STATIC DETECTION OF DEADLOCKS IN POLYNOMIAL TIME

BY STEPHEN P. MASTICOLA

A dissertation submitted to the

Graduate School—New Brunswick

Rutgers, The State University of New Jersey

in partial fulfillment of the requirements

for the degree of

Doctor of Philosophy

Graduate Program in Computer Science

Written under the direction of

Dr. Barbara G. Ryder

and approved by

_____

_____

_____

_____

_____

New Brunswick, New Jersey

May, 1993

© 1993
Stephen P. Masticola
ALL RIGHTS RESERVED

ABSTRACT OF THE DISSERTATION

Static Detection of Deadlocks in Polynomial Time

by Stephen P. Masticola, Ph.D.

Dissertation Director: Dr. Barbara G. Ryder

Parallel and distributed programming languages often include explicit synchronization primitives, such as rendezvous and semaphores. Such programs are subject to synchronization anomalies; the program behaves incorrectly because it has a faulty synchronization structure. A deadlock is an anomaly in which some subset of the active tasks of the program mutually wait on each other to advance; thus, the program cannot complete execution.

In static anomaly detection, the source code of a program is automatically analyzed to determine if the program can ever exhibit a specific anomaly. Static anomaly detection has the unique advantage that it can certify programs to be free of the tested anomaly; dynamic testing cannot generally do this. Though exact static detection of deadlocks is NP-hard [Tay83a], many researchers have tried to detect deadlock by exhaustive enumeration of synchronization states, using Petri nets or other program representations. In practice, programs often have large enough state spaces to render this approach impractical.

Our approach, rather, is to make an approximate analysis of the program in time polynomial in the size of the source code. Our approximation is *safe*: if we certify a

program free of deadlock, it will never deadlock. We do this using iterative flow analysis techniques to detect (but not enumerate) "deadlock cycles" in the program's control and synchronization structure. We identify four constraints on deadlock cycles which we use to prune invalid cycles, thus avoiding false alarms. One pruning method uses a "can't happen together" relation between statements; we show how such information can be found, and how it may be of value in other analyses.

We have implemented our analysis for the rendezvous synchronization of the Ada language, and have tested it on over 100 programs obtained from government and industrial sources. We demonstrate that our technique is quite accurate compared to exhaustive state generation; few false alarms are seen in practice. Our technique is also well behaved in execution time, running faster than the exhaustive technique for all programs except those with the simplest state spaces.

To demonstrate the generality of our methods, we show preliminary algorithms for detecting deadlock in two very different synchronization paradigms: binary semaphores, and the dynamic, pointer-based process control of Concurrent C.

Preface

Holy deadlock.[1] *Sir Albert Patrick Herbert*

[1] Title of a novel [1934] satirizing the paradoxes of the British divorce law.

Acknowledgements

I'd like to thank the members of my committee, in alphabetical order:

- Alex Borgida, for his thoroughly professional teaching and his welcome encouragement.
- Lori Clarke, for her fresh and knowledgeable intuition and high standards.
- Thomas J. Marlowe, for his keen insight, hard work, and unfailing good humor.
- Barbara G. Ryder, for her generous guidance and support, technical, financial, and personal.
- Don Smith, for his patient and wise counsel throughout the second half of my graduate career.

I'd also like to thank the members of the Rutgers Programming Languages Reading Group, for providing a springboard for discussion and a needed source of information.

Dedication

so, so you think you can tell / heaven from hell / blue skies from pain
can you tell a green field from a cold steel rail / a smile from a veil

do you think you can tell? and did they get you to trade
your heroes for ghosts
hot ashes for trees
hot air for the cool breeze
cold comfort for change
did you exchange

a walk-on part in a war
for a lead role in a cage?

Pink Floyd
Wish You Were Here

I dedicate this thesis to all the people who made a difference in my life through the course of my graduate study.

- To my parents, Robert and Nancy Masticola, for their unwavering support.
- To my friends, both inside and outside academia. I'll not name names; there are many, and they know who they are.
- And lastly, to Marcella Sullivan Price, Edward L. Maher, and Christina McAuliffe. Each inspired one person in his or her own way; thus, their lives had meaning.

Table of Contents

Abstract ii

Preface iv

Acknowledgements v

Dedication vi

List of Tables xxii

List of Figures xxiv

List of Abbreviations xxix

1. Overview. 1

1.1. The deadlock problem. 2

1.1.1. Tasking and synchronization. 2

1.1.2. The importance of detecting possible deadlocks. 3

Our main research goal. 4

1.1.3. A short comparison with existing methods. 4

Dynamic testing. 4

Static analysis. 5

State enumeration. 5

Formal proof methods. 6

1.1.4. Our requirements. 6

1.2. Our approach: Detection of constrained deadlock cycles. 7

1.3. Main contributions. 8

1.4. Guide to the thesis. 9

2. General framework. 10
2.1. Introduction. 11
2.1.1. Terminology and notation. 11
2.1.2. A short sketch of the process. 11
2.2. Theoretical work. 13
2.2.1. Graph model of synchronization structure. 13
2.2.2. Execution model and state generation. 15
Abstraction of the program state. 16
Execution model. 17
Sufficient conditions for transitions. 17
Necessary conditions for transitions. 17
Example. 18
States of entries. 18
State enumeration algorithm. 18
2.2.3. A structural definition of deadlock and starvation. 19
Artifactual deadlock cycles. 21
2.2.4. Constraints on deadlock cycles. 21
Four constraints. 21
Enforcing the constraints in cycle detection. 23
2.3. Developing a deadlock analyzer. 24
2.3.1. Non-concurrency analysis. 25
Widening and pseudotransitivity in *CHT*. 26
B4 analysis. 30
A troublesome non-concurrency relationship. 31
2.3.2. Finding constraint information other than *CHT*. 32
2.3.3. Refining the sync graph. 32
2.3.4. Multi-source lattice frameworks. 33

Our formulation: the cross-product of homogeneous lattice frameworks. 34
Cross-product of heterogeneous lattice frameworks. 36
2.3.5. Cycle location graphs. 37
2.3.6. The *DeadPath* problem. 39
2.4. Experimentation and evaluation. 40

3. Lattice frameworks and properties. 42
3.1. Lattice frameworks and properties for deadlock detection. 43
3.1.1. Motivating the analysis. 43
3.1.2. General lattice framework properties. 43
Closure properties. 44
Convergence time for iterative lattice problems. 46
3.1.3. Tarjan's path problem methods. 47
3.1.4. Kam-Ullman "rapid" problems in irreducible graphs. 49
3.1.5. Ordering of monotone inflationary functions in composition. . . . 50
3.1.6. Incremental edge elimination. 51

4. Modeling a subset of Ada programs. 53
4.1. Introduction. 54
4.2. A basic subset of Ada. 55
4.2.1. Tasking behavior and rendezvous in the basic Ada subset. 55
4.2.2. Sync graphs programs in the basic Ada subset. 56
4.2.3. Model of program execution in the basic Ada subset. 57
4.2.4. Constraints on deadlock cycles in the basic Ada subset. 58
4.2.5. Cycle location graphs in the basic Ada model. 59
4.2.6. Theoretical bounds on complexity of deadlock cycle detection. . 61
Discussion. 68
4.3. `Accept...do` construct. 69

4.3.1. Sync hypergraphs. 69
Representing control flow. 70
Representing synchronization. 70
4.3.2. Execution model. 71
4.3.3. Infinite wait anomalies in the `Accept...do` model. 72
Characteristics of anomalies. 72
Deadlocks involving hyperedges. 73
"Waits on" relationship and hyperedges. 74
4.3.4. Hyperedge cycle location graphs. 75
4.3.5. Scope depth and feasible deadlocks. 75
4.3.6. Impact on constraints. 78
4.4. `Select` construct. 81
4.4.1. Representing conditional entries. 82
4.4.2. Conditional entries without `else` or `delay`. 82
Guard clauses. 82
Commitment and sync hyperedge paths. 84
Scope depth and conditional entries. 85
4.4.3. Conditional entries with `else` clauses or `delay` alternatives. . . . 85
`Else` clauses and `delay` alternatives without rendezvous statements inside the clause. 86
`Else` clauses and `delay` alternatives containing head nodes. . . . 86
`Else` clauses and `delay` alternatives containing tail nodes. 87
4.4.4. Conditional and timed entry calls. 89
Representing conditional entry calls. 89
Conditional entry calls and deadlock. 89
4.4.5. Impact to constraints. 90

5. The *CHT* relation in Ada programs. 93

5.1. Computing *CHT* in the Ada model. 94
5.1.1. *B4* analysis. 94
B4 lattice framework. 95
Completor edges. 95
Function space. 97
Using *CHT* information to eliminate completor edges. 101
Incremental update of *B4* sets. 101
Augmenting *CHT* information using *B4* analysis. 101
5.1.2. Pinning analysis on successors of a rendezvous. 103
Example of pinning analysis. 106
Pinning analysis when the begin node is in $CPreds(n)$. 108
Pinning and the `accept-do` and `select` constructs. 110
Predecessors of a node. 110
Sisters. 110
Algorithm for pinning analysis. 111
Propagating pinning information. 113
5.1.3. Strict intervals. 114
Algorithm to find strict intervals. 116
5.1.4. Critical section analysis. 118
An example of a critical section. 119
Critical section structures. 121
Possibility 1 125
Possibility 2 125
Possibility 3 125
Possibility 4 126
Possibility 5 126
Relaxing the structural requirements on critical sections. 127
Eliminating the stickiness check. 127

Multiple critical section bodies. 128
Scope depth and critical sections. 129
Identifying critical sections. 131
Expanding *CHT* information using critical section structures. . . 133
Deadlocks with critical section nodes as head nodes. 135
Example. 135
Bracket node propagation and deadlock cycle detection. 137
Signaling head node propagation and critical section structures. . 143
Critical sections which call other critical sections. 144
A naive solution. 145
Critical section call graphs. 145
Nested calls to simple critical sections. 146
Algorithm for constructing the simple critical section call graph. 148
5.1.5. A more general definition of critical sections. 150
Two-task mutual exclusion. 151
An example of *CHT* with three tasks. 155
5.1.6. Remote procedure call analysis. 157
5.1.7. Iterative *CHT* analysis. 159
5.1.8. Total time for *CHT* analysis. 161
5.1.9. Additional uses of the *CHT* relation. 162

6. Detecting deadlock in Ada programs. 163
6.1. Signaling count lattice framework. 164
6.1.1. Edge functions. 165
6.1.2. Signal counts and 1-semiboundedness. 168
6.1.3. Mapping function. 169
6.1.4. Time for the total deadlock path algorithm. 173
6.2. Signaling node count/accept scope depth lattice framework. 174

6.2.1. Edge functions. 176
6.2.2. Mapping function. 181
6.3. Total time for the algorithm. 185

7. Experimental results. 186
7.1. Introduction. 187
7.1.1. Objectives of the experiment. 187
Objective 1: Deadlock analysis characterization. 187
Objective 2: *CHT* analysis characterization. 187
Objective 3: Pruning information effect characterization. 187
Data gathering objectives. 188
A note on terminology. 188
7.2. Architecture of the deadlock analyzer software. 189
7.2.1. Overview of the analyzer. 189
All-states analysis and deadlock detection. 189
7.2.2. Additional controls for analysis. 190
7.2.3. Outputs and diagnostic information. 190
7.2.4. Source files. 191
7.2.5. Limitations of the analyzer implementation. 192
7.3. Sources of test programs. 192
7.4. Experiment design. 194
7.4.1. Data gathered. 194
7.5. Statistical methods. 196
7.5.1. Least mean squares regression. 196
Regression to polynomials of independent variables. 197
Leaps and bounds. 197
Holdbacks. 197
7.5.2. Regression quality metrics. 198

