. Therefore, the length of probe thread is upper bounded by $|V| + 1$.

For the B-probe computation, the goal is to mark every member of $TS(d)$. Since this goal can be accomplished by the primary B-probes, the non-primary B-probes, if received, can be discarded immediately.

For the F-probe computation, the goals are twofold (1) to explore every member of $RS(d)$ and (2) to return to the initiator d to carry out the strength computation. The first goal can be accomplished by the primary F-probes; therefore, there

is no need to continue the propagation of the non-primary F-probes. To accomplish the second goal, all F-probes (primary and non-primary) are required to get back to d if they have been traveling within the range of $TS(d)$.

Lemma 6.6 A knot $RS(d)$ is found if and only if a d-computation terminates with $S_d^d = 1$.

Proof: The same as the probes in Algorithm 6.0, F-probes in Algorithm 6.1 only split or join at the initiator. Thus, Invariant 6.1 is still true for Algorithm 6.1 and, similarly, $S_d^d = 1$ means all F-probes are propagated back to the initiator.

By Lemma 6.3 and 6.5, in finite time, the whole $RS(d)$ will be explored by the primary F-probes. Similarly, by Lemma 6.4 and 6.5, in finite time, the whole $TS(d)$ will be marked by the primary B-probes. Also, by Lemma 6.5, within finite time, the d-computation will terminate which, in turn, implies all non-primary F-probes are either propagated back to d or simply stopped at a vertex $v \notin TS(d)$. Therefore, in finite time, when d-computation terminates, both $RS(d)$ and $TS(d)$ are completely explored and all the non-primary F-probes are propagated back to d if necessary.

A primary F-probe will eventually reach the initiator d if and only if every vertex it has visited is in $TS(d)$. A non-primary F-probe is directly sent back to its initiator d if and only if the vertex u it is visiting is a member of $TS(d)$ (which also implies all the vertices the probe has visited are in $TS(d)$). Also, all the vertices that can be reached by the F-probes define the set $RS(d)$. Therefore, in a d-computation all F-probes (primary and non-primary) may eventually find their way back to the initiator d if and only if $RS(d) \subseteq TS(d)$. Consequently, a d-computation may terminate with $S_d^d = 1$ if and only if $RS(d) \subseteq TS(d)$. Since by Definition 6.4 $RS(d) \subseteq TS(d)$ defines a knot $RS(d)$, the lemma is proven. ∎

Following the above lemmas, we can conclude that:

Theorem 6.2 In a static system, if a computation described in Algorithm 6.1 is initiated at every vertex in GRG, (1) all knots, if any, will be detected in finite time and (2) there will be no false detection of any knot.

Proof: Suppose there exist a knot K which contains at least a vertex d. Since Algorithm 6.1 is initiated at every vertex in GRG, it must be initiated at d. By Lemma 6.6, knot K is guaranteed to be declared as $RS(d)$ (when $S_d^d = 1$). Lemma 6.5 assures such a d-computation will detect the knot $RS(d)$ in finite time. This proves part (1) of the theorem. Since the system is static, whenever a vertex d reports $S_d^d = 1$, by Lemma 6.6 there exist a knot $RS(d)$. This proves part (2) of the theorem. Therefore, all knots and only genuine knots will be detected in finite time if Algorithm 6.1 is applied to every vertex in a static GRG. ■

6.2.3 Single Detection of Knots

In Algorithm 6.1 a KDC, say a d-computation, is required to be initiated at every vertex d in a static GRG in order to detect all the knots that exist in the graph. Each d-computation is executed independently and will declare a knot $K = RS(d)$ if it exists. In other words, knot K will be declared multiple times by all the vertices $d \in K$. The disadvantages of the multiple detection of a single knot is at least twofold: inefficiency and extra synchronizations required for deadlock recovery. It is inefficient in the sense that redundant messages are passed around to detect one knot many times. Also, since a knot may be detected by all its members, extra synchronizations are needed among the members to reach a consensus as when and how to recover from the deadlock. Yet another unbounded storage problem might occur later when the algorithm is extended to deal with dynamic systems. In Algorithm 6.1 the sizes of the storage for F_v^d, B_v^d, and S_v^d at each vertex v should be as large as the number of possible concurrent d-computations in a dynamic system. The size of a dynamic GRG could be arbitrarily large, and the execution time of the d-computation could be arbitrarily long. Although a garbage collection mechanism

may be used to clean up the out-of-date information stored in these buffers, their required sizes are still unbounded.

A widely used idea for the single detection of knots is to assign a unique ID (or priority, timestamp, etc.) to each vertex, say d, and its initiated d-computation (e.g., [68, 105], also [35, 84, 100, 114, 118, 126] have proposed similar methods for single detection of cycles). Whenever two d-computations encounter each other (i.e., their probes meet at a vertex), according to certain probe propagation rules which compare probe ID's, one of them may be dropped from further propagation. Suppose there is a knot K in GRG, eventually one and only one of the d-computations initiated at the vertices $d \in K$ is allowed to proceed to declare K. The goal of this section is to develop similar probe competition rules in order to refine the knot detection Algorithm 6.1 so that knots can be declared exactly once. Also, since only one d-computation is the final winner to declare a knot, one set of local storage at each vertex is sufficient to support the algorithm. This will help in eliminating the unbounded storage problems which might emerge in the next two algorithms for dynamic systems.

Each time that a probe competition rule is applied at a vertex v, based on v's local knowledge of the received probe ID's, a decision is made if a received probe should be dropped from further propagation. The effects of such a decision, however, are not only local to v but also are widely spread in that they may reach other places and affect other d-computations. Therefore, we need to carefully examine the behavior of probe propagations and competitions to develop a set of rules so that the effect of executing these rules is exactly a single detection of knots.

First, let's examine different types of d-computation competitions. A vertex v is said to be "in-knot" if its $RS(v)$ defines a knot; otherwise, v is "out-knot." A d-computation is an "in-knot" computation if it is initiated at an in-knot vertex; otherwise, it is an "out-knot" computation. According to Algorithm 6.1 a knot is supposed to be detected by its in-knot d-computations. Based on this classification, the following types of competitions are possible:

1. A competition among the in-knot d-computations may be:

   (a) an **intra-knot competition**: occur at a vertex v in a knot K, where all the involved d-computations are initiated in K, or

   (b) an **inter-knot competition**: otherwise (i.e., the competition occurs at an out-knot vertex or its involved d-computations are initiated in different knots).

2. A competition among the out-knot d-computations is referred to as an **out-knot competitions.**

3. A competition between in-knot and out-knot d-computations is referred to as an **in-knot-out-knot competition.**

The following properties help us understand where and how the results of these competitions might reach and affect other d-computations. By Definition 6.2, any member of a tie may only reach other members of the same tie. Therefore, we have the following property.

Property 6.3 There is no path from a vertex in a tie to a vertex outside the tie.

Since a knot is a special type of stable tie (see Property 6.1), Property 6.3 is also valid for knots.

Property 6.4 There is no path from a vertex in a knot to a vertex not in the knot.

Also, Property 6.1 implies that a knot contains no other knots. Therefore, it is easy to see that (1) there is no path from a vertex in a knot to a vertex in another knot and (2) there is no common part that may be shared between any two knots. Consequently, we have the following "Isolation Property" for the knots.

Property 6.5 (Isolation) Knots do not overlap and cannot be reached from each other.

Although Property 6.4 tells us that there are no outgoing paths from knots, there still may be incoming paths to knots. In Algorithm 6.1, for any knot, there are

no outgoing F-probes but there may be incoming ones. On the other hand, for any knot, there are no incoming B-probes but there may be outgoing ones. Also, the Isolation Property implies that the inter-knot competitions may never happen at an in-knot vertex. Therefore, the only situation in which an inter-knot competition might occur is one where the B-probes initiated in-knot meet with each other at an out-knot vertex. Since B-probes may never enter any knot, the effects of such inter-knot competitions (i.e., carried by the survived B-probes) may never reach and affect any knot. Since in Algorithm 6.1 only the in-knot probe propagations are the necessary and sufficient operations for knot detections, the effects of inter-knot competitions can be ignored. This result inspires the idea that each knot may be treated independently by applying the following rules:

Rule 6.1 *Prevent F-probes from entering into any knot from outside of that knot.* By applying this rule, there will be no in-knot-out-knot competitions at in-knot vertices. And the result of out-knot and in-knot-out-knot competitions occurring at the out-knot vertices will have no chance of reaching in-knot vertices. Therefore, the effects of both out-knot and in-knot-out-knot competitions can be ignored.

Rule 6.2 *Vote for a unique in-knot d-computation to complete the detection computation and declare the existence of the knot.* In the following algorithm we choose to let the computation with the larger ID to continue whenever two d-computations meet at a vertex. Therefore, by applying this rule, the intra-knot competitions in a knot K, if it exists, will vote for a d-computation with the largest ID to complete its computation and declare the knot K.

Note that the ID associated with each d-computation may be arbitrarily assigned. The only requirement is that each ID be unique. Here, in Rule 6.2, we may also choose the d-computation with the smallest ID to win the competition. However, in the dynamic systems to be discussed in the following sections, an ID for a d-computation is required to reflect its initiation logical timestamp. A commonly used method is to increment a logical time counter before its value is assigned to a new d-computation. When two ID's are compared, one of the concerns is to decide

which probe carries the more up-to-date information. Consequently, the probe with the smaller ID is usually the victim to be ignored.

Also note that by Rule 6.2, we can always let a d-computation continue if it has a larger ID, but we may not be able to drop a d-computation with a smaller ID immediately if it has arrived at a vertex earlier and has already been propagated. However, in the algorithm presented later, each vertex is assigned an ID, say d, and a presumed d-computation a priori. The total effect of the a priori assigned ID is that all the computations with a smaller ID will be dropped eventually.

There is no straightforward way of realizing Rule 6.1 because knots themselves are the targets to be detected. Before all knots in a GRG are found, we cannot say a probe is currently "out-knot" (it might be in an undetected knot) nor can we prevent such a "out-knot" (suppose it is really out-knot) from being propagated into any knot (there might be some undetected knots). However, Rule 6.1 may be accomplished by applying a necessary condition to the existence of a knot. By Definition 6.4, if a vertex u is in a knot K, it must be in a cycle because $u \in K = RS(u)$. In other words, a vertex is out-knot if it is out-cycle. A cycle is a necessary (but not a sufficient) condition of the existence of a knot. If F-probes are prevented from entering into any cycle from outside of that cycle, they will never enter into any knot. In a d-computation, if an F-probe is sent to a vertex u where a B-probe has visited, both the F-probe and u are currently in the same cycle. Therefore, Rule 6.1 can be modified and realized as:

Rule 6.1-A *Prevent F-probes from entering into any cycle from outside of that cycle.* In other words, in a d-computation an F-probe will only be propagated to where a primary B-probe has visited.

Rule 6.1-A also suggests that the voting process of Rule 6.2 is done primarily by the B-probes. This is because at any vertex an F-probe may only be received after a matched B-probe (i.e., with the same ID which, in turn, means from the same d-computation) is received. The competition among d-computations, therefore, is

primarily done by comparing B-probe ID's. A refined version of Algorithm 6.1, which detects every knot in a GRG exactly once, is as follows.

Algorithm 6.2

Convention: A probe (d, u, v, s, dir) means it is initiated at d and is currently transferred from u to v with strength s and direction dir. At each vertex u in the GRG the probe information for the currently involved d-computation is recorded as follows. A variable B_u is used to store the largest B-probe ID that has been known to u. All the smaller B-probe ID's are discarded, which means u will be only involved in the largest d-computation that has been known to it. Also, there is a flag F_u to indicate that if a primary F-probe from the involved d-computation has been received. If a primary F-probe is received at u, its strength is divided by the number of outgoing edges $|RN(d)|$ and is temporarily stored at the buffer P_u for further propagation. Also, for each outgoing edge $\langle u \rightarrow v \rangle$ at u, the largest B-probe ID received along the edge is stored in O_u^v. Once again, at the initiator d of each d-computation, a variable S_d is used to accumulate the strength of the returned probes. Originally when a GRG is constructed, each vertex $d \in V$ is presumably involved in its own d-computation; that is, initially, $B_d := d$, $F_d := TRUE$, $P_d := 1/|RN(d)|$, and $S_d := 0$. Also, initially, $\forall (u, v) \in E$, $O_u^v := 0$.

In a GRG, a KDC (d-computation) is initiated at each vertex $d \in V$ as follows.

1. Initialization phase of a d-computation at vertex d:

   if $(d = B_d)$ **then** -- d is ready to initiate a d-computation
       Send a B-probe (d, d, u, x, B) to every vertex $u \in TN(d)$.
   else -- d has already received a larger B-probe ID
       null; -- not to initiate another d-computation .... (6.2.1)
   end if;

2. For each vertex $w \in V$, if w receives a B-probe (d, v, w, x, B) from v it does:

```
if (d > B_w) then   -- it is a primary B-probe with a larger ID
    B_w := d;  F_w := FALSE;  P_w := 0;  O^v_w := d; ........ (6.2.2)
    Send a B-probe (d, w, u, x, B) to every vertex u ∈ TN(w);
elsif (d = B_w) then   -- it is a non-primary B-probe
    O^v_w := d;  -- B-probe ID's are propagated in ascending order
    if (F_w = TRUE) then   -- a matched F-probe is ready
        Send an F-probe (d, w, v, P_w, F) to v; ................. (6.2.3)
    end if;
else  -- smaller ID
    null;  -- discard the received probe ...................... (6.2.4)
end if;
```

3. For each vertex $w \neq d$, if w receives an F-probe (d, u, w, s, F) from u it does:

```
if (d = B_w) then   -- the ID is match
    if (F_w = FALSE) then   -- it is a primary F-probe
        F_w := TRUE;  P_w := s/|RN(w)|; .................. (6.2.5)
        Send an F-probe (d, w, v, P_w, F) to every vertex v
            where O^v_w=d; .................................. (6.2.6)
    else   -- it is not a primary F-probe
        Send an F-probe (d, w, d, s, F) to the initiator d; ....... (6.2.7)
    end if;
else  -- the ID is not match
    null;  -- discard the received probe ...................... (6.2.8)
end if;
```

4. After initiating a probe computation, the initiator d starts collecting returned F-probes (d, v, d, s, F) and accumulating the returned probe strength in S_d as long as the d-computation it is involved is still the one initiated by itself (i.e., $B_d = d$). If the accumulated probe strength $S_d = 1$, a knot is found. ◇

In Algorithm 6.2, Rule 6.1-A requires every vertex u to keep track of the largest B-probe ID received along each of its outgoing edges. In real systems, this information can be easily kept in the task and resource control tables. For brevity, we use O_u^v to store the information for each outgoing edge at u. Even in dynamic systems, the number of outgoing edges of u is fixed once it becomes blocked. Therefore, unlike the storage for F_v^d, B_v^d, and S_v^d at each vertex v in Algorithm 6.1, the size of the O_u^v is known when u becomes blocked. Also, O_u^v is always updated whenever a B-probe is received through the edge $\langle u \rightarrow v \rangle$ until u leaves the blocked state. No clean up mechanism for O_u^v is necessary.

Algorithm 6.2 is modified from Algorithm 6.1 based on the requirements of Rules 6.1-A and 6.2. Comparing these two algorithms, Algorithm 6.2 differs from Algorithm 6.1 as follows.

1. The only situations where an F-probe may be propagated are at lines (6.2.3), (6.2.6), and (6.2.7). Line (6.2.7) is used to handle non-primary F-probes. A primary F-probe is propagated from w to v only when (1) w receives a non-primary B-probe from v while a local matched primary F-probe is ready (see line (6.2.3)) or (2) w receives a primary F-probe while a matched B-probe has been received earlier from v (see line (6.2.6)). These observations provide enough evidence that Rule 6.1-A is faithfully realized; and the differences can be summarized as

   (a) Out-knot F-probes are stopped before entering into any cycle from outside of that cycle.

(b) The propagation of an in-knot F-probe from u to v may be postponed until a matched B-probe is received at v.

2. Line (6.2.1) asserts that a d-computation should not be initiated if a larger ID is known (i.e., from a reachable vertex). If a d-computation is initiated, its computation may be interrupted at the following situations: (1) A received probe is discarded if its ID is smaller than a previous one (see lines (6.2.4) and (6.2.8)). (2) At a vertex, its currently involved d-computation is overridden if a larger B-probe ID is received (see line (6.2.2)). To summarize, a d-computation may be interrupted

   (a) at the initialization stage when the initiator d find itself is currently involved in a detection computation with an ID $B_d > d$, or

   (b) when its F- and B-probes are overridden/discarded at vertices w where a larger B_w is found.

   These differences are due to the realization of Rule 6.2. The correctness of Lemmas 6.7 and 6.8 in the following proves that Rule 6.2 is faithfully realized.

In Algorithm 6.2 a probe strength is split and assigned to several "new" F-probes whenever an F-probe is to be propagated to several vertices (see line (6.2.5)). Although not explicitly stated in the algorithm code, the "new" F-probes are assumed to be temporarily "stored" at the vertex where the "old" one split. Some of the new F-probes may be propagated immediately at line (6.2.6) while the other F-probes may be waiting until line (6.2.3). The only situation that results in new F-probe being propagated occurs when a matched B-probe has been received from the edge on which the new F-probe is supposed to be sent. Although the algorithm does not keep track of which outgoing edges have propagated a new F-probe, there will be no duplicated probe strength propagated through any single edge. This is because in a d-computation, at most one B-probe may be propagated along each edge in a GRG. Consequently, suppose there are no discarded probes in a d-computation (i.e., d-computation is the winner under Rule 6.2), Invariant 6.1 is preserved by

Algorithm 6.2. The following Lemmas are proven based on the differences listed above and the conditional Invariant 6.1 preserved by Algorithm 6.2.

Lemma 6.7 By Algorithm 6.2 a knot, if it exists, will be detected at least once in finite time.

Proof: By the Isolation Property, knots do not overlap and cannot be reached from each other. Therefore, we can consider each knot independently. As we have discussed earlier, for any knot, F-probes may be coming in and B-probes may be going out. According to the difference 1(a) listed above, Rule 6.1-A prevents F-probes from entering any cycle from outside of that cycle which, in turn, can effectively prevent out-knot F-probes from entering any knot from outside of that knot. Also, the B-probes which exit from a knot can never come back or reach any other knot, hence, can be ignored. Consequently, we can restrict the proof within the scope of a single knot, say K, if it exists.

Suppose K is a non-empty set which is composed of $\{d_1, d_2, \ldots, d_m, d_n\}$, where the vertex ID's $d_1 < d_2 < \cdots < d_m < d_n$. The set of d-computation ID's which may be seen at any vertex in K is exactly the set K itself because they may be propagated by B- and F-probes to anywhere in K. If we try to initiate a detection computation for each vertex in K, at least d_n will succeed because none of the other d-computation ID's in K is larger than d_n. Since d_n is the largest ID in K, its probe propagations will never be interrupted by an even larger ID. Therefore, Rule 6.2 does not affect the d_n-computation which, in turn, means Invariant 6.1 holds for the d_n-computation by Algorithm 6.2.

Suppose, for comparison, another d_n-computation is initiated and executed independently by Algorithm 6.1. The only difference left is that, in Algorithm 6.2, due to Rule 6.1-A the propagation of an F-probe from d_i to d_j may be postponed until a primary B-probe is received at d_j. Since the size of K is finite, primary B-probes can reach any vertex in K in finite time (Lemma 6.4 applies here). The extra delay due to Rule 6.1-A, therefore, is finite. By Theorem 6.2, Algorithm 6.1 guarantees that d_n can declare knot K

in finite time. Taking this extra finite delay into account, d_n can still declare knot K in finite time by executing Algorithm 6.2. Consequently, knot K will be declared at least by d_n in finite time. ■

__Lemma 6.8__ By Algorithm 6.2 a knot, if it exists, will be detected at most once.

Proof: This Lemma can be proven by contradiction. Following from the proof of Lemma 6.7, knot K should at least be declared by d_n. Suppose, in K, there is another d_i-computation initiated at $d_i < d_n$ may also declare the knot K. Every vertex in K, including d_n, will be visited by B- and F-probes from d_i-computation in finite time. Initially B_{d_n} equals d_n and may only be increased whenever a higher ID is received. All the B-probes from the d_i-computation will be discarded at d_n because its ID $d_i < d_n \leq B_{d_n}$. Therefore, no F-probe from d_i-computation is able to reach d_n (Rule 6.1-A). At least one primary F-probe of d_i-computation will be stopped before reaching d_n. Consequently, S_{d_i} will never reaches one which, in turn, means d_i will never declare the knot K. ■

Combine Lemmas 6.7 and 6.8, we can conclude the following theorem.

__Theorem 6.3__ In a static system, if we initiate a computation described in Algorithm 6.2 at every vertex in GRG,

(1) all knots, if any, will be detected exactly once in finite time and

(2) there will be no false detection of any knot.

Proof: The statement (1) of the theorem is true due to Lemmas 6.7 and 6.8. In the proof of these two lemmas, Invariant 6.1 holds for the d_n-computation (the very one which declares knot K) by Algorithm 6.2. Similar to the reasoning of Lemma 6.6, a knot $K = RS(d_n)$ exists if d_n-computation terminates with $S_{d_n} = 1$. This proves part (2) of the theorem. ■

6.3 Knot Detection in Dynamic Systems

First, let's summarize characteristics of dynamic systems in which OR deadlocks may be found. In a dynamic system the GRG may be updated dynamically while the knot detection computations are running. In such a DGRG, a vertex is in the "waiting" state if there are outgoing edges; otherwise, it is "free." A task vertex is waiting if it is blocked and is waiting for its outstanding requests to be granted. A resource vertex is waiting if all its units are assigned and is waiting to be relinquished by the holders. On the other hand, a task vertex is free if it is active, and a resource vertex is free if there are available units. A set of edges $\{\langle u \rightarrow v_i \rangle\}$ are added to the DGRG if a vertex u starts to wait for a set of vertices $\{v_i\}$. Examples are: a task vertex starts waiting when initiating a set of requests but none of the requests may be satisfied immediately; a consumable resource vertex starts waiting for a set of producers to satisfy the request when it is created; and a reusable resource vertex starts waiting for its holders to release its units when the last unit is assigned. Once u becomes waiting, no more outgoing edges from u will be added to the DGRG. One of the outgoing edges $\{\langle u \rightarrow v_i \rangle\}$ may be deleted by a free vertex v_i which, in turn, will cause the waiting vertex u to delete its whole set of outgoing edges.

Since in a knot, vertices are waiting for each other and none of them is free to satisfy (i.e., to delete) others' waiting needs (i.e., outgoing edges), the whole group of waiting situations will persist. Therefore, we have the following *Stable Property* for knots in dynamic systems.

Property 6.6 (Stable) Once a knot is formed in a dynamic system, it persists until it is detected and resolved.

The stable property is the foundation of the methodology developed in Chapter 4. In the following sections, synchronization mechanism of dynamic algorithms are developed based on the guidelines suggested by that methodology. Algorithms 6.1 and 6.2 are then extended for dynamic systems (Algorithms 6.3 and 6.4).

6.3.1 A Guaranteed Knot Detection Algorithm for Dynamic Systems

In this section, we develop a dynamic algorithm Algorithm 6.3 based on Algorithm 6.1 using the methodology suggested in Chapter 4. Due to an efficiency concern, a minor improvement is then made in Algorithm 6.3. Following the guideline described in Section 4.4.2.2, a modified version of Algorithm 6.1 (Algorithm 6.1′) is used to simulate the projected KDC's due to Algorithm 6.3. The correctness of Algorithm 6.1′ is analyzed without a formal proof since it is very similar to that of Algorithm 6.1. The developed Algorithm 6.3 should satisfy both Spatial and Temporal consistency criteria and each of its KDC's should be equivalent to a legal execution of Algorithm 6.1′. Similar to Algorithm 6.1, both Algorithm 6.1′ and Algorithm 6.3 allow an individual KDC to be treated independently without any interference from other KDC's. Therefore, in the following discussions, we focus on the interactions between a KDC and the underlying DGRG.

To maintain the meaningfulness of a KDC, Theorem 4.1 requires that at each vertex (1) the detection of white edges should be avoided and (2) the connectivity of the adjacent operations should be ensured. We will first deal with the problem of white edges. The connectivity concern will be discussed in the cases when probes are propagated.

In the OR deadlock model, an edge may be deleted in the forward direction (e.g., a waiting vertex may become free after it is granted one of its requests and, then, withdraw all other requests) or in the backward direction (e.g., a free vertex may grant other's requests). Therefore, both F-probes and B-probes may be received from white edges. Based on Assumptions 4.1 and 4.3, the only situation where a white edge may be detected is when an edge is being canceled from one end of the edge and, almost at the same time, a probe is being sent from the other end. Since each vertex could keep track of its incoming and outgoing edges locally, the receiving end should always know if a probe was from an unknown edge (i.e., a white edge) which, in turn, can eliminate the detection of white edges. This idea can be realized by the following rule.

Rule 6.3 Each vertex keeps track of its incoming and outgoing edges locally and when a probe is received from an unknown edge, it is discarded immediately.

In a static GRG, each vertex has an invocation of Algorithm 6.1. Similarly, Algorithm 6.3 could be applied to dynamic systems as follows. In a DGRG, suppose at time t_{d_i} the state of a vertex d is changed from free to waiting. Within a finite delay Δ_{d_i}, a KDC ((t_i, d)-computation[4]) is initiated at time t_i for vertex d.

As required in Algorithm 6.1, a set of storages F_u^{d,t_i}, B_u^{d,t_i}, and S_u^{d,t_i} should be pre-allocated and initialized at every vertex $u \in V$ whenever a (t_i, d)-computation is initiated. This can be done easily in static systems but is very difficult to do in dynamic systems. In a dynamic system, such storage can be dynamically allocated and initialized when a vertex u receives a primary B- or F-probe from (t_i, d)-computation.

According to Algorithm 6.1, each probe is processed immediately after it is received. The result of processing each probe is final when the GRG is static. For example, a probe propagation "terminates" when it reaches a leaf vertex (i.e., a free vertex u with $RN(u) = \emptyset$ or a waiting vertex v with $TN(v) = \emptyset$). However, in a DGRG, the situation at each vertex may be dynamically changing. Consider the situation that, at a vertex v, a probe is received and later v may have new edges built to connect to neighbors (e.g., becomes waiting if it was free or starts to be waited on by other vertices). Theorem 4.2 suggests that as long as a probe received at v is meaningful, it should be allowed to continue its propagation after v has built new connections. If the continuation of the propagations are allowed when vertices connect to new edges, the connectivity of the adjacent probe propagation operations in such situations need to be examined to keep a KDC meaningful. If the connectivity of the adjacent operations are well maintained, Algorithm 6.3 can generate KDC's which are equivalent to the legal executions of Algorithm 6.1. Thus, the following rule may be applied to a dynamic algorithm.

[4]Here a (t_i, d)-computation is similar to a d-computation in the static systems. Since in a dynamic system a vertex d may initiate KDC's multiple times, the notation (t_i, d)-computation is used to distinguish them from each other.

Rule 6.4 *In a dynamic system, probes are saved after they are processed at each vertex. If new edges are connected to a vertex, the saved probes may continue their propagation if necessary.*

However, to maintain information for the examination of the connectivity of the adjacent operations incurs overhead at the free vertices. An alternative solution is to render a KDC obsolete whenever the connectivity requirement is not met. We will come to this solution later.

One of the situations which may occur in dynamic systems but does not exist in the static systems is that vertices in waiting state may become free. When a vertex u is in the waiting state, it may be involved in several (t_i, d)-computations. A set of storages is arranged at u to keep track each of its involved (t_i, d)-computations. When u goes free, it becomes a leaf vertex (or becomes an isolated vertex if no vertices are waiting for it) and is not effectively involved in any KDC. All of the probes previously propagated through u become meaningless. This is because the operations previously triggered at u become meaningless and, therefore, the probes propagated by these operations are meaningless. As far as Theorem 4.1 is concerned, nothing need be done beyond the deletion of u's outgoing edges in DGRG (by sending control messages to $RN(u)$). Since these operations are known to be meaningless and, according to Theorem 4.2, their information should no longer be used, the following rule should be enforced in the algorithm.

Rule 6.5 *Abandon all of the currently involved KDC's when a vertex becomes* free *from a* waiting *state.* In a dynamic system, when a vertex u becomes free from a waiting state, all of the old records kept at u for its previously involved (t_i, d)-computations could be deleted and their storage spaces could be de-allocated.

Another side effect of Rule 6.5 is that it solves the problem of the connectivity of the adjacent probe propagations without requiring a complicated mechanism to examine the status of each saved probe. First, let's consider the B-probes. The B-probes are always received and processed at waiting vertices. The connectivity of two

adjacent B-probe propagations is directly ensured by Rule 6.5. Then, let's consider the F-probes. Two adjacent F-probe propagations over the edges $\langle a \rightarrow b \rangle$ and $\langle b \rightarrow c \rangle$ may not be connected if the latter is triggered after the former edge is deleted. The deletion of the edge $\langle a \rightarrow b \rangle$ means the vertex becomes free which, in turn, means that a can reach free vertices while it is waiting. Therefore, for each KDC K in which a was involved, part of its probe strength is destroyed when a abandons it. This violates Invariant 6.1 and K is rendered obsolete although its meaningless probes may still continue their propagations through vertex b. Consequently, by Theorem 4.1, Rules 6.4 and 6.5 ensure that every detected knot must be a genuine one.

With Rule 6.4, there is an efficiency improvement that could be made to Algorithm 6.3. Since both primary and non-primary F-probes may be saved at a leaf vertex while it is free, the probe strength could be accumulated together and propagated as the strength of a primary probe later when the vertex becomes waiting. This modification simplifies the algorithm as well as improves its efficiency in terms of reducing non-primary F-probes. However, with this modification, Algorithm 6.1 needs to be modified to simulate the projected KDC's due to the improved Algorithm 6.3. Algorithm 6.1′ is derived from Algorithm 6.1 by delaying the propagation of the primary F-probes at each vertex to simulate the situation that the vertex is in the free state. While a primary F-probe is waiting at a vertex for further propagation, it collects the strength of received non-primary F-probes. A non-primary F-probe is discarded immediately if its strength is transferred to a primary F-probe. Each delay time is a randomly selected finite duration.

The modification in Algorithm 6.1′ may reduce the number of messages required for sending the non-primary F-probes back to the initiator of a KDC but it may also delay the declaration of a knot. However, the latter disadvantage is not the case in Algorithm 6.3 since the "delay" is the time while a primary F-probe is waiting at a free leaf vertex where no outgoing edges exist.

Algorithm 6.1′ can be easily proven correct since the modification only adds finite delay time to the F-probe propagations (i.e., Lemma 6.3 still holds) and

does not violate Invariant 6.1. Also, Lemmas 6.5 and 6.6 still hold after taking into account that some of the non-primary probes may be merged with the primary probes. Consequently, Algorithm 6.1′ can be proven in a similar way as Theorem 6.2.

The following algorithm consists of two parts: (1) the synchronization mechanism which deals with dynamically changing nature of the underlying system and (2) the description of a KDC. The synchronization mechanism in part (1) is derived from Rules 6.3, 6.4, and 6.5. The KDC described in (2) is basically the same as that specified in Algorithm 6.1 with the modification mentioned earlier.

Algorithm 6.3

Convention: A probe (t, d, v, s, dir) carries five parameters. The first parameter t is the timestamp when the computation is initiated. The other four parameters d, v, s, and dir are the same as those in Algorithm 6.1. At each $u \in V$ a pool of storages are reserved for the following data structures used for keeping track of each (t_i, d)-computation which u has been involved. These data structures are dynamically allocated when the first probe arrives at u from a (t_i, d)-computation. Two flags, $F_u^{t_i,d}$ and $B_u^{t_i,d}$, are used to indicate if u has received F- and B-probes from (t_i, d)-computation, respectively. If a flag is not found at vertex u (i.e., not allocated), its value is assumed to be *FALSE* for brevity. Similarly, a dynamically allocated variable $S_u^{t_i,d}$ at u is used for temporary accumulation of the received F-probe strength from (t_i, d)-computation. If $S_u^{t_i,d}$ is not found at u, its value is assumed to be zero. The state of each vertex $u \in V$ is recorded in $ST(u)$ with a value of waiting or free.

Synchronization Mechanism:

- In a DGRG, if a vertex d becomes blocked:
  1. Construct a set of reachable neighbors $RN(d)$.
  2. Send a control message to every vertex $v \in RN(d)$ to register d in $TN(v)$.

3. Set its state $ST(d)$:= waiting (at time t_{d_i}).

4. At d, for each previously received F-probe (i.e., for each $u \neq d$, where its $S_d^{u,t_j} \neq 0$ and flag $F_d^{u,t_j} = TRUE$), send an F-probe $(t_j, u, v, S_d^{u,t_j}/|RN(d)|, F)$ to every vertex $v \in RN(d)$ and then reset $S_d^{u,t_j} := 0$.

5. Send a control message to every vertex $v \in RN(d)$ to poll B-probes (i.e., probes with flag $B_v^{u,t_j} = TRUE$, where $u \neq d$). Upon receiving this polling message, vertex v will propagate all its B-probes to d.

- If one of the requests of a vertex d is satisfied:

  1. Send a control message to every vertex $v \in RN(d)$ to de-register d in $TN(v)$.

  2. De-allocate all F_d^{u,t_j}, B_d^{u,t_j} and S_d^{u,t_j} found at d.

  3. Set the state $ST(d)$:= free.

- When a probe is received from an unknown edge, it is discarded immediately.

Description of a KDC:

1. With a finite delay Δ_{d_i} after t_{d_i}, a KDC ((t_i, d)-computation) is initiated at time t_i as follows.

   $S_d^{t_i,d}$:= *0*; -- allocate and initialize $S_d^{t_i,d}$
   Send a B-probe (t_i, d, u, x, B) to every vertex $u \in TN(d)$;
   Send an F-probe $(t_i, d, v, 1/|RN(d)|, F)$ to every vertex $v \in RN(d)$.

2. For each vertex $v \neq d$, if v receives an F-probe (t_i, d, v, s, F):

   if ($F_v^{t_i,d}$ = *FALSE*) **then** -- it is a primary F-probe
       $F_v^{t_i,d}$:= *TRUE*; -- allocate and initialize $F_v^{t_i,d}$
       $S_v^{t_i,d}$:= *0*; -- allocate and initialize $S_v^{t_i,d}$
       if ($ST(v)$ = waiting) **then**
           Send an F-probe $(t_i, d, w, s/|RN(v)|, F)$
               to every vertex $w \in RN(v)$;

```
            else  -- free and has no RN(v)
                S_v^{t_i,d} := s;
            end if;
        else  -- it is not a primary F-probe
            if (B_v^{t_i,d} = TRUE) then
                -- a primary B-probe has been received
                Send an F-probe (t_i, d, d, s, F) to the initiator d;
            else  -- no B-probe has yet been received
                -- Accumulate the strength.
                S_v^{t_i,d} := S_v^{t_i,d} + s;
            end if;
        end if.
```

3. For each vertex $u \neq d$, if u receives a B-probe (t_i, d, u, x, B):

```
    if (B_u^{t_i,d} = FALSE) then  -- it is a primary B-probe
        B_u^{t_i,d} := TRUE;  -- allocate and initialize B_u^{t_i,d}
        Send a B-probe (t_i, d, w, x, B) to every vertex w ∈ TN(u);
        if (S_u^{t_i,d} > 0) then
            -- non-primary F-probes have been received
            Send an F-probe (t_i, d, d, S_v^{t_i,d}, F) to the initiator d;
            S_v^{t_i,d} := 0;  -- S_v^{t_i,d} can be de-allocated here
        end if;
    else  -- it is not a primary B-probe
        null;  -- discard the received probe
    end if.
```

4. After initiating a knot detection computation, the initiator d starts collecting returned (t_i, d)-computation F-probes and accumulating the returned probe strength in $S_d^{t_i,d}$. If the accumulated probe strength $S_d^{t_i,d} = 1$, a knot is found. ◇

Note that in Algorithm 6.3, there is a finite delay, say Δ_{d_i}, allowed between when a vertex d_i goes to waiting at t_{d_i} and when a (t_i, d)-computation is initiated. There are two reasons that this finite delay Δ_{d_i} is necessary. First, it is to recognize the fact that the two events always occur in series. Second, time is always needed to satisfy a vertex's outstanding request in a distributed environment. In a real

system, there is a minimum delay between the time a vertex starts waiting and the time its request could be granted. In some systems, a minimum delay time Δ_{d_i} may be implemented as a performance parameter.