Standard error. . . . 198
Adjusted r^2. . . . 198
The F ratio. . . . 199
7.5.3. Confidence intervals. . . . 199
7.5.4. Some pitfalls of statistical analysis of execution time. . . . 199
7.6. Overall experiment summary. . . . 200
7.7. Polynomial deadlock analysis. . . . 202
7.7.1. Estimated execution time of polynomial deadlock analysis. . . . 202
The best approximation. . . . 204
7.8. *CHT*. . . . 207
7.8.1. Time requirements. . . . 207
7.8.2. Overall accuracy. . . . 208
Comparison with [DS91] 210
7.8.3. Contributions of the refinements of *CHT*. . . . 210
7.8.4. Sync edge pruning. . . . 211
7.8.5. Effectiveness of *CHT* in deadlock detection. . . . 211
7.8.6. Synergy effects, deadlock, and *CHT* analysis. . . . 214
7.9. Effectiveness of the *SACnt* summaries. . . . 216
7.10. All-states deadlock analysis. . . . 219
7.10.1. Accuracy of all-states analysis. . . . 219
7.10.2. Predicting execution time. . . . 220
Wampler's hypothesis. . . . 220
Conclusions about predicting all-states execution time. . . . 222
7.11. Conclusions from the experiment. . . . 222

8. Binary semaphores — an initial study. . . . 224
8.1. Introduction. . . . 225
Integer semaphores. . . . 225

8.2. Modeling binary semaphore synchronization for deadlock detection. . . . 225
8.2.1. Mapping semaphores to the rendezvous model. 226
The need for additional state information. 228
8.2.2. Graph representations of binary semaphore programs. 229
8.2.3. Execution model of binary semaphore programs. 230
8.2.4. Constraints on deadlocks in the binary semaphore model. 231
Single-task deadlocks. 232
8.3. Overview of deadlock detection in binary semaphores. 233
8.4. *CHT* and binary semaphores. 236
8.4.1. Mutual exclusion using semaphores. 236
Structural definition of a mutex. 236
Identifying mutex structures. 238
8.4.2. Semaphore critical section structures. 240
An example of a semaphore critical section structure. 240
Structural definition of SEM critical section structures. 241
Identifying SEM critical section structures. 245
8.4.3. *B4* analysis of semaphores. 247
An example of limited *B4* analysis in the semaphore model. . . . 247
Lattice framework for limited *B4* analysis in binary semaphores. 247
Completor edges in the *B4* sync graph. 248
B4 lattice. 249
B4 function space and mapping for binary semaphores. 249
8.4.4. Eliminating spurious sync edges. 250
Examples of spurious sync edges. 251
Formal definitions of spurious sync edges. 251
8.4.5. The "semaphore kill" problem. 251
The semaphore kill (*SemKill*) lattice. 253
The $F_{SemKill}$ function space. 254

SemKill mapping function. 257
Convergence properties of the *SemKill* framework. 258
8.4.6. Time complexity of *CHT* analysis. 259
8.5. Finding nodes at which semaphores must always be set. 260
8.5.1. Theoretical background for the *MustSet* relation. 260
8.5.2. The *MustSet* lattice framework. 261
The $G_{MustSet}$ graph structure. 262
The $L_{MustSet}$ lattice space. 262
The $F_{MustSet}$ function space. 262
The $M_{MustSet}$ mapping function. 263
Refining *MustSet* with critical section information. 264
Methodology and time bounds. 264
Additional refinements. 265
8.5.3. The *MustClear* relation. 266
8.6. Finding deadlock cycles in semaphore programs. 266
8.6.1. The CDSEM sync graph. 268
8.6.2. Lattice framework for deadlock cycle detection. 268
Lattice space. 268
Function space. 269
Mapping function in the SEM lattice framework. 270
8.7. Complexity of deadlock analysis in binary semaphores. 272

9. Concurrent C — an initial study. 273
9.1. Overview of Concurrent C. 274
9.1.1. Differences between Concurrent C and Ada. 274
Process variables. 275
Transaction variables. 275
Other Concurrent C synchronization constructs. 275

Passing variables between processes. 276
An example of Concurrent C synchronization. 276
Initial simplifying assumptions. 277
Reasons for the choice of language features. 278
The assumptions. 278
9.2. Modeling of Concurrent C program execution. 280
9.2.1. CC sync graphs. 280
Definition of the CC sync graph. 281
Nodes. 281
Control edges. 281
Signal types, sync variables, and sync edges. 282
Semantics of nodes and sync edges. 283
Sync variables in the execution model. 283
CC execution graphs of dynamic program executions. 285
9.2.2. An alternate static graph representation. 285
A problem with the CC sync graph. 285
An alternative graph representation. 288
9.2.3. Execution wave. 290
9.3. Constraints on deadlock cycles in the Concurrent C model. 292
9.3.1. The use of constraint 3 in pruning deadlock cycles. 294
Reaching definitions and *CHT*. 295
9.3.2. Partitioning. 296
An algorithm for partitioning. 299
9.4. Computing *CHT* sets in Concurrent C. 303
9.4.1. Initializing *CHT*. 303
9.4.2. The *CHTSelf* relation. 304
Properties of *CHTSelf*. 304
9.4.3. Pinning analysis. 307

Dynamic conditions for pinning. 307
Static conditions for pinning. 309
Finding pinning in Concurrent C programs. 310
Pinning analysis and non-head nodes. 312
Formalism. 312
Algorithm for pinning analysis in Concurrent C. 313
9.4.4. Critical sections. 315
Weak critical section structures. 315
Formal definition of Concurrent C critical section structures. . . 317
An alternate definition of critical section structures for Concurrent C. 319
Algorithms for critical section structures in Concurrent C programs.319
9.4.5. *B4* analysis in Concurrent C programs. 322
Preconditions to deduce *B4*. 322
Lattice framework for limited *B4* analysis in Concurrent C. . . . 324
Completor edges in Concurrent C programs. 324
B4 mapping function for Concurrent C. 325
9.4.6. Time complexity of *CHT* analysis for Concurrent C. 326
9.5. Deadlock cycle detection in Concurrent C programs. 326
9.5.1. Invalid deadlock paths within a single process. 327
Control paths with accept head and tail nodes of the same signal type. 327
Control paths that traverse create edges. 328
The cycle detection (CDCC) sync graph. 329
9.5.2. CC cycle location graphs. 331
9.5.3. CC lattice framework for cycle detection. 331
Lattice space. 332
Function space. 332

Mapping function in the CC lattice framework. 334
9.6. Complexity of the entire analysis. 336
9.6.1. Complexity of analyzing the entire program. 336
9.7. Conclusions. 337

10. Related work. 338
10.1. Introduction. 339
10.2. Problem characterization. 339
Concise overview. 340
10.2.1. Coffman and Denning [CD73] 340
10.2.2. Complexity of static synchronization analysis: Taylor [Tay83a] . 341
10.2.3. Classification of Ada deadlocks: Cheng [Che90] 343
10.3. Static nontermination anomaly detection. 345
Concise overview. 345
10.3.1. Flow graph methods. 346
Concise overview. 346
Concurrency state enumeration: Taylor [Tay83b] 346
Concurrency analysis and symbolic execution: Young and Taylor [YT88] 347
Control edges as task interaction: Long and Clarke [LC89] 349
Task clans: McDowell [McD89] 350
Deadlock cycle pruning in resource graphs: Saxena [Sax77] 352
Iterative refinement: Bristow et. al. [BDER79] 353
Linear algebraic methods: Zhou, Yeh, and Ng [ZYN86] 354
10.3.2. Petri net methods. 355
Concise overview. 355
Petri net invariants: Murata, Shenker, and Shatz [MSS89] , and related work. 356
Reduction transforms. 357

Software tool kits. 357
Extended Petri nets: Cheng and Ushijima [CU91] 358
10.3.3. Livelock-induced starvation: Karam and Buhr [KB90] 359
10.3.4. Survey of Ada deadlock detection methods: Cheng [Che91] . . . 359
10.4. Static race detection and concurrency analysis. 361
Concise overview. 362
10.4.1. Non-concurrency: Callahan and Subhlok [CS88, CKS90] 362
10.4.2. Interprocedural analysis: Duesterwald and Soffa [DS91] , [Due91] .364
10.4.3. Congruences: Mercouroff [Mer92] 365
10.4.4. Classification of determinacy: Emrath and Padua [EP88] 367
10.4.5. Balasundaram and Kennedy [BK89] 369
10.5. Formal proof methods. 370
Concise overview. 371
10.5.1. Constrained regular expressions: Avrunin, Dillon, Wileden, and Riddle [ADWR86]. 371
10.5.2. Isolation-based proofs: Dillon [Dil90]. 372
10.5.3. Optimizing the proof process: Apt [Apt83] 373
10.5.4. Model checking. 374
10.6. Summary of static analysis methods. 374
10.7. Static intertask data flow analysis. 375
Concise overview. 377
10.7.1. Classic iterative analysis: Long and Clarke [LC91] 377
10.7.2. Abstract interpretation: Reif and Smolka [RS90] 379
10.7.3. Data dependency graph construction: Moser [Mos90] 380
10.7.4. Task dependence nets: Cheng [Che92] 381
10.8. Dynamic deadlock detection. 381
Concise overview. 382
10.8.1. Survey of dynamic deadlock detection: Knapp [Kna87] 382

10.8.2. Deadlock and interactive debugging: Helmbold and Luckham [HL85] . . . 384
10.8.3. Real-time systems, resource allocation, and dynamic deadlock: Shih and Stankovic [SS90] . . . 385
10.8.4. Cheng, Kasahara and Ushijima [CKU91] . . . 386
10.9. Conclusions . . . 387

11. Future work. . . . 388
11.1. Future work in static deadlock detection. . . . 389
11.2. Additional applications for non-concurrency analysis . . . 391
Debuggers and program understanding tools. . . . 392
Data flow analysis. . . . 392
Optimizing rendezvous. . . . 392
Dead code elimination. . . . 393
Race detection. . . . 393

Appendix A. Boolean matrix notation. . . . 394
A.1. Subtraction, negation, union, and intersection. . . . 395
A.2. Transpose. . . . 397
A.3. Boolean matrix multiplication. . . . 397
A.4. Multiplication by a matrix of all ones. . . . 398

References . . . 399

Vita . . . 413

List of Tables

4.1. Modes of the *Waits* relation. 74

5.1. Pinning analysis - iteration 1. The "Head node?" column for a node n contains "No" if n's partners all do rendezvous only with n and members of $CHT(n)$, "Yes" otherwise. 107

5.2. Pinning analysis - iteration 2. 108

5.3. *CHT* Relation for the program of Figure 5.11 137

5.4. Worst-case time for each form of *CHT* analysis. 161

6.1. Summary of worst-case timing analysis for static deadlock detection using the worklist algorithm. 185

6.2. Summary of worst-case timing analysis for static deadlock detection using Tarjan's path expression algorithm. 185

7.1. Independent variables in the deadlock experiments. 195

7.2. Dependent variables for polynomial time analysis in the deadlock experiments. 195

7.3. Dependent variables for all-states in the deadlock experiments. 196

7.4. Overall experimental results. 200

7.5. Coefficients for the best fit to the deadlock program execution time, in decreasing order of absolute value. Omitted variables were eliminated by the leaps-and-bounds analysis and have a value of zero. 204

7.6. Goodness-of-fit estimators for holdback data in the polynomial analysis. 204

7.7. Number of programs with *CHT* contributions from each refinement. . . 211

7.8. Percentages of programs for which each refinement to *CHT* had synergy or neutral effects on the total *CHT* relation, or prevented spurious deadlock reports. 216

7.9. Programs for which signal counts, accept depth, or both prevented non-top *SACnt* summaries from being propagated, and for which they prevented false alarms. Percentages are relative to the 138 total programs. 217

8.1. SEM critical section structures for Figure 8.3 241

8.2. Time for each component of *CHT* analysis in binary semaphore programs.259

8.3. Complexity of deadlock analysis for binary semaphore programs. 272

9.1. Corresponding synchronization features in Ada and Concurrent C. . . . 274

9.2. Time for each form of *CHT* analysis in Concurrent C programs. 326

9.3. Complexity of deadlock analysis for partitions of Concurrent C programs.336

10.1. Summary of related synchronization and non-concurrency analysis techniques. 376

List of Figures

1.1. The "four-way Sicilian deadlock" [Bur70] 2

1.2. Ada code fragment corresponding to Figure 1.1 3

2.1. Process of developing a polynomial-time deadlock detection algorithm. . 12

2.2. A program fragment in which nodes m and n both represent instances of the same non-synchronization statement. 14

2.3. An Ada program fragment without remote procedure calls, and its sync graph. 22

2.4. An Ada program with a deadlock cycle that violates constraint 4, and its sync graph. 23

2.5. A typical deadlock analyzer. 24

2.6. Propagation of "must happen before" information along control and completor edges. 31

2.7. Program for which the binary CHT_{perf} misses information. 31

2.8. Node split operation used in representing multi-source problems in a heterogeneous cross-product framework. 37

2.9. Node splitting in creating the CLG. 39

4.1. Loop unrolling transform in the basic Ada model. 60

4.2. Sync graph of a program constructed corresponding to the 3-CNF expression $(a + b+ \sim c) \times (a + c+ \sim d)$. 63

4.3. Templates for positive literal and anti-ordering (a), negative literal (b), and ordering (c) tasks. 64

4.4. Ordered pair of negative top nodes. 66

4.5. A program fragment and its sync hypergraph. 70

4.6. Deadlocks involving hyperedges. 74
4.7. Hyperedge cycle location graph (HCLG) transform. 76
4.8. Representing conditional entries in the sync hypergraph. 83
4.9. Co-executability and `else` clauses of selective wait statements. 88
4.10. Representing conditional entry calls in the sync hypergraph. 90
5.1. Completors of node x and completor edges to x. 96
5.2. A simple sync graph. The thick horizontal bars represent the bounds of loop bodies. 104
5.3. A "vanishing partner" of n. 105
5.4. Pinning analysis example. 106
5.5. Pinning analysis example after second iteration. 107
5.6. Pinning analysis where the distinguished begin node b is in $CPreds(n)$. . 108
5.7. Propagation of pinning information to the successors of a node m 114
5.8. A critical section. 119
5.9. A critical section structure. 122
5.10. Example from Lemma 50 130
5.11. A program with nested calls to critical sections. 136
5.12. Spurious deadlock cycle which does not indicate a real "waits-on" relationship. 138
5.13. Deadlock cycle with a critical section exit node as head node. 139
5.14. Deadlock cycle with a call body return node as head node. 140
5.15. Cycles involving exit nodes. 141
5.16. Cycles involving multiple call nodes. 142
5.17. A simplified version of the "dining philosophers" problem with two philosopher tasks. 144
5.18. Critical section call graph for the program of Figure 5.11 146
5.19. Simple critical section structure examples from Lemma 61. 148
5.20. *CHT* example from Theorem 10. 153

5.21. Maximal X set that induces a W that fails to satisfy the preconditions of Theorem 10 155

5.22. A program in which $m \in CHT(m)$ is enforced by three tasks. 156

6.1. Splitting a hypothesized head node. (a) Original HCLG segment. (b) Node a_i split. (c) Node s_d split. 168

6.2. Types of edges in the HCLG, and the classes of edge functions to which they correspond. 170

7.1. Times for polynomial and all-states analysis, versus number of nodes. Scales are log-log for clarity. 201

7.2. Deadlock detection time using the polynomial algorithm, versus each of the 12 independent variables. Scales are log-log for clarity. 203

7.3. Residuals on holdback data for each of the test programs for the best adjusted-r^2 fit found. Short vertical lines are the 90% confidence bounds of each prediction. 205

7.4. Residuals on holdback data for each of the test programs and for each of the 71 best fits found by leaps and bounds. 206

7.5. Time to estimate CHT. The scale is log-log for clarity. 207

7.6. Histogram of the number of CHT refinement iterations, versus the number of programs needing that many iterations. 208

7.7. Number of pairs in calculated versus "perfect" CHT relations. Scale is log-log for clarity. 209

7.8. Histogram of the ratio CHT/CHT_{perf} to number of programs with that ratio. 209

7.9. Histogram of the ratio CHT/CHT_{perf} where CHT was estimated using only the $B4$ refinement, to number of programs. 210

7.10. Histograms of the relative number of bits contributed by each of initialization, pinning, $B4$, remote procedure call, and critical section analyses. 212

7.11. Percent of sync edges pruned, versus number of sync edges in the program. The x axis is logarithmic for clarity. 213
7.12. Histogram of percent of sync edges pruned, versus number of programs. 213
7.13. Synergy effects for each of the four refinements of *CHT* analysis. The x axis is logarithmic for clarity. 215
7.14. Suppression of *SACnt* summary propagation by signaling counts, accept depth, or both. The x axis is logarithmic for clarity. 218
7.15. Histogram of suppression of *SACnt* summary propagation. 218
7.16. Deadlock detection time using all-states analysis, versus each of the 12 measured parameters. 221
7.17. Wampler's hypothesis on the number of reachable states in a program. . 223
8.1. A rendezvous task to simulate the behavior of an initially-clear semaphore.227
8.2. A rendezvous task to simulate the behavior of an initially-set semaphore. 227
8.3. A program fragment which uses initially-clear semaphores to simulate rendezvous synchronization, and its semaphore sync graph. 230
8.4. Process of detecting deadlocks in a program using binary semaphore synchronization, in polynomial time. 234
8.5. A mutex for initially-set semaphore S, with three bodies. 237
9.1. A Concurrent C program showing the use of process and transaction variables. 276
9.2. The parse-form CC sync graph of the Concurrent C program of Figure 9.1 277
9.3. A CC execution graph for an execution of the Concurrent C program of Section 9.1.1 286
9.4. Example of a case in which collapsing all instances of a single process into a single subgraph results in loss of information. 287
9.5. Create sets. All processes with the same coloring are in the same create set. 289
9.6. Non-pseudotransitivity of the *CHTSelf* relation. 306

9.7. Transactor and partner instances of nodes. The three forms of transactor instances are shown. 308
9.8. Some examples of pinning scenarios in Concurrent C. 311
9.9. A weak critical section structure. 316
9.10. An alternative structure that may be used as a critical section. 320
9.11. A Concurrent C program with a spurious cycle. 329
9.12. CDCC sync graph of the partition on the loop in `main` of Figure 9.2 . . 330
A.1. An Ada program fragment and its sync graph. 396

List of Abbreviations

$AInSameType(n)$	is the set of accept-in nodes of the same type as n, if n is a signaling node. If n is not a signaling node, $AInSameType(n)$ is empty.
$Accept?(E, y)$	is a predicate which yields true iff the set (or execution wave) E contains an accept-in node of signal type y.
$B4(n)$	is a pair of sets of nodes (C, S), such that, for any node $m \in C \cup S$, all instances of m complete before any instance of n starts. By abuse of notation, we also use $m \in B4(n)$ to denote the same relation between m and n.
$CallB(c, r)$	is the call body defined by the node pair (c, r), i.e., the set of nodes in the control flow interval (c, r), minus $\{c\}$.
$CallB_{set}$	is a set of call bodies of some critical section structure.
$CallSameInst(n)$	is the set of nodes m such that, on all control paths from n to m that contain only one instance of n, m always calls the same process instance as n.
$CHT(n)$	is a set of nodes that never execute at the same time as n.
$CHT_{perf}(n)$	is the set of all nodes that cannot execute at the same time as n, under the assumptions that all control paths may be taken and any combination of guard conditions is possible.
$CHTSelf(n)$	is a set of nodes m such that, if any instance m_i of m can ever happen together with n_j of n, then $m_i = n_j$.
$CHTSelf_{Perf}(n)$	is the set of all nodes that may be in any $CHTSelf(n)$ set, under the assumption that all control paths are executable.
$Completors(x)$	is the set of the most immediate control ancestors of x which must complete a rendezvous before x executes.
$CompleteRend(x)$	is the set of nodes which may complete a rendezvous with x. If x is a signaling node, $CompleteRend(x)$ is the set of accept-out nodes of the same type; if x is an accept-out node, $CompleteRend(x)$ is the set of signaling nodes of the same type. Otherwise, $CompleteRend(x)$ is empty.
$CPred(x_i)$	is the unique control predecessor instance of node instance x_i.
$CPreds(x)$	is the set of control predecessors of node x.

$CReach(x)$	is the set of nodes n such that there is a control path from n to x, possibly of length zero.
$Crits$	is the set of critical section structures in the program.
$CSB(n,x)$	is the critical section body defined by the node pair (n,x), i.e., the set of nodes in the control flow interval (n,x), minus $\{n\}$.
CSB_{set}	is a set of critical section bodies for some critical section structure.
$CSStruct_{y_1,y_2}$	is the critical section structure defined by signal types y_1 and y_2.
$DeadPath(n)$	is the set of hypothesized head nodes h such that there may be a valid deadlock path from h to n.
Def_v	is a program point where a sync variable v is defined.
$DOM(n)$	is the set of control dominators of node n.
E_C	is the set of control edges of a sync graph.
$E_{Completors}$	is the set of completor pseudo-edges of a sync graph.
E_{MayHT}	is the set of node pairs (x,y) such that x may happen together with y and x is not the same as y.
E_{PCM}	is the set of edges in the process creation multigraph.
E_S	is the set of sync edges of a sync graph.
I	is the identity array.
$InClause(n)$	is, for each accept-in node n, the set of nodes contained within the do clause of n (including n's accept-out node). If n is not an accept-in node, then $InClause(n)$ is empty.
$L(x)$	is the loop body most immediately containing node x.
$LE(x)$	is the end node of the loop body most immediately containing node x.
$Loops$	is the set of loops in the program.
$LS(x)$	is the begin node of the loop body most immediately containing node x.
$MayCall(y)$	is the set of signaling nodes that may call signals of type y.
$MustCall(y)$	is the set of signaling nodes that must call signals of type y.
$MustClear(n)$	is a set of semaphores that must be clear whenever n executes.
$MustSet(n)$	is a set of semaphores that must be set whenever n executes.
$MutexB(n,x)$	is the mutex body with entry node n and exit node x.