According to the dynamic system model described at the beginning of Section 6.3, there are four types of system activities in a DGRG, i.e., (1) the creation of a vertex, (2) the deletion of a vertex, (3) the change of the state of a vertex from free to waiting, and (4) the change of the state of a vertex from waiting to free. It could be assumed that a vertex is always created as an isolated free vertex and the deletion of a vertex can only take place when the vertex becomes free and isolated. Therefore, only the state changes of the vertices in a DGRG need to be considered. This confirms that the synchronization mechanism in Algorithm 6.3 does cover all the possible situations that could happen in a DGRG.

To prove the correctness of the algorithm, we need to argue that: (1) any existing knot will be detected eventually, and (2) a detected knot must be a real one. These are established in the following theorem.

Theorem 6.4 In a dynamic system, if a (t_i, d)-computation described in Algorithm 6.3 is initiated at a vertex $d \in V$ within a finite delay at time t_i after d starts waiting, (1) there will be no false detection of any knot, and (2) any existing knot will be detected in finite time.

Proof: It is easy to see in the development of Rules 6.3, 6.4, and 6.5 that Algorithm 6.3 has been kept as close to Algorithm 6.1 as possible. The three rules are used to deal with dynamic situations and to ensure that Algorithm 6.3 satisfies Spatial and Temporal consistency criteria. When designing the three rules, we have carefully examined them so that the projected KDC's due to these rules are equivalent to the legal executions of Algorithm 6.1 in static states. The efficiency improvement made to Algorithm 6.3 can be simulated by a modification in Algorithm 6.1′. The verification of Algorithm 6.1′ is very similar to that of Algorithm 6.1 as we have described. Consequently, we have a correct Algorithm 6.1′ which could simulate every possible projected KDC's due to the execution of Algorithm 6.3 in dynamic systems.

This, in turn, means Algorithm 6.3 can correctly recognize knots in terms of meaningful operations in the projected KDC's. By Theorem 4.1, Rules 6.3 further ensures that Algorithm 6.3 may only detect genuine knots. Part (1) of the theorem is, therefore, proven.

Suppose there is a knot K which exists in a DGRG. Knot K is a finite non-empty set of vertices $\{d_i\}$, where $i = 1, \ldots, n$. Suppose that each vertex d_i starts waiting and becomes a member of K at time t_{d_i}, where $t_{d_1} \leq t_{d_2} \leq \cdots \leq t_{d_n}$. When the youngest member d_n becomes waiting, it is already in the existing knot K. By the Stable Property, K can be treated as a static sub-GRG after t_{d_n}. According to Algorithm 6.3, a (t_n, d_n)-computation is initiated at d_n at time t_n with a finite delay after t_{d_n}. By Theorem 4.2, at least the (t_n, d_n)-computation could detect the knot K in finite time. Hence, part (2) of the theorem is proven. ■

The idea behind Rules 6.4 and 6.5 is that all probes are allowed to continue in dynamic systems except the ones that are known to be meaningless. In other words, every meaningful KDC could freely continue its probe propagation until it finds a knot. Theorem 4.2 suggests the use of this policy to give every meaningful KDC a chance to continue to find an existing knot. On the other hand, Theorem 6.4 has proven that there exist at least one such KDC and, therefore, an existing knot is guaranteed to be declared in finite time.

A KDC may become inactive (including termination) if all of its probes cease propagating due to the following reasons: (1) a probe returns to the initiator, (2) a probe is discarded because it is found meaningless (e.g., due to Rule 6.5), or (3) a probe reaches a leaf vertex in the system. If all the probes of a KDC are stopped due to (1) or (2), it is permanently terminated; otherwise, it may resume its computation later if some of the probes are in the situation (3). All the data storage of a permanently terminated KDC are de-allocated as described in Algorithm 6.3. The meaningless probes in situation (3) may eventually be cleaned up if the leaf vertices where they are located change from the waiting state to the free state (i.e., due to Rule 6.5) or are deleted in the free state. If a leaf vertex remains in a

free state for a long time, it may need a large heap of storage for the KDC's that ever reached it between its two free states; otherwise, the algorithm may fail due to storage overflow at some vertices. Therefore, there is no upper bound for the size of this reserved storage at each vertex. Alternatively, the cleaning process for the meaningless probes under situation (3) could be speeded up by a periodically invoked garbage collection mechanism or by clean messages spread out from the vertices where probes are found meaningless. Although these mechanisms may reduce the frequency of storage overflow problems, they do incur an overhead.

6.3.2 Single Detection of Knots in Dynamic Systems

In this section, we develop a dynamic algorithm based on Algorithm 6.2. As discussed in Section 6.2.3, we need to rule out some of the possible interferences between KDC's. The Isolation property (Property 6.5) states that, in a static GRG, knots do not overlap and cannot be reached from each other. In dynamic systems if both Spatial and Temporal Consistency criteria are fulfilled, the Isolation property can be rephrased as follows:

Lemma 6.9 In a dynamic system, knots do not overlap and cannot be reached from each other in terms of meaningful operations generated by a dynamic algorithm which satisfies Spatial and Temporal consistency criteria.

Proof: Suppose we have a dynamic algorithm, say $ALGO_D$, which satisfies the Spatial and the Temporal consistency criteria. The projected KDC's due to $ALGO_D$ should be equivalent to some KDC's generated by a static algorithm, say $ALGO_S$. Here, we don't care if $ALGO_S$ can correctly detect knots. The only thing that concerns $ALGO_S$ is that it propagates probes which can simulate projected KDC's due to $ALGO_D$.

Suppose in a system state ST^f, a dynamic KDC $DKDC_D$ is projected as $SKDC_D$. $SKDC_D$ is equivalent to a legal execution of $ALGO_S$, say $SKDC_S$. Since Property 6.5 could be applied to any static KDC generated by $ALGO_S$ in ST^f, it also holds for any projected KDC such as $SKDC_D$

in ST^f. $SKDC_D$ consists of all the meaningful operations in $DKDC_D$ that could be found in ST^f. Consequently, in ST^f, knots do not overlap and cannot be reached from each other in terms of meaningful operations found in the projected KDC's such as $DKDC_D$. Since this is true in an arbitrarily composed example, it is true in general that in a dynamic system, knots do not overlap and cannot be reached from each other through meaningful operations generated by a dynamic algorithm which satisfies Spatial and Temporal consistency criteria. ■

Since Rules 6.1-A and 6.2 are based on the static Isolation property (Property 6.5), Lemma 6.9 implies that, if a derived dynamic Algorithm 6.2 satisfies Spatial and Temporal consistency criteria, these rules should still apply. In other words, a knot will be declared exactly once if a synchronization mechanism properly maintains Spatial and Temporal consistency criteria.

The set of Rules 6.3, 6.4, and 6.5 derived in Section 6.3.1 provide a good mechanism to deal with the dynamically changing nature of the underlying system. Rule 6.3 prevents any operation in a KDC from detecting a white edge. This rule does not interfere with Rules 6.1-A and 6.2 and, hence, can be directly applied to Algorithm 6.4. However, there is a little concern that Rules 6.4 and 6.5 might affect the effectiveness of Rule 6.2. Rule 6.4 suggests that a meaningful KDC should be allowed to continue when new edges are attached to the existing graph. Rule 6.5 suggests that a vertex should abandon all of its currently involved KDC's when it goes free from a waiting state. Both of them may conflict with Rule 6.2. How these three rules are realized might affect the validity of the algorithm.

In dynamic systems, a vertex might invoke multiple KDC's during its lifetime. It is not sufficient to uniquely identify a KDC by its initiator's ID as has been done in Algorithm 6.2. In addition, each KDC needs a timestamp as part of its ID to distinguish itself from other KDC's initiated at the same vertex. The timestamp could be a real-time or a logical clock which specifies the causal relations between KDC's. Since there is usually no common global clock in distributed environments,

a logical clock is used in our algorithm. We need a protocol to generate logical clocks for KDC's when they are initiated. This logical clock protocol is an attempt to realize Rules 6.4 and 6.5 while retaining the effectiveness of Rule 6.2.

When a vertex d becomes waiting, a new KDC is initiated and a logical timestamp t is generated to go with it. The ID of a KDC is now a pair (t, d) where t is the logical timestamp and d is its initiator's ID. To find the greater ID between two KDC's, their logical timestamps are compared first. The KDC with a smaller ID is logically initiated before a KDC with a greater one. The initiator's ID is used to determine a total ordering when two KDC's have the same logical timestamp. In terms of meaningful KDC's, as long as their ID's are uniquely assigned and a total ordering can be determined, Rule 6.2 is satisfied. Combined with Rule 6.3, Rules 6.4, and 6.5 are responsible for the meaningfulness of KDC's.

One of the characteristics of Algorithm 6.2 is that only B-probes are involved in the voting process and F-probes may be received at a vertex only if a primary B-probe of the same KDC has been received (Rule 6.1-A). Therefore, only B-probes coming from $RN(d)$ are considered in the process of generating a new logical timestamp whenever a vertex d becomes waiting. Since no B-probes may be received while d is free, to fulfill Rule 6.5, d could override all its previously involved KDC's when it becomes waiting. In other words, d should be involved in a new KDC (either d initiates a new one or it joins in an existing one which is meaningful at d) with a greater ID when it becomes waiting.

To fulfill Rule 6.4, no higher logical timestamp need be created provided d can find any existing meaningful KDC via B-probes received from $RN(d)$ after it becomes waiting. This may be sufficient in implementing Rule 6.4 but is not necessary if we plan a little bit ahead. Suppose among the larger B-probes collected at d that (1) the one from u carries the largest ID but is meaningless and (2) the one from v is the largest meaningful B-probe. We may let the KDC from v continue its propagation through d. However, there are two possible situations that u may turn to eventually: (1) grants d's request and renders the KDC obsolete at v or (2) becomes waiting and initiates or propagates a B-probe to override the KDC

from v. In any case, the KDC from v will become meaningless in the future. We may (a) wait for u's result or (b) initiate an even larger KDC at d. Approach (a) can reduce the number of probe message but it tends to delay the declaration of knots. On the other hand, approach (b) may cost more probe message overhead, but knots may be detected much earlier than approach (a).

Consequently, to fulfill all three rules (i.e., Rules 6.2, 6.4, and 6.5), d should initiate a new KDC only if it finds that the largest KDC ID in the set $\{d\} \cup RN(d)$ is meaningless. If that is the case, the new logical timestamp of the initiated KDC is the increment of this largest ID. Based on this approach, rules developed in the previous two algorithms can be realized in Algorithm 6.4 as follows.

Algorithm 6.4

Convention: A probe (t, d, u, v, s, dir) means it is initiated at d and is currently transferred from u to v with strength s and direction dir. The first parameter t is the timestamp when the computation is initiated. The pair (t, d) is the ID of the computation (i.e., (t, d)-computation). The timestamp t is a logical clock which defines the causal relation between KDC's. If two KDC carries the same timestamp t, a total ordering is determined by their initiator's ID d which is unique in the whole system. At each vertex u in GRG the probe information for the currently involved (t, d)-computation is recorded as follows. A variable K_u stores the latest KDC ID initiated at u. A variable B_u is used to store the largest B-probe ID has been known to u. All the smaller B-probe ID's are discarded, which means u will be only involved in the largest d-computation has been known to it. Flag F_u indicates that if a primary F-probe from the involved (t, d)-computation has been received. If a primary F-probe is received at u, its strength is divided by the number of outgoing edges $|RN(d)|$ and is temporarily stored at the buffer P_u for further propagation. Also, a flag V_u is used to show if the recorded B-probe is valid. A B-probe from a free vertex is invalid. An invalid probe is marked $dir = INVALID$. For each outgoing edge $\langle u \rightarrow v \rangle$ at u, the largest B-probe ID received along the edge is stored in O_u^v. At

the initiator d of each d-computation, a variable S_d is used to accumulate the strength of the returned probes. Initially when a vertex v is created, $K_v := (0,0)$, $B_v := (0,0)$, $V_v := INVALID$, $F_d := FALSE$, $P_d := 0$, and $S_d := 0$. Also, $\forall(v,u) \in E$, $O_v^u := (0,0)$.

Synchronization Mechanism:

- In a DGRG, if a vertex d becomes blocked:
  1. Construct a set of reachable neighbors $RN(d)$.
  2. Send a control message to every vertex $v \in RN(d)$ to register d in $TN(v)$ and to poll B-probes. Upon receiving of this control/polling message, a vertex v will propagate its B-probe to d if $ST(v)$ = waiting; otherwise, v will send its latest B-probe marked $dir := INVALID$ if $ST(v)$ = free.
  3. Collect all the B-probes from $v \in RN(d)$ and record them in O_d^v if the received probe is valid (i.e., $dir = B$). Compare all the received probe ID including the invalid ones with B_d and put the largest ID in B_d. If this largest ID belongs to a valid B-probe, set $V_d := TRUE$.
  4. Set its state $ST(d)$:= waiting (at time t_{d_i}).
- If one of the requests of a vertex d is satisfied:
  1. Send a control message to every vertex $v \in RN(d)$ to de-register d in $TN(v)$.
  2. Set $F_d := FALSE$ and $V_d := INVALID$.
  3. Set the state $ST(d)$:= free.
- When a probe is received from an unknown edge, it is discarded immediately.

Description of a KDC:

1. With a finite delay Δ_{d_i} after t_{d_i}, the vertex d does:

```
if (V_d = INVALID) then
    -- d is ready to initiate a KDC
    t_i := B_d.t + 1;  -- increment the logical clock
    B_d := (t_i,d);  K_d := B_d;
    F_d := TRUE;  P_d := 1/|RN(d)|;
    S_d := 0;  V_d = TRUE;
else  -- d has already received a larger B-probe ID
    null;  -- not to initiate another d-computation
end if;
Send a B-probe (t_i, d, d, u, x, B) to every vertex u ∈ TN(d).
```

2. For each vertex $w \in V$, if w receives a B-probe (t_i, d, v, w, x, B) from v:

```
if ((t_i, d) > B_w) then
    -- it is a primary B-probe with a larger ID
    B_w := (t_i, d);  F_w := FALSE;
    O_w^v := (t_i, d);  P_w := 0;
    Send a B-probe (t_i, d, w, u, x, B) to every vertex u ∈ TN(w);
elsif ((t_i, d) = B_w) then  -- it is a non-primary B-probe
    O_w^v := (t_i, d);  -- B-probe ID's are propagated in ascending order
    if (F_w = TRUE) then  -- a matched F-probe is ready
        Send an F-probe (t_i, d, w, v, P_w, F) to v;
    end if;
else  -- smaller ID
    null;  -- discard the received probe
end if;
```

3. For each vertex $w \neq d$, if w receives an F-probe (t_i, d, u, w, s, F) from u:

```
if ((t_i, d) = B_w) then  -- the ID is match
    if (F_w = FALSE) then  -- it is a primary F-probe
        F_w := TRUE;  P_w := s/|RN(w)|;
        Send an F-probe (t_i, d, w, v, P_w, F) to every vertex v
            where O_w^v=(t_i, d);
```

```
        else  -- it is not a primary F-probe
            Send an F-probe (t_i, d, w, d, s, F) to the initiator d;
        end if;
    else  -- the ID is not match
        null;  -- discard the received probe
    end if;
```

4. After initiating a probe computation, the initiator d starts collecting returned F-probes (t_i, d, v, d, s, F) and accumulating the returned probe strength in S_d as long as the (t_i, d)-computation it is involved is still the one initiated by itself (i.e., $B_d = K_d$). If the accumulated probe strength $S_d = 1$, a knot is found. ◇

The following theorem proves the correctness of Algorithm 6.4 based on the rules used and Theorems 4.1, 4.2, and 6.3.

Theorem 6.5 In a dynamic system, Algorithm 6.4 guarantees that (1) any existing knot will be detected exactly once in finite time and (2) there will be no false detection of any knot.

Proof: The set of Rules 6.3, 6.4, and 6.5 sufficiently maintains meaningfulness of each KDC in DGRG. Also, the realization of Rules 6.2, 6.4, and 6.5) is such that a vertex d, which has just became waiting, will initiate a new KDC with a greater logical timestamp only if it finds the largest KDC ID in the set $\{d \cup RN(d)\}$ is meaningless. Imagining that if this is not the case and d still initiates a new KDC but with a smaller logical timestamp, this newly initiated smaller KDC will be overridden by a larger KDC from $RN(d)$ immediately. In terms of projected meaningful operations, Algorithm 6.4 is equivalent to Algorithm 6.2.

Suppose there is a knot K formed in a system state GRG, say ST^f, when a vertex $d \in K$ starts waiting. An KDC will be initiated at d in any case (it may initiate a smaller KDC as described above). By the Stable Property and Isolation Property (i.e., Lemma 6.9, knot K can be treated as a separated static sub-GRG and there will be at least one KDC running

on K, i.e., the one initiated at d). In terms of meaningful operations on K starting from ST^f, Algorithm 6.4 will be equivalent to Algorithm 6.2. By Theorems 6.3 and 4.2 knot K will be detected exactly once in finite time. This proves part (1) of the theorem. Also, since only meaningful KDC's may declare knots and they are equivalent to KDC's due to Algorithm 6.2, Theorem 4.1 and 6.3 guarantee that all declared knots are the real ones. This proves part (2) of the theorem. ■

6.4 Knot Detection with Timing Constraints

Finally, in this section we take into account that the timing constraints which are used in many real-time systems may be associated with vertices in DGRG. A waiting vertex in a real-time system may become voluntarily free due to timing constraints associated with it. For example, a task may wait for a a resource with a specified deadline. Upon the expiration of the deadline, the waiting task may relinquish its request and, hence, become free voluntarily. Consequently, edges in a real-time system DGRG may disappear if one of its end vertices becomes free voluntarily due to timing constraints. As discussed in Section 2.6.1, some of the deadlocks may be temporal. Many systems do not consider these Temporal deadlocks to be real deadlocks since they do not persist forever. However, our performance data (Chapter 7) has shown that detecting and resolving these temporal deadlocks in a system may significantly improve the system performance. Therefore, it is worthwhile to attempt to detect deadlocks even if they are temporal.

Without loss of generality, we assume that timing information is available at every vertex in a DGRG. Whenever a vertex starts waiting, it knows the maximum length of time the waiting state will last. When a task vertex starts waiting, it may set a deadline for the waiting state according to its own timing constraints. If a resource vertex is waiting it means the resource is held or is to be produced by a task. The deadline of a waiting resource depends on the timing constraints associated with the task which is holding or to produce it. There are many policies

that could be used to determine the deadline of a waiting state. How to determine a deadline for a waiting state is an interesting performance problem which is beyond the scope of this dissertation study. We assume that the deadline of a waiting vertex can be properly obtained.

First, let's assume that there is a well synchronized system clock at each site in a distributed system. When a probe P reaches a waiting vertex v, P can check how long v will remain waiting. An operation, say OP_k, may then propagate P to a neighbor of v, say u. The meaningfulness of the operation OP_k relies on the existence of an edge between vertices v and u. If the deadline has expired at either v or u, the edge between v and u will be deleted from DGRG and, hence, the operation OP_k will become meaningless. This, in turn, will cause all the subsequent operations from OP_k to be meaningless. If a KDC is not aware of this problem in real-time applications, it may detect false knots with meaningless edges.

To resolve the problem stated above, we need to keep track of the temporal meaningfulness of a KDC all over the places where its probes have ever reached. To do this, we can use probes to propagate deadline information as follows. Each probe carries a deadline such that it indicates when the KDC at a certain vertex will become meaningless. In other words, the deadline carried by a probe should indicate when one of the previous operations will become meaningless due to the expiration of a vertex deadline. Consequently, the following rule should be applied to probes.

<u>Rule 6.6</u> *A probe needs to keep track of the earliest vertex deadline while it is propagated in a DGRG.*

Also, each vertex needs to keep track of the meaningfulness of the KDC's in which it is currently engaged. If a vertex d is the initiator of a KDC, it needs to keep track of the meaningfulness of all the returned F-probes. When d sums up the returned probe strength, all of the returned F-probes should have valid deadlines. On the other hand, a non-initiator vertex v which is engaged in the KDC needs to maintain the information so that it knows if it can reach the initiator d, i.e., if

$v \in TS(d)$, at a certain time. This is required both in Algorithms 6.3 and 6.4 so that v can decide if a received non-primary F-probe should be directly sent back to the initiator. Moreover, Rule 6.1-A of Algorithm 6.4 requires this information to decide if an F-probe can be received and processed at v. Consequently, the following rule should be applied to the vertices which are engaged in a KDC.

Rule 6.7 *In a KDC, the initiator needs to keep track of the earliest deadline of the F-probes that have returned while all the other engaged vertices need to keep track of the latest deadline of the B-probes received from the KDC.*

Based on Rules 6.6 and 6.7, we derive the following algorithm (Algorithm 6.5) which describes the deadline computation for a KDC. Each of Algorithms 6.3 and 6.4 may then be integrated with Algorithm 6.5 for knot detections in real-time applications.

Algorithm 6.5 (Deadline Computation of a KDC)

Convention: *P.dl* denotes the deadline *dl* carried by a probe P. *v.dl* denotes the deadline of a vertex v. Also, at each engaged vertex the computation deadline is maintained at *KDC.dl*.

In a KDC, when a probe P is received at an engaged vertex v, the deadline computation is carried out as follows:

1. Both *P.dl* and *v.dl* are checked first. If either one is expired, discard the received probe immediately.
2. If $v \neq d$, i.e., v is not the initiator of the KDC:

```
if (P is a B-probe) then
    if (P.dl > v.dl) then
        P.dl := v.dl;
    end if;
    if (P.dl > KDC.dl) then
        KDC.dl := P.dl;
    end if;
```

```
elsif (P is a F-probe) then
    if (KDC.dl <= current_time) then
        -- the KDC is not meaningful at v
        null;  -- discard the received probe
    elsif (P.dl > v.dl) then
        P.dl := v.dl;
    end if;
end if;
```

3. If $v = d$, i.e., the initiator of the KDC:

```
if (KDC.dl <= current_time) then
    -- the KDC is not meaningful at v
    null;  -- discard the received probe
elsif (P is a F-probe) then
    if (P.dl < KDC.dl) then
        KDC.dl := P.dl;
    end if;
    accumulate the probe strength in S_d;
    declare a knot if S_d=1;
end if;
```

◇

The basic idea of Algorithm 6.5 is that in a real-time system we can precisely predict when an existing edge will disappear from the DGRG. By Rule 6.6, when a vertex v receives a probe P, v could determine when the operation in which P is received will become meaningless due to some other vertex having deadline expire. Since a vertex may receive multiple probes from a certain KDC, Rule 6.7 suggests a way to only maintain the necessary timing information at each engaged vertex of a certain KDC. It is clear that Algorithm 6.5 could effectively help to keep a KDC from declaring false knot.

For a Stable Knot (i.e., a Stable Deadlock defined in Section 2.6.1), all of the involved vertices have an infinite deadline and, hence, Algorithm 6.5 won't be effectively applied to it. Both Theorems 6.2 and 6.3 are still true for Stable Knots in real-time applications. Therefore, Stable Knots will be correctly detected as that has been proven in these two theorems.

On the other hand, a Temporal Knot (i.e., a Temporal Deadlock) is not guaranteed to be detected in time. This is because the delay of a KDC may be longer than the tightest timing constraints in a Temporal Knot. For example, suppose there is a Temporal Knot K formed in a DGRG and the earliest KDC which may detect K will finish its computation at time t_d while one of the edges in K will disappear at time t_k. The knot K may be successfully detected only if $t_d \prec t_k$. Algorithm 6.5 is an attempt to avoid the declaration of knot K in the case that $t_d \succ t_k$.

So far we have assumed that the clocks in a distributed real-time system are precisely synchronized. We now relax this assumption a little bit so that clocks may be allowed to drift apart to a upper limit, say Δt_{drift}, and will be re-synchronized to maintain a certain accuracy requirement. Still, this will not affect the correctness of the detection of the Stable Knots. But now Algorithm 6.5 cannot precisely predict when a certain operation will become meaningless. An operation may be treated as a meaningless one a little bit too early or too late within the clock accuracy Δt_{drift} specified in the system. In the former case, a Temporal Knot may go undetected as it is supposed to be. This is not a serious problem because such an undetected Temporal Knot will be broken very soon within Δt_{drift}. However, in the later case, a broken false knot may be declared. If a knot is detected very close to the earliest deadline in the knot, e.g., within Δt_{drift}, it is very likely that the detected knot may be already broken. Since this knot will not exist after a short time Δt_{drift} in any case, we could simply ignore it.

6.5 Application to Ada Environments

We are also interested in applying our algorithms to distributed real-time systems such as Ada environments. In this section, we will show how our algorithms can be applied to detect both Ada rendezvous deadlocks and task terminations.

6.5.1 Ada Rendezvous and Task Termination

As discussed in Section 2.3.3, the Ada rendezvous mechanism may cause OR model deadlocks. Also, a **delay** statement may be incorporated in an Ada rendezvous. The **delay** statement is designed to support some of the real-time features. Timing constraints, therefore, may be involved in Ada rendezvous deadlocks. For example, due to timing constraints attached to each task, a task which is blocked in a rendezvous may need to be timed out or even abnormally aborted. This can be accomplished by specifying a **delay** duration in a rendezvous. Consequently, a task may not stay in a waiting state forever and a "deadline" may be associated with a waiting state as modeled in Section 6.4. Therefore, Ada rendezvous deadlocks may not be stable in real-time applications. Currently, there are no good solutions for Ada rendezvous deadlocks, especially when timing constraints are involved.

There are similarities between the detection of an OR model deadlock and the detection of the termination of a group of cooperating tasks in a distributed computation [5, 6, 39, 41, 42, 48, 50, 98]. In a distributed system tasks cooperate with each other in a computation by means of message exchange. A distributed computation is said to be globally terminated if it reaches a *final* state which, in turn, relies on its member tasks reaching their final states and being ready to terminate locally. When a task has finished its local computation, other cooperating tasks still may want to communicate with it. The termination problem arises when tasks are ready to terminate locally, but they still agree to communicate with other cooperating tasks. The *global termination condition* is defined as the condition that each of the cooperating tasks in a distributed computation is either terminated or ready to terminate. A global termination condition is not satisfied if *any* of the cooperating tasks in a distributed computation is still active and not ready to terminate. The distributed termination problem can be treated as a special case of an OR deadlock where all the cooperating tasks in a distributed computation are involved in a "deadlock" (in the sense that every task in the computation is waiting for others to terminate). If any of these cooperating tasks is not ready to terminate (i.e., still active), there is no "deadlock" and, hence, the global termination condition

is not satisfied. Therefore, any knot detection algorithm for the OR deadlocks can also be tailored for solving the distributed termination problem and vice versa.

In Ada, the **terminate** statements may be used in association with rendezvous (e.g., in *selective wait* statements). A *selective wait* statement may allow a task to **terminate** if all its sibling tasks and their dependent tasks which belong to the same root creator have terminated or are waiting at a **terminate** alternative (Section 2.3.3). This is a pessimistic solution of the termination problem in that it assumes all the active sibling tasks may want to make an entry call to the ones which are ready to terminate in a *selective wait* statement.

Ada has the deficiency of not providing well defined task dependencies. For example, in Ada rendezvous semantics, the calling task is not provided in the **accept** statement. It is because of this deficiency, such a pessimistic definition for the condition of task termination is suggested. Also, the deficiency results in difficulties of rendezvous deadlock detection. Without special compiler and runtime support, it is impossible to detect certain deadlocks involved in Ada rendezvous. We have found that with this additional Ada rendezvous semantic information available at runtime, not only does the detection of rendezvous deadlocks become possible, but also the detection of task termination becomes more efficient. As an example, in Section 6.5.3, we present an integrated algorithm (to be referred to as "OR Algorithm") for both the detection of Ada rendezvous deadlocks and the detection of task terminations.

6.5.2 Design Assumptions Concerning the Ada Runtime Environments

In the development of the following OR Algorithm, we made several assumptions. First, we assume that runtime tasking in the distributed environment is supported by a *Distributed Runtime Tasking Supervisors* [116] (DRTS's). Each of the nodes in a distributed system is equipped with a copy of DRTS. The DRTS's provide services by sending messages to each other. A DRTS could be a separate entity or embedded in the operating system (OS) or kernel. Information concerning

task interactions and synchronizations which are managed by the OS, kernel, or DRTS should be available for deadlock detection. For example, the state of the local task synchronization (rendezvous/termination) should be available in the local OS or kernel, the state of the inter-site task synchronization should be provided by DRTS's.

In addition, information concerning implicit task interactions and synchronizations should be supported both by the compiler and the runtime environment. For example, in Ada rendezvous semantics, the calling task is not provided in the **accept** statement. Without special compiler and runtime support, this feature makes the deadlock problem unsolvable. It is required that a correct and up-to-date DGRG is built at runtime to support correct deadlock detection operations. One possible solution is to ask the compiler to provide extra data structure and program code (not explicitly programmed) for deadlock detection. The extra code provided by the compiler is to be executed at runtime to maintain the deadlock detection related data structure. For example, two kinds of tables are to be built to support deadlock detection for the Ada **accept** statement, one *reachable entry calls* (REC) table for each task, and one *possible calling tasks* (PCT) table for each entry point declared in a task. Initial values for these tables should be entered by the compiler. Program code for maintaining the REC table should be inserted at the proper places in a task by the compiler. Each task, therefore, can update its REC table whenever it is necessary at runtime. When a task is blocked by an **accept** statement at an entry point, the deadlock detection agent is triggered to search every REC table in every possible calling task which is listed in the PCT table of that entry point. An edge is added to the DGRG if there is a matched reachable entry call in a possible calling task.

In real-time systems a timing constraint is usually associated with a task. The runtime system should be able to detect a missed deadline and abort the task. This deadline information is not available from Ada. As summarized in Section 2.3 other aspects of real-time processing are supported by Ada's *selective wait* statement. Using a combination of the **delay** statement and the **else** alternative in Ada's **select**

statement, one can provide an escape in the event that no open alternatives exist or that open alternatives are unduly delayed in their selection. These delays and the ways to terminate them have an effect on whether the task makes its deadline. Similarly, using Ada's *timed* or *conditional* entry calls, a calling task can ensure that it will not be blocked forever impacting its ability to make its deadline. Primarily, we are concerned with task deadlines, and to meet a task's timing constraints, time-out durations are associated with the timed entry calls for the task calling an entry and the delay alternative in the selective wait statement for the task which is waiting for an entry call. How to pick up an appropriate time-out duration for each operation (a rendezvous attempt or a resource request) is beyond the scope of this paper. A simple choice which is assumed in the following discussion is to set the time-out of an operation by the task deadline which, of course, means that if it times out it will not make the deadline.

6.5.3 Algorithm for the Detection of Ada Rendezvous Deadlocks and Task Terminations

In this section we present an OR Algorithm which can be used to detect Ada rendezvous deadlocks and task terminations. The OR Algorithm is based on Algorithm 6.4 integrated with Algorithm 6.5. Therefore, the OR Algorithm carries the features discussed in both Algorithm 6.4 and 6.5.

Whenever a task becomes *BLOCKED*, a KDC is initiated. There are two types of KDC's, i.e., *DEADLOCK* detections and *TERMINATION* detections, in the OR Algorithm for the detection of deadlocks or termination conditions, respectively. A task which is waiting for termination may still be involved in a deadlock. Only if all tasks in the same knot are waiting for termination, a global termination condition is satisfied. This means that a *DEADLOCK* detection computation works on the entire DGRG while a *TERMINATION* detection computation only works on a subset of DGRG where tasks are waiting for termination in rendezvous. If a task is waiting for termination and a KDC is initiated, the task and all the probes of the KDC will

be labeled as *TERMINATION* type. A *TERMINATION* probe can only go through *TERMINATION* tasks.

The data structures for the probes, tasks, and resources are defined in the Figures 6.3, 6.4, and 6.5, respectively. We assume that the resources are not associated with timing constraints and, hence, they are not involved in the deadline computation in a KDC. Therefore, there are no deadline fields in the data structure of resources. However, the deadline information may be added to the resources similar to the tasks if necessary. The DGRG is directly recorded at every task and resource. At each task, there is a ***holding_table*** which contains the resource held by it (incoming edges) and a ***pending_table*** which contains its pending requests (outgoing edges). At each resource, there is a ***waiting_queue*** which contains the tasks which are waiting for the resource (incoming edge) and a ***producer_queue*** which contains the task which may be the producers of the resource (outgoing edges).

```
type PROBE_TS_TYPE is range 0..INTEGER'LAST;
type TASK_ID_TYPE is range 0..INTEGER'LAST;
type DETECTION_TYPE is (DEADLOCK, TERMINATION);
type PROBE_DIR_TYPE is (FORWARD, BACKWARD, INVALID);
type PROBE_STR_TYPE is range 0.0..1.0;
type PROBE_TYPE is
   record
      probe_ts : PROBE_TS_TYPE := 0;  -- probe timestamp
      initr_id : TASK_ID_TYPE := 0;  -- the ID of a probe initiator
      probe_dl : DURATION := 0.0;
         -- Probe deadline is determined by the earliest vertex deadline in its travelling path.
      probe_det : DETECTION_TYPE := DEADLOCK;
      probe_dir : PROBE_DIR_TYPE := BACKWARD;  -- direction
      probe_str : PROBE_STR_TYPE := 0.0;  -- strength
   end record;
```

Figure 6.3: Data structure for probes in the OR Algorithm.

A DGRG is a bipartite graph that vertices are divided into two disjoint subsets, a set of resources vertices and a set of task vertices, such that there are no edges connecting vertices from the same subset. Request edges may be created in DGRG due to that a task is waiting for a set of consumable resources (rendezvous) and

```
type TASK_STATE_TYPE is (ACTIVE, BLOCKED);
type RESOURCE_ID_TYPE is range 0..INTEGER'LAST;
type TASK_TYPE is
   record
      task_id : TASK_ID_TYPE := 0;
      task_state : TASK_STATE_TYPE := ACTIVE;
      task_det : DETECTION_TYPE := DEADLOCK;
      task_req_dl : DURATION := 0.0;  -- deadline of task's request
      task_kdc_dl : DURATION := 0.0;  -- deadline of a KDC in which the task is engaged
      probe_init_ts : PROBE_TS_TYPE;  -- TS of the initiated probe
      probe_buf : PROBE_TYPE;  -- probe in buffer
      holding_table : RES_TABLE_TYPE;  -- resources held by the task
      pending_table : RES_TABLE_TYPE;  -- pending requests of the task
      request_count : NATURAL := 0;  -- the number of pending requests of the task
      forward_res_table : FORWARD_TABLE_TYPE;
         -- a table of resource ID's which are ready to receive forward probes
      rec_probe_str : PROBE_STR_TYPE := 0.0; -- received probe strength
   end record;
```

Figure 6.4: Data structure for tasks in the OR Algorithm.

```
type RESOURCE_ID_TYPE is range 0..INTEGER'LAST;
type RESOURCE_STATE_TYPE is (FREE, HELD);
   -- For a consumable resource, it is FREE if it is produced but is not consumed yet; on the
   -- other hand, it is HELD by its producer if it is requested but is not produced yet.
type RESOURCE_TYPE is
   record
      resource_id: RESOURCE_ID_TYPE:=0;
      resource_state : RESOURCE_STATE_TYPE := FREE;
      probe_buf : PROBE_TYPE;  -- probe in buffer
      waiting_queue : QUE_TYPE;  -- waiting queue of the resource
      producer_queue : QUE_TYPE;  -- producer queue of the resource
      producer_count : NATURAL := 0;  -- number of tasks in producer_queue
      forward_task_table : FORWARD_TABLE_TYPE;
         -- a table of task ID's which are ready to receive forward probes
   end record;
```

Figure 6.5: Data structure for resources in the OR Algorithm.

the corresponding producer edges (point to the tasks which might satisfy the rendezvous) are created immediately after the creation of these consumable resources. Unlike assignment edges, which may be created at any time when a resource is assigned to a task, producer edges are created only after a new consumable resource is initiated due to a request from a task. Since all these edges are created at once as a group, it is sufficient that only one KDC is initiated when necessary. The best candidate for the initiation of a KDC for a group of newly created edges is the task which first requested the set of consumable resources. Consequently, in the OR Algorithm, KDC's may be initiated only when a task becomes *BLOCKED* in the rendezvous. In Figure 6.6, **procedure** *TASK_INIT_PROBE* describes how a KDC is initiated.

In Figures 6.7 **procedure** *TASK_RCV_B_PROBE* describes how tasks handle B-probes and in Figures 6.8 **procedure** *TASK_RCV_F_PROBE* describes how tasks handle F-probes. These two procedures are the integration of Algorithms 6.4 and 6.5. Also, **procedure** *RESOURCE_RCV_B_PROBE* (Figures 6.9) and **procedure** *RESOURCE_RCV_F_PROBE* (Figure 6.10) describe how resources handle B-probes and F-probes, respectively. The differences between the procedures for tasks and the procedures for resources are: (1) resources do not perform deadline computation since there is no assumed deadline associated with it and (2) resources do not perform strength checking for knot declaration since all the KDC's are initiated at tasks. When a probe is received at a resource vertex, its deadline is checked as part of the probe validity checking.

There are several procedures and functions used in the OR Algorithm which are not explicitly described in the figures. They are briefly defined as follows:

1. **procedure** *SEND* (*P*, *ID*) sends a probe *P* to a task or a resource identified as *ID*.

2. **procedure** *RECEIVE* (*P*, *ID*) receives a probe *P* from a task or a resource identified as *ID*.

```
procedure TASK_INIT_PROBE (T: in out TASK_TYPE) is
  -- This procedure is invoked when an ACTIVE task T requests a set of HELD resources. The
  -- deadline of T's request is T.task_req_dl. The task T is in transition from the ACTIVE
  -- state to the BLOCKED state. The B-probes are collected from R.probe_buf at each of the
  -- waited resources R along with the request pending response. A period of waiting time Δt
  -- which may be chosen as a function of task T's deadline might be inserted right before the
  -- calling of this procedure.
  R : RESOURCE_TYPE;    R_ID : RESOURCE_ID_TYPE;    P, BP : PROBE_TYPE
begin
  CLEAR_TABLE(T.forward_res_table);
  P.probe_ts := T.probe_buf.probe_ts
  P.probe_dir := INVALID; -- it is an INVALID probe
  for R_ID in T.pending_table loop
    RECEIVE (BP, R_ID); -- BP is the B-probe from R_ID
    if (((P.probe_det = TERMINATION) and then
        (T.task_det = DEADLOCK)) or else
       (BP.probe_ts < T.probe_buf.probe_ts)) then
       null; -- discard the received probe
    elsif (BP.probe_ts > P.probe_ts) then
       P := BP;
       CLEAR_TABLE(T.forward_res_table);
       ADD_TO_TABLE(T.forward_res_table, R_ID);
    elsif ((BP.probe_ts = P.probe_ts) and then
         (BP.initr_id = P.initr_id)) then
       ADD_TO_TABLE(T.forward_res_table, R_ID);
    end if;
  end loop;
  if ((P.probe_dir = INVALID) or else
     (P.probe_dl <= current_time)) then
    -- The largest KDC is meaningless; initiate a new probe to override it.
    P.probe_ts := P.probe_ts + 1;      P.initr_id := T.task_id;
    P.probe_det := T.task_det;  -- DEADLOCK or TERMINATION
    P.probe_dir := BACKWARD;      P.probe_str := 0.0;
    P.probe_dl := T.task_req_dl;      T.task_kdc_dl := P.probe_dl;
    T.probe_init_ts := P.probe_ts;      T.probe_buf := P;
    T.probe_buf.probe_dir := FORWARD; -- prepare a F-probe
    T.probe_buf.probe_str := 1.0 / T.request_count;
    T.rec_probe_str := 0.0;  -- reset received probe strength
  else  -- otherwise, no probe is initiated.
    if (P.probe_dl > T.task_req_dl) then  P.probe_dl := T.task_req_dl;  end if;
    T.probe_init_ts := 0;  T.probe_buf := P;
    T.task_kdc_dl := P.probe_dl;
  end if;
  for R_ID in T.holding_table loop
    SEND (P, R_ID);  -- propagate BACKWARD
  end loop;
end TASK_INIT_PROBE;
```

Figure 6.6: Procedure for the probe initiation in the OR Algorithm.

```
procedure TASK_RCV_B_PROBE (T: in out TASK_TYPE;
            P: in PROBE_TYPE; R: in RESOURCE_TYPE) is
   -- This procedure is invoked whenever a BLOCKED task T receives a B-probe P from a
   -- resource R.
   R_ID : RESOURCE_ID_TYPE
begin
   if ((not IN_TAB(R, T.pending_table)) or else -- from a white edge
      ((P.probe_det = TERMINATION) and then (T.task_det = DEADLOCK)) or else
      (P.probe_ts < T.probe_buf.probe_ts) or else
      ((P.probe_ts = T.probe_buf.probe_ts) and then
       (P.initr_id < T.probe_buf.initr_id)) or else
      (T.task_req_dl <= current_time) or else
      (P.probe_dl <= current_time)) then
      null; -- discard the received probe
   elsif (T.probe_buf.probe_dir = BACKWARD) then
      if ((P.probe_ts = T.probe_buf.probe_ts) and then
         (P.initr_id = T.probe_buf.initr_id)) then
         ADD_TO_TABLE(T.forward_res_table, R.resource_id);
         if (P.probe_dl > T.task_kdc_dl) then  T.task_kdc_dl := P.probe_dl;  end if;
      else  -- a new one received
         CLEAR_TABLE(T.forward_res_table);
         ADD_TO_TABLE(T.forward_res_table, R.resource_id);
         if (P.probe_dl > T.task_req_dl) then  P.probe_dl := T.task_req_dl;  end if;
         T.probe_buf := P;     T.task_kdc_dl := P.probe_dl;
         for R_ID in T.holding_table loop
            SEND (P, R_ID);  -- propagate backward
         end loop;
      end if;
   else  -- there is a F-probe in the buffer
      if ((P.probe_ts = T.probe_buf.probe_ts) and then
         (P.initr_id = T.probe_buf.initr_id)) then
         if (P.probe_dl > T.task_req_dl) then  T.task_kdc_dl := T.task_req_dl;
         elsif (P.probe_dl > T.task_kdc_dl) then  T.task_kdc_dl := P.probe_dl;
         end if;
         SEND (T.probe_buf, R.resource_id) -- R is the sender of the probe P
      else  -- the new one is larger, hence, override the current one
         CLEAR_TABLE(T.forward_res_table);
         ADD_TO_TABLE(T.forward_res_table, R.resource_id);
         if (P.probe_dl > T.task_req_dl) then  P.probe_dl := T.task_req_dl;  end if;
         T.probe_buf := P;     T.task_kdc_dl := P.probe_dl;
         for R_ID in T.holding_table loop
            SEND (P, R_ID);  -- propagate backward
         end loop;
      end if;
   end if;
end TASK_RCV_B_PROBE;
```

Figure 6.7: Procedure for tasks handling received B-probes in the OR Algorithm.

```
procedure TASK_RCV_F_PROBE (T: in out TASK_TYPE;
            P: in PROBE_TYPE; R: in RESOURCE_TYPE) is
   -- This procedure is invoked whenever a BLOCKED task T receives a F-probe P from a
   -- resource R.
   R_ID : RESOURCE_ID_TYPE
begin
   if ((not IN_TAB(R, T.holding_table)) or else -- from a white edge
      ((P.probe_det = TERMINATION) and then (T.task_det = DEADLOCK)) or else
      (P.probe_ts < T.probe_buf.probe_ts) or else
      ((P.probe_ts = T.probe_buf.probe_ts) and then
       (P.initr_id < T.probe_buf.initr_id)) or else
      (T.task_req_dl <= current_time) or else
      (P.probe_dl <= current_time)) then
      null; -- discard the received probe
   elsif (T.probe_buf.probe_dir = BACKWARD) then
      if ((P.probe_ts = T.probe_buf.probe_ts) and then
         (P.initr_id = T.probe_buf.initr_id)) then
         if (P.probe_dl > T.task_req_dl) then P.probe_dl := T.task_req_dl; end if;
         P.probe_str := P.probe_str / T.request_count;
         T.probe_buf.probe_dir := FORWARD;    T.probe_buf.probe_str := P.probe_str;
         for R_ID in T.forward_res_table loop
            SEND (P, R_ID); -- only send to forward_res_table
         end loop;
      end if;
   else -- there is a F-probe in the buffer
      if ((T.task_kdc_dl > current_time) and then
         (P.probe_ts = T.probe_buf.probe_ts) and then
         (P.initr_id = T.probe_buf.initr_id)) then
         if ((P.probe_ts = T.probe_init_ts) and then
            (P.initr_id = T.task_id)) then
            if (P.probe_dl < T.task_kdc_dl) then T.task_kdc_dl := P.probe_dl; end if;
            T.rec_probe_str := T.rec_probe_str + P.probe_str;
            if (T.rec_probe_str = 1.0) then
               A deadlock is found;
            end if;
         else -- non-primary F-probe, send it back to its initiator
            SEND (P, P.initr_id);
         end if;
      end if;
   end if
end TASK_RCV_F_PROBE;
```

Figure 6.8: Procedure for tasks handling received F-probes in the OR Algorithm.

```
procedure RESOURCE_RCV_B_PROBE (R: in out RESOURCE_TYPE;
                   P: in PROBE_TYPE; T: in TASK_TYPE) is
   -- This procedure is invoked when a resource R receives a B-probe P from a task T.
   T_ID : TASK_ID_TYPE
begin
   if ((not IN_QUE(T, R.producer_queue)) or else -- from a white edge
      ((P.probe_ts = R.probe_buf.probe_ts) and then
      (P.initr_id < R.probe_buf.initr_id)) or else
      (P.probe_ts < R.probe_buf.probe_ts) or else
      (P.probe_dl <= current_time)) then
      null; -- discard the received probe
   elsif (R.probe_buf.probe_dir = BACKWARD) then
      if ((P.probe_ts = R.probe_buf.probe_ts) and then
         (P.initr_id = R.probe_buf.initr_id)) then
         ADD_TO_TABLE(R.forward_task_table, T.task_id);
      else -- a new one received
         CLEAR_TABLE(R.forward_task_table);
         ADD_TO_TABLE(R.forward_task_table, T.task_id);
         R.probe_buf := P;
         for T_ID in R.waiting_que loop
            SEND (P, T_ID); -- propagate backward
         end loop;
      end if;
   else -- it is a FORWARD probe in the buffer
      if ((P.probe_ts = R.probe_buf.probe_ts) and then
         (P.initr_id = R.probe_buf.initr_id)) then
         SEND (R.probe_buf, T) -- T is the sender of the probe P
      else -- the new one is larger, hence, override the current one
         CLEAR_TABLE(R.forward_task_table);
         ADD_TO_TABLE(R.forward_task_table, T.task_id);
         R.probe_buf := P;
         for T_ID in R.waiting_que loop
            SEND (P, T_ID); -- propagate backward
         end loop;
      end if;
   end if;
end RESOURCE_RCV_B_PROBE;
```

Figure 6.9: Procedure for resources handling received B-probes in the OR Algorithm.

```
procedure RESOURCE_RCV_F_PROBE (R: in out RESOURCE_TYPE;
          P: in PROBE_TYPE; T: in TASK_TYPE) is
   -- This procedure is invoked when a resource R receives a F-probe P from a task T.
   T_ID : TASK_ID_TYPE
begin
   if ((not IN_QUE(T, R.waiting_queue)) or else  -- from a white edge
      ((P.probe_ts = R.probe_buf.probe_ts) and then
       (P.initr_id < R.probe_buf.initr_id)) or else
      (P.probe_ts < R.probe_buf.probe_ts) or else
      (P.probe_dl <= current_time)) then
      null; -- discard the received probe
   elsif (R.probe_buf.probe_dir = BACKWARD) then
      if ((P.probe_ts = T.probe_buf.probe_ts) and then
         (P.initr_id = T.probe_buf.initr_id)) then
         P.probe_str := P.probe_str / T.producer_count;
         R.probe_buf.probe_dir := FORWARD;    R.probe_buf.probe_str := P.probe_str;
         for T_ID in R.forward_task_table loop
            SEND (P, T_ID); -- only send to forward_task_table
         end loop;
      end if;
   else  -- it is a FORWARD probe in the buffer
      if ((P.probe_ts = T.probe_buf.probe_ts) and then
         (P.initr_id = T.probe_buf.initr_id)) then
         -- non-primary probe, send it back to its initiator
         SEND (P, P.initr_id);
      end if
   end if
end RESOURCE_RCV_F_PROBE;
```

Figure 6.10: Procedure for resources handling received F-probes in the OR Algorithm.

3. **function** *IN_TAB* (*R*, *TAB*) returns a value *TRUE* if a resource R is found in a table *TAB*, where *TAB* could be a task's *pending_table* (for the outstanding requests) or *holding_table* (for its granted resources); otherwise, a value *FALSE* is returned.

4. **function** *IN_QUE* (*T*, *QUE*) returns a value *TRUE* if a task T is found in a queue *QUE*, where *QUE* could be a resource's *waiting_queue* (for the waiting tasks) or *producer_queue* (for the potential producers of the resource); otherwise, a value *FALSE* is returned.

5. **procedure** *CLEAR_TABLE*(*F_TAB*) clears table *F_TAB*, where *F_TAB* is a table of ID's to where an F-probe should be forwarded.

6. **procedure** *ADD_TO_TABLE(F_TAB, ID)* adds an *ID* into *F_TAB*, where *ID* is a *resource_id* or a *task_id*.

The procedures *SEND* and *RECEIVE* are used for probe propagations. The functions *IN_TAB* and *IN_QUE* are used to check if a received probe was coming from a white edge. And the procedures *CLEAR_TABLE* and *ADD_TO_TABLE*, which is an alternate way of implementing O_v^u in Algorithm 6.4, maintains a table of outgoing address (ID's) to where an F-probe should be forwarded.

All tasks are *ACTIVE* when created and all consumable resources (rendezvous) are *HELD* by the producers when requested. In Ada, synchronization between two tasks occurs when the task issuing an entry call and the task accepting an entry call are ready to establish a rendezvous. A rendezvous is a consumable resource. Either one of the calling and called tasks arriving at the rendezvous first will wait, and, hence, becomes the "consumer" of the "rendezvous." The second task which establishes a rendezvous is always the "producer" of the "rendezvous." After a rendezvous is established, the calling task becomes *BLOCKED* while the called task is executing corresponding statements following the accept statement. The called task, therefore, is *ACTIVE* and acts as the producer during the rendezvous period.

6.6 Comparison to Other Work

In this section we compare our algorithms to related work. First, we briefly survey some previously proposed OR deadlock and knot detection algorithms in Section 6.6.1. Our proposed Algorithm 6.4 is then compared to the most similar one in the literature. In Section 6.6.2, we briefly survey a related work on the dynamic detection of deadlocks for Ada programs and compare our Ada application to it.

6.6.1 The OR Deadlock and Knot Detection Algorithms

Dijkstra and Scholten [42] introduced the notion of *diffusing computation* (See Section 3.2.3) and suggested an algorithm to detect the termination of an arbitrary diffusing computation in any network environment. Most of the OR deadlock detection algorithms proposed in the literature are basically derived from their notion of diffusing computation. For example, Misra and Chandy [98] discussed how a diffusing computation based termination detection algorithm can detect communication deadlocks. Following this Chandy, Misra and Haas [27] presented an algorithm for the communication deadlock detection based on diffusing computation.

However, the structure of a diffusing computation does not directly define a *knot* which is the necessary and sufficient condition for the existence of an OR deadlock. Therefore, many efforts have been made to adapt diffusing computations to detect knots which cause OR deadlocks. For example, Misra and Chandy [97] presented an algorithm based on diffusing computation for distributed knot detection. Also, Natarajan [105] modified Chandy, Misra and Haas's algorithm in [27] to include a voting phase so that the algorithm can explicitly detect each knot exactly once.

Another disadvantage of using a diffusing computation for OR deadlock detection is its prolonged two phased structure. Although there is no explicit boundary between the two phases of a diffusing computation, *query* messages have to travel to every reachable vertex and *reply* signals may be bounced back when it sees a cyclic wait. Chandy, Misra, and Haas's communication deadlock detection algorithm [27] is a typical application of the diffusing computation. Natarajan's algorithm [105] includes an additional voting phase before the invocation of a diffusing computation to detect knots. This make the delay of the detection of an OR deadlock even longer. Huang proposed an algorithm [68] which is based on Natarajan's algorithm. Huang uses a probe strength computation to combine the two phases of a diffusing

computation. However, Huang's algorithm still employs two phases, i.e., a voting phase and a knot detection phase.

Our proposed Algorithm 6.4 differs from Huang's algorithm in two respects:

1. The voting process in Algorithm 6.4 is blended with the B-probe computation. To be more specific, in Algorithm 6.4, Rule 6.1-A requires that, in order to receive and process an F-probe, a primary B-probe from the same KDC must be received at a vertex. In Huang's algorithm, to accomplish the voting phase, an F-probe cannot be processed at a vertex until at least a B-probe from the same KDC is received at each of the vertex's outgoing edges.
2. Algorithm 6.4 may initiate a new KDC for a newly blocked vertex d only if it finds that the largest KDC ID in the set $\{d \cup RN(d)\}$ is meaningless. The idea is that the meaningful KDC's should be preserved as much as possible so knots may detected earlier. On the other hand, Huang's algorithm attempts to detect knots by a "latest" KDC. Whenever a vertex becomes blocked, a new KDC is initiated with a "latest" logical timestamp. This "latest" KDC may override previous meaningful KDC's which may detect the same knot earlier.

In addition, Algorithm 6.5 may be integrated with Algorithm 6.4 for real-time applications. This is a feature has not yet been found in the literature.

6.6.2 Deadlock Detection Algorithms in Ada Applications

Most research work on the deadlock problem in Ada programs is in the area of static program analysis. There has been very limited research in the dynamic detection of deadlock for Ada programs. One noteworthy exception is the work performed by German, Helmbold, and Luckham [52, 53, 61, 62]. Their work uses a runtime monitor to detect a class of Ada deadness errors including some limited forms of deadlock. There are four major differences between their work and the work presented here. First, the monitoring mechanism used in their work is a centralized mechanism, while our work focuses on distributed systems. Second,

deadlock problems are analyzed in very different ways. In their work, deadness errors are classified into three types: Circular Deadlock, Global Blocking, and Local Blocking. Circular Deadlock is extremely restricted and involves only entry calls. It does not account for deadlock that may occur for any other reason such as circular conflicts over a disk resource. Their Circular Deadlocks model is equivalent to the Single-Resource deadlock model, but (again) is used only on entry calls. In addition to handling Circular Deadlocks for entry calls, they can also detect a Global Blocking situation. A Global Blocking means that all the tasks in the system are stopped, regardless of the reason. This implies that when they find all tasks stopped it may have been due to problems that would have to be modeled by the OR model, or the AND-OR model, or the C(n,k) model. However, to find such a problem requires either the unlikely situation that all tasks are stopped, or waiting long enough for all tasks which are still running to terminate successfully, or to eventually call a blocked task. This is unreasonable for a real-time system. A third type of deadness error that they deal with is called Local Blocking. Local Blocking means that a subset of interacting tasks are blocked. However, neither the Circular Deadlock nor the Local Blocking consider deadlocks which involve the OR logic wait-for dependency due to **accept** statements. Consequently, their monitor cannot detect any local deadlock with OR model or AND-OR model, or C(n,k) model complexity unless the situation evolves into a Global Blocking situation. Third, timing constraints are not considered in their work. This is due, in part, because it was not their intent, and in part, because they do not treat a task which is executing a **delay** of a selective wait statement as in the blocked state. Fourth, their monitor is built at the user application level rather than as part of the underlying runtime system. Most of their research effort was spent on the program transformations necessary to gain program visibility at the user level. In our approach, we assume the compiler and underlying system can provide the proper information, and then focus on the development of efficient, distributed, real-time deadlock detection algorithms for Ada programs.

6.7 Concluding Remarks

This chapter basically contains two parts: the development of knot detection algorithms and its application to Ada environments.

In the first part we have carefully developed a series of knot detection algorithms in three refinement steps as suggested in Section 4.1. In the first step, the underlying system is assumed static and we then could focus on the properties and principles of knot detection computations. To detect knots in distributed systems we need to use a "local view definition" which defines a knot from the point of view of a vertex in a GRG. From a vertex d, this "local view definition" suggests that we should construct a set of vertices $RS(d)$ to which d can find a directed path in GRG and a set of vertices $TS(d)$ from where we can find a path directs to d. The set $RS(d)$ is a knot if and only if $RS(d) \subseteq TS(d)$. This definition leads to the idea that if probes are sent out along d's outgoing edges to search $RS(d)$ all the probes should come back if $RS(d) \subseteq TS(d)$. To determine if all the probes come back, a strength is associated with each probe [68]. Whenever a probe split, the strength splits; wherever probes merge, the strength merges. If the initiator of a KDC sees its original probe strength come back, all the probes are back. This principle is presented and proven in Algorithm 6.0.

One of the problems with Algorithm 6.0 is that an infinite loop may be involved in the probe forwarding path and, hence, a KDC may not terminate in finite time. To remedy the infinite loop problem, we developed the notion of "primary" probes, i.e., the probes (forward and backward) from a certain KDC received at a vertex for the first time. Only the primary probes are fully propagated in a GRG. A non-primary probe will be discarded or directly sent back to the initiator if necessary. This idea is presented and proven in Algorithm 6.1.

In the development of Algorithm 6.2 we analyze how KDC's might interact with each other in a GRG and derive probe competition rules to reduce the detection of a knot to be exactly once. These rules also help in eliminating the unbounded storage problems which might emerge in the dynamic algorithms.

In the second step of the algorithm development, based on our methodology presented in Chapter 4, synchronization mechanisms are developed to extend Algorithms 6.1 and 6.2 into dynamic systems. We have tried to optimize our synchronization mechanisms so that an existing knot could be detected as early as possible. The results are presented and proven in Algorithms 6.3 and 6.4. Since our algorithms are directly derived from properties of the OR deadlock model and knot structures, they are more efficient in the sense that the computation delay of declaring knots is in general shorter than previous work.

In the last step, Algorithms 6.3 and 6.4 are extended for real-time applications. We analyze the properties of Algorithms 6.3 and 6.4 and the implications of timing constraints in real-time systems to derive a Deadline Computation Algorithm (Algorithm 6.5). Algorithm 6.5 may be integrated with Algorithms 6.3 or 6.4 for real-time applications.

In the second part of this chapter, our algorithms are applied to Ada environments. We analyze Ada rendezvous deadlock and task termination problems and proposed an integrated solution for them (i.e., the OR Algorithm). One unique feature of the OR Algorithm is that it is able to properly handle both **delay** and **terminate** statements in Ada's rendezvous which has never been addressed in any previous work.

CHAPTER 7

IMPLEMENTATION AND PERFORMANCE EVALUATION

In this chapter, we report on a performance study based on the deadlock detection algorithms and various schemes implemented in a distributed real-time database testbed. The testbed system is briefly described in Section 7.1. The implementation of the algorithms and the baseline schemes (for comparison) are described in Section 7.2. Experiment parameters and performance metrics are discussed in Section 7.3. The experimental results are presented in Section 7.4. Finally, a summary and conclusions are found in Section 7.5.

7.1 The Real-Time Database Testbed RT-CARAT

RT-CARAT (Real-Time Concurrency And Recovery Algorithm Testbed)[67] is a distributed real-time database testbed system. RT-CARAT is designed to be a flexible tool for the testing and performance evaluation of real-time database protocols. The testbed is implemented as a set of cooperating server processes which communicate via efficient message passing mechanisms. Figure 7.1 illustrates the processes and message structure of RT-CARAT for any two sites of the system. In each site there is a TM (Transaction Manager) server process and a pool of DM (Data Manager) server processes, which are created during system start up. TR (Transaction) processes are created by users to execute database transactions. For the purpose of performance measurement, TR processes are created automatically by the TD (Test Driver) process according to a specified workload. The DC (Data Collector) process is responsible for collecting performance statistics. The SPY process can monitor and display the system status while the transactions are executing and it is used primarily for debugging purposes.

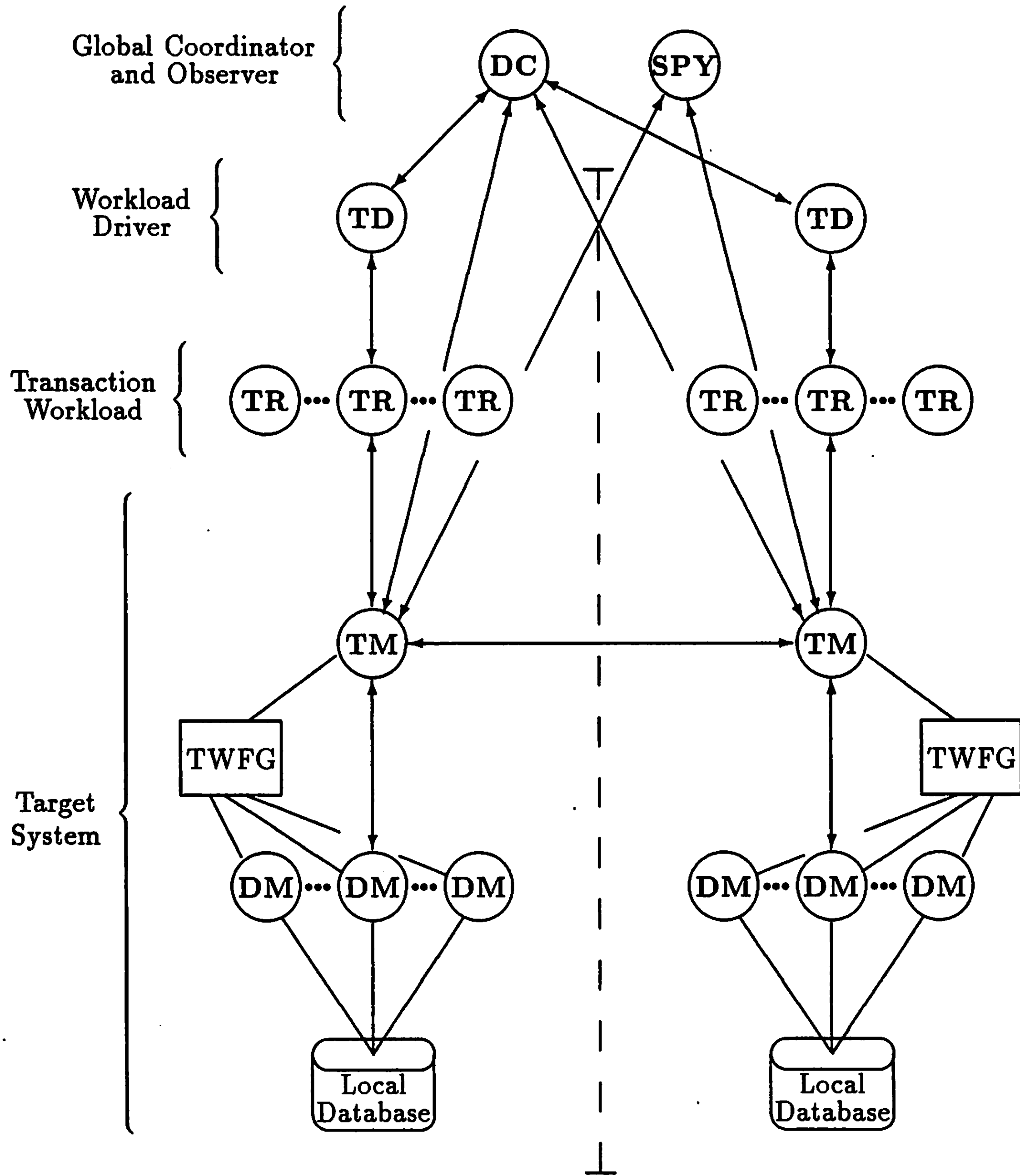


Figure 7.1: RT-CARAT processes and message structure

When a TR initiates a transaction, it registers a unique ID with its local TM (i.e., its coordinator). In the RT-CARAT system, there is one database at each site (together they may be treated as a non-replicated database that is partitioned and distributed among all the involved sites). To open a database, the TR process sends a DBOPEN message to its coordinator TM. If the database is located at the local site, the coordinator TM assigns a local DM server to this transaction and then forwards the DBOPEN message to the assigned DM. If the database is located at a remote site, the coordinator TM will forward the DBOPEN message to the remote site (slave) TM which in turn assigns a DM server to this transaction and then forwards the DBOPEN message to the assigned DM. When a TR initiates a transaction, a deadline which is randomly selected from a pre-specified range called "deadline window" is associated with the transaction. This deadline information is then passed to the TM's and the designated DM's along with the DBOPEN messages. Before a transaction reaches its commit point (i.e., the transaction completes all its database operations and initiates the two-pahse-commitment protocol) it will be aborted upon the expiration of the transaction deadline by any of the involved TR, TM's, and DM's processes. To access (read/write) records, the TR process sends a TDO message to its coordinator TM. A TDO message may contain multiple data accesses to any single-site database. In the Single-Resource and the AND model system, the data granules requested in a TDO step are processed in sequence and each of them must be granted before the transaction can proceed. On the other hand, in the OR deadlock model system, a transaction may proceed if any data granule requested in a TDO step can be granted. If a TDO step accesses data granules in the local site, the coordinator TM forwards the TDO message to the assigned DM server for the actual database operations. If a TDO step accesses data granules from a remote site, the coordinator TM forwards the TDO message to the remote slave TM which in turn forwards it to the designated DM server. When a transaction completes by its deadline, its coordinator TM is informed to perform the two-phase-commitment protocol. If a transaction is to be aborted, its coordinator TM is informed to broadcast the ABORT messages to each of slave sites.

The physical environment for the experiments includes three microVAX II sites. The sites are connected by a local area DECnet. System clocks are synchronized at the beginning of each test and they may drift up to 100 msecs apart in an hour long test. We believe that this clock drift rate does not significantly affect the experiment results since most of the transaction deadlines are set to tens of seconds.

For concurrency control and recovery, we use the two-phase locking protocol and after-image journaling mechanism from the RT-CARAT implementation. In each site, a TWFG (Transaction Wait-For Graph) is maintained by the TM and the DM processes (see Figure 7.1).

In RT-CARAT, when a DM process is in a waiting state, it cannot process any message until it is woken up. Therefore, probes are propagated by the TM and active DM processes immediate after they update the TWFG. In RT-CARAT, locks are performed at the page level and each database page contains 6 records (data granules). When a DM wants to read (or write) a record, it acquires a read (or write) lock on the page which contains the record. If there are lock conflicts, the DM will update the TWFG and perform deadlock detection operations (propagate probes according to the TWFG). After a DM finishes all of the possible local probe propagation operations, it checks to see if there are probes needed to be propagated to the remote sites. If so, the local TM is informed to initiate inter-site probe propagations. TM's then cooperate in the inter-site probe propagations. Since TM's are responsible for the inter-site TDO message transfers, they will update their local TWFG's whenever remote TDO messages are sent or received. When a coordinator TM forwards a TDO message to a remote slave TM, a remote wait outgoing edge record (the state of the receiving end of the edge is not shown locally) is added to the local TWFG. When a slave TM receives a TDO message from a remote coordinator TM, an incoming edge record (the state of the sending end of the edge is not shown locally) is added to the local TWFG. When a slave site finishes a TDO and responds back to the coordinator site, the above two edge records will be reversed by the involved TM's. Necessary inter-site probe propagations

will be cooperatively performed by the involved TM's after they update their local TWFG's.

When a transaction is chosen as the victim of a deadlock resolution, it is aborted. No rollback is necessary to abort a transaction in RT-CARAT since after-image journaling is used. An aborted transaction will be restarted immediately if its deadline has not expired.

In RT-CARAT, the CPU is scheduled based on transaction priority with preemption using the underlying VAX/VMS operating system real-time priorities (priorities 16-30). The CPU scheduler is embedded in the TM. Upon receiving a transaction execution request from a TR, the scheduler assigns a priority to the transaction according to the chosen CPU scheduling policy, e.g., EDF (Earliest Deadline First) [67]. The scheduling operation is done by mapping the assigned transaction priority to the real-time priority of the DM process which carries out the transaction execution. At this point, an executing DM will be preempted if it is not the highest priority DM process at the moment, otherwise it will continue to run.

In our experiments, we primarily assume that no specific real-time scheduling protocol is used in the underlying system. Most of the experiments focus on the effects of various schemes related to the deadlock problems in the system. All the DM processes are running in the regular priorities (priorities 4-15) which are manipulated by the underlying VAX/VMS operating system. However, for comparison, a commonly used real-time CPU scheduling protocol EDF is used in one of the experiments.

RT-CARAT is a secondary storage database system. In a secondary storage database system, disk I/O is an important performance factor. However, the disk I/O performance is beyond the scope of our experiments. Instead of performing physical disk I/O operations, which is under the control of disk controllers, we simulate I/O operations in our experiments. Each I/O operation is simulated as a fixed delay time protected by a FIFO critical section. For each read operation,

a page of database records is directly generated in the main memory. The delay time is chosen so that all of the tests in an experiment will not cause the system to become I/O bound or CPU bound.

7.2 The Schemes for the Experiments

To verify the deadlock detection algorithms and to study their performance, we have implemented three classes of schemes in the RT-CARAT system:

1. Break by deadline: This is one of the baseline schemes. In the RT-CARAT system, each transaction is associated with a deadline. An transaction is aborted when its deadline expires. This scheme allows a deadlock to persist until one of its participants is aborted by deadline. This class of schemes is denoted as NDD (No Deadlock Detection).
2. Timeout and retry: This is a class of baseline schemes. Instead of allowing transactions to wait until deadline expiration, this class of schemes sets a short timeout period for the transactions' waiting states. A transaction will stop waiting and rollback upon the expiration of the timeout period or the transaction deadline. A transaction will be restarted after the rollback if its deadline is not expired. Three timeout settings are used in the experiments: 1 second (denoted as W01), 5 seconds (W05), and 10 seconds (W10).
3. Deadlock detection/resolution: For deadlock detection/resolution schemes, three proposed algorithms are implemented: the Single-Resource Algorithm (denoted as SRD, see Section 5.3.4 for the detail of the algorithm), the AND Algorithm (AND, Section 5.4.4), and the OR Algorithm (ORD, Section 6.5.3). Also, there is a previously implemented AND deadlock detection algorithm which is based on Chandy, Misra, and Haas's resource deadlock detection algorithm proposed in [27] (denoted as CMH). Some of the implementation details are discussed in the following paragraphs.

In the original SRD, AND, and ORD algorithms, the code is composed of two parts: one specifies the operations performed at the tasks and the other part

specifies the operations performed at the resources. The resource part of the algorithm code is basically simplified from the task part so that resources do not initiate probes and do not declare deadlocks. These algorithms are designed for the systems where tasks and resources are managed separately and GRG's can be easily formed. In RT-CARAT, transactions can be treated as tasks in the deadlock detection. However, since RT-CARAT only supports TWFG, the resource part of the proposed algorithms is eliminated in the implementations. Probes are directly transferred between transactions according to the TWFG in RT-CARAT.

Also, in the original ORD (Section 6.5.3), the detection of a task termination condition is treated as a special case of the deadlock. If a deadlock is detected in which every participant is also waiting for the termination, a termination condition is declared. Since, in RT-CARAT, transactions do not wait for each other to terminate, the termination detection part of the ORD algorithm is eliminated in the implementation.

The CMH algorithm uses the notion of *dependent set* to stop foreign probes from being repeatedly propagated in a cycle (see Section 5.2). At each site, the TM keeps track of a dependent set for each of the transactions it manages. A dependent set contains the probe ID's received at a transaction. A probe will not be propagated to the remote sites if it is found in the dependent set of the transaction where the probe is processed. A dependent set is cleared when its corresponding transaction leaves a remote waiting state. The probe ID is a combination of the initiator's transaction ID and site ID and a logical timestamp maintained at each DM process such that a probe ID can be uniquely identified in the whole system.

Both the SRD and the AND algorithms are optimized to reduce the number of probe initiations such that each cycle may be declared exactly once. Also, transaction deadline information is considered in the SRD and the AND algorithms to reduce the possible false detection of Temporal deadlocks. On the other hand, the implementation of CMH algorithm

- is optimized to reduce the inter-site probe propagations. In the CMH implementation, a DM will finish searching local cycles before it informs TM to propagate probes to the remote sites. Inter-site propagation will not be initiated if a DM finds local cycles and aborts the transaction it serves. This is because in the CMH implementation, a DM only searches cycles in which the transaction it serves is involved.

- may cause a cycle to be declared (and broken) at more than one places. This is because in the CMH algorithm, every transaction in a cycle may initiate a probe computation and may detect the cycle and abort itself (actually, all these operations are done by the transaction's designated DM and TM processes).

- does not take transaction deadline into consideration. In the CMH algorithm, a detected Temporal deadlock may never exist or may have been broken by a transaction which is aborted due to deadline expiration.

7.3 Parameters and Performance Metrics

In RT-CARAT, tests are automatically executed according to the parameters specified in a configuration file. Some of the important parameters are summarized as follows.

TR_Length – transaction length. In all our experiments, transactions are of two lengths, long and short. In a test configuration, the transaction length is specified as $[x_l, y_l; x_s, y_s]$. The first two numbers are used to specify the long transactions and the last two numbers are used for the short ones. The x values specify the number of TDO step per transaction and the y values specify the number of records per TDO steps.

Pr_Long_TR – the probability of initiating a long transaction.