N	is the set of nodes in a sync graph.
N_{Accept}	is the set of accept nodes in the sync graph.
$N_{Accept}(y)$	is the set of nodes in a sync graph that accept signals of type y.
$N_{Bracket}(S)$	is the set of nodes which are bracket nodes of mutex or critical section structure S.
N_{Create}	is the set of create nodes for the program.
$NextStates((E, A))$	is a set of execution waves which can be produced from the wave (E, A) by completing one abstract tasking or synchronization event.
N_{Head}	is the set of possible deadlock head nodes in the program (i.e., $N - N_{NonHead}$).
$NodeInsts$	is the set of node instances that may ever be created by the program.
$N_{NonHead}$	is a set of nodes that cannot be the head nodes of a valid deadlock cycle.
N_{PCM}	is the set of nodes in the process creation multigraph.
N_{Signal}	is the set of signaling nodes in the sync graph.
$NodeType(n)$	is the node type of n (i.e., signaling, accept, etc.)
ONE	is an array of all ones.
ONE_{Vector}	is a vector of all ones.
$PartInst(n_i)$	is the unique node instance which is a partner instance of n_i.
$Partners(n)$	is the set of successors of those nodes which can rendezvous with the predecessors of n. At least one member of $Partners(n)$ must appear on the execution wave whenever n starts.
$Pins(n)$	is the set containing all nodes if n pins its partners, $\emptyset$ if not.
$Partners_{other}(n)$	is the set of partners of n which may transact with process instances other than that containing the instance of n to which it is partnered.
$Partners_{Pinned}(n)$	is the set of partners n pins, or $\emptyset$ if n does not pin its partners.
$POSTDOM(n)$	is the set of control postdominators of node n.
$Process(n)$	is the process containing node n.
$ProcInst(n_i)$	is the process instance containing node instance n_i.
$ProcInsts$	is the set of process instances that may ever be created by the program.
$ProcRef(n_i)$	is the process instance referred to by instance i of signaling node n.

$Procs$	is the set of all processes in the program.
$SACnt_h(n)$	is a lower-bound count of the number of signaling head nodes of each signal type, and an upper-bound count on the scope depth of accept-in head nodes of each signal type, on all deadlock paths from h to n.
$SigCnt_h(n)$	is a lower-bound count of the number of signaling head nodes of each signal type, on all deadlock paths from h to n.
$Sigs$	is the set of signal names (of the form `task.entry`) in the program.
$Scope(n, y)$	is the scope depth of node n with respect to signal type y.
$Scope_{set}(E, y)$	is the sum, for all nodes n in the set (or execution wave) E, of the scope depth of n with respect to signal type y.
$SigN_{set}(E, y)$	is the number of signaling nodes of signal type y in the set (or execution wave) E.
$Sisters(n)$	is the reflexive set of sisters of n, i.e., accept-in nodes in the same conditional entry construct, if n is an accept-in node. Otherwise, $Sisters(n)$ is empty.
$Starters(n)$	is the set of most immediate control descendants of n which can start a rendezvous.
$Starters_{other}(n)$	is the set of nodes which are starters of n, but which may transact with process instances other than the one which transacts with p.
$StartRendSig(n)$	is the set of signaling nodes that can start a rendezvous with n If n is an accept-in node, $StartRendSig(n)$ is the set of signaling nodes of the same signal type as n. If n is not an accept-in node, then $StartRendSig(n)$ is empty.
$StrictInt(G, \mathcal{P})$	is the set of strict intervals on graph G as defined by the partition $\mathcal{P}$ of the nodes of G. (Section 5.1.3)
$SigType(n)$	is the signal type of n.
$SyncVars$	is the set of sync variables in the program.
$SyncVarInsts$	is the set of sync variable instances in all process instances created in the execution of the program.
$TakeOff(n)$	is the set of nodes which can take n off the execution wave by starting or completing a rendezvous.
$Task(n)$	is the set of nodes in the task to which node n belongs.
$Tasks$	is the set of all tasks in the program.
$TaskFrontNodes(t)$	is the set of nodes in task t which may be placed on the execution wave at the start of the program.

$Trans(n)$	is a set of nodes such that an instance of some node m from $Trans(n)$ must complete at the same time as any instance of n.
$TransInst(n_i)$	is a node instance which is known to complete at the same time node instance n_i does.
$UngSis?(E, y)$	is a predicate which yields true iff the set (or execution wave) E contains an accept-in node of type y, or which has an unguarded sister of type y.
$UnionB4((C, S))$	is the set of nodes $C \cup S$.
Use_v	is a program point where a sync variable v is used.
$Waits(E, x, y)$	is a relation such that, in execution wave E, node x cannot complete execution, but x could complete execution if y were on E, and y is a control successor of a node now on E. (I.e., "x waits on y in E.")
$Widen(CHT, S, T)$	is an array which is a superset of the CHT array. The refinement of CHT is made under the assumption that, for any node n, $S(n)$ is a set of nodes, such that some member of $S(n)$ must happen together with n (if $S(n)$ is not empty). T is a set of nodes such that, for each node n in the program, n must execute together with members of T. Typically, T will be either the nodes of a single task or the total set of nodes in the program.

Chapter 1

Overview.

1.1 The deadlock problem.

> **Deadlock:** In computing, an error condition in which processing cannot continue because each of two elements of the process is waiting for an action from the other. Synonymous with *deadly embrace.* (Dennis Longley, [Lon86].)

An example of a deadlock, not from the computer domain, is shown in Figure 1.1. Each of the four cars in the figure is waiting for the car in front of it to move out of the way. For reasons of honor, none is willing to break the deadlock by backing up. Thus, none of the cars is making any progress. This example can be translated directly to computer programs; Figure 1.2 shows an Ada code fragment with the same behavior.

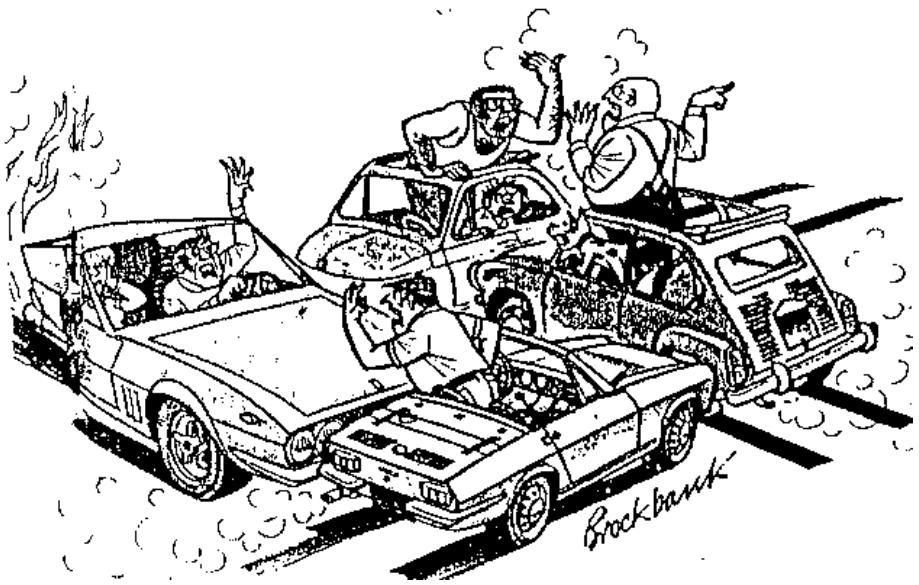

Figure 1.1: The "four-way Sicilian deadlock" [Bur70].

1.1.1 Tasking and synchronization.

Each of the drivers in Figure 1.1 corresponds to a *task* in the program of Figure 1.2. Tasks define blocks of code which may execute concurrently. In the two figures, each task has only one *task instance*, or executing copy, and all tasks execute concurrently throughout the running time of the program. Other models of tasking exist, and may behave differently.

The programmer may introduce *synchronization* between the tasks. In general,

```
task body Car1 is begin
  Car4.moved;            -- Wait for car 4 to move
  accept moved;          -- Signal that car 1 has moved
end Car1;

task body Car2 is begin
  Car1.moved;            -- Wait for car 1 to move
  accept moved;          -- Signal that car 2 has moved
end Car2;

task body Car3 is begin
  Car2.moved;            -- Wait for car 2 to move
  accept moved;          -- Signal that car 3 has moved
end Car3;

task body Car4 is begin
  Car3.moved;            -- Wait for car 3 to move
  accept moved;          -- Signal that car 4 has moved
end Car2;
```

Figure 1.2: Ada code fragment corresponding to Figure 1.1.

synchronization causes tasks to wait, at various points in their execution, for other tasks to take some action. We assume that synchronization is specified by *synchronization statements* in the bodies of the tasks. The *synchronization model* of the programming language defines the semantics of these statements. Note that the synchronization model of a language is separate from its tasking model.

In Figure 1.2, we use a *rendezvous* model of synchronization. In the rendezvous model, the "signaling" statements (of the form `<task name>.<signal name>`) and "accepting" statements (of the form `accept <signal name>`) mutually wait on each other. For instance, the `accept moved` statement in task `Car1` cannot complete execution until some other task has started to execute a `Car1.moved` statement, and vice versa. (In the example, task `Car2` is the only task that has such a statement. This is not a necessary condition on the structure of the program.)

1.1.2 The importance of detecting possible deadlocks.

Deadlocks may occur in any concurrent or parallel programming language in which tasks synchronize with each other during execution. Deadlock is an *anomaly*, or a failure of the program to function correctly. (Other, more widely known anomalies include potentially uninitialized variables and non-terminating loops.)

Good programming practice demands that the programmer must eliminate anomalies wherever they may occur. Safety-critical and mission-critical applications have even more stringent needs for anomaly-free software; a mistake in such a system may cause damage to the environment, the loss of large sums of money, or even danger to human life. For these applications, and even for more mundane ones, automatic detection of any possible anomaly is clearly a worthwhile research goal.

Many researchers have sought to automate the detection of synchronization anomalies, including deadlocks. However, deadlocks have resisted efficient automatic detection. Programmers have thus been forced to rely on difficult, unsafe manual test set generation to find deadlocks in their code. We believe there is a better alternative.

Our main research goal.

Our research goal is to develop practical program analysis systems, which can be used to prove that concurrent programs are free of deadlock anomalies. If our analyses detect potential deadlocks, we should aid the programmer in identifying and removing the deadlocks from the program. Our broad goal is to construct a general framework for deadlock detection, which can be specialized to a wide variety of synchronization and tasking models.

1.1.3 A short comparison with existing methods.

As we show in Chapter 10, most of the past efforts in deadlock detection have relied on dynamic testing, state enumeration methods, or formal proof methods. Each of these methods has disadvantages that make it impractical for large systems. We will describe each of these methods briefly in turn.

Dynamic testing.

Dynamic testing, while a valuable diagnostic tool, cannot prove that a program is free from deadlocks; the possibility always exists that the test set simply does not expose the deadlock. Dynamic testing is therefore *unsafe*; programs are judged either "probably all

right" or "definitely incorrect." We feel that this is inadequate for critical applications; programs must be *proved* free of the tested anomalies.

Static analysis.

We must therefore use some method of *static analysis* [Hec77], or formal proofs of correctness, to guarantee safety. Static analysis finds specific properties of the program which are true in *all* possible executions; it thus has the unique ability to prove that an arbitrary program is free from deadlock. The previous work in static analysis falls into two categories: state enumeration and formal proof methods.

State enumeration.

During the execution of a program, the *concurrency state* of the program is a set of synchronization statements, one from each executing task instance, which may be reached in execution without executing any intervening synchronization statement in the same task instance. (The end of the task may also be taken to be a synchronization statement.) The method of *state enumeration* is a form of static analysis. The analyzer lists a set of concurrency states, including at least all the possible concurrency states of the program. Each state is checked for deadlock.

The program must always be in one of the enumerated states during any execution. State enumeration methods usually make the *static analysis assumption*: that all branches in the program may be taken. Thus, state enumeration may also generate concurrency states that will never actually occur.

If the program can deadlock, then state enumeration will detect it. State enumeration is thus safe, in the sense we require; it can be used to prove that programs are free from deadlock.