Pr_Write_TR – the probability of initiating an write (update) transaction. When a transaction is initiated, it is either a write transaction or a read-only transaction. Only an write transaction may issue write TDO steps.

Pr_Write_TDO – the probability of issuing a write TDO in an update transaction.

Pr_Dist_TR – the probability of initiating a distributed transaction. When a transaction is initiated, it is either a distributed transaction or a local one. Only a distributed transaction may request records from remote sites.

N_Rem_Sites – the number of remote sites from which a distributed transaction may request records.

IO_Delay – simulated I/O delay. In our experiments, each disk I/O operation is simulated as a constant delay protected by a critical section.

MPL – multiprogramming level. This parameter is primarily determined by the number of TR processes in the system.

DB_Size – database size. The maximum database size at each site is 3000 pages with 6 records per page.

DL_Window – deadline window. The deadline window is specified as $[x_l\text{-}y_l, x_s\text{-}y_s]$. The notation $x_l\text{-}y_l$ (values are in seconds) specifies the lower (x_l) and the upper (y_l) limits of the deadlines for the long transactions. Similarly, $x_s\text{-}y_s$ specifies the lower (x_s) and the upper (y_s) limits of the deadlines for the short transactions.

Acc_Pattern – data accessing pattern. This parameter specifies how records are selected from a database for a TDO step. In our experiments, two types of accessing patterns are used: random and contiguous. A random accessing pattern means that each record is randomly selected from the whole database specified by the *DB_Size*. On the other hand, a contiguous accessing pattern means that the records accessed in a TDO step are contiguous (only the first one is randomly selected).

CPU_Scheduling – (real-time) CPU scheduling policy. This parameter specifies the real-time CPU scheduling policy used in the underlying system. In our

experiments, we primarily assume that there is no specific real-time scheduling protocol is used in the underlying system. However, for comparison, EDF (Earliest Deadline First) scheduling is used in one of the experiments.

DD_Scheme – deadlock detection scheme. The schemes (described in Section 7.2) used in an experiment.

The following metrics and statistics are used to evaluate and compare the proposed algorithms and baseline schemes:

- For the overall performance comparison, the following metrics are used:
  - Deadline Guarantee Ratio – the percentage of transactions in a test that are committed by deadline.
  - Record Throughput – the amount of useful work done in one second. Since transactions are of two lengths in the tests, instead of using Transaction Throughput, Record Throughput (records/second) is used for this measurement.
- To evaluate the efficiency of the implemented algorithms,
  - we measure the CPU Utilizations for the deadlock detection, the locking management, and the total system usage.
  - we collect the statistics for the number of locks requested and blocked (i.e., not immediately granted locks), the number of probe computations initiated, and the number of local and global (inter-site) probe messages during the period of a test run time (30 minutes).

7.4 Experimental Results

In addition to the verification of the implemented algorithms (described in Experiment 0), we report on the following four sets of experiments:

1. **The AND Deadlock Model**: The purpose of this experiment is to study and evaluate the performances of various algorithms and baseline schemes in

a typical database environment (i.e., the AND deadlock model system). The algorithms and schemes used in this experiment are: CMH, AND, NDD, W01, W05, and W10.

2. **The CPU Scheduling and the Deadlock Detection**: This experiment is an extension to the first experiment. In this experiment, we study the effect of adding the real-time scheduling policy EDF to systems both with and without the deadlock detection/resolution or the timeout/retry schemes.

3. **The Single-Resource Deadlock Model**: A Single-Resource deadlock model system can be simulated by imposing a restriction on the RT-CARAT system such that a data granule can only be requested and granted exclusively (i.e., write locks only). This is a special case of the system studied in the first experiment (i.e., the AND model system). In this experiment, we also study the efficiency of different types of probe algorithms (forward vs. backward propagations, and simple probes vs. complex chain probes). The algorithms and schemes used in this experiment include CMH, AND, SRD, and NDD.

4. **The OR Deadlock Model**: The purpose of this experiment is to study and evaluate the performances of the OR algorithm and the baseline schemes in a OR deadlock model system. The algorithms and schemes used in this experiment are: ORD, NDD, W01, W05, and W10.

Our data collection is based on the method of *replication*. In the experiments, each test consist of five runs where each run is 30 minutes long. The data are collected and averaged over the five runs of each test. The major performance metric "deadline guarantee ratio" is used to determine the length of each run and the number of runs needed. Thirty minutes is chosen to reduce the transient effect during the test startup period. The deadline guarantee ratio during a test startup period is better than that during the steady state because deadlocks are less likely to happen in the startup period. We gradually increased the test run time and observed that there is no significant differences between the 30 minutes and any run time longer than that. The choice of five runs for each test was determined so that the width of the 95% confidence intervals for the deadline guarantee ratio is

less than 5% (actually, after five runs, it is less than 2% in most of the tests) of the estimated point (i.e., the mean value of the deadline guarantee ratios over the runs in a test).

7.4.1 Experiment 0: Algorithm Verification

The verification of the implemented algorithms was largely done in the debugging phase. The following verification procedures were performed on each of the implemented algorithms:

Execution Trace Analysis : Debugging a distributed program is rather difficult. The technique we use to debug the RT-CARAT implementations is to keep a message log and a trace log for each of the RT-CARAT processes. At the final stage of the implementation, the unnecessary information is eliminated from the trace log files. After each test run, the trace log files are analyzed to check if

- simple local cycles can be quickly detected by the DM's,
- long and nested local cycles (or long and complex local knots) can be correctly detected by the DM's whenever they appear in the local TWFG,
- global cycles (or knots) which involves more than one site can be efficiently detected by the TM's,
- transaction deadlines can be properly handled in the probe propagations such that false detection of the Temporal deadlocks are rare, and
- probe computations can terminate properly (i.e., no infinite propagation loops).

Exhaust Test : A set of exhaust tests are conducted to verify the progress concern of the implemented algorithms (i.e., if a deadlock can be properly detected in finite time). We use very long transaction deadlines (more then 30 minutes) in the long exhaustive tests (more than two hours each). The results of these exhaust tests have shown that (1) thousands of deadlocks are detected and

resolved in each of these tests and (2) all the transactions in these tests are committed by their deadlines. Suppose all types of deadlock situations may have appeared in each of these exhaust tests, the 100% deadline guarantee ratio indicates that all of them are detected and resolved. These results imply that the possibility of severe implementation error is very low.

7.4.2 Experiment 1: The AND Deadlock Model

We begin our experiments with the AND deadlock model system. The AND deadlock model system is very common in databases. In RT-CARAT, if both "read" and "write" locks can be acquired for a database page, the TWFG depicts an AND deadlock system. For example, suppose two transactions share a read lock to a database page and a third transaction wants a exclusive write lock to the same page. The third transaction has to wait for *both* read lock holders to release the lock to the page.

Table 7.1 summarizes the parameter settings of this experiment. When a transaction is initiated,

- it is 50% chance short and 50% chance long. A short transaction has 4 TDO steps while a long one has 12 TDO steps. Each TDO step accesses 4 records. Each record is randomly selected from a database with a size specified by the parameter *DB_Size*.
- it is 50% chance read-only and 50% chance write. In a write transaction, a TDO step is 50% chance read-only and 50% chance write-only.
- it is 50% chance local and 50% chance distributed. In a distributed transaction, TDO steps are evenly distributed over three sites: one coordinator site and two remote slave sites.

There are 8 TR processes at each site. The database size is varied from 1000 to 3000 pages per site. The shortest deadline window [20-80, 6-24] is determined by a set of pilot tests. In these pilot tests, we observe that 60-80% of the transactions can commit by deadline with this deadline window setting. In the experiment, the

deadline window is varied from this shortest setting to a four times longer setting. The algorithms and the baseline schemes used in this experiment are: CMH, AND, NDD, W01, W05, and W10.

Table 7.1: Experiment 1 Parameter Settings

Parameters	Settings
Fixed Settings	
TR_Length	[12, 4; 4, 4]
Pr_Long_TR	50%
Pr_Write_TR	50%
Pr_Write_TDO	50%
Pr_Dist_TR	50%
N_Rem_Sites	2
MPL	8 TR's per site
Acc_Pattern	random
CPU_Scheduling	non-real-time
IO_Delay	200 msecs
Varied Settings	
DB_Size	1000, 1667, 2333, and 3000 pages per site
DL_Window	[20-80,6-24], [40-160,12-48], [60-240,18-72], [80-320,24-96], and [∞-∞,∞-∞]
DD_Scheme	CMH, AND, NDD, W01, W05, and W10

Figures 7.2 and 7.3 compare all of the schemes listed above with respect to deadline guarantee ratio. Figures 7.4 and 7.5 compare these schemes with respect to record throughput. Overall, the two detection/resolution schemes, i.e., the AND and the CMH schemes, perform about the same and they are the best schemes in this experiment. Slightly below these two detection/resolution schemes are the two timeout/retry schemes W05 and W10. Among the three timeout/retry schemes W01 is significantly worse than the other two. This indicates that most of the outstanding requests can be granted within 5 seconds, but the one second timeout period is too short.

NDD is the worst scheme in this experiment. In terms of deadline guarantee ratio, NDD is not sensitive to the deadline window (Figure 7.2). However, in Figure 7.4, the performance of NDD drops dramatically when the transaction deadlines become longer. This result indicates that without any scheme (either detection/resolution or timeout/retry) to deal with deadlocks, a longer deadline window setting might actually degrade the system performance because it causes transactions' average waiting state longer.

Figures 7.6 and 7.7 show that the majority of lock requests (in the period of 30 minutes test run time) are granted immediately. Only a small number of the lock requests are blocked and part of the blocked requests may cause the initiation of deadlock detection computations. Also, in Figures 7.8 and 7.9, the amount of CPU power used for deadlock detection is very small (less than 2%) compared to the CPU utilizations for the locking management (about 30%) and the system total (70-80%). Although more than ten thousand probes may be processed in a 30 minutes test run time (Figures 7.10 and 7.11), they only consume less than 2% of the CPU power. All these results show that a deadlock detection algorithm implemented in a system can be very efficient and incurs very little overhead. These results also explain why there are no significant differences between the two deadlock detection algorithms tested in this experiment in terms of deadline guarantee ratio and record throughput.

The probe statistics (Figures 7.10 and 7.11) show that the AND algorithm invokes probe initiation less frequently and requires less probe propagations than the CMH algorithm. This makes the AND algorithm more attractive than the CMH algorithm in the systems where the probe propagations are complicated and expensive. For example,

- in a wide area communication network where each probe message propagation is routed through many intermediate sites, the delay of a probe propagation is long and the overhead it incurs is not low.
- in some systems, the resolution of a detected deadlock requires information (e.g., the criticalness, the deadline, and the priority of each deadlocked task)

which should be collected by the probes. Processing and propagating such complicated probes can be very expensive.

To reduce the frequency of both the probe computation initiation and the probe propagation is crucial in such systems.

As described in Section 7.2, the CMH algorithm is optimized to reduce the inter-site probe propagations. However, this optimization does not work well under high data contention workloads. In Figure 7.11, the global probe message incurred by the CMH algorithm is more than that incurred by the AND algorithm when the data contention is high (i.e., database size is 1000 or 1667 pages). This is because under the high data contention workload, the TWFG becomes more complicated (edges are created due to data competitions), and the situation that a DM finishes a local search without finding any cycle while the inter-site probe propagations are needed is more likely to happen in a more complicated TWFG. Consequently, the inter-site global probe message rate is relatively higher under the high data contention workloads.

7.4.3 Experiment 2: The CPU Scheduling and the Deadlock Detection

This experiment is an extension of Experiment 1. In this experiment, we study the effect of a real-time scheduling policy EDF in systems both with and without the deadlock detection/resolution or the timeout/retry schemes. Since EDF scheduling utilizes VAX/VMS real-time priorities 16-30, the maximum number of DM servers at each site has to be limited to 15 (in the worse case, each DM needs to be assigned to a site-wide unique priority). Therefore, the parameter settings of this experiment (shown in Table 7.2) are scaled down from Experiment 1 as follows. The multiprogramming level is reduced to 4 TR processes at each site. To maintain a similar data contention level, the database size is varied from 500 to 2000 pages per site. Since the multiprogramming level is reduced to half of the setting in Experiment 1, the average response time will be about half of that in Experiment 1. Therefore the deadline window is set to 1/2 and 2/3 of those settings

in Experiment 1 for the long and the short transactions respectively. The 2/3 deadline window reduction reflects the fact that constant overhead (e.g., transaction start and commitment overhead) is significant in short transactions. Again, the algorithms and the baseline schemes used in this experiment include CMH, AND, NDD, W01, W05, and W10.

Table 7.2: Experiment 2 Parameter Settings

Parameters	Settings
Fixed Settings	
TR_Length	[12, 4; 4, 4]
Pr_Long_TR	50%
Pr_Write_TR	50%
Pr_Write_TDO	50%
Pr_Dist_TR	50%
N_Rem_Sites	2
MPL	4 TR's per site
Acc_Pattern	random
IO_Delay	200 msecs
Varied Settings	
DB_Size	500, 1000, 1500, and 2000 pages per site
DL_Window	[10-40,4-16], [20-80,8-32], [30-120,12-48], and [40-160,16-64]
CPU_Scheduling	non-real-time and EDF
DD_Scheme	CMH, AND, NDD, W01, W05, and W10

Overall, the EDF scheduling increases the deadline guarantee ratio by about 2-5% and the throughput by about 1-2 records/second (Compare Figures 7.12, 7.14, 7.16, and 7.18 with Figures 7.13, 7.15, 7.17, and 7.19 respectively). All the performance trends remain the same as observed in Experiment 1.

Figures 7.20-7.23 depict the probe statistics with and without EDF scheduling. When EDF scheduling is in use, the probe messages are moderately reduced. One interesting phenomenon we have observed is that the number of probe computation initiations by the AND algorithm is decreased 50% by EDF scheduling while the reduction of the AND probe messages does not reflect this ratio. This is because

in the AND algorithm a transaction can "append" itself to the chain of an existing probe instead of creating a new one. Under the EDF scheduling, the AND algorithm is less likely to initiate new probes and, hence, the ongoing probe computations get more chance to continue. This also implies that, in average, a deadlock cycle can be detected earlier by the AND algorithm when EDF scheduling is in use.

7.4.4 Experiment 3: The Single-Resource Deadlock Model

The Single-Resource deadlock model system is a special case of the AND model system. A Single-Resource deadlock model system can be simulated by imposing a restriction on the RT-CARAT system such that a data granule can only be requested and granted exclusively (i.e., write locks only). The goal of this experiment is to study the efficiency of different types of probe algorithms (forward vs. backward propagations, and simple probes vs. complex chain probes). Table 7.1 summarizes the parameter settings of this experiment. These parameter settings are similar to those in Experiment 1 except that all the transactions are of the write-only type and the number of remote slave sites for the distributed transactions is varied from 1 to 2. The algorithms and the baseline schemes used in this experiment include CMH, AND, SRD, and NDD.

Again, overall, all the three algorithms perform about the same with respect to both the deadline guarantee ratio (Figures 7.24-7.27) and the record throughput (Figures 7.28-7.31).

One of the goals of this experiment is to study and compare the forward and the backward probe propagations. Since the parameter *N_Rem_Sites* which sets the number of slave sites for the distributed transactions will affect the backward propagation more than the forward propagation, probe statistics are compared under both 1 and 2 slave sites settings. Note that there are about 5-10% more lock requests in the 1 slave site tests than in the 2 slave sites tests (Figures 7.32-7.35). This is because the mean response time for the 1 slave site transactions is shorter

Table 7.3: Experiment 3 Parameter Settings

Parameters	Settings
Fixed Settings	
TR_Length	[12, 4; 4, 4]
Pr_Long_TR	50%
Pr_Write_TR	100%
Pr_Write_TDO	100%
Pr_Dist_TR	50%
MPL	8 TR's per site
Acc_Pattern	random
CPU_Scheduling	non-real-time
IO_Delay	200 msecs
Varied Settings	
DB_Size	1000, 1667, 2333, and 3000 pages per site
DL_Window	[20-80,6-24], [40-160,12-48], [60-240,18-72], [80-320,24-96], and [∞-∞,∞-∞]
N_Rem_Sites	1, 2
DD_Scheme	CMH, AND, SRD, and NDD

than that for the 2 slave sites transactions. With this in mind we now proceed with the following analysis.

Figures 7.36-7.39 show the number of probe computation initiations for these three algorithms. When the system workload is changed from 2 to 1 slave site, the probe initiation rates for both the CMH and the SRD algorithms increase slightly. However, the probe initiation rates decrease slightly for the AND algorithm when the system is changed to 1 slave site workload. A possible explanation is that the TWFG is simpler (less incoming edges from the remote sites) under the 1 slave site workload and the AND algorithm can benefit from it.

Figures 7.40-7.43 show the statistics for the probe messages. In most of the tests, the AND algorithm generates the smallest number of local and global probe messages. In general, the CMH algorithm requires the most local messages while the SRD algorithm requires the most global messages. The number of probe messages

due to the SRD algorithm is very sensitive to the number of slave sites, the deadline window, and the data contention. In terms of the probe message rate, the SRD algorithm becomes the best among the three algorithms when the system workload is set to 1 slave site, the shortest deadline window, and the largest database size.

7.4.5 Experiment 4: The OR Deadlock Model

The purpose of this experiment is to study and evaluate the performances of the OR algorithm and the baseline schemes in the OR deadlock model systems. In an OR deadlock model system, a transaction may proceed if *any* of the records requested in a TDO step can be granted. If the number of the records requested in a TDO step is increased, not only does the probability that a TDO step is blocked decrease but also the average size of the knots increase. Also, knots are more complicated than cycles because, in general, more transactions and more wait-for edges are involved in a knot than in a cycle. In this experiment, we need to greatly reduce the database size to raise the data contention to a level such that enough number of knots can be generated in the tests. In a set of pilot tests we have found that a satisfactory setting is such that each TDO step should be limited to two requests and the database size should be smaller than 100 pages. Since in each TDO step only one database operation is actually performed, to maintain the average length of the transactions' execution time to be comparable to the other experiments, both the number of TDO steps per transaction and the I/O delay are increased by 50%. In this experiment, in addition to the random accessing pattern, the contiguous accessing pattern is also tested. In a test with the contiguous accessing pattern, 5/6 of TDO steps will map their two requested records into the same page. In other words, 5/6 of the transactions in the TWFG will have only one outgoing edge while the remaining 1/6 of the transactions will have two outgoing edges. Comparing to the random accessing pattern setting, the contiguous accessing pattern will result in a higher frequency of deadlock occurrences and, in general, the deadlock structure is simpler. The algorithms and the baseline schemes used in

this experiment are: ORD, NDD, W01, W05, and W10. Table 7.4 summarizes the parameter settings used in this experiment.

Table 7.4: Experiment 4 Parameter Settings

Parameters	Settings
Fixed Settings	
TR_Length	$[18, 2; 6, 2]$
Pr_Long_TR	50%
Pr_Write_TR	100%
Pr_Write_TDO	100%
Pr_Dist_TR	50%
N_Rem_Sites	2
MPL	8 TR's per site
CPU_Scheduling	non-real-time
IO_Delay	300 msecs
Varied Settings	
DB_Size	30, 50, 70, and 90 pages per site
DL_Window	[20-80,6-24], [40-160,12-48], [60-240,18-72], and [80-320,24-96]
Acc_Pattern	random and contiguous
DD_Scheme	ORD, NDD, W01, W05, and W10

Figures 7.44-7.47 compare the algorithms and the schemes with respect to the deadline guarantee ratio. Figures 7.48-7.51 compare the algorithms and the schemes with respect to the record throughput. Again, the NDD is the worst scheme in this experiment.

Overall, among the three timeout/retry schemes, W05 outperforms the other two. In Figures 7.46, 7.47, 7.50, and 7.51, W10 outperforms W01 when the data contention is low (i.e., *DB_Size* = 90). The reverse is the case when the data contention is high (i.e., *DB_Size* = 30). Also, the nearly linear curves shown in these figures imply that W01 may be the best scheme when *DB_Size* > 90 and W01 may be the best scheme when *DB_Size* < 30. This observation tells us that the choice of a timeout period is not an easy task.

When the record accessing pattern is random, the SRD algorithm performs worse than W05 in most of the tests. There are three possible causes for the relatively poor performance of the SRD detection/resolution scheme:

1. To rollback a transaction is simple in RT-CARAT which favors timeout/retry schemes.

2. The resolution of a detected knot is not good enough to allow most of the surviving transactions to meet their deadlines.

3. There are probably many long wait-for paths in the TWFG.

The first cause is obvious. To see why the latter two are possible causes let's consider two more cases:

- When the record accessing pattern is contiguous, the SRD algorithm performs very close to the W05 scheme.

- In Experiment 1, all the detection/resolution schemes outperform all the timeout/retry schemes.

Table 7.5 shows the average cyclic wait lengths found in the tests of the above three cases. When a deadlock is a knot, the length of the longest cycle in the knot is used as the cyclic wait length of that knot. Although the average cyclic wait length of the knots only range from 4.1 to 6.5 transactions, the size of a knot is usually more than 10 transactions. The cyclic wait length of a deadlock (a cycle or a knot) can be used as an index to indicate how "long" the surviving transactions should be waiting. Also, the average cyclic wait length in a system is related to the average length of the wait-for paths in the TWFG.

Table 7.5: Mean Cyclic Wait Length

Test Settings	Mean Cyclic Wait Length
OR system/random access	6.0 – 6.5 transactions
OR system/contiguous access	4.1 – 4.6 transactions
AND system/random access	2.8 – 3.5 transactions

Table 7.5 suggests that when the mean cyclic wait length is longer than 4 transactions in a test, we should consider

- using a smarter resolution policy to break detected deadlocks.
- breaking a long wait-for path even before it forms a deadlock.

This also suggests that an integrated solution can be developed for both Non-deadlocked Blocking situations and Temporal deadlocks in a real-time system (Section 2.6.1).

7.5 Summary of Results and Conclusions

To verify the algorithms and to study their performance, we have implemented three classes of schemes in the RT-CARAT system: (1) break by deadline, (2) time-out/retry, and (3) detection/resolution. For the detection/resolution schemes, four algorithms are implemented and evaluated in the experiments. These algorithms include the Single-Resource Algorithm (Section 5.3.4), the AND Algorithm (Section 5.4.4), the OR Algorithm, and the resource deadlock detection algorithm proposed by Chandy, Misra, and Haas [27]. In general, our experimental results indicate the following:

- Distributed deadlock detection can be efficiently realized. Because they can be efficiently realized, the overall performance differences (in terms of deadline guarantee ratio and throughput) between related algorithms are not significant.
- Compared to the baseline scheme break by deadline, any of these deadlock detection algorithms can significantly improve the system performance in terms of the deadline guarantee ratio and the throughput.
- Compared to the timeout/retry schemes, the detection/resolution scheme performs better when the deadlocks are simple and short (e.g., the Single-Resource and the AND deadlocks). However, the detection/resolution scheme

performs worse than timeout and retry when the average cyclic wait length is long (e.g., the OR deadlocks).

- EDF scheduling slightly improves the overall system performance but does not affect the trend of the performance results.
- In terms of the probe message rate, the AND Algorithm is the best (i.e., requires the least probe messages) among the three cycle detection algorithms in most of the tests. However, the Single-Resource Algorithm becomes the best one when the system workload is set to 1 slave site, the shortest deadline window, and the largest database size.

The results of the experiments conducted in this chapter indicate that when the mean cyclic wait length is long, a good deadlock resolution policy is necessary to manage to allow most of the surviving transactions to meet their deadlines. Also, the length of the mean cyclic wait length in a system is related to the average length of the wait-for paths in the TWFG of that system. This suggests that an integrated solution should be developed for both the Non-deadlocked Blocking situations and the Temporal deadlocks in a real-time system.

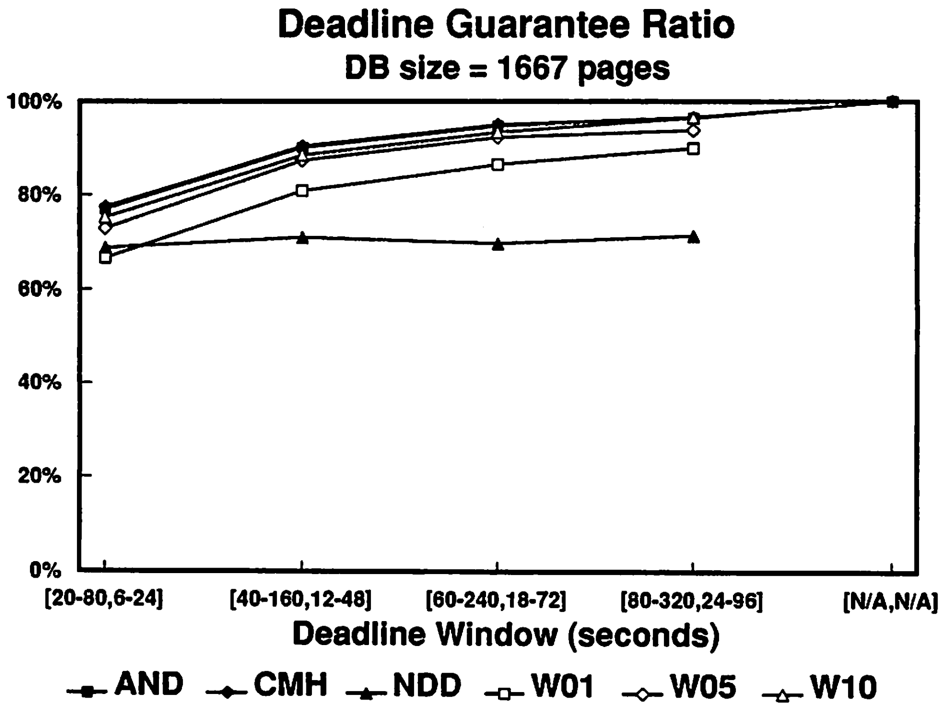


Figure 7.2: Deadline Guarantee Ratio, AND Model System, DB Size 1667 Pages

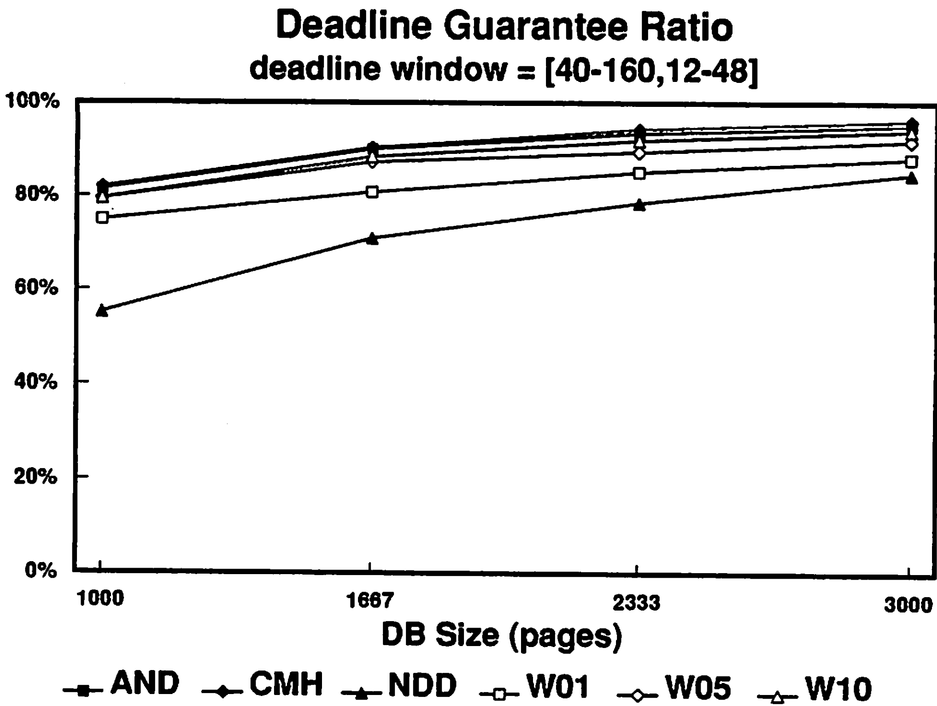


Figure 7.3: Deadline Guarantee Ratio, AND Model System, DL Window [40-160,12-48]

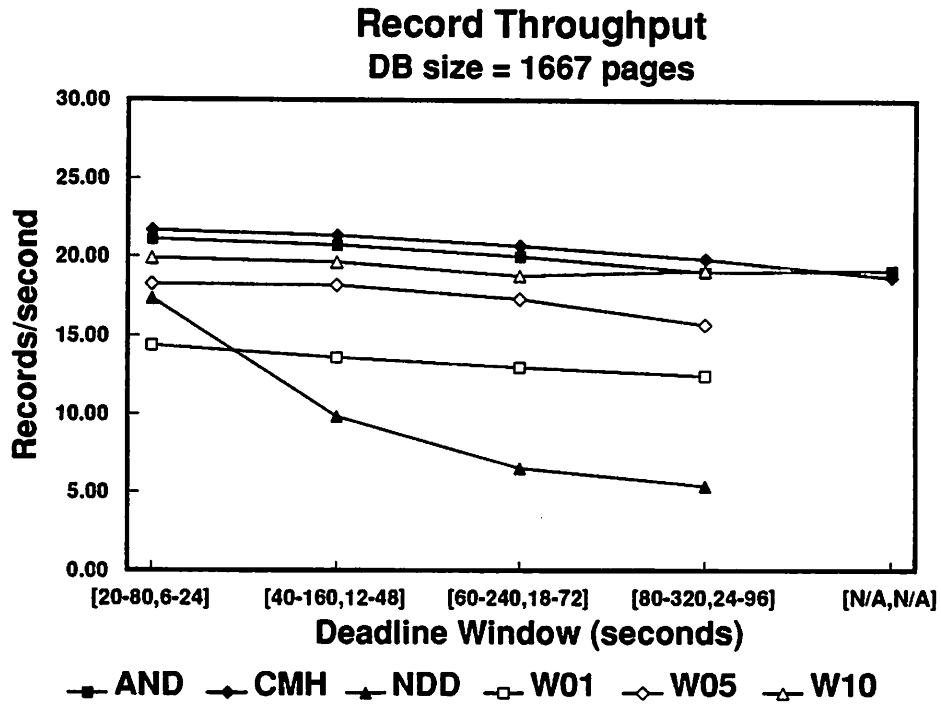


Figure 7.4: Record Throughput, AND Model System, DB Size 1667 Pages

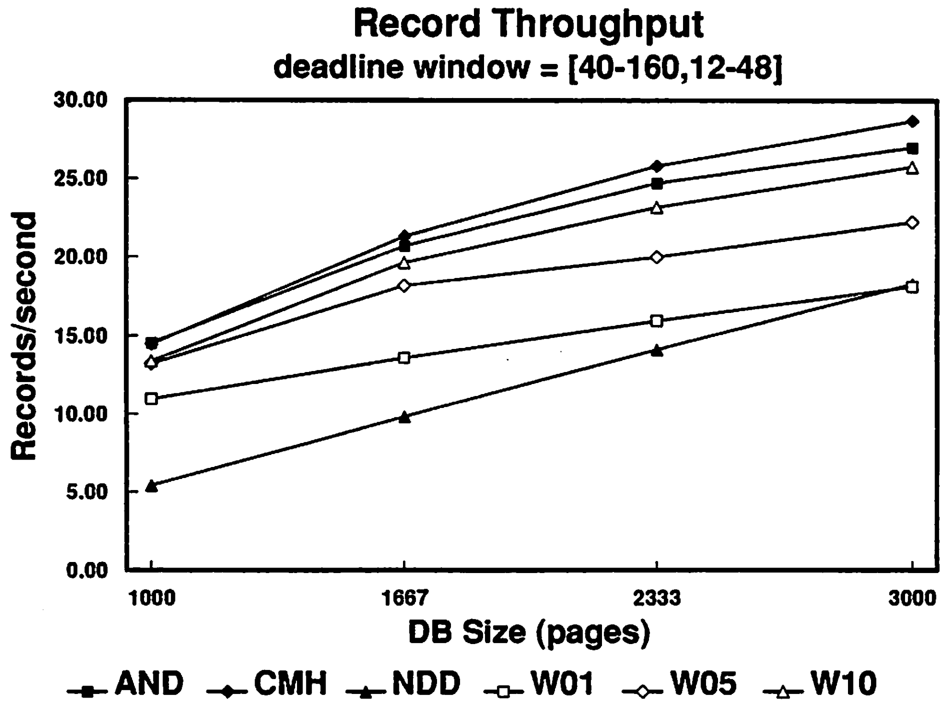


Figure 7.5: Record Throughput, AND Model System, DL Window [40-160,12-48]

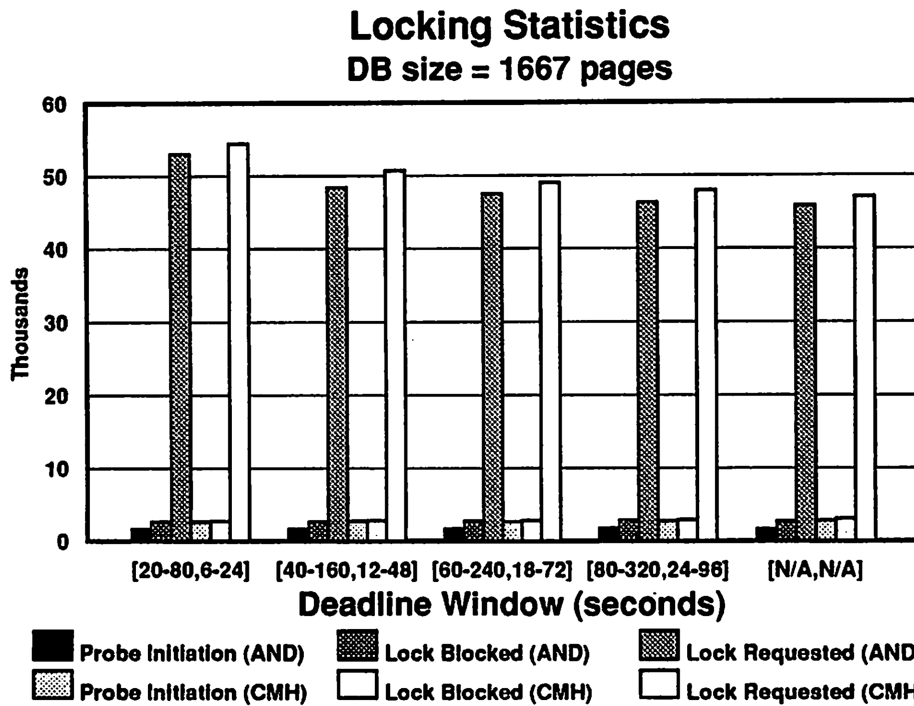


Figure 7.6: Locking Statistics, AND Model System, DB Size 1667 Pages

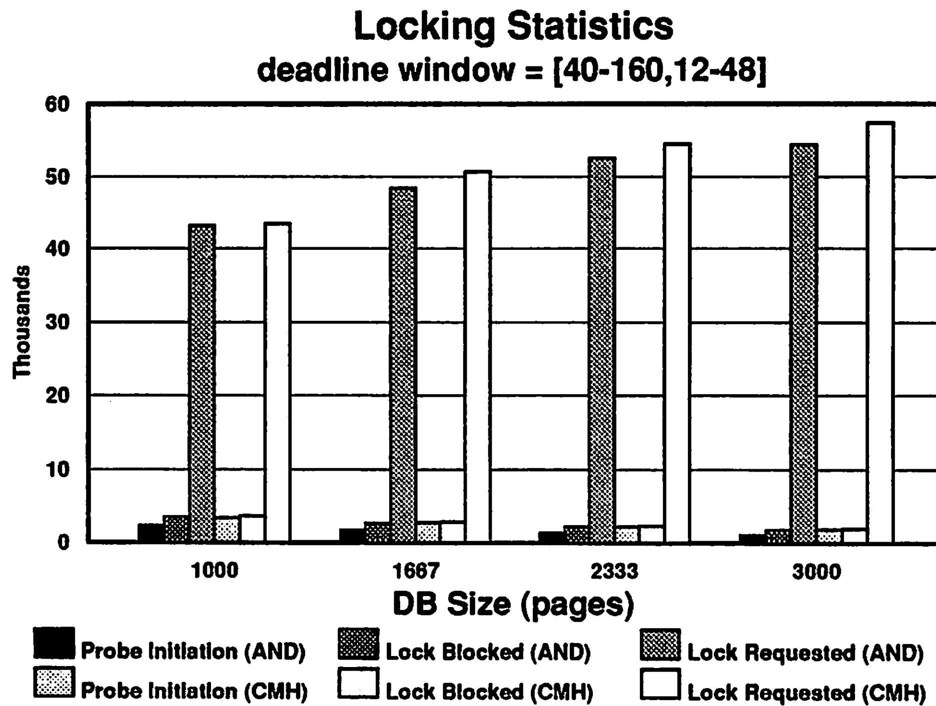


Figure 7.7: Locking Statistics, AND Model System, DL Window [40-160,12-48]

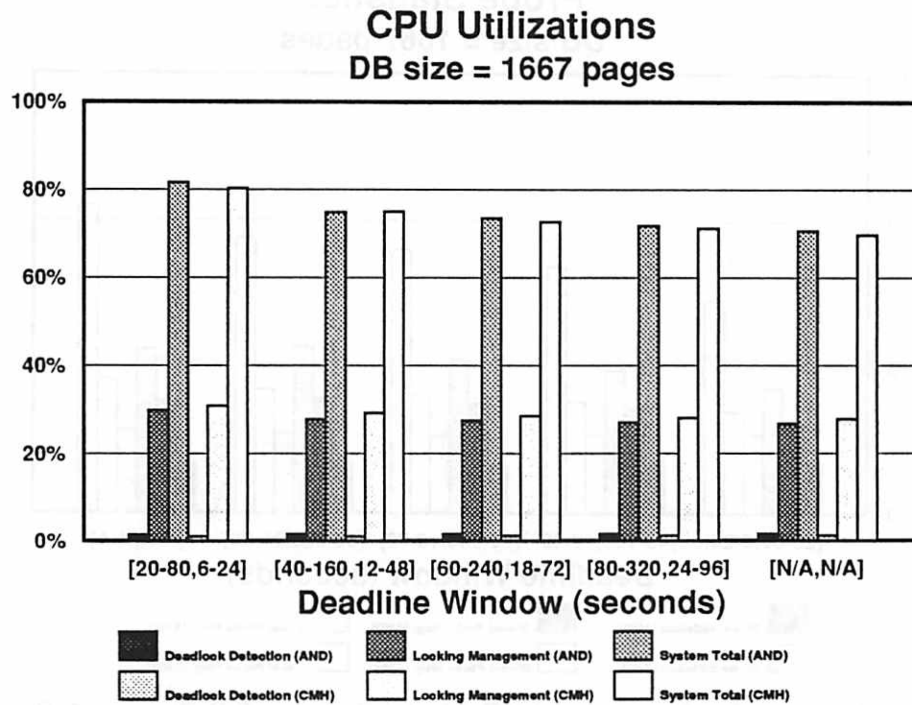


Figure 7.8: CPU Utilization, AND Model System, DB Size 1667 Pages

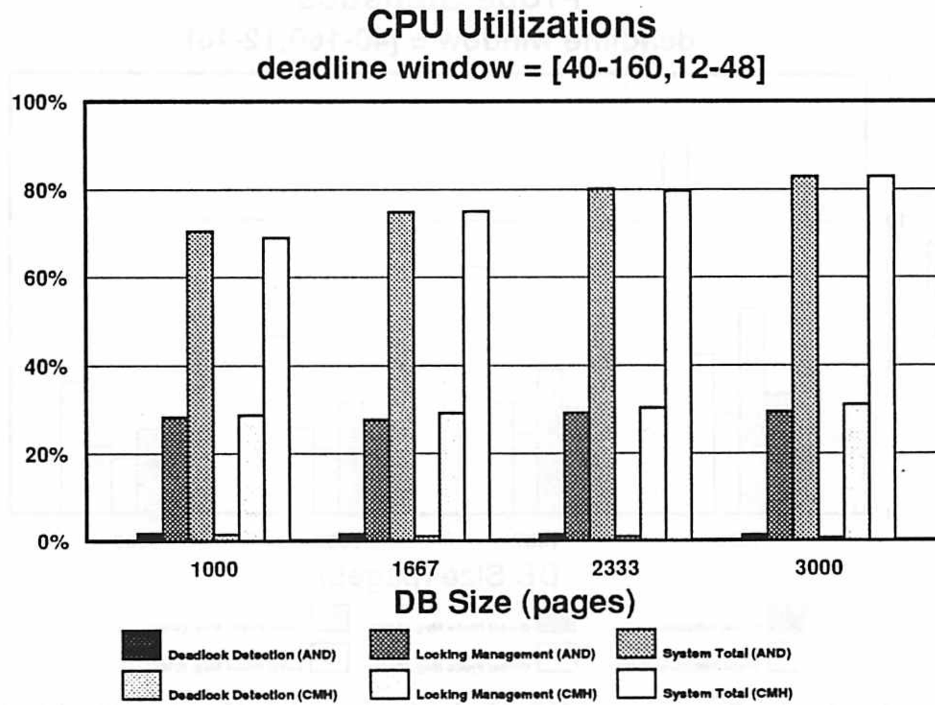


Figure 7.9: CPU Utilization, AND Model System, DL Window [40-160,12-48]

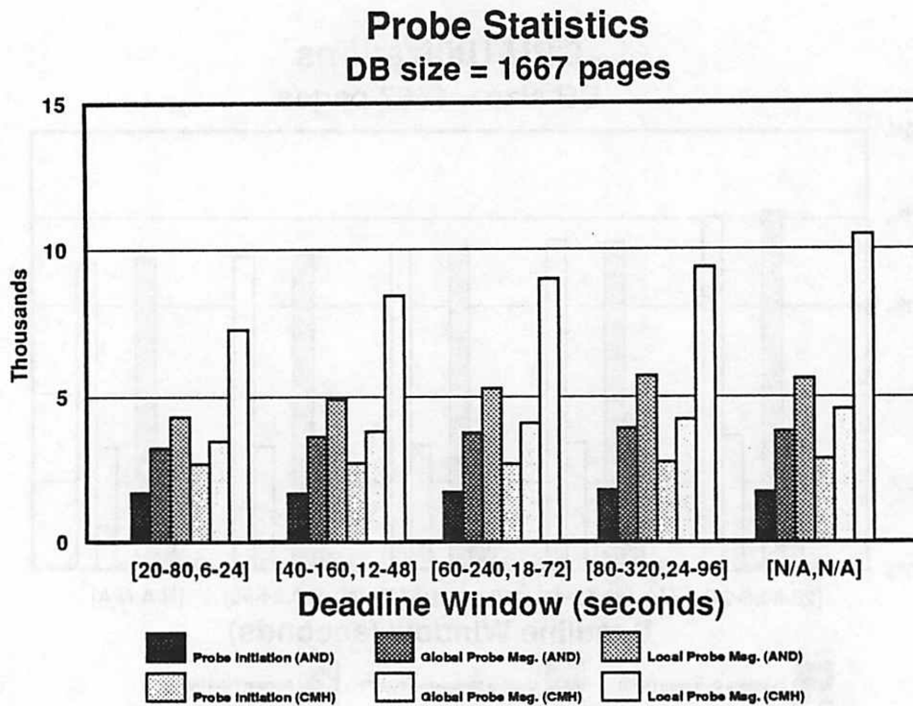


Figure 7.10: Probe Statistics, AND Model System, DB Size 1667 Pages

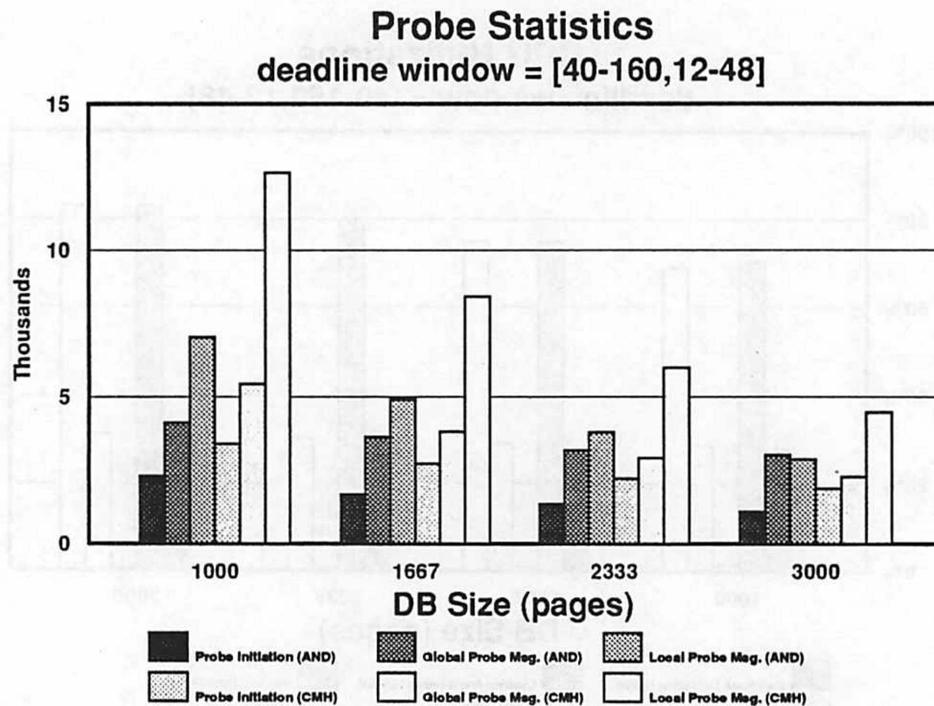


Figure 7.11: Probe Statistics, AND Model System, DL Window [40-160,12-48]

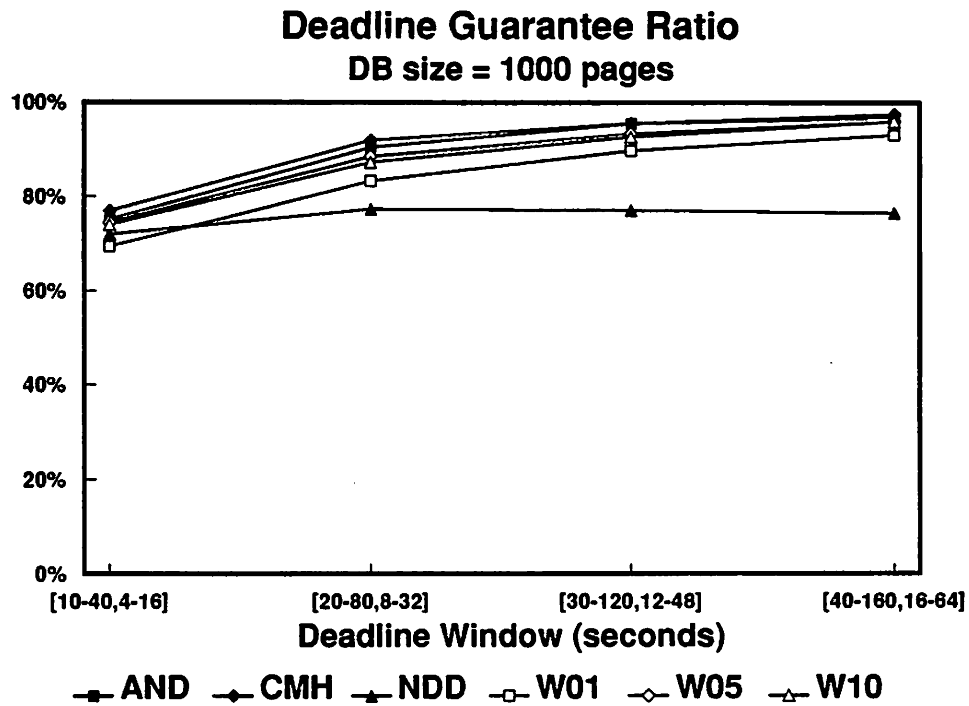


Figure 7.12: Deadline Guarantee Ratio, AND Model System, Non-Real-Time Scheduling, DB Size 1000 Pages

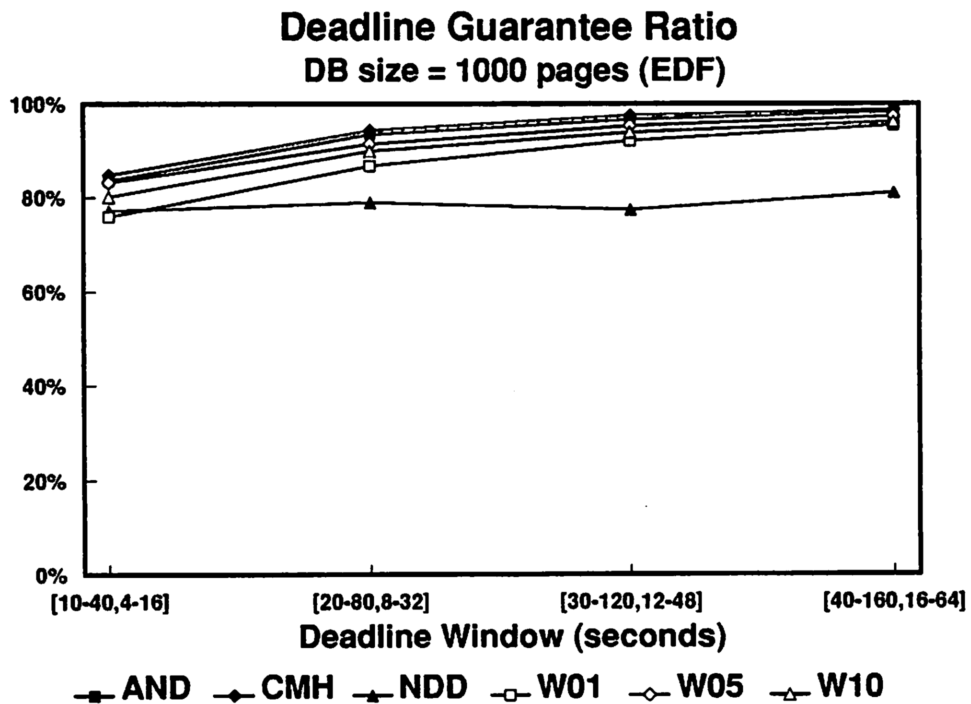


Figure 7.13: Deadline Guarantee Ratio, AND Model System, EDF Scheduling, DB Size 1000 Pages

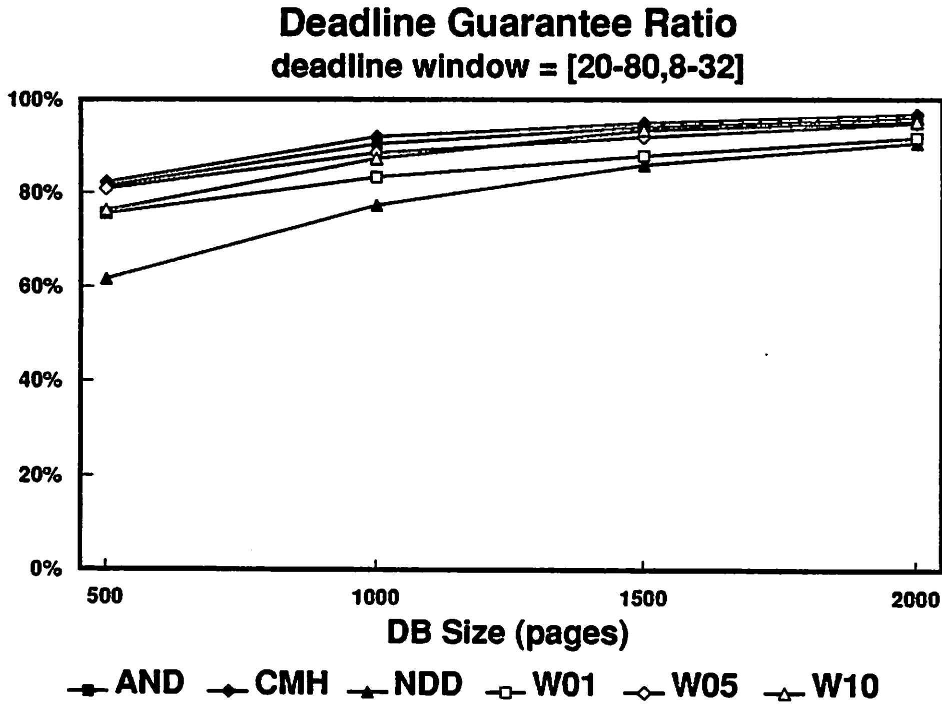


Figure 7.14: Deadline Guarantee Ratio, AND Model System, Non-Real-Time Scheduling, DL Window [20-80,8-32]

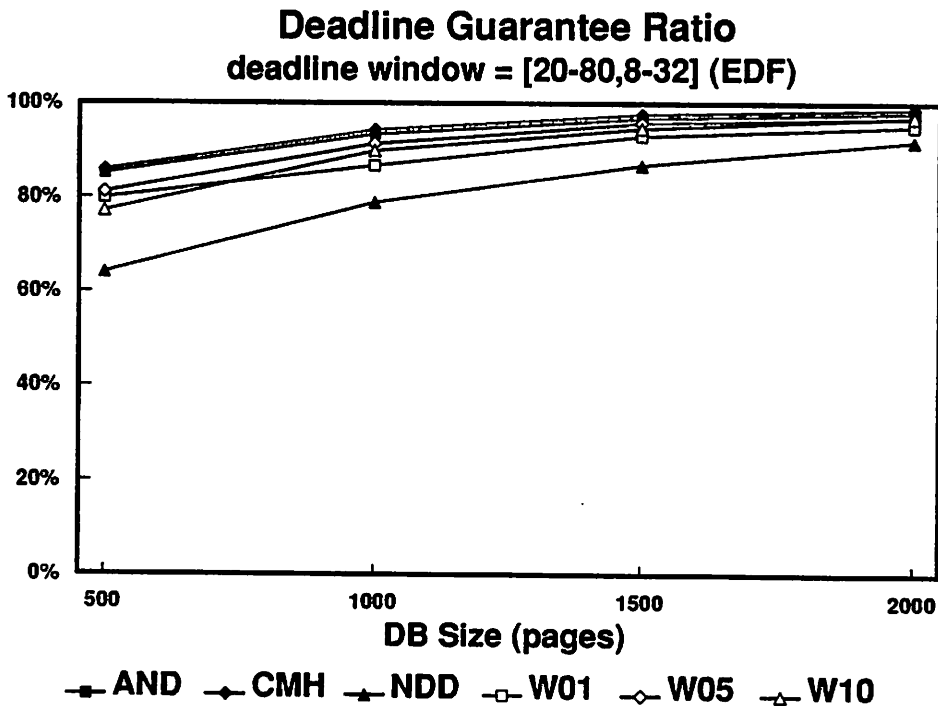


Figure 7.15: Deadline Guarantee Ratio, AND Model System, EDF Scheduling, DL Window [20-80,8-32]

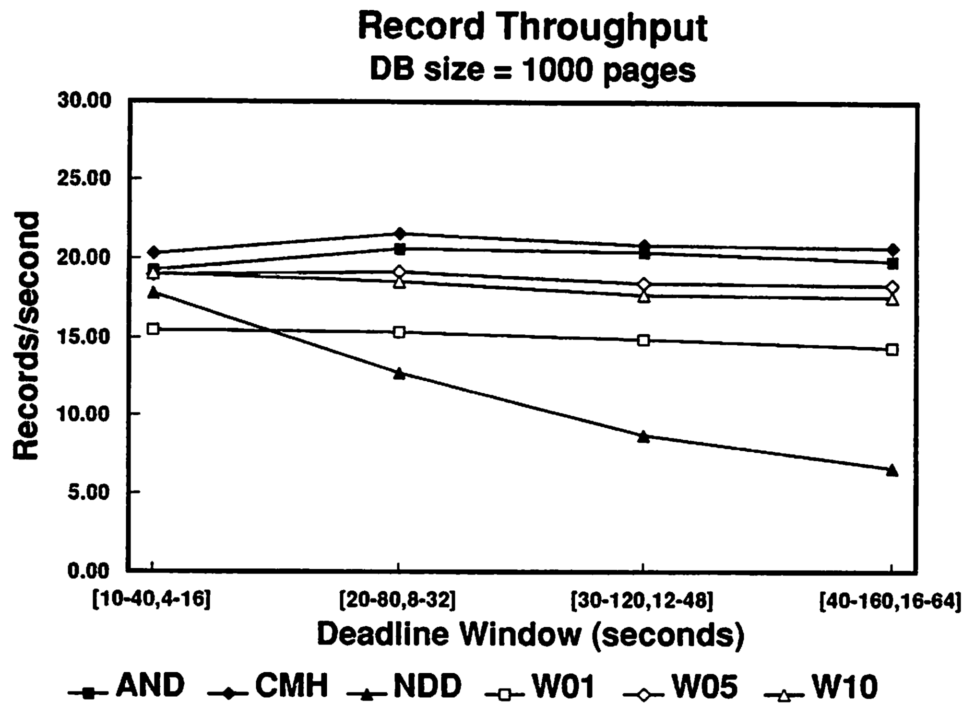


Figure 7.16: Record Throughput, AND Model System, Non-Real-Time Scheduling, DB Size 1000 Pages

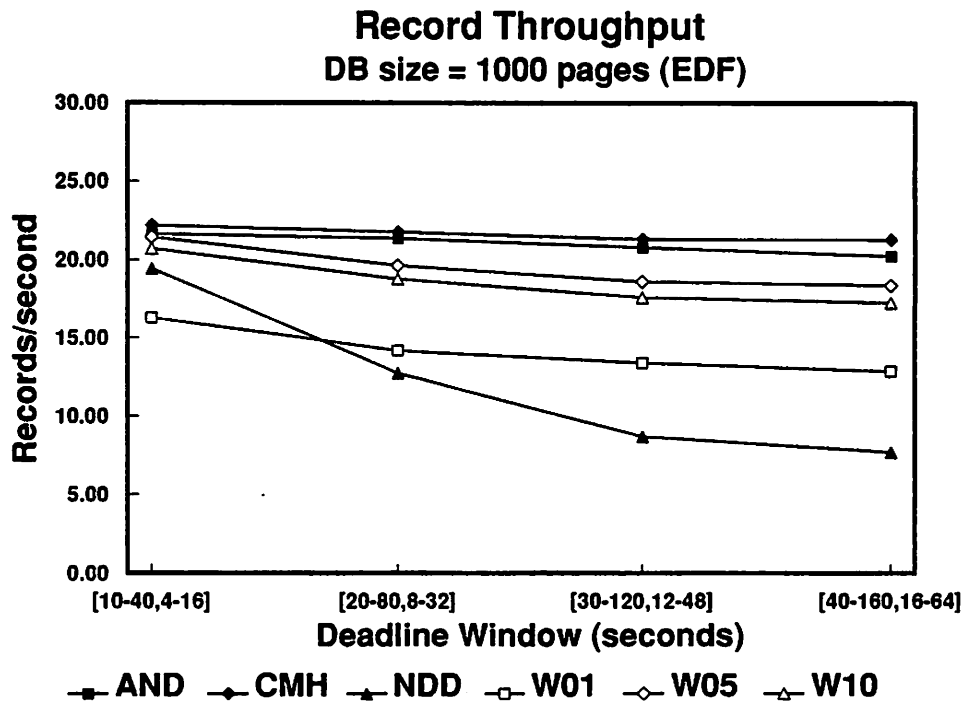


Figure 7.17: Record Throughput, AND Model System, EDF Scheduling, DB Size 1000 Pages

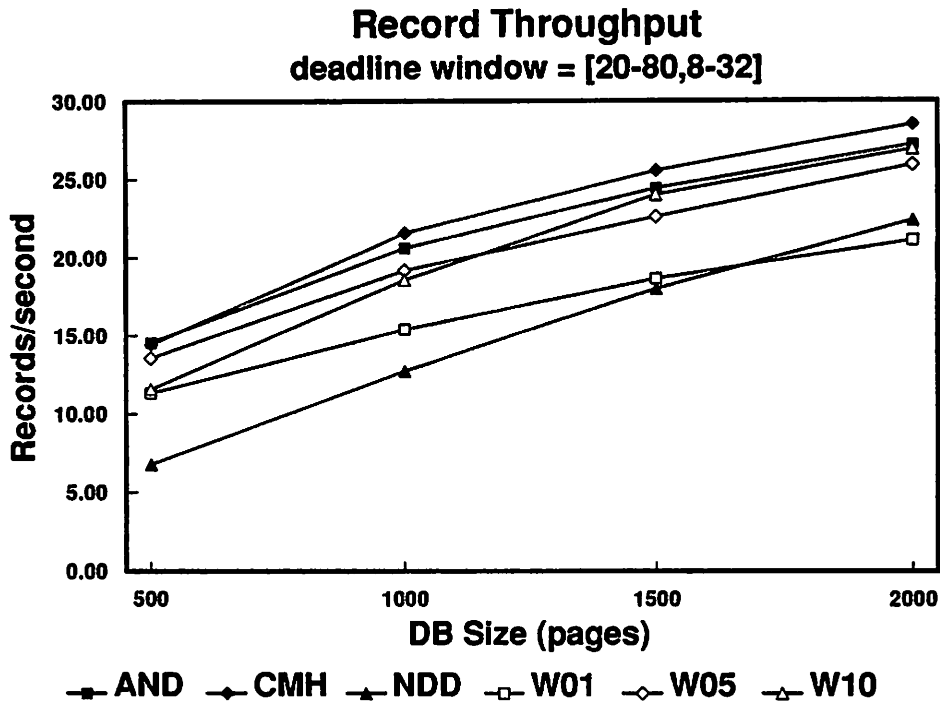


Figure 7.18: Record Throughput, AND Model System, Non-Real-Time Scheduling, DL Window [20-80,8-32]

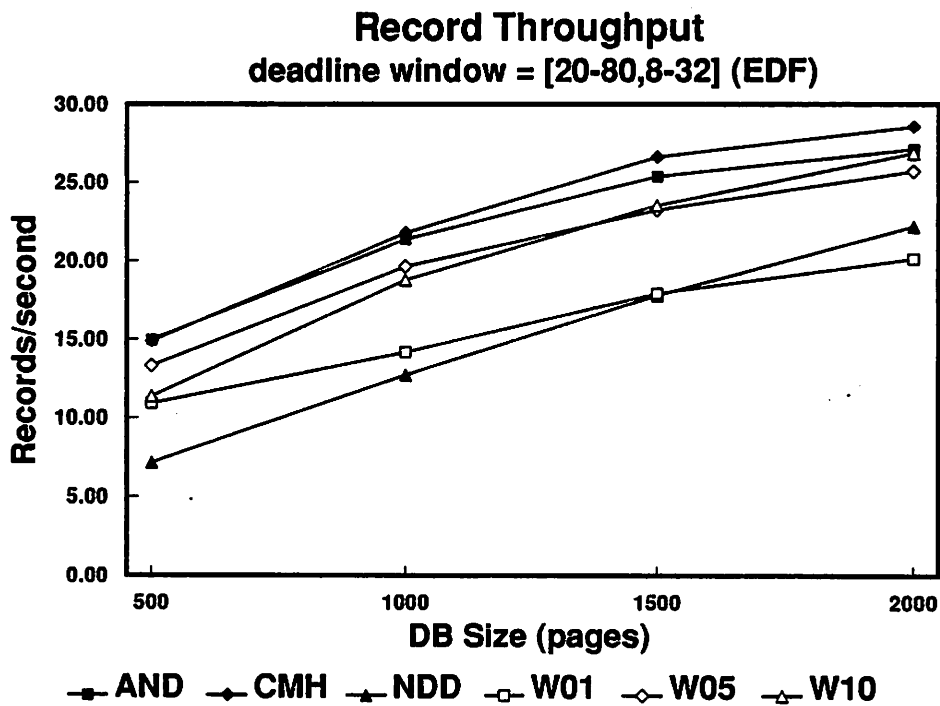


Figure 7.19: Record Throughput, AND Model System, EDF Scheduling, DL Window [20-80,8-32]

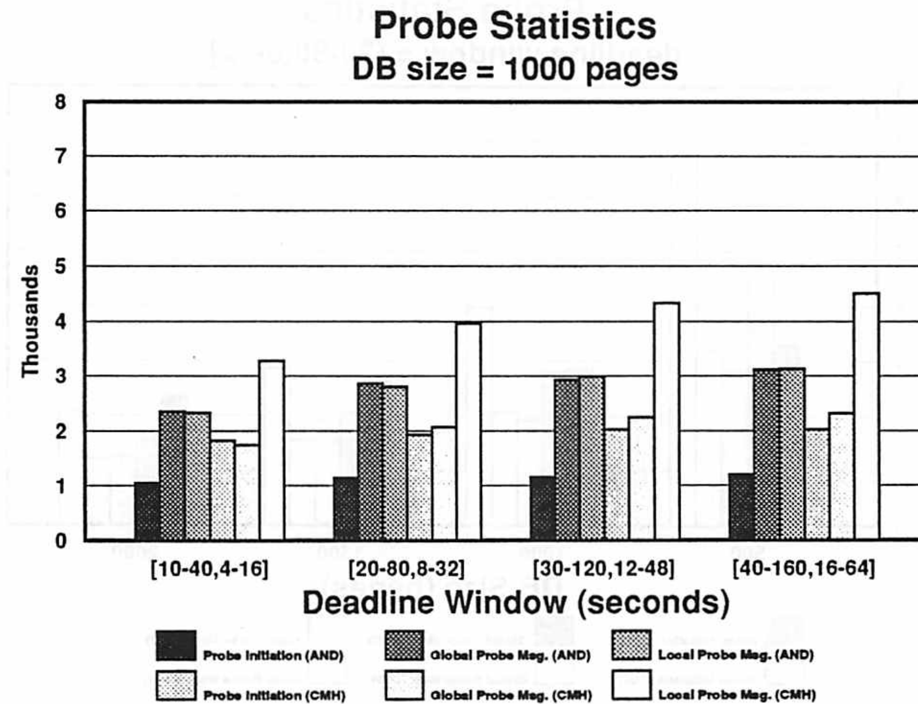


Figure 7.20: Probe Statistics, AND Model System, Non-Real-Time Scheduling, DB Size 1000 Pages

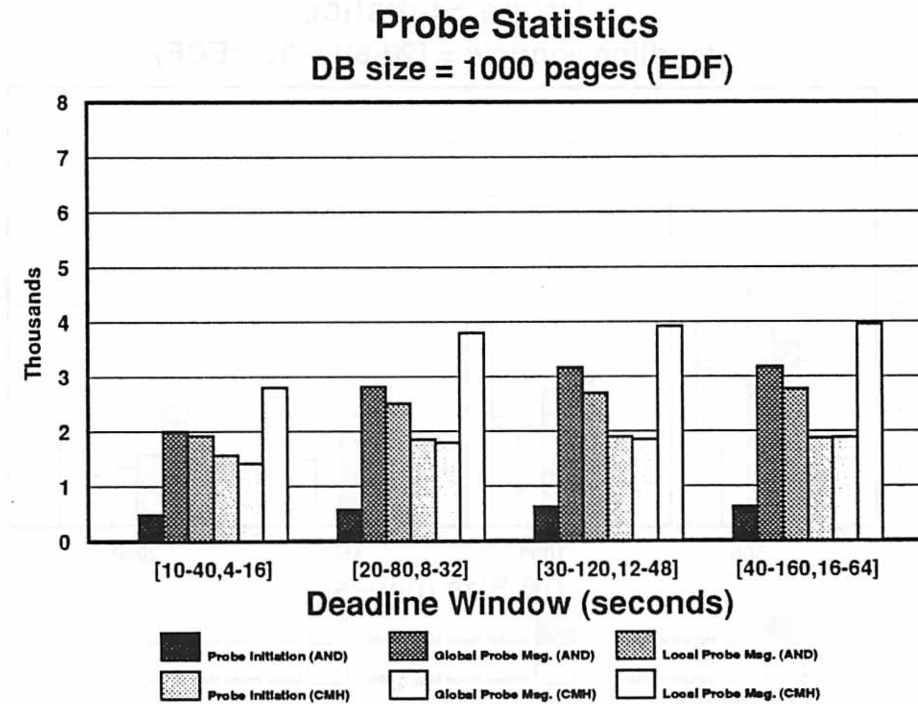


Figure 7.21: Probe Statistics, AND Model System, EDF Scheduling, DB Size 1000 Pages

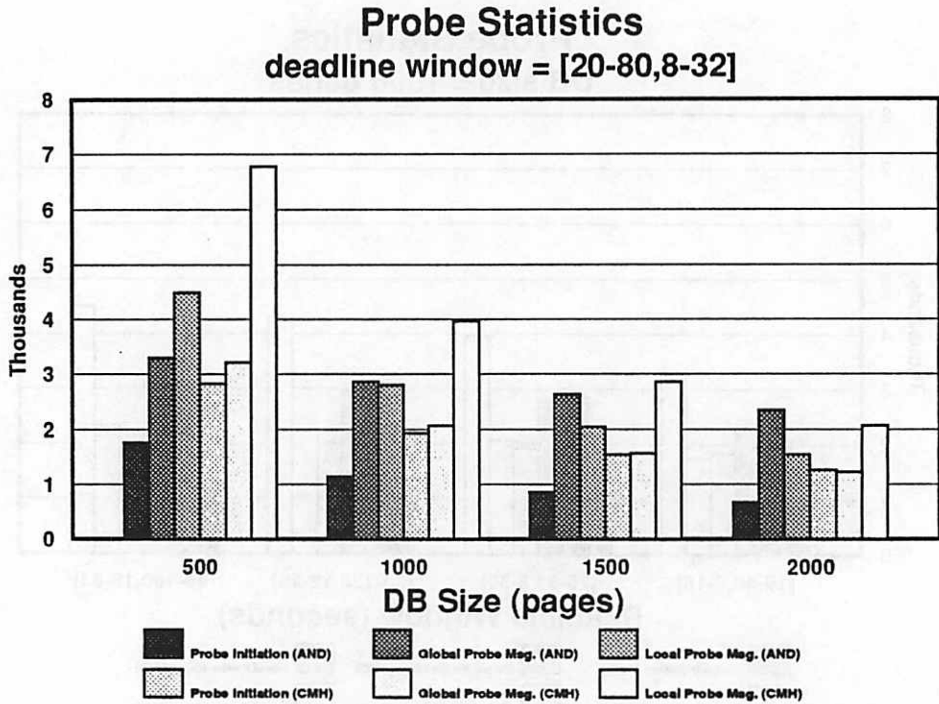


Figure 7.22: Probe Statistics, AND Model System, Non-Real-Time Scheduling, DL Window [20-80,8-32]

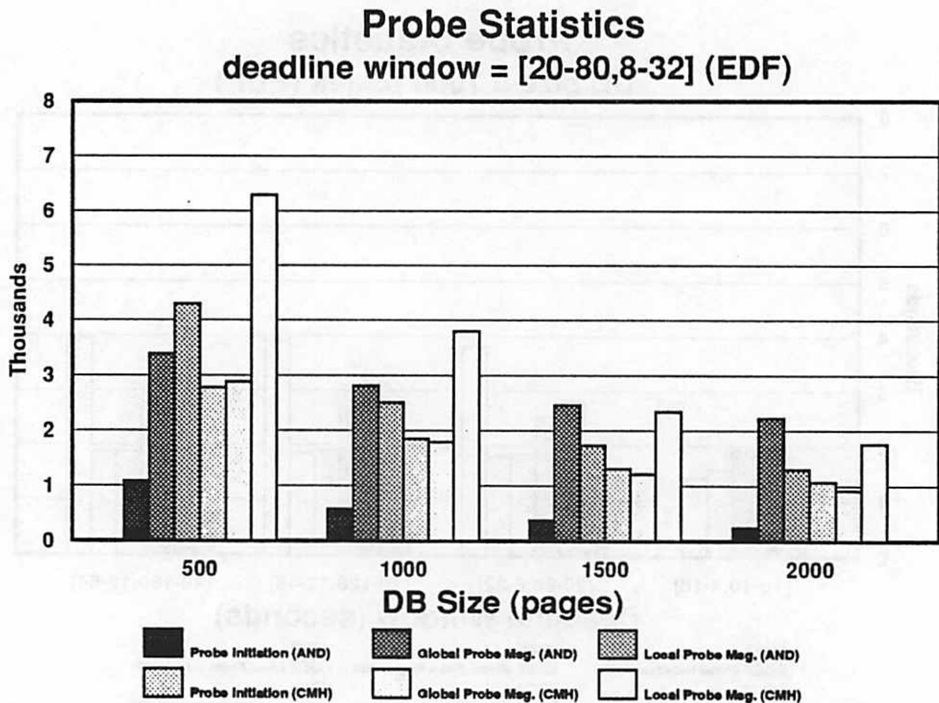


Figure 7.23: Probe Statistics, AND Model System, EDF Scheduling, DL Window [20-80,8-32]