The problem with state enumeration is that the number of concurrency states for even a simple program may be enormous, even unbounded in the number of lines of source code (assuming that the tasking model permits an unbounded number of instances of the same task to be created.) Thus, state enumeration may fail to produce

any result in a reasonable time. In Chapter 7, we show experimental evidence that state enumeration often "explodes combinatorially" when large programs are analyzed.

Formal proof methods.

Finally, we consider formal proof methods, which are actually a sophisticated form of static analysis. The idea is to construct a mathematical proof that the program cannot enter a deadlock state.

Formal proof methods are safe, and very precise, but they cannot be fully automated with current technology. They are far too labor-intensive for all but those purposes in which safety is much more critical than development cost. They are also sharply limited; at this writing, no software system even as complex as a compiler has ever been formally proven correct.

Concurrent programs pose special difficulties for the current formal proof technology. The programmer must prove freedom from deadlock for *each* of the program's concurrency states, either by proving that the state does not represent a deadlock, or that it cannot be reached in any program execution. Effectively, the formal proofs are built on top of state enumerations. This makes formal proof methods even more time-consuming.

Last, but certainly not least, the entire proof may have to be re-done from scratch if the program is changed. We can see that formal proof methods, while appealing, are not yet practical for large software systems.

1.1.4 Our requirements.

To avoid the problems inherent in other methods, we impose the following requirements upon our systems for deadlock detection:

- *Safety.* Wherever a deadlock exists in the program, we report that it may exist.
- *Efficiency.* Our deadlock detection algorithm should avoid combinatorial explosions in detection time. Specifically, we require that our analysis complete in a time which is polynomially related to the number of source lines of the program

under test. We further require that our algorithm have a modest execution time in practice, though it need not take less absolute time than other methods in all cases.

- *Ease of use.* Our method should be completely automatic, and should provide meaningful information to the programmer about any suspected deadlocks.
- *Effectiveness.* Our method should have as few "false alarms" as is possible, given that the other constraints are satisfied.

1.2 Our approach: Detection of constrained deadlock cycles.

In the most general terms, our approach is to detect possible cyclic "waits-on" relationships among the tasks in the program. The detection amounts to a constrained search on a graph representation of the program's tasking and synchronization behavior. The potential existence of deadlock cycles is *detected* by this search, but the cycles themselves are not *enumerated*.

While searching for deadlock cycles, we prune away as many paths as possible that cannot be part of deadlock cycles. Such paths are said to violate *constraints* on the deadlock cycles. An example of a constraint would be that the deadlocked statements represented by the cycle must be able to happen together. We use information on which statements *can* happen together, to prune paths which contain statements that *never* happen together.

Prior to the cycle search, we solve static analysis problems to gather pruning information. During the search, we retain approximate pruning information, which holds for all deadlock paths between two nodes in the graph. We require that the approximate pruning information be *safe*, i.e., weaker than the exact information. Also, the cycle search must complete in polynomial time; this limits the detail of the pruning information we can retain.

As long as the preliminary information gathering and the cycle search are polynomial, the entire deadlock detection process is polynomial. The graph model of the program, the set of constraints, the information gathering algorithms, and the cycle

detection algorithm must be specialized for a particular synchronization type. However, we demonstrate, by example, that our general model can be specialized to a wide variety of synchronizations.

1.3 Main contributions.

We consider these to be the central contributions of this research:

- A general, informal framework for developing static deadlock detection systems. In effect, we present here a manual on how to develop a deadlock detection algorithm.
- Specific algorithms for deadlock analysis in three widely different tasking and synchronization models: (a) a large subset of Ada, (b) binary semaphores, and (c) a subset of Concurrent C. These synchronization models have vastly different characteristics; we thus claim that our general framework applies to a wide range of synchronization models.
- *Non-concurrency analysis*, a methodology for finding pairs of statements which can never happen together. Our non-concurrency analysis framework can incorporate, and improve upon, the results of other researchers in the field [CS88, DS91]. We show experimental evidence that non-concurrency analysis is crucial for effective deadlock detection.

  We also claim that non-concurrency analysis has important practical applications other than deadlock detection. These include program understanding tools, race anomaly detection, dead code elimination, optimization of rendezvous, and static data flow analysis of concurrent programs, as detailed in Section 11.2.
- Lattice framework formulations for *multi-source problems*, a useful set of static analysis problems in which information at each node arises from more than one source. In such problems, edges in the program graph are partitioned naturally into classes (e.g., control and synchronization), and information propagates separately along edges of each class. The "must happen before" analyses of

[CS88, DS91], are multi-source problems, which have not been formulated as lattice frameworks prior to this work; our multi-source framework formulations can also accommodate new data flow problems we introduce.

- Experimental investigation and validation of deadlock detection for the Ada subset. Our experiments show that deadlock detection can be efficient and effective, as we require. They also show, in detail, the effectiveness of each type of pruning information in preventing false alarms. Finally, our empirical work has identified important areas for future research.

We present Chapter 11 as an outline of the needed continuing work in this area. For the present, we feel that we have made important progress in developing static, polynomial-time deadlock detection as a practical tool of the software engineer.

1.4 Guide to the thesis.

This thesis is intended to do four things. First, it presents an intuitive view of the process of polynomial-time static deadlock detection. Second, it supports our claims for the correctness, effectiveness, and generality of polynomial-time static deadlock detection. Third, it provides sufficient information for our work to be duplicated by other researchers. Fourth, it reviews the previous work in the field, and relates our work to it. The more casual reader will not necessarily need all this information.

For an initial, intuitive overview of the thesis, we suggest reading about the general framework, the basic Ada model, and the experimental work on it. This material is contained in Chapters 1, 2, Section 4.2, and Chapters 5 (subsection headers only), 7 and 11. [MR93] presents a reasonable synopsis of Chapter 5.

Chapters 8 and 9 provide evidence that the deadlock detection framework can be generalized to synchronization and tasking primitives which are quite different from the Ada model; they represent work in progress. Chapter 10 describes previous research in detail and relates our work to it; for a concise overview of the related work, we recommend reading only the subsection headers and Table 10.1.

Chapter 2

General framework.

2.1 Introduction.

This chapter presents the informal general framework we will use to develop polynomial-time deadlock detection algorithms. We will present both the process of development and a skeletal version of the algorithm.

2.1.1 Terminology and notation.

Throughout this thesis, we use the following graph terminology.

- If there is an edge (m, n) in a graph G, then m is a *predecessor* of n, and n is a *successor* of m, in G.
- If there is a control path $(m, ..., n)$ of length ≥ 1 in G, then m is an *ancestor* of n, and n is a *descendant* of m, in G.
- Instance i of node n or task T are denoted n_i and T_i, respectively.

2.1.2 A short sketch of the process.

Figure 2.1 illustrates the process of developing a deadlock detection algorithm, and also outlines this chapter. Given a synchronization and tasking paradigm, we first gain a theoretical understanding of its behavior. To this end, we create a *sync graph* representation of programs, and an abstract program execution model based on the sync graph representation. Once the graph and execution models have been created, we can use them to develop an algorithm to enumerate concurrency states exhaustively (assuming that the number of abstract states is finite). This algorithm will be used as a correctness check and benchmark for the implementation of the polynomial algorithm.

Deadlocks correspond to cycles in the sync graph, but not all cycles correspond to deadlocks. Therefore, we specialize a set of general constraints on deadlock cycles to the execution model. A simple example of such a constraint is that all the deadlocked statements must be able to execute concurrently. Given these constraints, we identify the types of information that are useful in pruning away the cycles that do not correspond to deadlock.

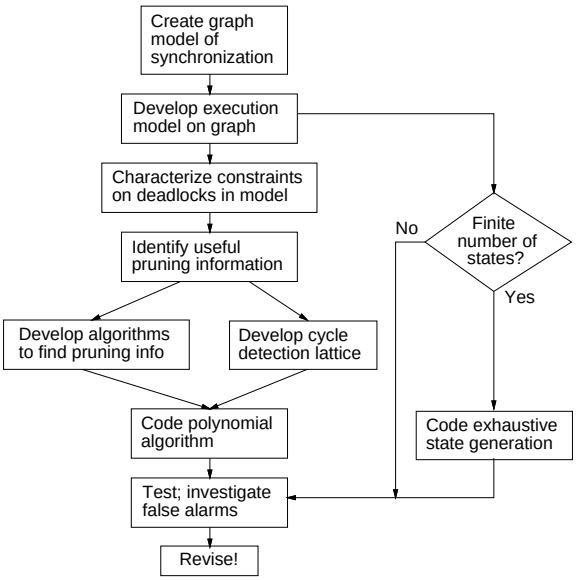

Figure 2.1: Process of developing a polynomial-time deadlock detection algorithm.

Once this pruning information has been identified, we develop static analysis algorithms to estimate it, given the sync graph and other information obtainable during program parsing. These algorithms are required to have provably polynomial worst-case completion times.

We also develop a lattice framework for finding deadlock cycles in the sync graph. The framework finds paths in the sync graph that could be part of deadlock cycles, and maintains conservative estimates of pruning information on these deadlock paths. As for the constraint information algorithms, we require that the framework provably terminate in polynomial time.

When the cycle detection lattice and pruning information algorithms have been identified, they are coded as a complete algorithm, and tested against the exhaustive state generation on realistic test data. The purpose of the testing is three-fold: to verify that the polynomial algorithm's implementation is conservative, to characterize the false-alarm rate and execution time of the polynomial algorithm versus state generation, and to identify any pruning information that does not eliminate many false alarms, and thus can be dispensed with. As an example of the last, the nesting depth of Ada `accept` statements was found to have little practical value in deadlock detection (Section 7.9).

If testing reveals accuracy problems in the polynomial algorithm, or if some pruning information is found not to be useful, we revise the algorithm. We may also revise the sync graph structure or execution model, if this will yield more information. The revision process continues until the polynomial implementation has satisfactory performance.

2.2 Theoretical work.

This section describes the theoretical steps necessary to develop a deadlock analyzer. These must be done before algorithm development can begin.

The first step in developing a deadlock detection algorithm is to develop a model of the execution of programs. The execution model consists of a graph representation of programs, and an execution model that generates the *concurrency states* of the program, i.e., the sets of statements in the program that may execute concurrently. The execution model should be an abstraction of program execution, producing at least the concurrency states that the source program itself can have.

Given the execution model, we then identify a set of constraints which must hold on any valid deadlock cycle in the graph representation. The constraints, specialized for the synchronization primitives we are analyzing, are used to prune spurious deadlock cycles.

2.2.1 Graph model of synchronization structure.

The sync graph model of a particular program is a triple (N, E_C, E_S). N is the set of nodes of the graph. Each node is akin to a basic block [Hec77], ending with a synchronization statement or tasking event rather than a branch. Node n_s represents all instances of the synchronization statement s, and all instances of non-synchronization statements or tasking events on paths ending at s that do not include other synchronization.

Note that different instances of a single non-synchronization statement may be represented by more than one node, as in the example in Figure 2.2. Here, node b represents

the beginning of task T. Node m represents all control paths from b to the `accept sig_m` statement, including the latter but not including b. Node n similarly represents all control paths from b to the `accept sig_n` statement. Thus, both m and n represent instances of code block C and statement d.

```
task T is begin      -- Node b
  boolean test;
  < Code block C >
  if test then       -- Statement d
    accept sig_m;    -- Node m
  else
    accept sig_n;    -- Node n
  end if;
end T;               -- Node e
```

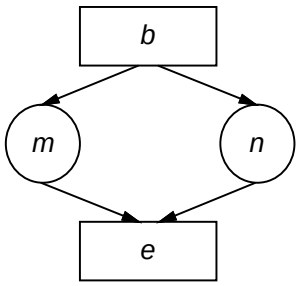

Figure 2.2: A program fragment in which nodes m and n both represent instances of the same non-synchronization statement.