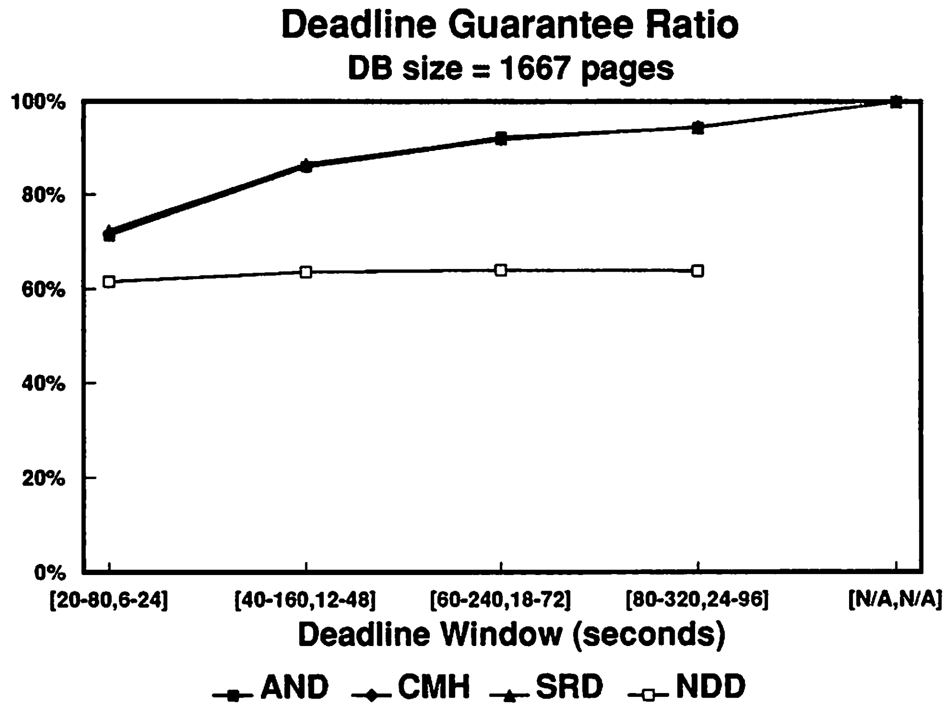


Figure 7.24: Deadline Guarantee Ratio, Single-Resource System, 2 Slave Sites, DB Size 1667 Pages

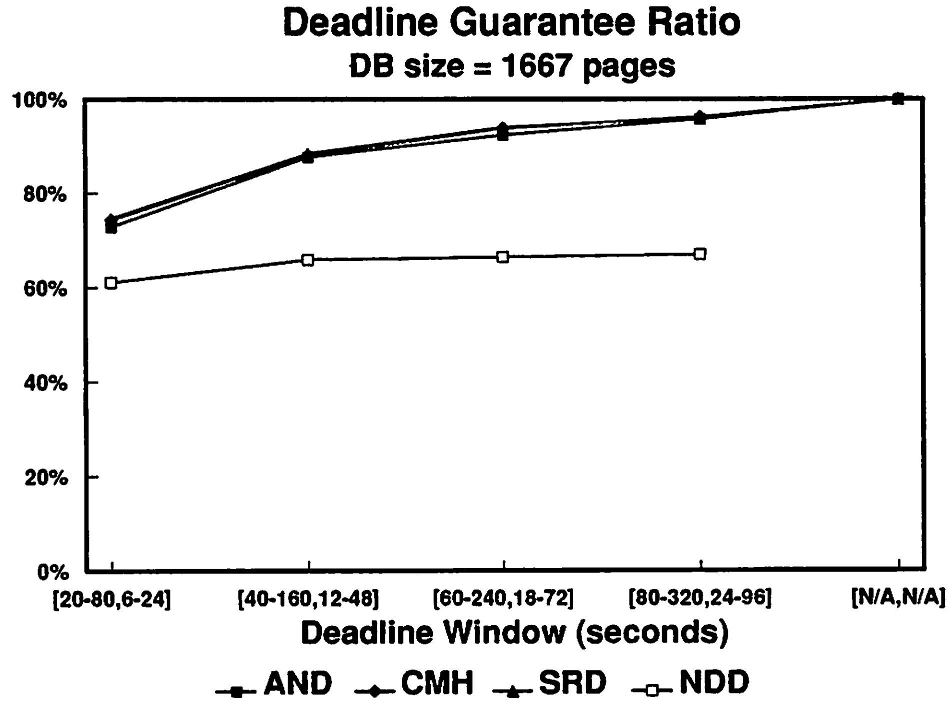


Figure 7.25: Deadline Guarantee Ratio, Single-Resource System, 1 Slave Site, DB Size 1667 Pages

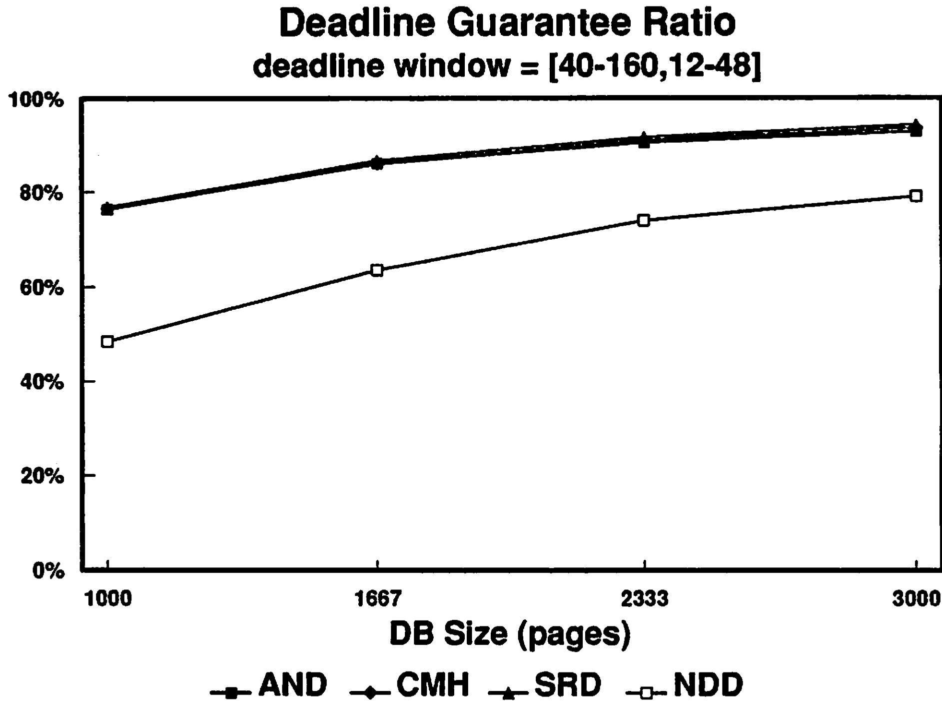


Figure 7.26: Deadline Guarantee Ratio, Single-Resource System, 2 Slave Sites, DL Window [40-160,12-48]

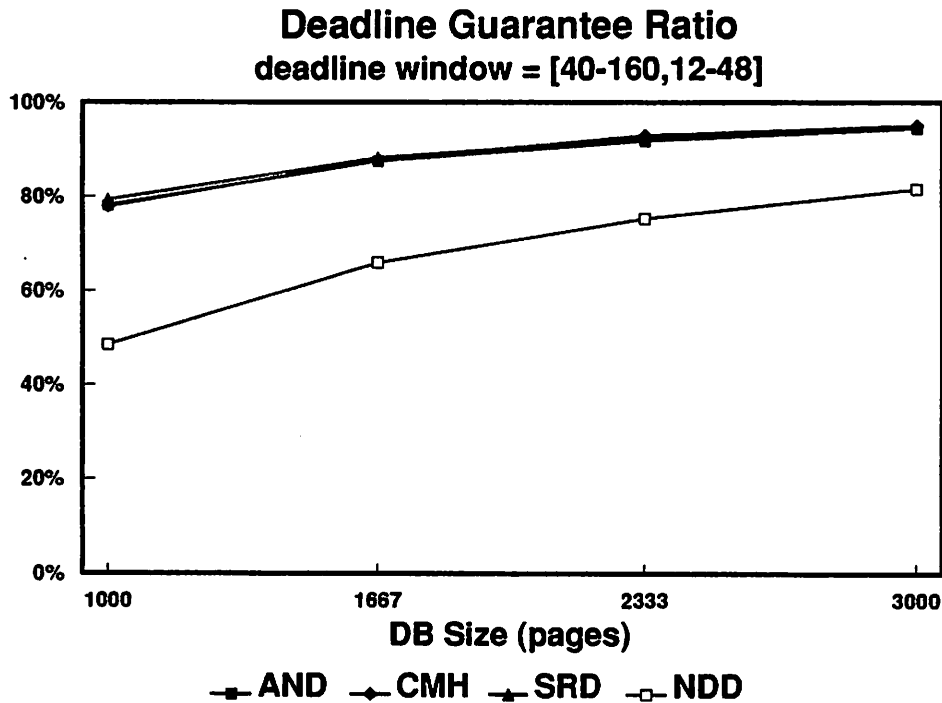


Figure 7.27: Deadline Guarantee Ratio, Single-Resource System, 1 Slave Site, DL Window [40-160,12-48]

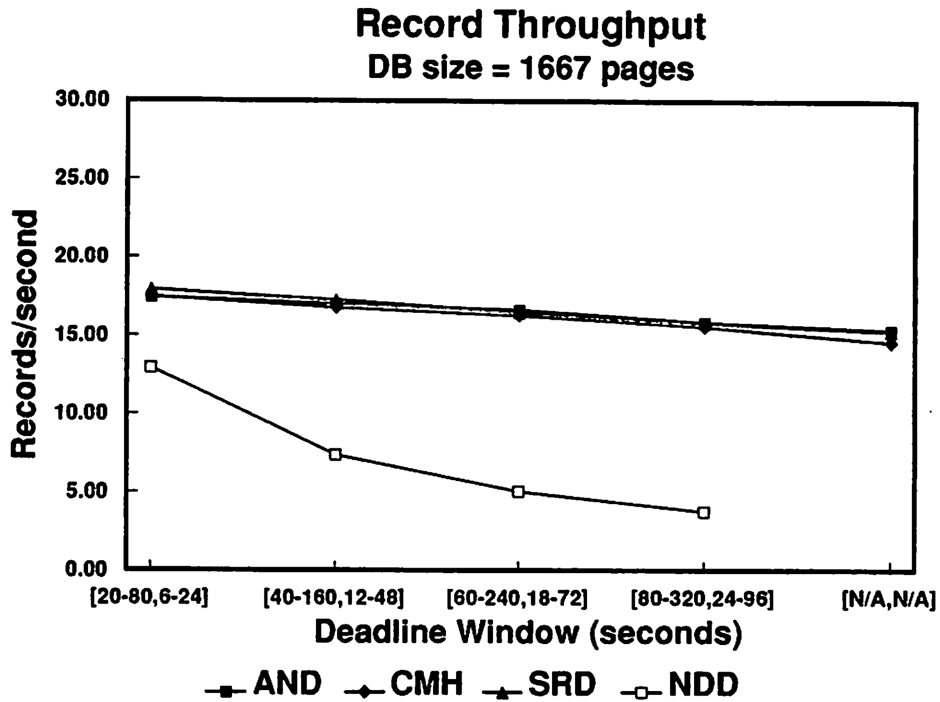


Figure 7.28: Record Throughput, Single-Resource System, 2 Slave Sites, DB Size 1667 Pages

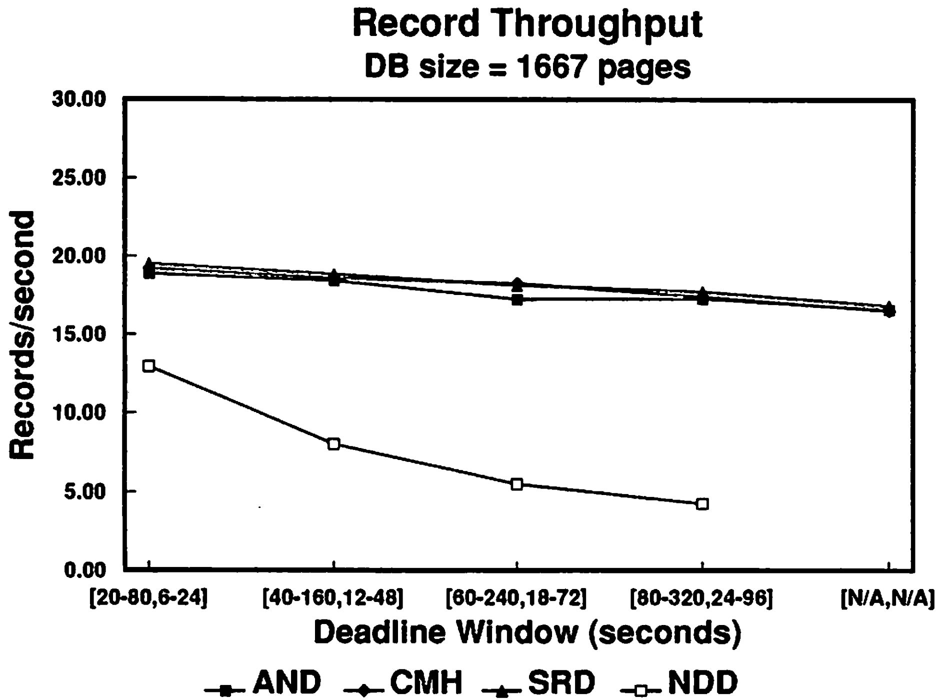


Figure 7.29: Record Throughput, Single-Resource System, 1 Slave Site, DB Size 1667 Pages

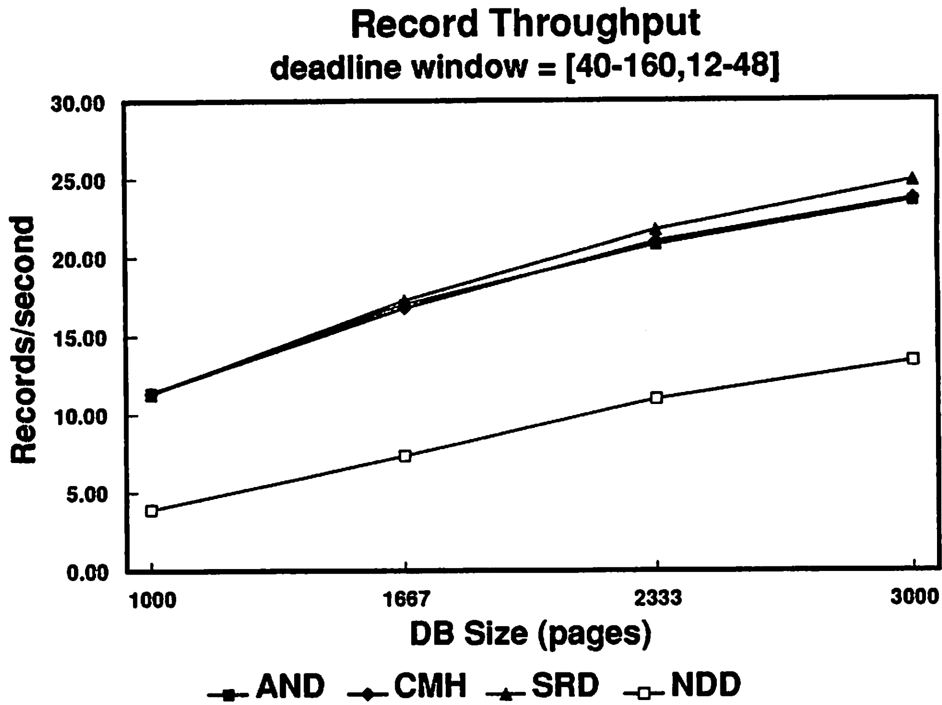


Figure 7.30: Record Throughput, Single-Resource System, 2 Slave Sites, DL Window [40-160,12-48]

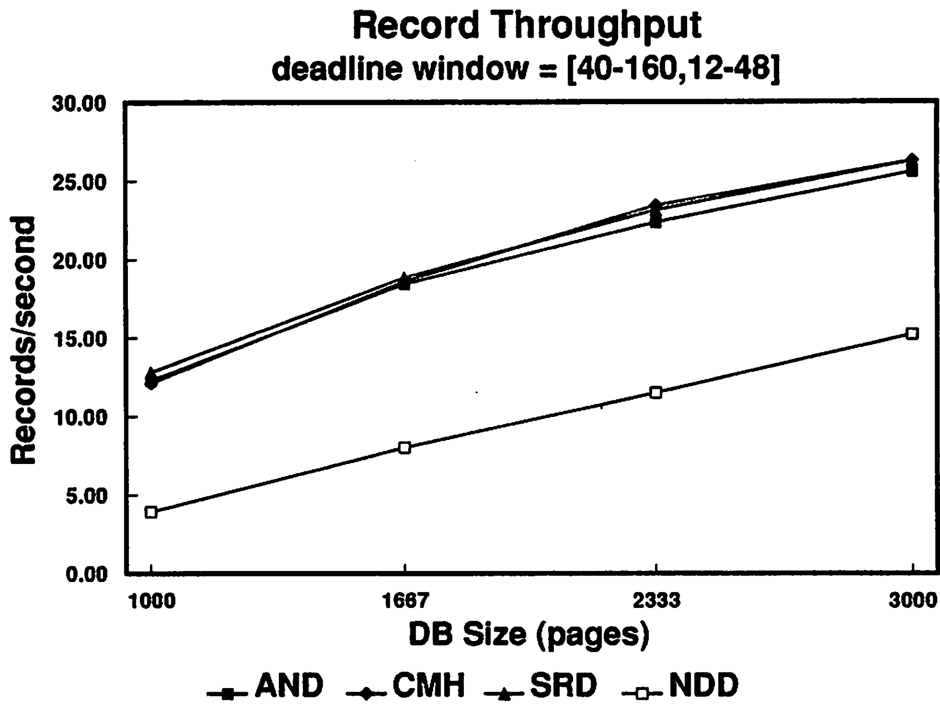


Figure 7.31: Record Throughput, Single-Resource System, 1 Slave Site, DL Window [40-160,12-48]

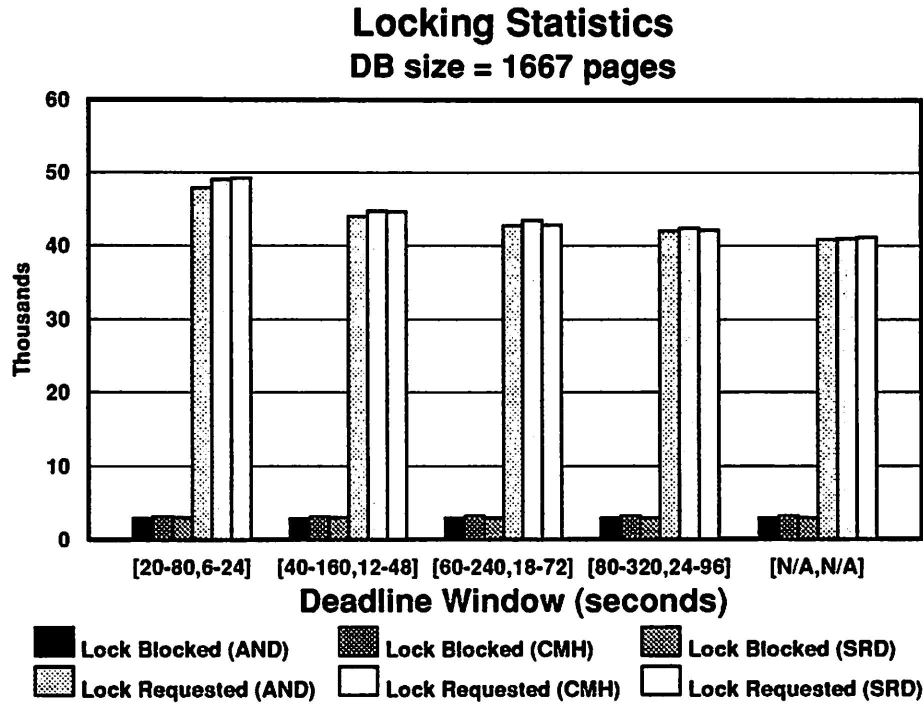


Figure 7.32: Locking Statistics, Single-Resource System, 2 Slave Sites, DB Size 1667 Pages

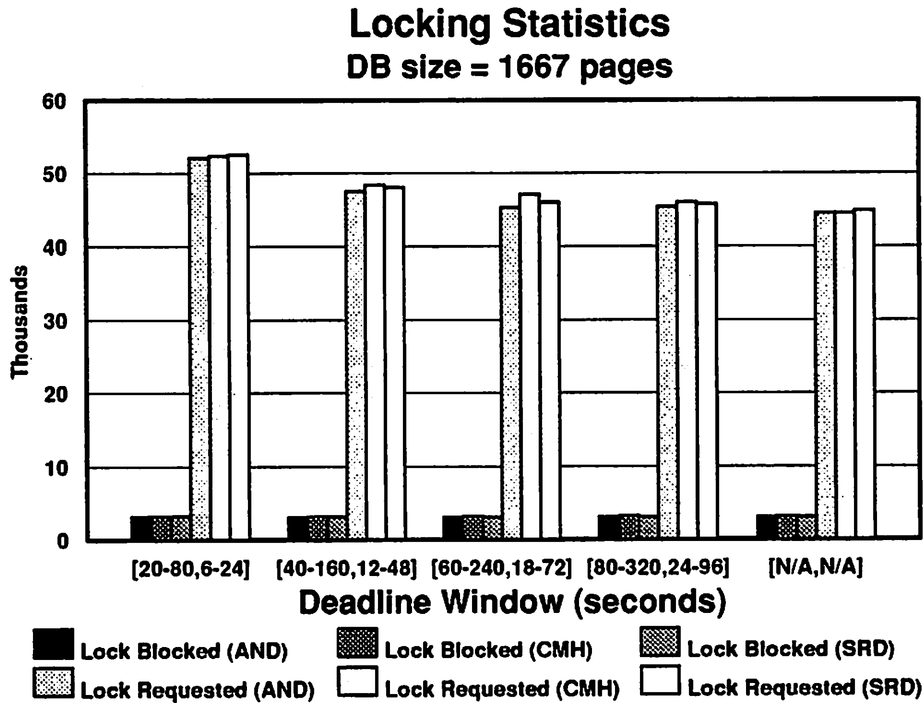


Figure 7.33: Locking Statistics, Single-Resource System, 1 Slave Site, DB Size 1667 Pages

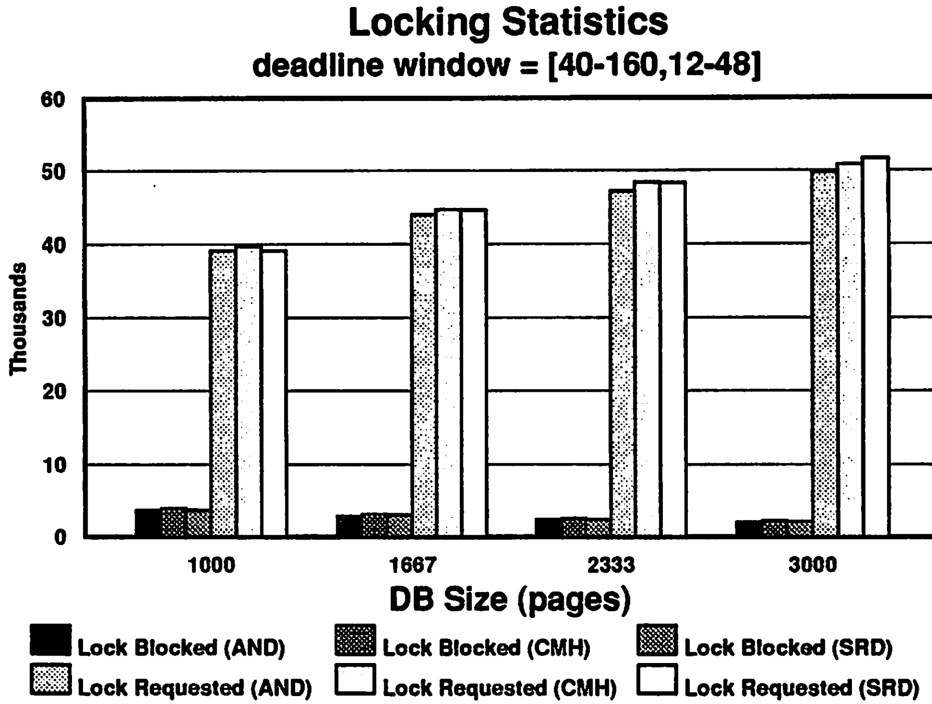


Figure 7.34: Locking Statistics, Single-Resource System, 2 Slave Sites, DL Window [40-160,12-48]

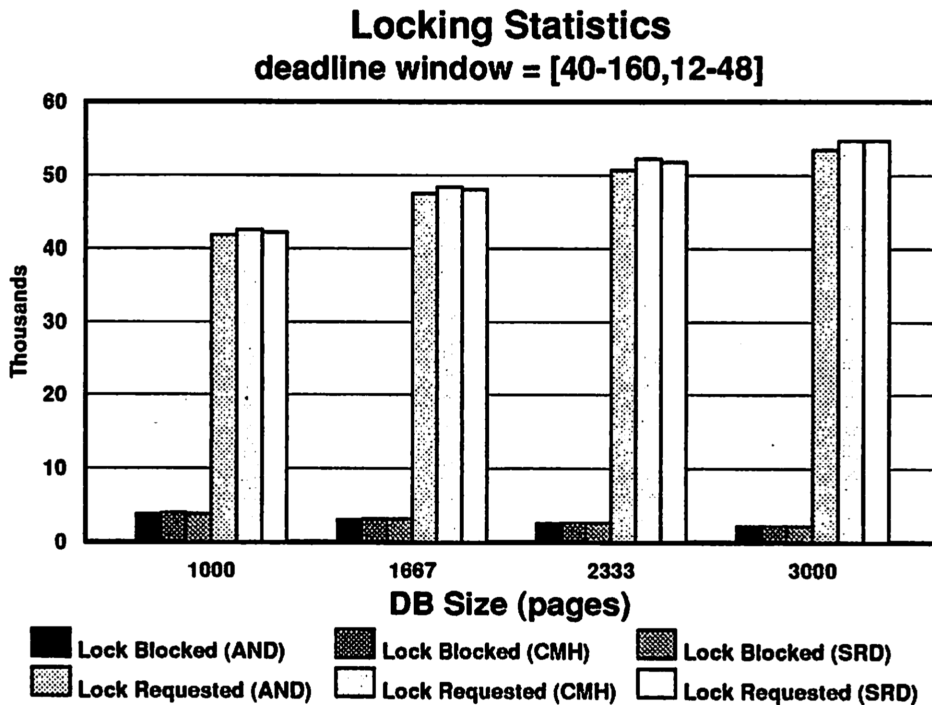


Figure 7.35: Locking Statistics, Single-Resource System, 1 Slave Site, DL Window [40-160,12-48]

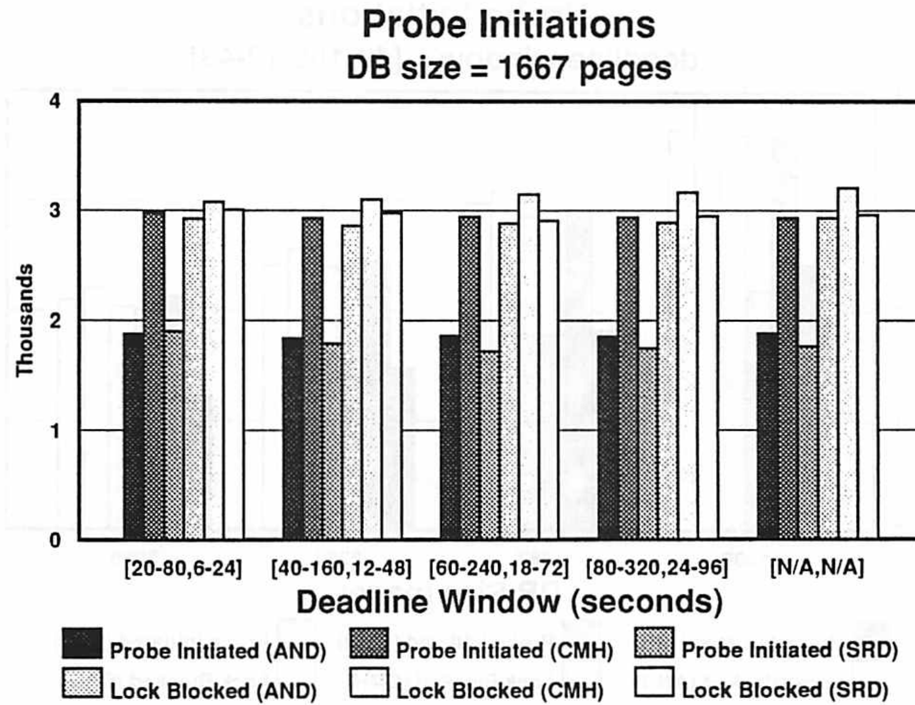


Figure 7.36: Probe Initiations, Single-Resource System, 2 Slave Sites, DB Size 1667 Pages

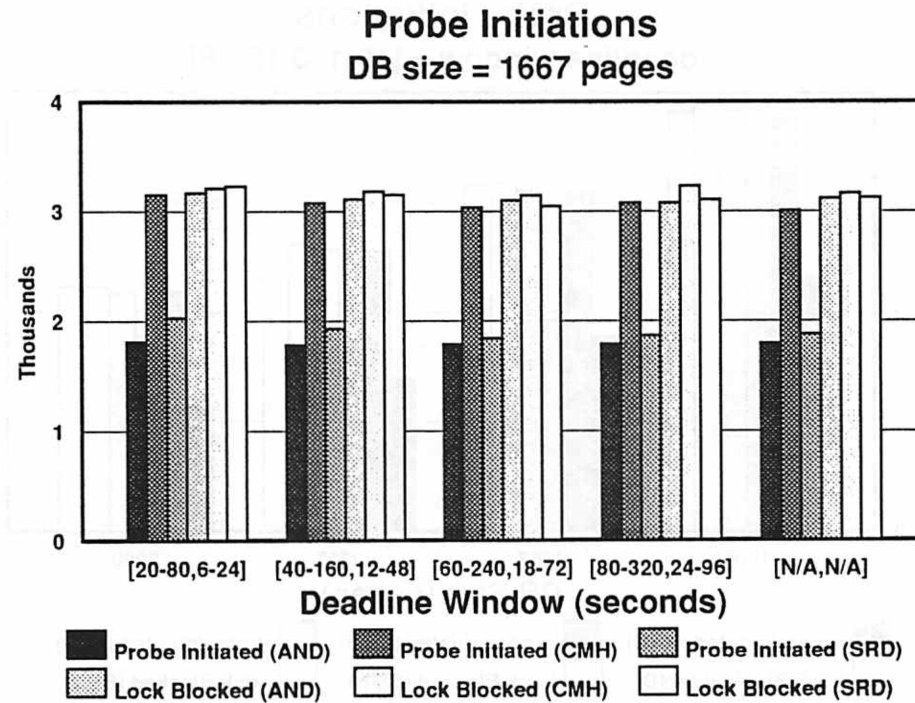


Figure 7.37: Probe Initiations, Single-Resource System, 1 Slave Site, DB Size 1667 Pages

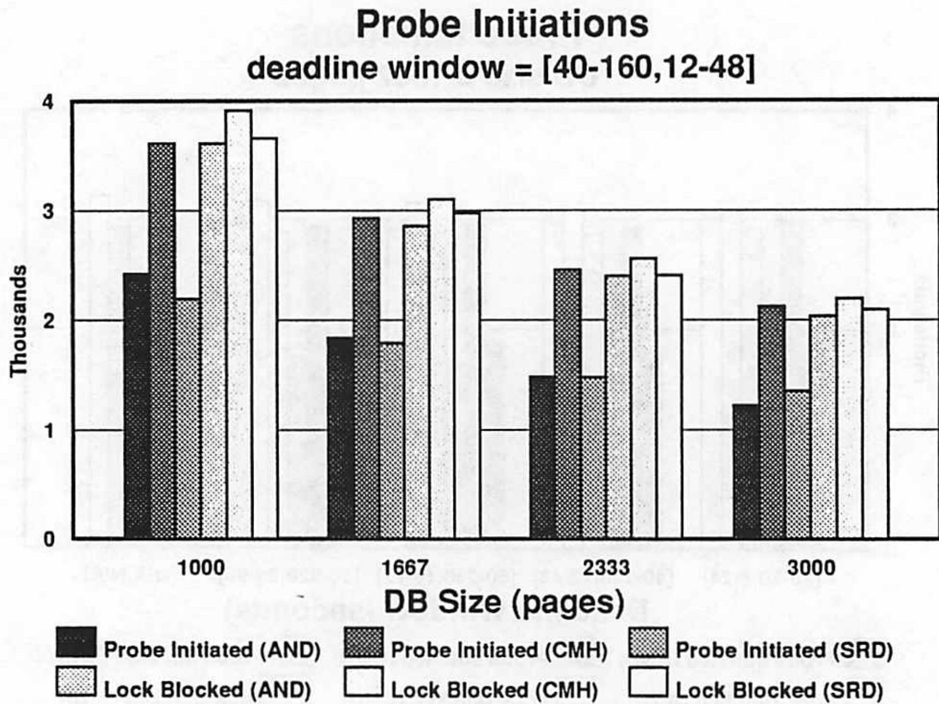


Figure 7.38: Probe Initiations, Single-Resource System, 2 Slave Sites, DL Window [40-160,12-48]

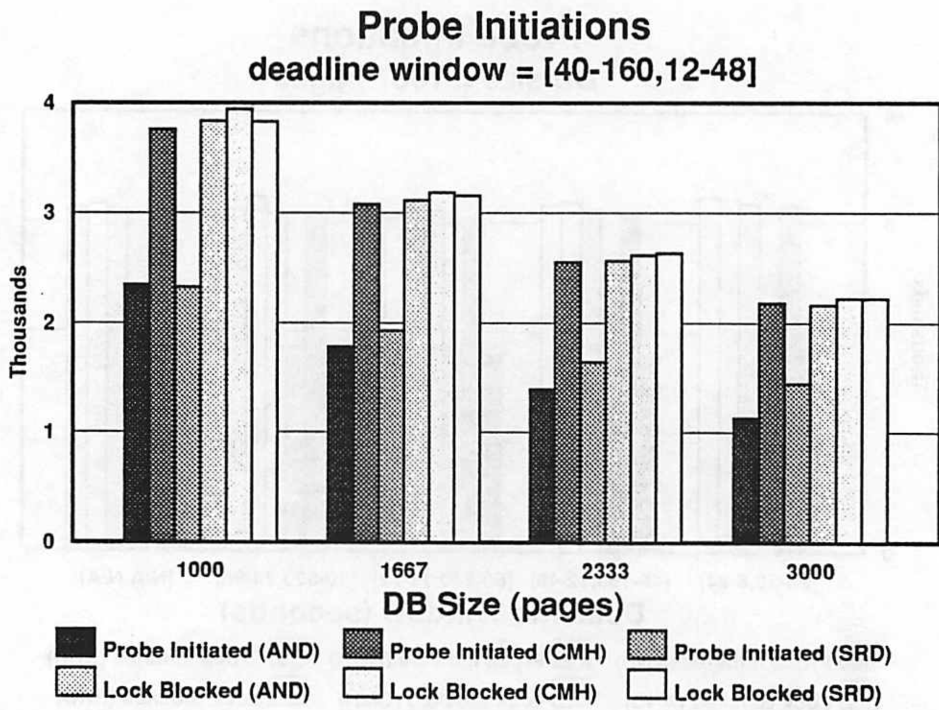


Figure 7.39: Probe Initiations, Single-Resource System, 1 Slave Site, DL Window [40-160,12-48]

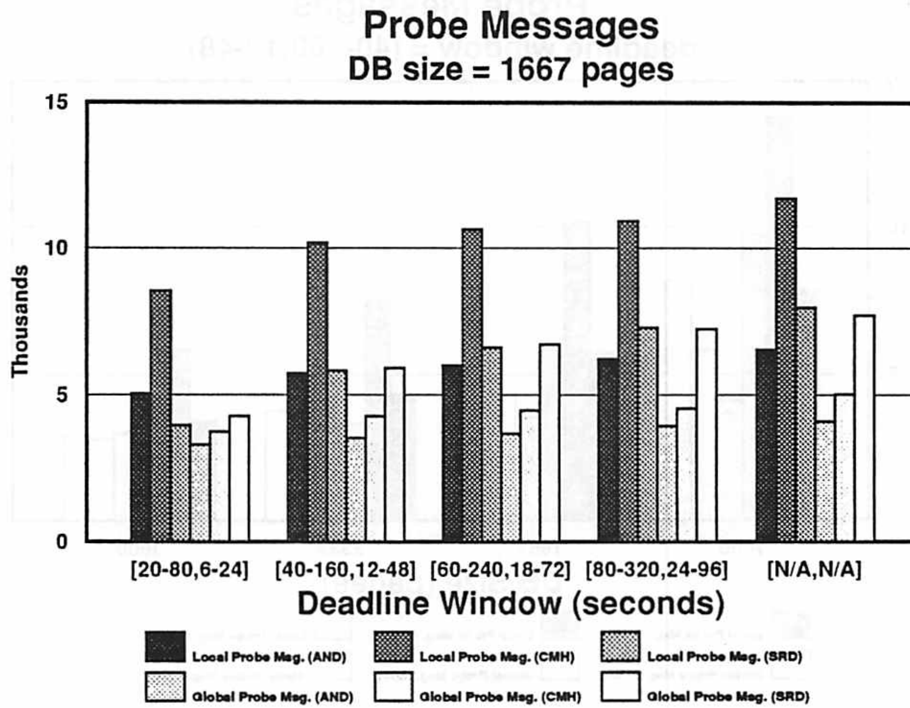


Figure 7.40: Probe Message Statistics, Single-Resource System, 2 Slave Sites, DB Size 1667 Pages

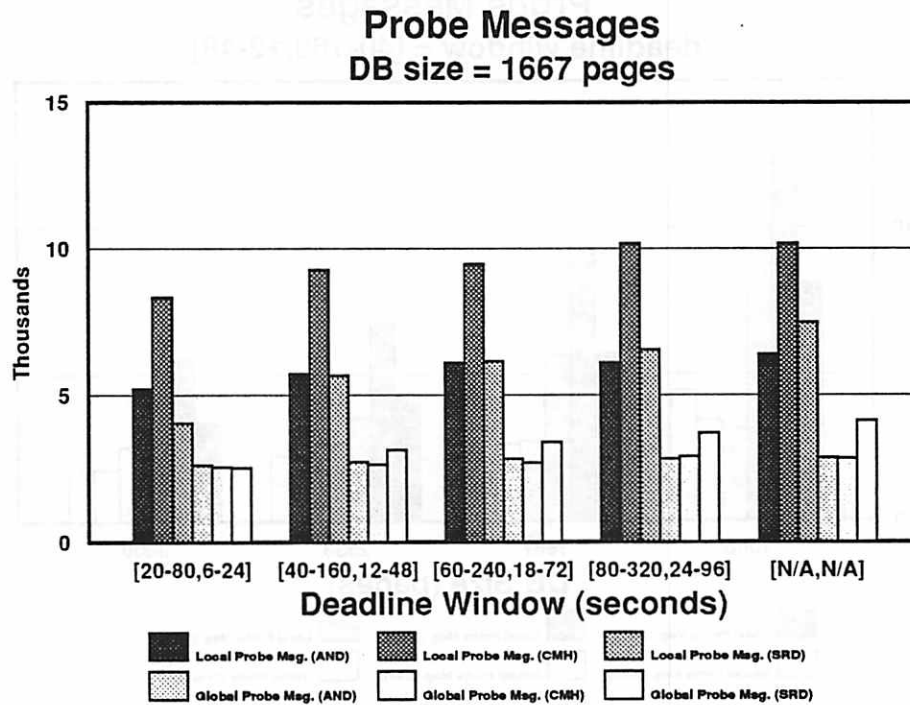


Figure 7.41: Probe Message Statistics, Single-Resource System, 1 Slave Site, DB Size 1667 Pages

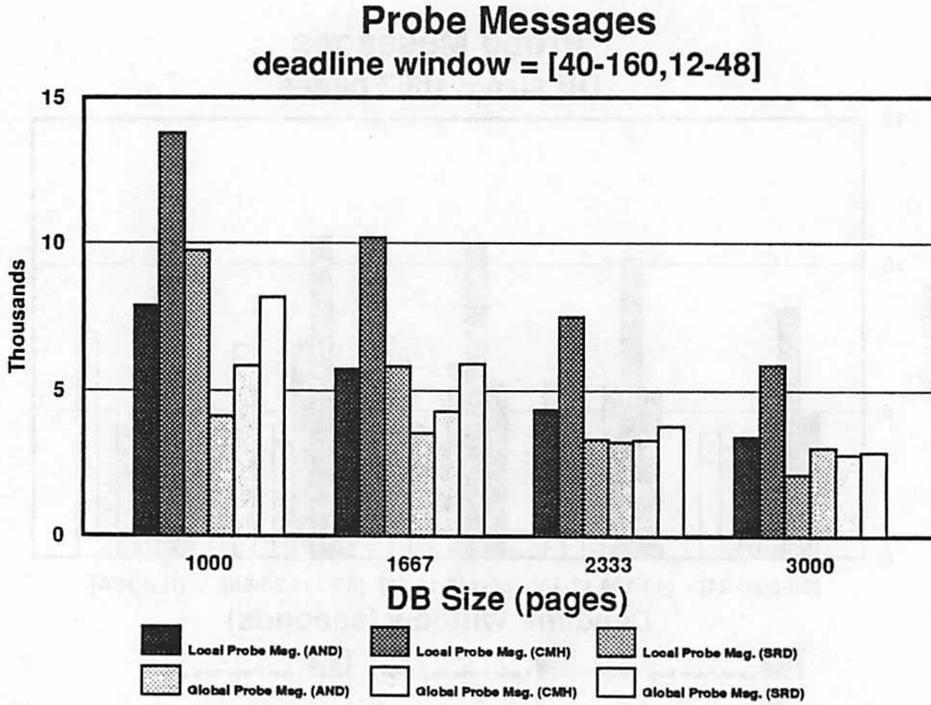


Figure 7.42: Probe Message Statistics, Single-Resource System, 2 Slave Sites, DL Window [40-160,12-48]

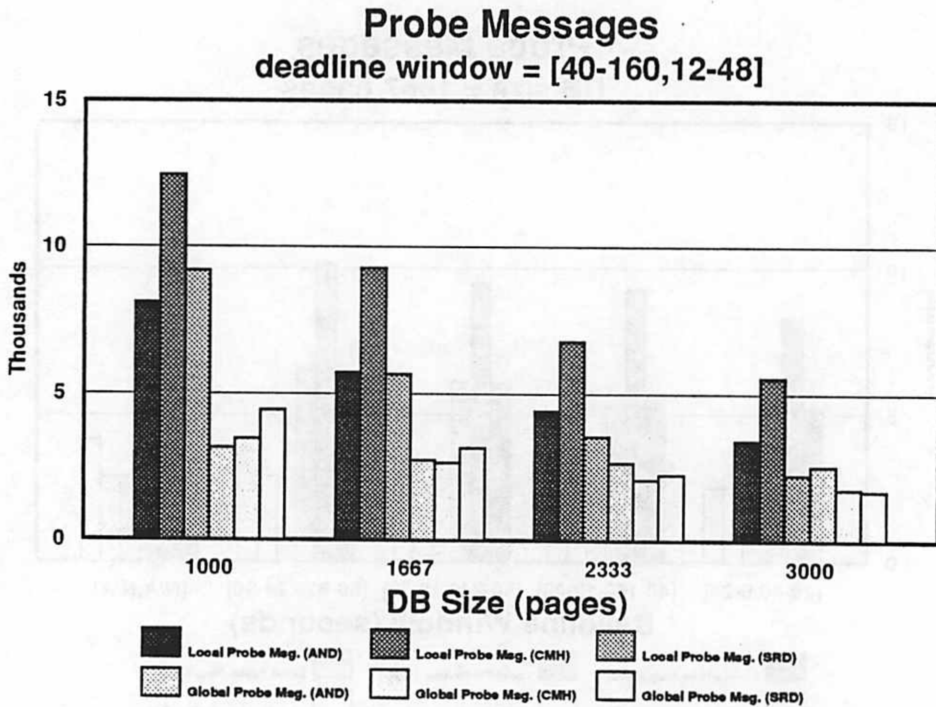


Figure 7.43: Probe Message Statistics, Single-Resource System, 1 Slave Site, DL Window [40-160,12-48]

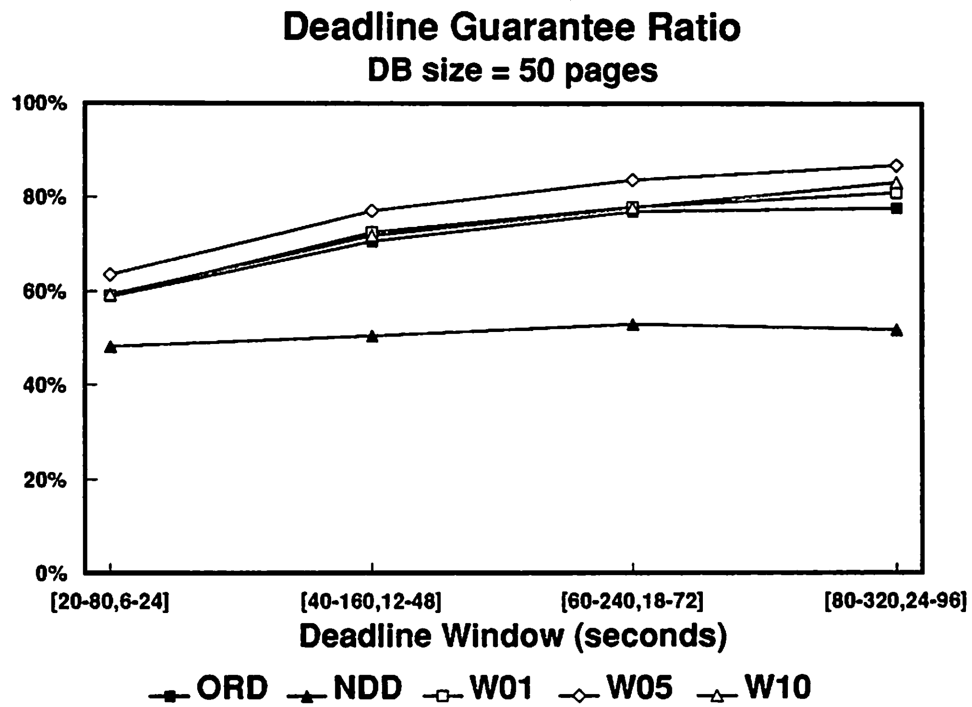


Figure 7.44: Deadline Guarantee Ratio, OR Model System, Random Accessing Pattern, DB Size 50 Pages

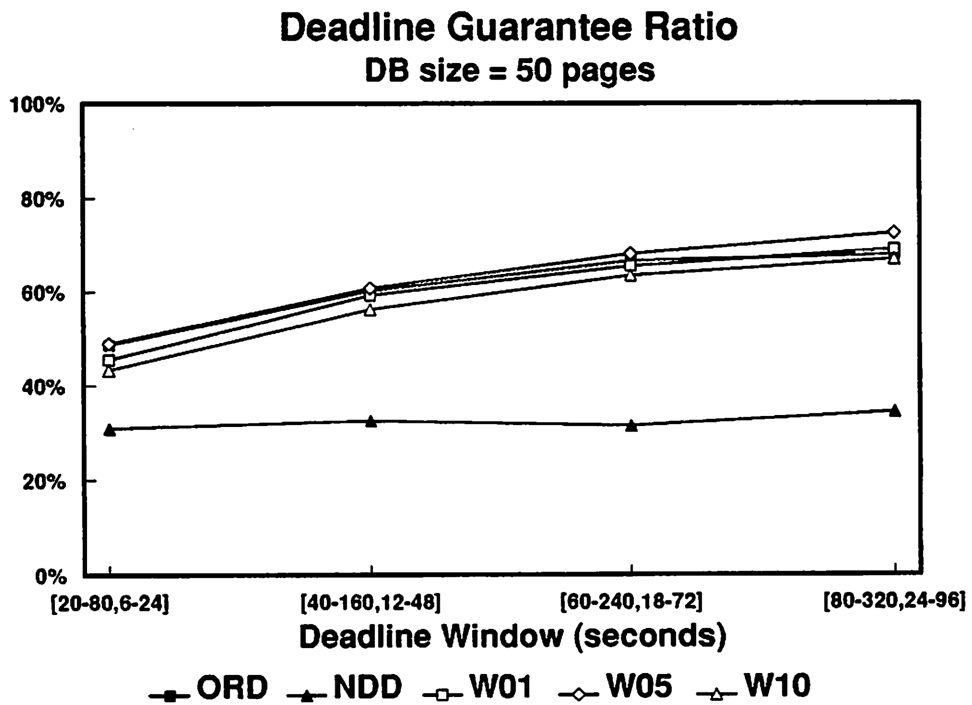


Figure 7.45: Deadline Guarantee Ratio, OR Model System, Contiguous Accessing Pattern, DB Size 50 Pages

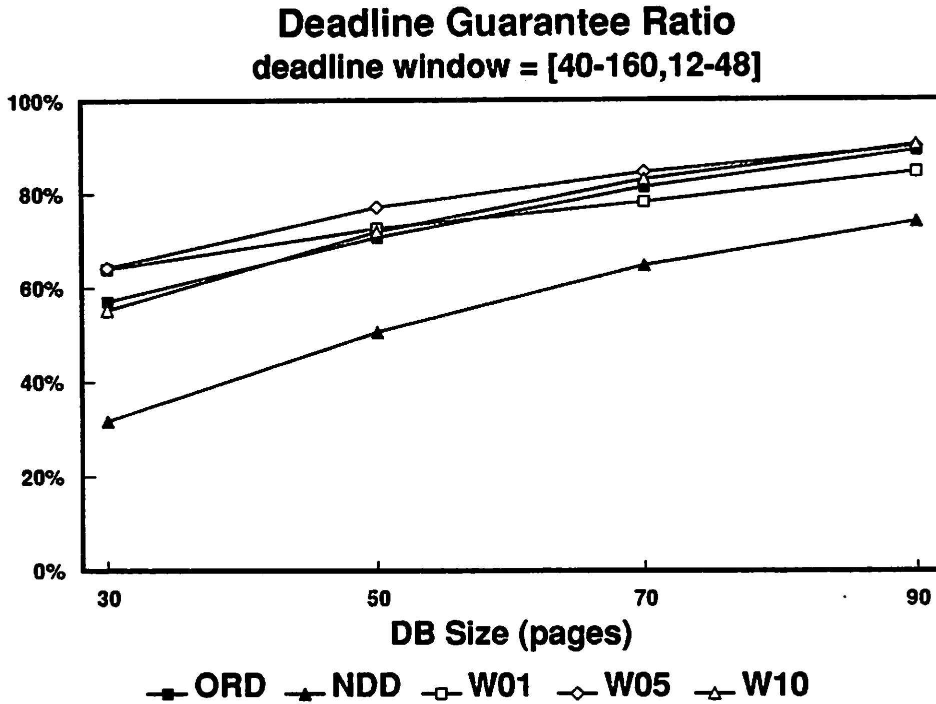


Figure 7.46: Deadline Guarantee Ratio, OR Model System, Random Accessing Pattern, DL Window [40-160,12-48]

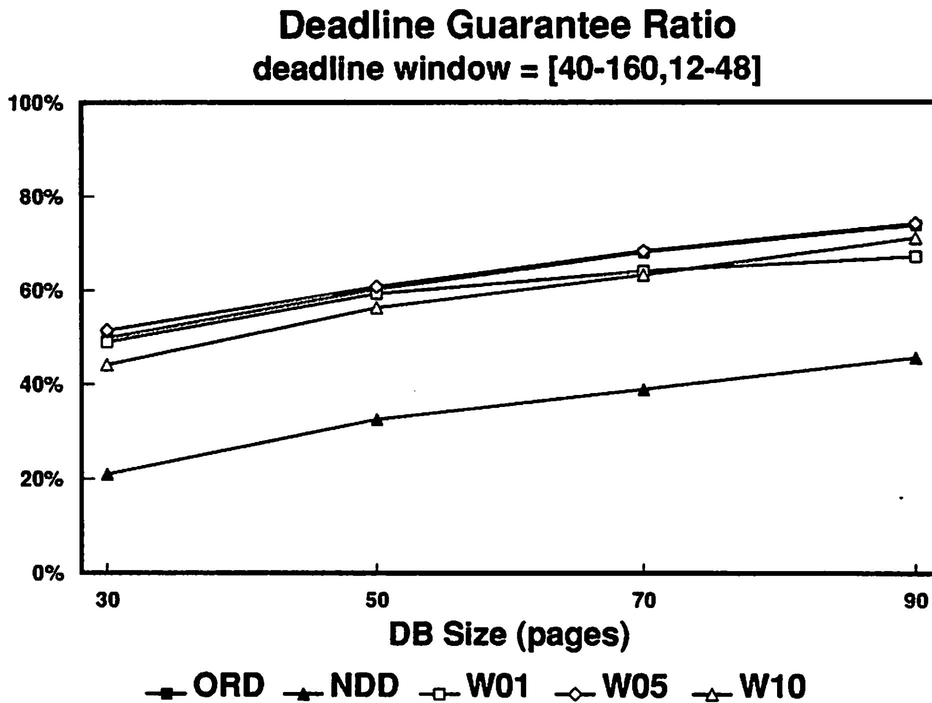


Figure 7.47: Deadline Guarantee Ratio, OR Model System, Contiguous Accessing Pattern, DL Window [40-160,12-48]

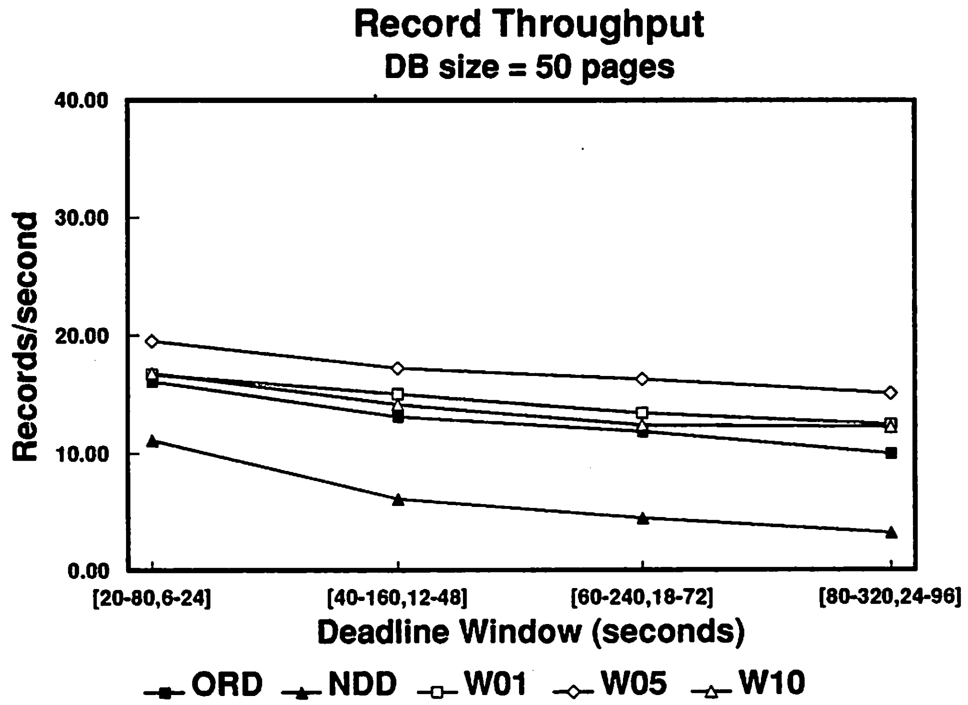


Figure 7.48: Record Throughput, OR Model System, Random Accessing Pattern, DB Size 50 Pages

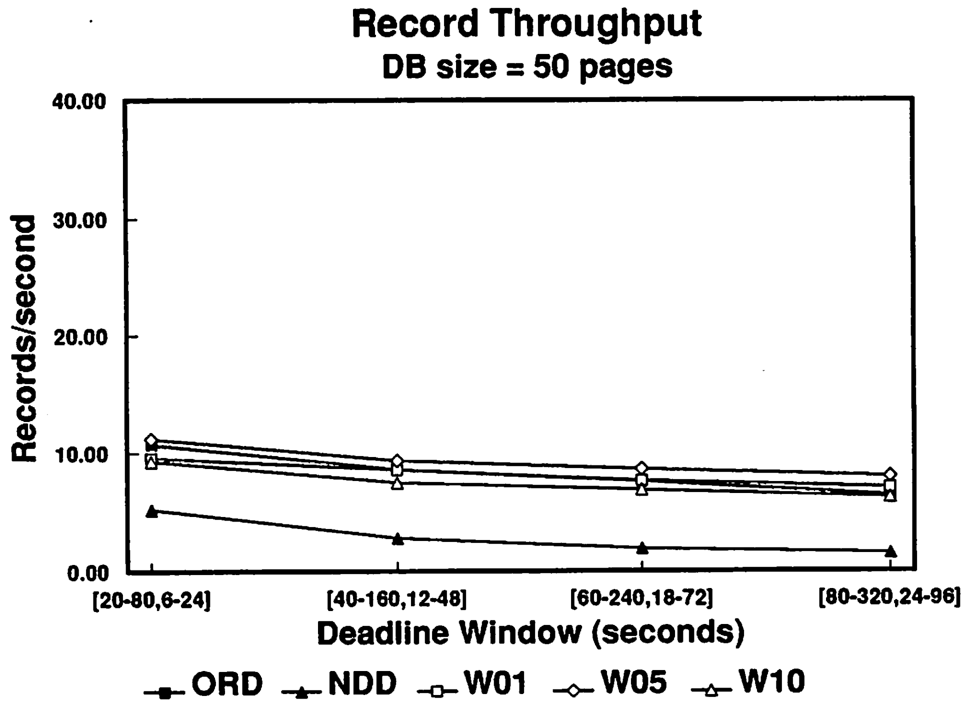


Figure 7.49: Record Throughput, OR Model System, Contiguous Accessing Pattern, DB Size 50 Pages

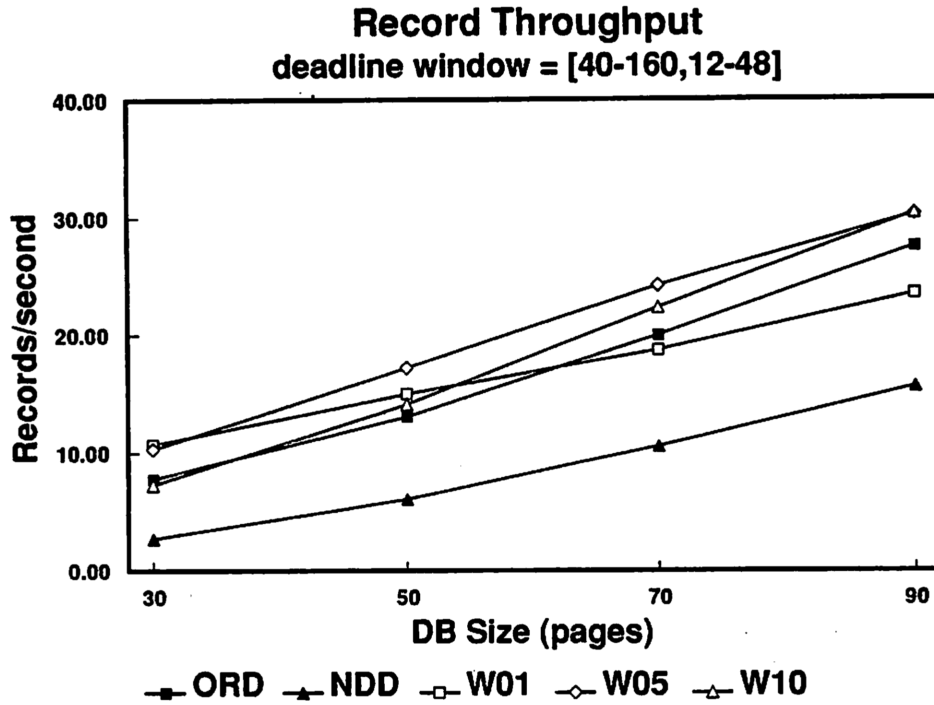


Figure 7.50: Record Throughput, OR Model System, Random Accessing Pattern, DL Window [40-160,12-48]

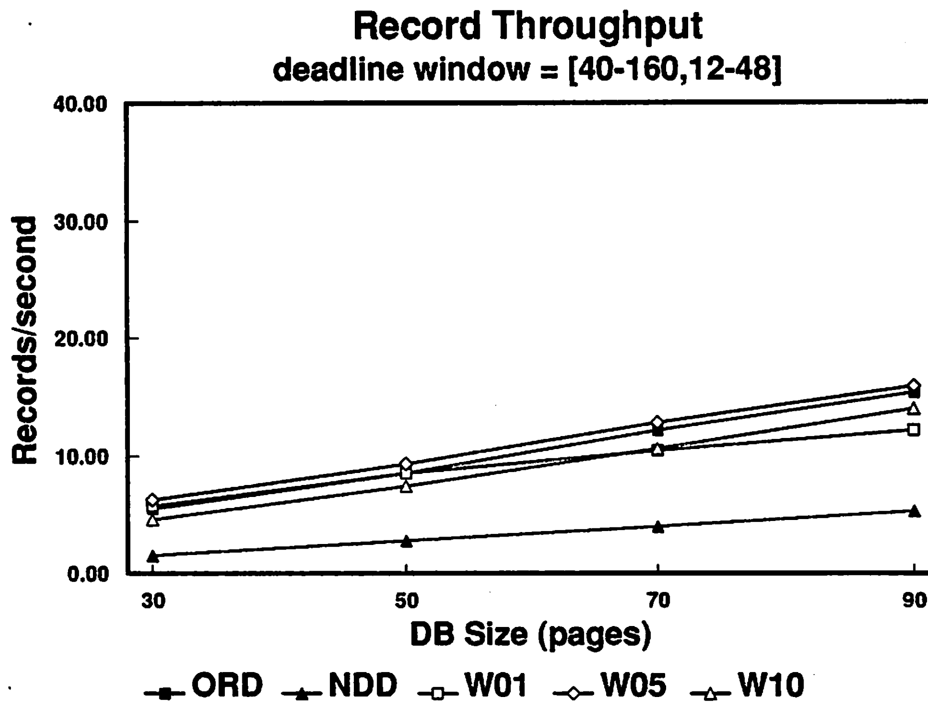


Figure 7.51: Record Throughput, OR Model System, Contiguous Accessing Pattern, DL Window [40-160,12-48]

CHAPTER 8

SUMMARY AND FUTURE DIRECTIONS

8.1 Summary of Conclusions

Many complex systems will be built using highly parallel and distributed computer technology. Many of these systems will require that various types of timing constraints be met, and that they be highly dependable. For this to occur many advances are required. In this dissertation we did not attempt to address the entire problem, rather we focused on a single but important problem – deadlock – that must be solved in a cost-effective manner.

To deal with the deadlock problem, we may pessimistically try to "prevent" or "avoid" deadlock situations in a system. These two strategies ensure that the system will *never* enter a deadlock state. In real-time systems, deadlock prevention and avoidance methods have received most of the attention and are the current strategies of choice. However, these strategies may work successfully in relatively simple systems, but may be inefficient and very difficult to design and verify in more complex systems such as multiprocessors or distributed systems.

Another option is to optimistically assume that the system rarely deadlocks and allow it to enter a deadlock state, which will then be detected and recovered from. Although the problem of deadlock detection may be very complex when the underlying system is distributed and when tasks have timing constraints, the solution (i.e., the detection algorithm) could perform very efficiently. A deadlock detection procedure may be invoked only when a system suspects the existence of deadlock states. A well designed deadlock detection algorithm should not significantly incur overhead in a system where deadlock situations are rare. Moreover, a deadlock detection algorithm can be used in a system where simple local deadlocks are prevented or avoided. Consequently, if a system is able to detect and

recover from deadlock states both with or without the prevention or the avoidance strategies, the dependability of the system has increased.

To apply deadlock detection techniques in real-time systems, we first analyzed and identified the problems and the issues of deadlock detection in distributed real-time applications. Some of the important conclusions from this analysis are:

- In real-time systems, timing constraints are attached to the tasks. A task may be timed out from a state in which it is waiting. Deadlocks may not be stable if timing constraints are considered. Three types of problems: "Stable Deadlock," "Temporal Deadlock," and "Non-deadlocked Blocking" were identified and discussed. A stable deadlock in real-time systems is the same as a traditional deadlock in non-real-time systems. A temporal deadlock, on the other hand, is a special kind of deadlock which is not treated as a deadlock or is assumed not to exist in non-real-time systems. Such a deadlock is *temporal* and hence not *stable*. The *stable property* which is assumed in most of the traditional deadlock detection algorithms can no longer be used to detect temporal deadlocks in real-time systems. Timing constraints must be taken into consideration in detecting temporal deadlocks. The timing information collected for detecting temporal deadlocks can also be applied to resolve many of the the problems associated with non-deadlocked blocking.
- It is generally too expensive to completely achieve both the Safety and the Progress criteria when designing algorithms for distributed real-time systems. These criteria may be violated due to timing constraints in real-time systems (e.g., a temporal deadlock may not be detected if one of the participant has a very tight deadline). Also, these criteria may not be fulfilled due to synchronization difficulties in distributed real-time systems (e.g., in the AND deadlock model, to break one of the nested cycles at the common part may cause the detection of false deadlocks). For soft real-time systems where violations will not cause any severe permanent faults, these two criteria can be relaxed to an acceptable level. When making trade-offs, in general, the Progress criterion should be considered more seriously than the Safety criterion. This is because

leaving the system in a deadlocked state is usually an uncontrollable fault whereas recovering from a false deadlock is a type of compensating action which usually impacts the system less severely.

- In real-time applications, when a deadlock is detected, the resolution decision should consider timing constraints. A deadlock is a cyclic wait situation. Depending on where a deadlock is broken, different timing dependencies might be formed. A bad resolution may cause more tasks to miss their deadlines even if they are not chosen as the victims to break the deadlock. Our experimental study also confirms the phenomenon that when a deadlock cycle is long, to break it at one point is sometimes not good enough to allow the remaining tasks to complete by their deadlines.

The deadlock detection algorithms proposed in the literature are designed for the detection of the Stable deadlocks. When these algorithms are applied to real-time applications, they may either ignore the existence of Temporal deadlocks or treat the Temporal deadlocks as Stable ones. Using the former approach a system is allowed to enter a Temporal deadlock state and then relies on some tasks to relinquish their requests to break the deadlock. This approach will cause a system to perform poorly if most of the tasks have long deadlines. On the other hand, the latter approach attempts to detect every deadlock without distinction. A Temporal deadlock may be detected after it is broken or a detected deadlock may never exist. Consequently, timing constraints should be taken into consideration in detecting deadlocks in real-time systems.

To extend an algorithm to real-time applications, we need to understand the principles and the properties of the algorithm. After completing a study of several existing algorithms in the literature, we have found that a better understanding of distributed deadlock detection is necessary for the extension of an algorithm to real-time applications as well as for the improvement (or the correction) to the original algorithm. We have derived a methodology for the development of distributed deadlock detection algorithms and some of the important results are summarized as follows.

- A good distributed deadlock detection algorithm should confine its knowledge about the global system state to local views that any individual site can see. It should also assume no physical common global clock.
- The problems of distributed deadlock detection can be divided into two parts, i.e., (1) how to correctly recognize a deadlock structure and (2) how to synchronize a deadlock detection computation with the underlying dynamically changing system. The first problem can be studied via static graphs and a "local view of a deadlock" is recommended for the definition of deadlocks and is suggested to be used in the development of "static" algorithms. To deal with problem (2), we have developed a systematic method to derive a "synchronization mechanism" which can be coupled with a known "static" deadlock detection algorithm to produce a "dynamic" algorithm.
- The deadlock detection computation and the underlying system were formally modeled based on Chandy and Lamport's notion of distributed "snapshots" [21]. Two consistency criteria must be fulfilled to maintain the "meaningfulness" of a deadlock detection computation. Based on the Stable property of the deadlocks, we derived "local" conditions for a synchronization mechanism to satisfy the Safety and the Progress concerns. By "local" we mean these conditions only require information that is local to each participant of a DDC. Consequently, these conditions provide a feasible way to realize a synchronization mechanism.
- Based on the understanding of the meaningfulness of deadlock detection computations, we know how to satisfy the Safety and the Progress concerns when developing an algorithm. An algorithm can be pushed to detect all the existing deadlocks as early as possible by preserving the meaningful computations in the system as much as possible. Also, by analyzing the implications of the timing constraints in the meaningfulness of the deadlock detection computations, deadlines can be properly incorporated in an algorithm for real-time applications.

According to our methodology, distributed deadlock detection algorithms can be developed in refinement steps as follows.

1. Static Algorithm: Develop principles for deadlock detection in a static graph.
2. Dynamic Algorithm:
   - To satisfy the Safety concern, avoid the detection of white edges (i.e., discard the probes received from a non-existing edge) and maintain the connectivity between adjacent probe operations.
   - Examine the situations when a new computation should be initiated to ensure the Progress concern.
3. Real-Time Applications: Associate deadlines with the probes to reflect the meaningfulness of a deadlock detection computation with the presence of the timing constraints in a system.
4. Maintain the principles developed in the first step throughout the rest of the design procedure.

Based on the above procedure, algorithms for the Single-Resource, the AND, and the OR deadlock models were developed. In the first step, cycle detection techniques were used for the Single-Resource and the AND deadlock models while a knot detection technique was used for the detection of the OR deadlocks. Some of these techniques are well known in the literature. In the second step, the static algorithms were adapted for use in dynamic systems. The development involves the improvements to the previously proposed algorithms. Finally, these algorithms were extended to the real-time applications.

For each of these three deadlock models, we have put together a practical algorithm (named the Single-Resource, the AND, and the OR Algorithms) which can be used in a general resource modeled system (i.e., Holt's General Resource System [66]). In particular, for the OR deadlock detection, we developed an integrated algorithm (the OR Algorithm) for the detection of both the Ada rendezvous deadlocks and the task termination conditions in distributed environments. One

unique feature of this OR Algorithm is that it is able to properly handle both the **delay** and the **terminate** statements in Ada rendezvous which has never been addressed before.

To verify the algorithms and to study their performance, we have implemented the following schemes in the RT-CARAT system where transaction timing constraints and soft real-time scheduling protocols are supported [67]:

1. Break by deadline: No deadlock detection algorithm is used in this scheme. A deadlock will be broken when one of its participant transactions is aborted by deadline.
2. Timeout and retry: In this scheme, a transaction will rollback and restart if its request cannot be granted in a preset short timeout period.
3. Deadlock detection/resolution: Together with a previously implemented AND deadlock detection algorithm (proposed by Chandy, Misra, and Haas [27]), three proposed algorithms for the Single-Resource, the AND, and the OR deadlock models are implemented for the verification and the performance study. If a transaction is chosen as the victim of the resolution of a deadlock, it will rollback and restart until its deadline expires.

Our experimental results show that distributed deadlock detection can be very efficient. Compared to the baseline scheme, i.e., break by deadline, any of these deadlock detection algorithms can significantly improve the system performance in terms of the deadline guarantee ratio and the throughput. Also, compared to the timeout and retry, the detection/resolution scheme performs better when the deadlocks are simple and short (e.g., the Single-Resource and the AND deadlocks). However, the detection/resolution scheme performs worse than timeout and retry when the average deadlock length is long (e.g., the OR deadlocks). The reason for the latter case is twofold: (1) the rollback is simple in RT-CARAT which favors timeout and retry scheme and (2) the resolution that break a long complex deadlock at one point is not good enough for real-time applications.

8.2 Future Directions

Our research work can be extended in several directions. First, the design of the algorithms for the AND-OR and the C(n,k) models is not complete. These two deadlock models are mathematically equivalent in that any AND-OR request can be transformed into C(n,k) request and vice versa. As discussed in Chapter 2, there is no simple construct of graph theory to describe the condition of the AND-OR or the C(n,k) model deadlocks. Similar to the problem in the AND deadlocks, several "deadlock core sets" may be nested which makes the Safety concern very difficult to be satisfied.

In principle, deadlock in the AND-OR model can be detected by applying the knot detection repeatedly, where each invocation operates on a subgraph of the AND part of the model. However, this strategy is not very efficient. Hermann and Chandy [63] proposed a more efficient algorithm for AND-OR deadlock. Their algorithm is based on a hierarchy of diffusing computations which they called a *tree* computation. The central idea of their algorithm is that when a diffusing computation reaches a blocked task: (1) the diffusing computation is propagated to its dependent set if it is an OR task, or (2) it initiates a separate tree computation if it is an AND task. This algorithm is not practical in that it detects "deadlock sets" instead of "deadlock core sets" which complicates the resolution procedure.