It should be evident that each concurrency state in the program can be represented by some set of nodes which, in turn, represent statement instances that can execute simultaneously. Thus, the set of concurrency states of the program can be represented by a powerset of nodes; the abstract concurrency history [Tay83b] of a program thus has a finite number of concurrency states.

Nodes may be of different types, corresponding to the class of statements or events they represent. In the models we have studied, nodes have been used to represent the following types of statements or events.

- The beginning and end of tasks and task instances, or of the entire program.
- Synchronization statements.
- The beginning and end of remote procedure execution.
- Dynamic task creation statements.

E_C is the set of directed *control edges* in the graph model. E_C contains the edge (m, n) if there is a control path from m to n in the program, which contains no synchronization statements or events other than m and n. Control edges (m, b_T) may also represent task creation; here, m would create an instance of task T, whose begin node is b_T.

E_S is the set of *sync edges*. Sync edges represent an "enable to complete" relationship between nodes. For a particular model, sync edges may be directed or undirected; they may have two ends, or they may be *hyperedges* with more than two endpoints. The E_S set for a program is initially constructed based only upon information found during parsing (e.g., signal types). Later refinements (Section 2.3.3) eliminate sync edges that cannot correspond to actual synchronization, because of statement ordering.

A directed sync edge (m, n) represents the relation, "An instance of m may enable an instance of n to complete execution." For example, m may set a semaphore that n awaits. Whether m ever actually *does* set a semaphore that n waits upon might not be known at the time the sync graph is constructed. This may be for any of several reasons, e.g.: m may never happen before n; the semaphore that m sets might not be identical to the one that n awaits (if semaphores are referenced through pointer variables); or the semaphore that m sets may always be modified before n starts execution. For these reasons, sync edges represent a superset of the possible synchronizations between nodes.

If m and n may mutually enable each others' completion (as in rendezvous), then we may simplify the model by using undirected sync edges $\{m, n\}$ instead. For more complex forms of synchronization, such as barriers or remote procedure calls, E_S might be a set of hyperedges. In the Ada remote procedure call, for example, a hyperedge (s, a_i, a_o) represents the relation, "Signaling node s may enable accept-in node a_i to complete. If it does, then only accept-out node a_o will enable s to complete."

2.2.2 Execution model and state generation.

An *execution wave* is an abstraction of a concurrency state and program store for the program. The abstract program store represents whether, and how, the program may proceed to the next state. We can envision the program execution advancing in a wave-like fashion through the sync graph, hence the name.

We will denote abstract execution waves as a pair (E, A), where E is an abstract concurrency state (i.e, an ordered list of nodes) and A is an abstract program store.

Since nodes can be repeated in E (representing concurrent instances of a single statement, in different instances of the same task), we will refer to the positions of E as *entries*. The i'th entry of abstract concurrency state E is denoted $E[i]$.

The abstract store A determines what execution waves can be generated next from (E, A). A typical example of A would be a map from binary semaphore variables to boolean values. In some other synchronization types, such as rendezvous, we can determine from E alone which states might be generated next; there, the abstract program store would be unneeded for the abstract model, and thus empty.

Abstraction of the program state.

In this section, we provide intuition on how the execution wave represents a set of abstract program states. We will not use the full formal machinery of *abstract interpretation* [CC77], but rather present this explanation so that it should indicate how an abstract interpretation model might be developed.

The concrete concurrency state of a program may be represented as a list of statements that are executing currently. The concrete program store is, likewise, a map from variables (including those affecting concurrency) to their current values.

We can map a concrete concurrency state to a *set* of abstract execution waves. In each execution wave (E, A), E is a set of sync graph nodes, one per task instance, which correspond to synchronization statements that can be reached from the concrete concurrency state, without executing any synchronization. Since tasks may have more than one instance, nodes may be repeated in E. A represents a possible value of the abstract store when the nodes of E are reached. Similarly, we may map an abstract execution wave (E, A) to the set of concrete program states which abstract to (E, A).

Since we do not make any assumptions about the relative execution speeds of instructions or communication through shared memory, programs in the same concrete state may have different concrete execution histories, due to races. Thus, the concrete program execution may act nondeterministically to choose different branches, given a single initial concrete state. This nondeterminism is carried through to the abstract

model, requiring us to map a single concrete state to multiple abstract execution waves.

Execution model.

We define a *transition function*, *NextStates*, that maps an execution wave to a set of possible next execution waves. *NextStates* simulates execution in the abstract program.

Sufficient conditions for transitions.

A sufficient, though not necessary, condition for (E', A') to be in $NextStates((E, A))$ is that it must be possible to transform (E, A) to (E', A') by completing a single (abstract) synchronization event. More formally, $(E', A') \in NextStates((E, A))$ if there exist concrete states $\mathcal{S}$ and $\mathcal{S}'$ such that:

- $\mathcal{S}$ abstracts to (E, A);
- $\mathcal{S}'$ abstracts to (E', A');
- $\mathcal{S}$ has $\mathcal{S}'$ as a descendant state;
- Exactly one synchronization or tasking operation occurs between $\mathcal{S}$ and $\mathcal{S}'$.

Necessary conditions for transitions.

For (E', A') to be in $NextStates((E, A))$, there must exist some program P, whose sync graph is also (N, E_C, E_S), and which has concrete states S and S' that satisfy the sufficient conditions above.

Note that not all possible functions *NextStates* meet the necessary conditions. For Figure 2.3, suppose we have an execution wave $((r, u), \emptyset)$, and that we have defined $NextStates(((r, u), \emptyset)) = \{((s, u), \emptyset), ((s, v), \emptyset)\}$. Obviously, $((s, v), \emptyset)$ satisfies the sufficient condition for at least one program (i.e., the program of Figure 2.3). It therefore satisfies the necessary condition, so it must be included. However, $((s, u), \emptyset)$ could only follow $((r, u), \emptyset)$ if r completed execution without a rendezvous. Such a resultant execution wave does not satisfy the sufficiency condition on *NextStates* for any program with the same sync graph. It therefore does not satisfy the necessary conditions, so we must explicitly exclude $((s, u), \emptyset)$ from $NextStates(((r, u), \emptyset))$.

Example.

As an example, consider binary semaphores.[1] Suppose that, for execution wave (E, A), $A(X) = \mathit{True}$, $E[i] = m$, m represents paths ending with a $p(X)$ statement, and $(m, n) \in E_C$. Then $(E', A') \in \mathit{NextStates}((E, A))$, where $E'[i] = n$, $A'(X) = \mathit{False}$, $\forall j \neq i : E'[j] = E[j]$, and $\forall Y \neq X : A'(Y) = A(Y)$.

States of entries.

In an execution wave (E, A), the entries of E may be in one of two states. $E[i]$ is in state READY if it can complete execution with abstract store A, under the assumption that all control paths represented by $E[i]$ are executable [Ban78]. $E[i]$ is in state WAITING otherwise.[2]

It is sometimes also convenient to think of a *firing function* Φ, which maps any execution wave (E, A) to a vector of states, one for each entry in E. $\Phi((E, A))[i]$ is READY if $E[i]$ can complete execution while executing along with the other entries of E, and with abstract store A.

If only one instance of any synchronization statement in the program can execute at any given time, then we may meaningfully speak of the states of *nodes*, as well as execution wave entries. The nodes which are not in E then can be said to have a state OFF-WAVE. We will use this concept in the models for Ada (Section 4.2.3) and binary semaphores (Section 8.2.3).

State enumeration algorithm.

Given the transition function, it is simple to define an algorithm to enumerate the abstract concurrency states of a program, assuming that there are finitely many of them. Such algorithms are useful in practice for verifying and benchmarking the implementation of polynomial-time deadlock detection.

---

[1] The reader unfamiliar with binary semaphores may find a definition in Section 8.1, on page 225.

[2] In our model of Ada remote procedure calls, we include an ENGAGED state for signaling nodes. This is a shorthand notation; the abstract store and abstract state together indicate whether s can proceed.

Since each program has a single initial concrete state $\mathcal{S}$, we find the corresponding execution wave $S = (E, A)$ such that $\mathcal{S}$ abstracts to $\{S\}$. We extend the domain of *NextStates* to sets of states, forming the function $NextStates_{set}$ in the obvious way. We define S_{Trans} as the transitive closure of $NextStates_{set}$ on $\{S\}$, i.e., the set of reachable abstract states. If there are only finitely many abstract program states, then the process must finish in finite time.

In some cases, such as dynamic task creation, there may not be finitely many abstract concurrency states. If that is true, then it is not possible to enumerate the concurrency states algorithmically, unless we know that the program generates only a finite number of states (regardless of whether it terminates).

2.2.3 A structural definition of deadlock and starvation.

Section 2.2.2 shows how to enumerate the set S_{Trans} of abstract execution states of a program. Any abstract state S in S_{Trans}, such that $NextStates(S) = \emptyset$ and S does not represent the end of the program, represents a possible infinite wait anomaly, since such a state S would have no successor state. Infinite wait anomalies are caused by deadlocks and/or starvation, as we shall see in this section.

Note that livelocks represent execution waves S such that each wave in $NextStates_{set}(S)$ has a successor, but that the wave representing the program's termination state is not in S_{Trans}. Thus, livelock does not fit this definition of an infinite wait anomaly.

We say that node m is *connected* to node n if:

- m and n may be in different task instances; and,
- m has a control successor m'; and,
- n represents a synchronization or tasking statement that might be able proceed if (and as soon as) m' could be reached in control flow.

In terms of the sync graph structure, there is a path from m to n which includes some control edges in m's task, and a sync edge from a control successor m' of m to n, and no other sync edges. m is *transitively connected* to n if m is connected to n, or, recursively,

if m is connected to some node p that is transitively connected to n.

Given an anomalous execution wave (E, A), we say that an entry $E[i]$ (other than the program end node) is *starved* if no other entry $E[j]$, and no control successor of any other entry in E, can enable $E[i]$ to complete. In structural terms, there are no sync edges connecting $E[i]$ to any entry on E or any of its control successors.

Similarly, entry $E[i]$ is a member of a *deadlock cycle* D if $E[i]$ is transitively connected to each member of D, and vice versa. Thus, some successor of any member of D could enable $E[i]$ to proceed, and vice versa, assuming that the successors could be reached in control flow.

Lemma 1 below shows that all infinite wait anomalies must contain starved nodes or deadlock cycles, or both.

Lemma 1 *Let $S = (E, A)$ be an execution wave representing an infinite wait anomaly. All entries in E (other than the program end node) are starved nodes, or are members of a deadlock, or are transitively connected to starved nodes or deadlocks.*

Proof: Suppose the contrary, i.e., that some entry $E[i]$ is not in a deadlock, is not starved, and is not transitively connected to another node that is. Since $E[i]$ is not starved, there must be a successor s of some other entry $E[j]$ which would enable $E[i]$ to proceed if s could execute. $E[j]$ likewise must not be deadlocked, starved, or transitively connected to other nodes that are.

Continuing this argument, we can trace along some chain of nodes that are transitively connected to $E[i]$ as far as possible. Either this chain has a unique first entry or it does not. If it has a beginning entry $E[h]$, then $E[h]$ must be starved, and so $E[i]$ would be transitively connected to a starved node, violating our assumption.