Bracha and Toueg [13] proposed to process a global system snapshot to find deadlocks in the C(n,k) model. The graph reduction techniques suggested by Holt [66] are used to determine the existence of deadlocks in a snapshot. Each of the active tasks in the snapshot can be scheduled to terminate and to release the resources it holds. The snapshot system state thus can be *reduced* to a new state (see Section 2.4.2). A snapshot is said to be *completely reducible* if there exists a sequence of graph reductions that reduces it to a set of isolated vertices. A task T_i is not deadlocked in state S if and only if there exists a sequence of reductions in the corresponding snapshot that leaves T_i unblocked. If a snapshot is completely

reducible, then the state it represents is not deadlocked. Again, this approach is not efficient.

The existence of cycles is a necessary condition for any type of deadlock. As our experimental results show, it is not worth it to detect and resolve large sized complex deadlocks in real-time applications. In real-time systems, a better way of dealing with the deadlocks in the AND-OR, the C(n,k), and even the OR model may be to quickly detect and break every cycle. A performance study could give us a better understanding of all these approaches.

In our study, the resolution of a detected deadlock is treated as a separate problem. However, the experimental results indicate that without good resolution schemes a system with deadlock detection may not perform very well. There are many issues involved in deadlock resolution. For example, as discussed in Chapter 2, depending on how a deadlock is broken, different timing dependencies among the surviving tasks might be formed. A bad resolution will cause more surviving tasks to miss their deadlines. Also, certain resolution schemes simply make the detection of the deadlocks unnecessary. For example, in a real-time system where deadlines are tight, a resolution policy requires that a deadlock should be broken into small pieces of wait-for paths with a length less than 3 transactions. In such a system, it is not necessary to detect a deadlock longer than 3 transactions. Any wait-for path in the system should be broken when it reaches a length of 3. This also suggests that an integrated solution can be developed for both the Non-deadlocked Blocking situations and the Temporal deadlocks in a real-time system. It is interesting to investigate various deadlock recovery schemes in a distributed environment both with and without timing constraints.

Another issue that has not been addressed well to date is the integration of the resolutions for the deadlock related problems. There is a class of deadlock related problems, such as livelock (a.k.a. effective deadlock or starvation), task termination problems, and orphan tasks, which must be detected dynamically at runtime. There is a commonalty among these problems and the deadlock problem that indicates that an integrated set of solutions may be possible. We believe that any large,

complex, long-lived, distributed system is susceptible to these problems. Moreover, some of these problems, e.g., task termination, even have the same property as the deadlock problem. (As an example, we have successfully integrated the detection of global task termination condition with Ada's rendezvous deadlock detection.) Consequently, if a system is able to detect and recover from deadlock and related problems both under timing constraints and not, then the dependability of the system has increased.

APPENDIX A

LIST OF ABBREVIATIONS

Following is a list of abbreviations which are frequently used in this dissertation.

CDC : Cycle Detection Computation (p. 116).

CSP : Communicating Sequential Processes (p. 15).

DDC : Deadlock Detection Computation (p. 70).

DGRG : Dynamic General Resource Graph (p. 69).

DP : Distributed Processes (p. 16).

GRG : General Resource Graph (p. 12).

GRS : General Resource System (p. 25).

KDC : Knot Detection Computation (p. 142).

TRG : Task(Transaction)-Resource Graph (p. 11).

TWFG : Task (Transaction) Wait-For Graph (p. 11).

APPENDIX B

LIST OF NOTATIONS

Following is a list of notations which are frequently used in this dissertation.

$RN(v)$: A set of directly reachable neighbor vertices from a vertex v (p. 13).

$TN(v)$: A set of neighbor vertices which can directly reach vertex v (p. 13).

$RS(v)$: A set of reachable vertices from a vertex v (p. 13).

$TS(v)$: A set of vertices which can reach vertex v (p. 14).

$\langle u \rightarrow v \rangle$: An edge from a vertex u to another vertex v (p. 71). The notation $\langle \vec{u} \rightarrow v \rangle$ denotes that the edge is locally recorded at the vertex u. Similarly, $\langle u \rightarrow \vec{v} \rangle$ denotes that the edge is locally recorded at the vertex v.

OP_k : An operation in a DDC (p. 81). The operation OP_k consists of three steps: (1) the evaluation of a trigger predicate tr_k, (2) the execution of the operation procedure op_k, and (3) the result predicate re_k.

ST : A snapshot of the global system state (p. 88). A subscript is used to specify the situation a snapshot is taken. For example, ST_k^{tr} denotes a snapshot which is taken after the evaluation of the trigger predicate tr_k of an operation OP_k. Superscripts are used to number a sequence of snapshots in time order, e.g., $ST^1 \prec ST^2 \prec \cdots \prec ST^n$.

$tr_k(ST_k^{re})$: A trigger predicate tr_k which is examined (i.e., re-evaluated) in the snapshot ST_k^{re} (p. 89).

$ALGO_D$: A deadlock detection algorithm for dynamic systems (p. 101).

$ALGO_S$: A deadlock detection algorithm for static systems (p. 101).

$DDDC_D(ST^i, ST^f)$: A dynamic deadlock detection computation performed by a dynamic algorithm from the system state ST^i to the state ST^f (p. 104).

$SDDC_D(ST^f)$: A projected static deadlock detection computation performed by a dynamic algorithm in the syatem state ST^f (p. 104).

$SDDC_S(ST^f)$: A static deadlock detection computation performed by a static algorithm in the syatem state ST^f (p. 105).

$CSDDC_S(ST^f)$: A complete static deadlock detection computation performed by a static algorithm in the syatem state ST^f (p. 105).

BIBLIOGRAPHY

[1] Afek, Y. and Saks, M., "Detecting Global Termination Conditions in the Face of Uncertainty," in *Proceedings of the Sixth Annual ACM Symposium on Principles of Distributed Computing*, (Vancouver, B.C., Canada), pp. 109–124, ACM SIGACT-SIGOPS, Aug. 1987.

[2] Agrawal, R., Carey, M. J., and McVoy, L. W., "The Performance of Alternative Strategies for Dealing with Deadlocks in Database Management Systems," *IEEE Transactions on Software Engineering*, vol. SE-13, no. 12, pp. 1348–1363, Dec. 1987.

[3] Andrews, G. R. and Levin, G. M., "On-the-fly Deadlock Prevention," in *Proceedings ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, (Ottawa, Ontario, Canada), pp. 165–172, Aug. 1982.

[4] Andrews, G. R. and Schneider, F. B., "Concepts and Notations for Concurrent Programming," *ACM Computing Surveys*, vol. 15, no. 1, pp. 3–43, Mar. 1983.

[5] Apt, K. R., "Correctness Proofs of Distributed Termination Algorithms," *ACM Transactions on Programming Languages and Systems*, vol. 8, no. 3, pp. 388–405, July 1986.

[6] Apt, K. R. and Francez, N., "Modeling the Distributed Termination Convention of CSP," *ACM Transactions on Programming Languages and Systems*, vol. 6, no. 3, pp. 370–379, July 1984.

[7] Awerbuch, B. and Micali, S., "Dynamic Deadlock Resolution Protocols," in *Proceedings of the Foundations of Computer Science*, (Toronto, Canada), pp. 196–207, IEEE, 1986.

[8] Badal, D. Z., "The Distributed Deadlock Detection Algorithm," *ACM Transactions on Computer Systems*, vol. 4, no. 4, pp. 320–377, Nov. 1986.

[9] Bal, H. E., Steiner, J. G., and Tanenbaum, A. S., "Programming Languages for Distributed Computing Systems," *ACM Computing Surveys*, vol. 21, no. 3, pp. 261–322, Sept. 1989.

[10] Balter, R., Berard, P., and Decitre, P., "Why Control of Concurrency Level in Distributed Systems is More Fundamental than Deadlock Management," in *Proceedings ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, (Ottawa, Ontario, Canada), pp. 183–193, Aug. 1982.

[11] Barbosa, V. C., "Strategies for the Prevention of Communication Deadlocks in Distributed Parallel Programs," *IEEE Transactions on Software Engineering*, vol. 16, no. 11, pp. 1311–1316, Nov. 1990.

[12] Bernstein, P. A., Hadzilacos, V., and Goodman, N., *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, 1987.

[13] Bracha, G. and Toueg, S., "A Distributed Algorithm for Generalized Deadlock Detection," in *Proceedings of the Third Annual ACM Symposium on Principles of Distributed Computing*, (Vancouver, B.C., Canada), pp. 285–301, ACM SIGACT-SIGOPS, Aug. 1984.

[14] Bracha, G. and Toueg, S., "Distributed Deadlock Detection," *Distributed Computing*, vol. 2, no. 3, pp. 127–138, Dec. 1987.

[15] Brinch Hansen, P., "Distributed Processes: A Concurrent Programming Concept," *Communications of the ACM*, vol. 21, no. 11, pp. 934–941, Nov. 1978.

[16] Brookes, S. D. and Roscoe, A. W., "Deadlock Analysis in Networks of Communicating Processes," *Distributed Computing*, vol. 4, no. 4, pp. 209–230, Apr. 1991.

[17] Burns, A., *Concurrent Programming in Ada.* Cambridge University Press, 1985.

[18] Burns, A., Lister, A. M., and Wellings, A. J., *A Review of Ada Tasking*, vol. 262 of *Lecture Notes in Computer Science.* Berlin Heidelberg, Germany: Springer-Verlag, 1987.

[19] Carriero, N. and Gelernter, D., "How to Write Parallel Programs: A Guide to the Perplexed," *ACM Computing Surveys*, vol. 21, no. 3, pp. 323–357, Sept. 1989.

[20] Ceri, S. and Pelagatti, G., *Distributed Databases: Principles and Systems.* McGraw-Hill, 1984.

[21] Chandy, K. M. and Lamport, L., "Distributed Snapshots: Determining Global States of Distributed Systems," *ACM Transactions on Computer Systems*, vol. 3, no. 1, pp. 63–75, Feb. 1985.

[22] Chandy, K. M. and Misra, J., "Asynchronous Distributed Simulation via a Sequence of Parallel Computations," *Communications of the ACM*, vol. 24, no. 11, pp. 198–206, Apr. 1981.

[23] Chandy, K. M. and Misra, J., "A Distributed Algorithm for Detecting Resource Deadlocks in Distributed Systems," in *Proceedings ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, (Ottawa, Ontario, Canada), pp. 157–164, Aug. 1982.

[24] Chandy, K. M. and Misra, J., "The Drinking Philosophers Problem," *ACM Transactions on Programming Languages and Systems*, vol. 6, no. 4, pp. 632–646, Oct. 1984.

[25] Chandy, K. M. and Misra, J., "How Processes Learn," *Distributed Computing*, vol. 1, no. 1, pp. 40–52, Jan. 1986.

[26] Chandy, K. M. and Misra, J., "An Example of Stepwise Refinement of Distributed Programs: Quiescence Detection," *ACM Transactions on Programming Languages and Systems*, vol. 8, no. 3, pp. 326–343, July 1986.

[27] Chandy, K. M., Misra, J., and Haas, L. M., "Distributed Deadlock Detection," *ACM Transactions on Computer Systems*, vol. 1, no. 2, pp. 144–156, May 1983.

[28] Chang, E. J. H., "Echo Algorithms: Depth Parallel Operations on General Graphs," *IEEE Transactions on Software Engineering*, vol. SE-8, no. 4, pp. 391–401, July 1982.

[29] Cheng, J., "A Classification of Tasking Deadlocks," *Ada Letters*, vol. X, no. 5, pp. 110–127, May/June 1990.

[30] Cheng, J., "Task-Wait-For Graphs and Their Application to Handling Tasking Deadlocks," in *TRI-Ada '90 Proceedings*, (Baltimore, MD), pp. 376–390, ACM/SIGAda, Dec. 1990.

[31] Cheng, J., "A Survey of Tasking Deadlock Detection Methods," *Ada Letters*, vol. XI, no. 1, pp. 82–91, January/February 1991.

[32] Choudhary, A. N., "Two Distributed Deadlock Detection Algorithms and Their Performance," Master's thesis, University of Massachusetts at Amherst, Feb. 1986.

[33] Choudhary, A. N., Kohler, W. H., Stankovic, J. A., and Towsley, D., "A Priority Based Probe Algorithm for Distributed Deadlock Detection and Resolution," in *The 7th International Conference on Distributed Computing Systems*, (Berlin, West Germany), pp. 162–168, IEEE-CS, Sept. 1987.

[34] Choudhary, A. N., Kohler, W. H., Stankovic, J. A., and Towsley, D., "Performance Evaluation of Two Distributed Deadlock Detection Algorithms," COINS Technical Report 89-13, University of Massachusetts at Amherst, Nov. 1988.

[35] Choudhary, A. N., Kohler, W. H., Stankovic, J. A., and Towsley, D., "A Modified Priority Based Probe Algorithm for Distributed Deadlock Detection and Resolution," *IEEE Transactions on Software Engineering*, vol. 15, no. 1, pp. 10–17, Jan. 1989.

[36] Choudhary, A. N., Kohler, W. H., Stankovic, J. A., and Towsley, D., "Correction to "A Modified Priority Based Probe Algorithm for Distributed Deadlock Detection and Resolution"," *IEEE Transactions on Software Engineering*, vol. 15, no. 12, p. 1644, Dec. 1989.

[37] Cidon, I., Jaffe, J. M., and Sidi, M., "Local Distributed Deadlock Detection with Finite Buffers," in *IEEE INFORCOM '86*, (Miami, Florida), pp. 478–487, Apr. 1986.

[38] Cidon, I., Jaffe, J. M., and Sidi, M., "Local Distributed Deadlock Detection by Cycle Detection and Clustering," *IEEE Transactions on Software Engineering*, vol. SE-13, no. 1, pp. 3–14, Jan. 1987.

[39] Cohen, S. and Lehmann, D., "Dynamic Systems and Their Distributed Termination," in *Proceedings ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, (Ottawa, Ontario, Canada), pp. 29–33, Aug. 1982.

[40] Dijkstra, E. W., "Co-operating Sequential Processes," in *Programming Languages* (Genuys, F., ed.), pp. 43–112, New York: Academic Press, 1968.

[41] Dijkstra, E. W., Feijen, W. H. J., and van Gasteren, A. J. M., "Derivation of a Termination Detection Algorithm for Distributed Computations," *Information Processing Letters*, vol. 16, no. 5, pp. 217–219, June 1983.

[42] Dijkstra, E. W. and Scholten, C. S., "Termination Detection for Diffusing Computations," *Information Processing Letters*, vol. 11, no. 1, pp. 1–4, Aug. 1980.

[43] Elmagarmid, A. K., "A Survey of Distributed Deadlock Detection Algorithms," *SIGMOD RECORD*, vol. 15, no. 3, pp. 37–45, Sept. 1986.

[44] Elmagarmid, A. K. and Datta, A. K., "Two-Phase Deadlock Detection Algorithm," *IEEE Transactions on Computers*, vol. 37, no. 11, pp. 1454–1458, Nov. 1988.

[45] Elmagarmid, A. K., Sheth, A. P., and Liu, M. T., "Deadlock Detection Algorithms in Distributed Database Systems," in *International Conference on Data Engineering*, (Los Angeles, California), pp. 556–564, IEEE-CS, Feb. 1986.

[46] Elmagarmid, A. K., Soundararajan, N., and Liu, M. T., "A Distributed Deadlock Detection and Resolution Algorithm and Its Correctness Proof," *IEEE Transactions on Software Engineering*, vol. 14, no. 10, pp. 1443–1452, Oct. 1988.

[47] Eswaren, K. P., Gray, J. N., Lorie, R. A., and Traiger, I. L., "The Notions of Consistency and Predicate Locks in a Database System," *Communications of the ACM*, vol. 19, no. 11, pp. 624–633, Nov. 1976.

[48] Francez, N., "Distributed Termination," *ACM Transactions on Programming Languages and Systems*, vol. 2, no. 1, pp. 42–55, Jan. 1980.

[49] Francez, N., "Corrigendum: Distributed Termination," *ACM Transactions on Programming Languages and Systems*, vol. 2, no. 3, p. 463, July 1980.

[50] Francez, N. and Rodeh, M., "Achieving Distributed Termination Without Freezeing," *IEEE Transactions on Software Engineering*, vol. SE-8, no. 3, pp. 287–292, May 1982.

[51] Gehani, N., *Ada: Concurrent Programming*. Englewood Cliffs, New Jersey: Prentice-Hall, Inc., 1984.

[52] German, S. M., "Monitoring for Deadlock and Blocking in Ada Tasking," *IEEE Transactions on Software Engineering*, vol. SE-10, no. 6, pp. 764–777, Nov. 1984.

[53] German, S. M., Helmbold, D. P., and Luckham, D. C., "Monitoring for Deadlocks in Ada Tasking," in *Proceedings of the AdaTEC Conference on Ada*, (Arlington, Virginia), pp. 10–25, ACM, Oct. 1982.

[54] Gligor, V. D. and Shattuck, S. H., "On Deadlock Detection in Distributed Systems," *IEEE Transactions on Software Engineering*, vol. SE-6, no. 5, pp. 435–440, Sept. 1980.

[55] Gray, J., Homan, P., Obermarck, R., and Korth, H., "A Straw Man Analysis of Probability of Waiting and Deadlock," Research Report RJ3066, IBM Research Laboratory, San Jose, California, Feb. 1981. Also appeared in *Fifth International Conference on Distributed Data Management and Computer Networks*, 1981.

[56] Gray, J. N., "Notes on Database Operating Systems," in *Operating Systems: An Advanced Course* (Bayer, R., Graham, R. M., and Seegmuller, G., eds.), pp. 393–481, Berlin: Springer–Verlag, 1978.

[57] Haas, L. M. and Mohan, C., "A Distributed Deadlock Detection Algorithm for a Resource-Based System," Research Report RJ 3765, IBM Research Laboratory, San Jose, California, Jan. 1983.

[58] Hao, K. and Yeh, R. T., "Detection of Inherent Deadlocks in Distributed Programs," in *The 3rd International Conference on Distributed Computing Systems*, (Miami/Ft. Lauderdale, Florida), pp. 518–523, IEEE-CS, Oct. 1982.

[59] Hélary, J.-M., Jard, C., Plouzeau, N., and Raynal, M., "Detection of Stable Properties in Distributed Applications," in *Proceedings of the Sixth Annual ACM Symposium on Principles of Distributed Computing*, (Vancouver, B.C., Canada), pp. 125–136, ACM SIGACT-SIGOPS, Aug. 1987.

[60] Helmbold, D. and Luckham, D., "Debugging Ada Tasking Programs," Technical Report No. 84-262 (Program Analysis and Verification Group Report No. 25), Stanford University, July 1984.

[61] Helmbold, D. and Luckham, D., "Debugging Ada Tasking Programs," *IEEE Software*, vol. 2, no. 2, pp. 47–57, Mar. 1985.

[62] Helmbold, D. and Luckham, D. C., "Runtime Detection and Description of Deadness Errors in Ada Tasking," Technical Report No. 83-249 (Program Analysis and Verification Group Report No. 22), Stanford University, Nov. 1983.

[63] Hermann, T. and Chandy, K. M., "A Distributed Procedure to Detect AND/OR Deadlock," Technical Report TR LCS-8301, Department of Computer Sciences, University of Texas, Austin, Texas, 1983.

[64] Ho, G. S. and Ramamoorthy, C. V., "Protocols for Deadlock Detection in Distributed Database Systems," *IEEE Transactions on Software Engineering*, vol. SE-8, no. 6, pp. 554–557, Nov. 1982.

[65] Hoare, C. A. R., "Communicating Sequential Processes," *Communications of the ACM*, vol. 21, no. 8, pp. 666–677, Aug. 1978.

[66] Holt, R. C., "Some Deadlock Properties on Computer Systems," *ACM Computing Surveys*, vol. 4, no. 3, pp. 179–196, Sept. 1972.

[67] Huang, J., Stankovic, J. A., Towsley, D., and Ramamritham, K., "Experimental Evaluation of Real-Time Transaction Processing," in *Proceedings of the 10th Real-Time Systems Symposium*, (Santa Monica, California), pp. 144–153, IEEE-CS, Dec. 1989.

[68] Huang, S.-T., "A Distributed Deadlock Detection Algorithm for CSP-Like Communication," *ACM Transactions on Programming Languages and Systems*, vol. 12, no. 1, pp. 102–122, Jan. 1990.

[69] Ichbiah, J. D., Barnes, J. G. P., Firth, R. J., and Woodger, M., *Rationale for the Design of the Ada Programming Language*. United States Government, 1986. The Ada Rationale was developed by Alsys and Honeywell under a contract from the United States Government (Ada Joint Program Office).

[70] Isloor, S. S. and Marsland, T. A., "The Deadlock Problem: An Overview," *IEEE Computer*, vol. 13, no. 9, pp. 58–78, Sept. 1980.

[71] Jagannathan, J. R. and Vasudevan, R., "A Distributed Deadlock Detection and Resolution Scheme: Performance Study," in *The 3rd International Conference on Distributed Computing Systems*, (Miami/Ft. Lauderdale, Florida), pp. 496–501, IEEE-CS, Oct. 1982.

[72] Jagannathan, J. R. and Vasudevan, R., "Comments on "Protocols for Deadlock Detection in Distributed Database Systems"," *IEEE Transactions on Software Engineering*, vol. SE-9, no. 3, p. 371, May 1983.

[73] Kamel, R. and Lo, S.-L., "Design Issues in the Implementation of Remote Rendezvous," in *Proceedings of the 10th International Conference on Distributed Computing Systems*, (Paris, France), pp. 245–251, IEEE Computer Society, May 28–June 1 1990.

[74] Karam, G. M. and Buhr, R. J. A., "Temporal Logic-Based Deadlock Analysis for Ada," *IEEE Transactions on Software Engineering*, vol. 17, no. 10, pp. 1109–1125, Oct. 1991.

[75] Katz, S. and Shmueli, O., "Cooperative Distributed Algorithms for Dynamic Cycle Prevention," *IEEE Transactions on Software Engineering*, vol. SE-13, no. 5, pp. 540–552, May 1987.

[76] Kawazu, S., Minami, S., Itoh, K., and Teranaka, K., "Two-Phase Deadlock Detection Algorithm in Distributed Databases," in *Proceedings 5th International Conference on Very Large Databases*, pp. 360–367, 1979.

[77] Knapp, E., "Deadlock Detection in Distributed Databases," *ACM Computing Surveys*, vol. 19, no. 4, pp. 303–328, Dec. 1987.

[78] Korth, H. F., "A Deadlock-Free, Variable Granularity Locking Protocol," in *Proceedings of the 5th Berkeley Conference on Distributed Data Management and Computer Networks*, pp. 105–121, Feb. 1981.

[79] Korth, H. F., "Edge Locks and Deadlock Avoidance in Distributed Systems," in *Proceedings ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, (Ottawa, Ontario, Canada), pp. 173–182, Aug. 1982.

[80] Korth, H. F., "Locking Primitives in a Database System," *Journal of the ACM*, vol. 30, no. 1, pp. 55–79, Jan. 1983.

[81] Korth, H. F., Krishnamurthy, R., Nigam, A., and Robinson, J. T., "A Framework for Understanding Distributed (Deadlock Detection) Algorithms," in *Proceedings of the Second ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, (Atlanta, Ga.), pp. 192–202, Mar. 1983.

[82] Kshemkalyani, A. D., *Characterization and Correctness of Distributed Deadlock Detection and Resolution.* PhD thesis, The Ohio State University, 1991.

[83] Kshemkalyani, A. D. and Singhal, M., "Characterization and Correctness of Distributed Deadlocks," Tech. Report OSU-CISRC-6/90-TR15, The Ohio State University, 1990.

[84] Kshemkalyani, A. D. and Singhal, M., "Invariant-Based Verification of a Distributed Deadlock Algorithm," *IEEE Transactions on Software Engineering*, vol. 17, no. 8, pp. 789–799, Aug. 1991.

[85] Lamport, L., "Time, Clocks, and the Ordering of Events in a Distributed System," *Communications of the ACM*, vol. 21, no. 7, pp. 558–565, July 1978.

[86] Ledgard, H., *ADA: An Introduction/Ada Reference Manual.* New York: Springer-Verlag, 1981/1980. The *Part II — Ada Reference Manual,* July 1980, was also published by the United States Government.

[87] Li, H. F., Radhakrishnan, T., and Venkatesh, K., "Global State Detection in NON-FIFO Networks," in *The 7th International Conference on Distributed Computing Systems*, (Berlin, West Germany), pp. 364–370, IEEE-CS, Sept. 1987.

[88] Lin, W.-T. K. and Nolte, J., "Communication Delay and Two Phase Locking," in *The 3rd International Conference on Distributed Computing Systems*, (Miami/Ft. Lauderdale, Florida), pp. 502–507, IEEE-CS, Oct. 1982.

[89] Liu, L., "Comments on "A Distributed Scheme for Detecting Communication Deadlocks"," *IEEE Transactions on Software Engineering*, vol. 15, no. 7, p. 926, July 1989.

[90] Lomet, D. B., "Coping with Deadlock in Distributed Systems," in *Proceedings of IFIP Working Conference on Data Base Architecture*, pp. 95–105, Venice, Italy: North-Holland Publishing Company, June 1979.

[91] Lynch, N. A. and Tuttle, M. R., "Hierarchical Correctness Proofs for Distributed Algorithms," in *Proceedings of the Sixth Annual ACM Symposium on Principles of Distributed Computing*, (Vancouver, B.C., Canada), pp. 137–151, ACM SIGACT-SIGOPS, Aug. 1987.

[92] Maekawa, M., Oldehoeft, A. E., and Oldehoeft, R. R., *Operating Systems – Advanced Concepts.* The Benjamin/Cummings Publishing Company, Inc., 1987.

[93] Marsland, T. A. and Isloor, S. S., "Detection of Deadlocks in Distributed Database Systems," *INFOR*, vol. 18, no. 1, pp. 1–20, Feb. 1980.

[94] Mattern, F., "Algorithms for Distributed Termination Detection," *Distributed Computing*, vol. 2, no. 3, pp. 161–175, Dec. 1987.

[95] Menasce, D. A. and Muntz, R. R., "Locking and Deadlock Detection in Distributed Data Bases," *IEEE Transactions on Software Engineering*, vol. SE-5, no. 3, pp. 195–202, May 1979.

[96] Misra, J., "Detecting Termination of Distributed Computations Using Markers," in *Proceedings of the Second Annual ACM Symposium on Principles of Distributed Computing*, (Montreal, Canada), pp. 290–294, ACM SIGACT-SIGOPS, Aug. 1983.

[97] Misra, J. and Chandy, K. M., "A Distributed Graph Algorithm: Knot Detection," *ACM Transactions on Programming Languages and Systems*, vol. 4, no. 4, pp. 678–686, Oct. 1982.

[98] Misra, J. and Chandy, K. M., "Termination Detection of Diffusing Computations in Communicating Sequential Processes," *ACM Transactions on Programming Languages and Systems*, vol. 4, no. 1, pp. 37–43, Jan. 1982.

[99] Misra, J., Chandy, K. M., and Smith, T., "Proving Safety and Liveness of Communicating Processes with Examples," in *Proceedings ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, (Ottawa, Ontario, Canada), pp. 201–208, Aug. 1982.

[100] Mitchell, D. P. and Merritt, M. J., "A Distributed Algorithm for Deadlock Detection and Resolution," in *Proceedings of the Third Annual ACM Symposium on Principles of Distributed Computing*, (Vancouver, B.C., Canada), pp. 282–284, ACM SIGACT-SIGOPS, Aug. 1984.

[101] Moss, J. E. B., "Nested Transactions: An Approach to Reliable Distributed Computing," Report MIT/LCS/TR-260, Laboratory for Computer Science, MIT, Apr. 1981.

[102] Muhanna, W. A., "Composite Programs: Hierarchical Construction, Circularity, and Deadlocks," *IEEE Transactions on Software Engineering*, vol. 17, no. 4, pp. 320–333, Apr. 1991.

[103] Murata, T., Shenker, B., and Shatz, S. M., "Detection of Ada Static Deadlocks Using Petri Net Invariants," *IEEE Transactions on Software Engineering*, vol. 15, no. 3, pp. 314–326, Mar. 1989.

[104] Murphy, S. L. and Shankar, A. U., "A Note on the Drinking Philosophers Problem," *ACM Transactions on Programming Languages and Systems*, vol. 10, no. 1, pp. 178–188, Jan. 1988.

[105] Natarajan, N., "A Distributed Scheme for Detecting Communication Deadlocks," *IEEE Transactions on Software Engineering*, vol. SE-12, no. 4, pp. 531–537, Apr. 1986.

[106] Obermarck, R., "Distributed Deadlock Detection Algorithm," *ACM Transactions on Database Systems*, vol. 7, no. 2, pp. 187–208, June 1982.

[107] Owicki, S. and Lamport, L., "Proving Liveness Properties of Concurrent Programs," *ACM Transactions on Programming Languages and Systems*, vol. 4, no. 3, pp. 455–495, July 1982.

[108] Papadimitriou, C., *The Theory of Database Concurrency Control.* Principles of Computer Science Series, Rockville, Maryland: Computer Science Press, 1986.

[109] Rajagopalan, S., "Deadlock Detection Techniques in Distributed Databases," Master's thesis, University of Massachusetts at Amherst, May 1985.

[110] Rana, S. P., "A Distributed Solution of the Distributed Termination Problem," *Information Processing Letters*, vol. 17, no. 1, pp. 43–46, July 1983.

[111] Reif, J. H. and Spirakis, P. G., "Real-Time Synchronization of Interprocess Communications," *ACM Transactions on Programming Languages and Systems*, vol. 6, no. 2, pp. 215–238, Apr. 1984.

[112] Riccardi, G. A. and Baker, T. P., "A Runtime Supervisor to Support Ada Tasking: Rendezvous and Delays," in *Ada in use - Proceedings of the Ada International Conference*, pp. 329–342, ACM and Ada-Europe, Cambridge University Press, May 1985.

[113] Roesler, M. and Burkhard, W. A., "Deadlock Resolution and Semantic Lock Models in Object Oriented Distributed Systems," in *Proceedings International Conference on Management Data (SIGMOD Vol. 17, No. 3)*, (Chicago, Illinois), pp. 361–370, ACM, June 1988.

[114] Roesler, M. and Burkhard, W. A., "Resolution of Deadlocks in Object-Oriented Distributed Systems," *IEEE Transactions on Computers*, vol. 38, no. 8, pp. 1212–1224, Aug. 1989.

[115] Roesler, M., Burkhard, W. A., and Cooper, K. B., "Efficient Deadlock Resolution for Lock-Based Concurrency Control Schemes," in *The 8th International Conference on Distributed Computing Systems*, (San Jose, California), pp. 224–233, IEEE-CS, June 1988.

[116] Rosenblum, D. S., "An Efficient Communication Kernel for Distributed Ada Runtime Tasking Supervisors," *Ada LETTERS*, vol. VII, no. 2, pp. 102–117, March-April 1987.

[117] Rosenblum, D. S., *Design and Verification of Distributed Tasking Supervisors for Concurrent Programming Languages.* PhD thesis, Stanford University, Mar. 1988. Also appear as Technical Report No. 88-375 (Program Analysis and Verification Group Report No. 38).

[118] Sanders, B. A. and Heuberger, P. A., "Distributed Deadlock Detection and Resolution with Probes," in *Proc. of the 3rd International Workshop on Distributed Algorithms*, (Nice, France), pp. 207–218, Sept. 1989.

[119] Shih, C.-S. and Stankovic, J. A., "Survey of Deadlock Detection in Distributed Concurrent Programming Environments and Its Application to Real-time Systems," COINS Technical Report 90-69, University of Massachusetts at Amherst, Aug. 1990.

[120] Shih, C.-S. and Stankovic, J. A., "Distributed Deadlock Detection in Ada Runtime Environments," in *TRI-Ada '90 Proceedings*, (Baltimore, MD), pp. 362–375, ACM/SIGAda, Dec. 1990.

[121] Shih, C.-S. and Stankovic, J. A., "Deadlock Detection in Distributed Real-Time Systems and Its Application to Ada Environments," *Computer Science and Informatics*, vol. 21, no. 1, pp. 1–28, July 1991. Invited Paper, CSI Journal, Computer Society of India.

[122] Shrivastava, S. K., "On the Treatment of Orphans in a Distributed System," in *3rd Symposium on Reliability in Distributed Software and Database Systems*, (Clearwater Beach, Florida), pp. 155–162, IEEE-CS/ACM, Oct. 1983.

[123] Shyu, S. C., Li, V. O. K., and Wang, C. P., "An Abortion-Free Distributed Deadlock Detection/Resolution Algorithm," in *Proceedings of the 10th International Conference on Distributed Computing Systems*, (Paris, France), pp. 167–174, IEEE Computer Society, May 28–June 1 1990.

[124] Singhal, M., "Deadlock Detection in Distributed Systems," *IEEE Computer*, vol. 22, no. 11, pp. 37–48, Nov. 1989.

[125] Sinha, M. K. and Natarajan, N., "A Distributed Deadlock Detection Algorithm Based on Timestamps," in *The 4th International Conference on Distributed Computing Systems*, (San Francisco, California), pp. 546–556, IEEE Computer Society, May 1984.

[126] Sinha, M. K. and Natarajan, N., "A Priority Based Distributed Deadlock Detection Algorithm," *IEEE Transactions on Software Engineering*, vol. SE-11, no. 1, pp. 67–80, Jan. 1985.

[127] Soundararajan, N., "Axiomatic Semantics of Communicating Sequential Processes," *ACM Transactions on Programming Languages and Systems*, vol. 6, no. 4, pp. 647–662, Oct. 1984.

[128] Spezialetti, M. and Kearns, P., "Efficient Distributed Snapshots," in *The 6th International Conference on Distributed Computing Systems*, (Cambridge, Massachusetts), pp. 382–388, IEEE-CS, May 1986.

[129] Sugihara, K., Kikuno, T., Yoshida, N., and Ogata, M., "A Distributed Algorithm for Deadlock Detection and Resolution," in *4th Symposium on Reliability in Distributed Software and Database Systems*, (Silver Spring, Maryland), IEEE-CS, Oct. 1984.

[130] Tay, Y. C., *Locking Performance in Centralized Databases*, vol. 14 of *Perspectives in Computing*. Academic Press, 1987.

[131] Tay, Y. C. and Loke, W. T., "A Theory for Deadlocks," Technical Report CS-TR-344-91, Department of Computer Science, Princeton University, Aug. 1991.

[132] Tsai, W.-C. and Belford, G. G., "Detecting Deadlock in a Distributed System," in *IEEE INFORCOM '82*, (Las Vegas, Nevada), pp. 89–95, Mar. 1982.

[133] Tsai, W.-C., Elmagarmid, A. K., and Hurson, A. R., "Deadlock Detection and Resolution in Distributed Database Systems," in *IEEE INFORCOM '87*, (San Francisco, California), pp. 77–86, Mar. 1987.

[134] Tu, S., Shatz, S. M., and Murata, T., "Applying Petri Net Reduction to Support Ada-Tasking Deadlock Detection," in *Proceedings of the 10th International Conference on Distributed Computing Systems*, (Paris, France), pp. 96–103, IEEE Computer Society, May 28–June 1 1990.

[135] Ullman, J. D., *Principles of Database Systems*. Pitman Publishing Limited, second ed., 1982.

[136] Volz, R. A. and Mudge, T. N., "Timing Issues in the Distributed Execution of Ada Programs," *IEEE Transactions on Computers*, vol. C-36, no. 4, pp. 449–459, Apr. 1987.

[137] Volz, R. A., Mudge, T. N., Naylor, A. W., and Mayer, J. H., "Some Problems in Distributed Real-time Ada Programs Across Machines," in *Ada in use – Proceedings of the Ada International Conference*, pp. 72–84, ACM and Ada-Europe, Cambridge University Press, May 1985.

[138] Wójcik, B. E. and Wójcik, Z. M., "Sufficient Condition for a Communication Deadlock and Distributed Deadlock Detection," *IEEE Transactions on Software Engineering*, vol. 15, no. 12, pp. 1587–1595, Dec. 1989.

[139] Wuu, G. T. and Bernstein, A. J., "False Deadlock Detection in Distributed Systems," *IEEE Transactions on Software Engineering*, vol. SE-11, no. 8, pp. 820–821, Aug. 1985.

[140] Zhou, B., Yeh, R. T., and Ng, P. A., "Principle of Deadlock Detection in Ada Programs," in *The 6th International Conference on Distributed Computing Systems*, (Cambridge, Massachusetts), pp. 572–579, IEEE Computer Society, May 1986.

[141] Zhou, B., Yeh, R. T., and Ng, P. A.-B., "An Algebraic System for Deadlock Detection and Its Applications," in *4th Symposium on Reliability in Distributed Software and Database Systems*, (Silver Spring, Maryland), IEEE-CS, Oct. 1984.

[142] Zöbel, D., "The Deadlock Problem: A Classifying Bibliography," *Operating Systems Review*, vol. 17, no. 4, pp. 6–15, Oct. 1983.