Now consider the situation when the chain connected to $E[i]$ has no beginning entry. E is of finite length, since the execution model only generates execution waves with finitely many nodes. Then the chain connected to $E[i]$ must have a cycle; such a cycle must be a deadlock cycle. This also violates our assumption. So E must contain a deadlock cycle or a starved entry. □

Artifactual deadlock cycles.

Some deadlock cycles represented by a set of entries D may be *artifactual*, i.e., there is some more fundamental anomaly than D which causes the infinite wait. The symptom of this is that the entries in the deadlock cycle are transitively connected to a set D' of entries outside the cycle, D' contains an anomaly, and none of the entries in D' are transitively connected to D. Thus, if the entries in D' could complete, then D might be enabled for completion, but not vice versa.

Artifactual deadlocks have been a particular problem in analysis of critical sections in Ada (Section 5.1.4). Our strategy in handling them has been to try to identify and prune deadlock cycles which *must* be artifactual.

2.2.4 Constraints on deadlock cycles.

We have identified a set of four general constraints on deadlock cycles. These constraints can be specialized to particular models of tasking and synchronization. They are extremely useful in helping the algorithm designer to identify information that prunes spurious deadlock cycles.

Four constraints.

In this section, we identify four general constraints on deadlock cycles. We use the Ada program fragment and sync graph of Figure 2.3 as a continuing example here, and throughout this document.

A set of nodes D may denote a deadlock cycle *only if* all four of the following constraints hold for D:

1. *Cycle Shape.* The cycle enters each task through a sync edge at a node in D, traverses at least one control edge within the task, and exits the task. The node in D at which the task is entered is called a *head node* of the deadlock cycle; D is the set of head nodes of the cycle. The node at which the task is exited is called a *tail node,* and is in D iff it is also a head node.

```
task body t1 is
begin
  t2.sig1;     -- (r)
  accept sig2; -- (s)
  t2.sig1;     -- (t)
end t1;

task body t2 is
begin
  accept sig1; -- (u)
  t1.sig2;     -- (v)
  accept sig1; -- (w)
end t2;
```

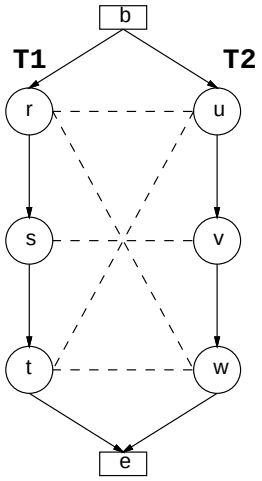

Figure 2.3: An Ada program fragment without remote procedure calls, and its sync graph.

The cycle may enter the task more than once *only if* more than one instance of the task may execute simultaneously.

In Figure 2.3, cycle (r, s, t, u, r) violates constraint 1, because it includes node u from task `t2`, but includes no control edges in task `t2`.

2. *Head Nodes Cannot Proceed.* The head nodes of D must all be of a node type that must wait for synchronization before completing. Furthermore, there must exist some abstract store A, such that the program can have store A while the nodes in D execute, and such that no nodes in D are in state READY with store A.

   In Figure 2.3, cycle (r, s, t, u, v, w, r) violates constraint 2, because the cycle has r and u as head nodes. These nodes can rendezvous with each other, so the program cannot deadlock.

3. *Head Nodes Can Happen Together.* The nodes of D must represent statement instances that can all execute simultaneously.

   In Figure 2.3, cycle (r, s, v, w, r) violates constraint 3, because the head nodes r and v can never happen together.

4. *Nodes Outside Deadlock Don't Break Infinite Wait.* There must exist some execution wave (E, A) such that no nodes in $E - D$ are in state READY.

In Figure 2.4, cycle (r, s, t, u, r) violates constraint 4, because t must always happen together with w, and t can rendezvous with w. Thus, node w, while outside the deadlock cycle, always breaks the deadlock.

```
task body T1 is begin
  T2.u;          -- (r)
  T2.tv;         -- (s)
end T1;

task body T2 is begin
  accept tv;     -- (t)
  accept u;      -- (u)
  accept tv;     -- (v)
end T2;

task body T3 is begin
  T2.tv;         -- (w)
end T3;
```

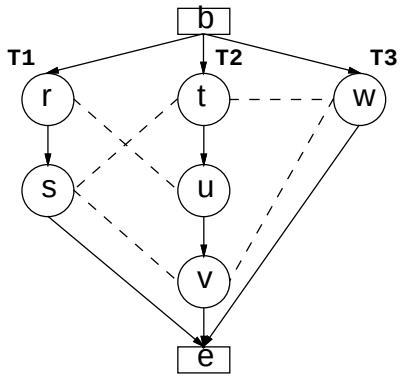

Figure 2.4: An Ada program with a deadlock cycle that violates constraint 4, and its sync graph.

Enforcing the constraints in cycle detection.

Constraint 1 is usually enforced in cycle detection by transforming the sync graph into a *cycle location graph*, or *CLG*. In the CLG, any cycle will automatically obey constraint 1, will entail one that does, or will violate some other constraint. The structure of the CLG depends upon the semantics of synchronization.

Constraint 2 is usually enforced by maintaining some safe[3] approximation of (a) the abstract stores A that may appear in execution waves with the nodes of D, or (b) the nodes (or types of nodes) that must be head nodes on deadlock paths starting at any given node $h \in D$, or (c) other structural information obtainable at parse time (such as the nesting of remote procedure entries), or (d) any combination of these.

For instance, in binary semaphores, we may be able to deduce that whenever h executes along with h', semaphore X is set. If h is a statement $p(X)$, then no valid deadlock cycle can contain both h and h', because h would always be able to complete. The same is also true if X must be set, and any deadlock path from h to h' must contain

[3]A *safe* approximation is always one that *avoids* pruning cycles that *might* represent deadlocks.

a $p(X)$ statement.

Constraint 3 is enforced by estimating a binary *can't happen together*, or CHT, relation. $h \in CHT(h')$ means that no instance of h can happen together with any instance of h'. Thus, no deadlock cycle can include both h and h' as head nodes.

Constraint 4 is often difficult to enforce, but it is sometimes possible to do so in special cases. For instance, in Figure 2.4, we may find that, whenever t executes, w, which can rendezvous with t, is also executing. w might not be in D, but will break any deadlock cycle that includes t. Thus, t cannot be the head node of a deadlock cycle at all. A possible refinement of this is to find deadlock-breaking nodes that must execute whenever a particular *pair* of head nodes executes.

2.3 Developing a deadlock analyzer.

Figure 2.5 shows the structure of a typical deadlock analyzer. This section details the individual algorithms in the analyzer.

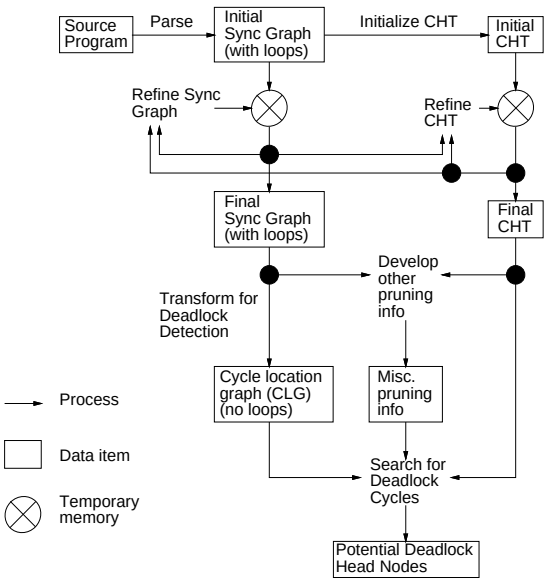

Figure 2.5: A typical deadlock analyzer.

A typical deadlock analyzer works as follows. The source program is parsed, and translated into a sync graph, with loops. In a process of iterative refinement, we extract CHT information from the sync graph, and use this information to refine the sync graph

by eliminating extraneous sync edges that do not correspond to actual synchronization.

Once the iterative refinement process stabilizes, we use the final sync graph and CHT as input to algorithms that find pruning information other than CHT. This pruning information is specific to the synchronization primitive being analyzed. We then transform the sync graph into a CLG, and find potential deadlock head nodes in this graph. If no deadlock cycles exist in the CLG, then we report that the program cannot deadlock; otherwise, we can identify the head nodes that might be part of a deadlock.

2.3.1 Non-concurrency analysis.

CHT information is very useful for pruning invalid deadlock cycles. Our usual method of estimating CHT is to make an initial conservative estimate of CHT, then to refine both the estimate and the sync graph. CHT information can be gathered by finding general structures that induce non-concurrency in the program (such as statements that must finish before others can start), or by searching for common programming idioms (such as critical section structures). In Sections 2.3.1 and 2.3.1, we discuss the two main classes of CHT *refinements*.

We define the set CHT_{perf} as the "perfect" CHT relation, with respect to the set of reachable states in the execution model. $m \in CHT_{perf}(n)$ iff there is no execution wave $(E, A) \in S_{Trans}$ such that $\{m, n\} \subseteq E$. A set CHT is conservative if, for all nodes n, $CHT(n) \subseteq CHT_{perf}(n)$.

Likewise, a refinement to CHT is conservative if it produces a conservative refined set CHT', given a conservative set CHT as input. We also require that refinements be monotone (i.e., $CHT_a \supseteq CHT_b \Rightarrow CHT'_a \supseteq CHT'_b$), and *inflationary*, i.e., $CHT' \supseteq CHT$.

The refinement process proceeds as follows:

1. Initialize CHT to a subset of CHT_{perf}. A typical initial value of CHT has $m \in CHT(n)$ iff m and n are members of the same single-instance task and $m \neq n$.
2. Repeat until CHT does not change:

(a) Refine the current estimate of CHT, using a series of conservative, monotone, and inflationary refinements that terminate in polynomial time.

(b) Using the current estimate of CHT, find sync edges in E_S that cannot correspond to any real synchronization, using an algorithm that terminates in polynomial time. Eliminate such edges from E_S.

3. Return the final CHT estimate and the final sync graph.

As long as the refinements meet our preconditions, the fixed point they reach is always the same, regardless of the ordering of the refinements (Theorem 2 in Section 3.1.5).

Widening and pseudotransitivity in CHT.

The CHT relation is not transitive in general. However, it admits a useful property, which we call *widening*. Widening is used in many of the refinement algorithms.

Suppose that we have a safe estimate of CHT, and we wish to augment the estimated $CHT(n)$ with additional nodes in $CHT_{perf}(n)$. The main idea of widening is to find a set of nodes S, such that some instance of some node in S is always executing whenever n executes. Note that the node in S whose instance executes along with n may change as the instance of n executes. CHT information may then be propagated from S to n, as follows.

Theorem 1 (Widening.) *If there exists a set of nodes S, such that some instance of some member of S must always execute together with any instance of node n, then the nodes in $\bigcap_{n' \in S} CHT(n')$ cannot happen together with n.*

Proof: Obvious. □

Corollary 1 *If there exists a set of nodes S, such that, at all times that any instance of node n is executing, either:*

- *Some instance of some member of S is executing; or,*
- *No instance of any such node is executing;*

then the nodes in $\bigcap_{n' \in S} CHT(n')$ *cannot happen together with* n.

Proof: Obvious. □

A simple example of such a set S may be found in the Ada remote procedure call; the nodes of a remote procedure body must execute together with some caller. So if n is a node in the remote procedure body, S would be the set of all nodes that can call the remote procedure.

Other examples of S include the entire set of program nodes other than n, and the nodes of a task that must execute whenever n does. These methods of widening are called *pseudotransitivity* and *task-wise pseudotransitivity*, respectively. These are explained in Corollaries 2 and 3, below.

Corollary 2 (Pseudotransitivity.) *The nodes in the set*

$$\bigcap_{n' \in N - CHT(n) - \{n\}} CHT(n')$$

cannot happen together with n.

Proof: The set $N - CHT(n) - \{n\}$ contains all nodes that may happen together with n. This becomes the set S of Theorem 1. □

Corollary 2 allows us to take the *pseudotransitive closure* of the CHT relation, in the obvious way. However, we do not do this explicitly. In pinning, critical section, and remote procedure analysis, we attempt to identify, for each node n, some set $S \subset N - CHT(n) - \{n\}$ satisfying Theorem 1. If no such set can be identified for n, we use $N - CHT(n) - \{n\}$ to refine CHT by pseudotransitivity as a by-product of the analysis.

Corollary 2 is a weak result, since $n \notin CHT(n)$ for any n that can execute at all. Thus, we do not use pseudotransitivity explicitly. However, we can widen CHT on a task-wise basis, as follows:

Corollary 3 (Task-wise pseudotransitivity.) *For any task* T *and node* $n \notin T$, *such that some instance of* T *is executing whenever any instance of* n *is executing, the nodes in the set*

$$\bigcap_{n' \in T - CHT(n)} CHT(n')$$

cannot happen together with n, where $CHT(n) \subseteq CHT_{perf}(n)$.

Using Corollary 3 will usually require one pseudotransitivity computation per task. In certain cases (e.g., Section 5.1.2), CHT is refined on a task-wise basis, so we can use task-wise pseudotransitivity without an excessive time penalty. In general, though, using Corollary 3 would increase the cost of a refinement algorithm by a factor of $\mathcal{O}(|\,Tasks|)$ time.

Algorithm 1 *Widening CHT using sets of nodes which must happen together with other nodes. Denoted $CHT' = Widen(CHT, S, T)$.*

Input:

- The boolean CHT array. The array has one row and column per variable; a 1 in location (m, n) means that $n \in CHT(m)$.[4]
- An array S, such that, for each node n, either $S(n) = \emptyset$ or some element of $S(n)$ must happen together with each node n.
- A vector T of nodes, such that, for each node $n \in N - \{b, e\}$, n must execute concurrently with members of T. We use T to try to apply pseudotransitivity (general or task-wise) to find CHT information for nodes not in S; doing so is inexpensive. To apply task-wise pseudotransitivity, T will be the set of nodes of a single task. To apply the more general pseudotransitivity of Corollary 2, $T = N$.

Output:

- The array $CHT' \supseteq CHT$.

Procedure:

1. $Mask \Leftarrow (S \times ONE)$, where ONE is the array of all ones, and $\times$ is the usual (Boolean) matrix product.[5]

[4]Boolean arrays can equivalently be viewed as functions from sets to sets; we adopt this notation system from computer graphics, and refer the reader to [CK84] as a precedent.

[5]Multiplying a set by a Boolean array can equivalently be thought of as function application.

The Boolean matrix notation used here, and subsequently in this chapter, is described in Appendix A.

2. $Tarray \Leftarrow (ONE_{Vector} \times T) \cap (-Mask)$.

   $Tarray$ is an array which duplicates T on every row for which S has no set bits.

3. $MustHT \Leftarrow (S \cup Tarray) - CHT - I$, where I is the $|N| \times |N|$ *identity array.*

   For each node n, $MustHT(n)$ is a set of nodes other than n such that some instance of at least one member of $MustHT(n)$ must (and can) happen together with any instance of n. If $S(n) = \emptyset$, then $MustHT(n) = T - CHT(n) - \{n\}$, to apply pseudotransitivity. If $S(n) \neq \emptyset$, then $MustHT(n) = S(n) - CHT(n) - \{n\}$, a set smaller than (and thus refined over) the set implied by the pseudotransitivity relation on n.

4. $NewCHT \leftarrow -(MustHT \times (-CHT))$.

   $NewCHT(n)$ is the set of nodes which can't happen together with any node that must happen together with n.

5. $CHT' \leftarrow CHT \cup NewCHT \cup NewCHT^T$.

   Since $NewCHT$ is not necessarily symmetric, we must union CHT with both $NewCHT$ and its transpose. Obviously, $CHT'(n) \supseteq CHT(n)$ for all n.

Lemma 2 (Correctness of the widening algorithm.) *If* $CHT \subseteq CHT_{perf}$ *and the parameters* S *and* T *are correctly specified to Algorithm 1, then* $CHT \subseteq CHT' \subseteq CHT_{perf}$.

Proof: From the discussion in the algorithm definition. □

Lemma 3 (Time bounds for the widening algorithm.) *Algorithm 1 requires* $\mathcal{O}(|N|^3)$ *time to execute in the worst case.*

Proof: The most expensive operation in Algorithm 1 is a multiplication of two $|N| \times |N|$ arrays; this requires $\mathcal{O}(|N|^3)$ in the worst case. □

Since Algorithm 1 must be used iteratively in some cases, it is also important to know the cost of updating $Widen(CHT, S, T)$ incrementally, as a function of the number of clear bits in any "old" version of CHT that are set in the subsequent "new" version.

Lemma 4 (Incremental time bounds for the widening algorithm.) *The cost of incrementally updating a one-bit change in the CHT parameter of* $Widen(CHT, S, T)$ *is* $\mathcal{O}(|N|)$.

Proof: Suppose that bit (m, n) is zero in the old CHT array and one in the new CHT array. We need not update $Mask$, since it depends only on S. However, we will need to clear bit (m, n) in $MustHT$. Similarly, we will have to clear bit (m, n) in $-CHT$.

$NewCHT$ may be updated by updating row m and column m alone. This update requires $\mathcal{O}(|N|)$ bit operations. From this, we may recompute CHT' in $\mathcal{O}(|N|)$ bit operations. Thus, it takes $\mathcal{O}(|N|)$ bit operations to update a one-bit change to the CHT parameter. □

B4 analysis.

The second main method of refining CHT is by using *B4 analysis*. We say that $m \in B4(n)$ if all instances of m must complete before any instance of n can start. If so, then $m \in CHT(n)$ and $n \in CHT(m)$.

B4 is usually found using a lattice framework. Initially, we know that $m \in B4(n)$ if m and n are contained in a single-instance task, and if there is a control path from m to n, but no such path from n to m.

We can also propagate *B4* information through control and sync edges. For instance, see Figure 2.6. The *B4* information is represented as a pair of sets, for control and synchronization, using a multiple edge type lattice framework as we discuss in Section 2.3.4. We deduce that $a \in B4(n)$, since a is in the *B4* sets of all of n's control predecessors. Similarly, b is in the *B4* sets of all nodes that perform synchronization operations allowing n's control predecessors to complete (called the *completors* of n), so b is also in $B4(n)$. The resulting information is passed along to the control and

completor edge successors of n, as shown.

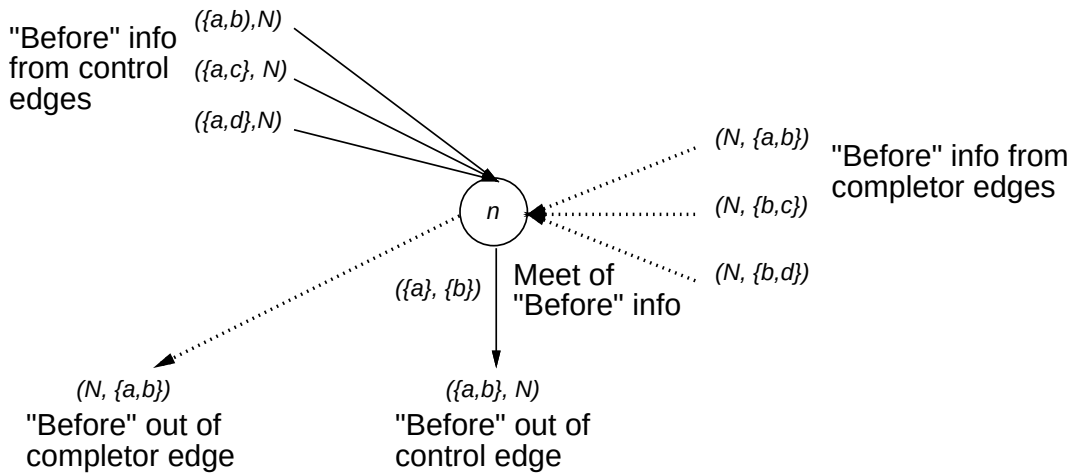

Figure 2.6: Propagation of "must happen before" information along control and completor edges.

A troublesome non-concurrency relationship.

Figure 2.7 shows an example of a sync graph having a non-concurrency relationship which is troublesome for deadlock detection. Here, any *pair* of nodes from $\{m, n, p\}$ can happen together, but not all three. However, all three would have to happen together for a valid deadlock cycle to exist.

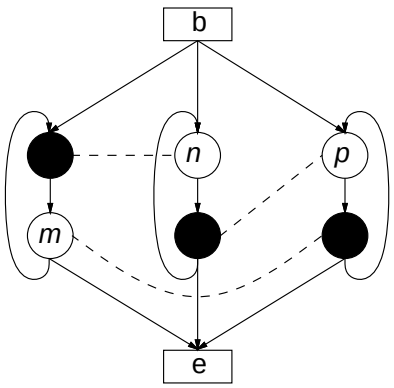

Figure 2.7: Program for which the binary CHT_{perf} misses information.

The sync graph of Figure 2.7 illustrates that there are some troublesome spurious

deadlock cycles that cannot be pruned by the binary *CHT* relation. Instead, the program requires a *ternary CHT* relation. Following Figure 2.7, it is easy to construct programs for which *n-ary CHT* relations are needed, where n is linearly related to the number of nodes. Since an approximate n-ary *CHT* relation may have up to 2^n node subsets, the time to construct one may be exponential in the number of nodes.

Fortunately, this behavior was seen only in a few benchmark programs in our experimental work, and was not seen in other, more realistic programs. The problem of this specific program construct may be addressed by detecting it explicitly, should we need to do this.

2.3.2 Finding constraint information other than *CHT*.

Information that some node *must* be in state READY whenever it is on the wave (e.g., node t in Figure 2.4) can be used to prune deadlock cycles under constraints 2 and 4. The specific information needed to do this depends upon the specifics of the synchronization.

In other models, the constraint information may involve the abstract store; the example for applying constraint 2 to binary semaphores (as we showed briefly in the example of Section 2.2.4) is such a case. Here, we must develop a conservative estimate of what semaphores must be set whenever given nodes execute, and what semaphores are waited upon by head nodes along all deadlock paths between two head nodes.

A conservative estimate of constraint 2 and 4 pruning information can often be kept for all deadlock paths between two nodes. This information is called a *path summary*, and is propagated around the sync graph, with appropriate updates, as described in Section 2.3.6.

2.3.3 Refining the sync graph.

Recall that sync edges should represent a direct "enable to proceed" relation between nodes. We may be able to find sync edges that do not represent such a relationship; these may be safely eliminated from the sync graph as irrelevant.

For instance, suppose that we have directed, two-ended sync edges, and that E_S includes the edge (m, n). If $n \in B4(m)$, then m cannot enable n to complete, since no instance of m can start until all instances of n complete. Therefore, we can safely eliminate the edge (m, n) from E_S.

The information necessary to eliminate irrelevant sync edges often depends on the synchronization model. For instance, in rendezvous, sync edge $\{m, n\}$ can be eliminated if $m \in CHT(n)$. This is not true in the binary semaphore model, in which m may never happen together with n, but m may set a semaphore X which n awaits, and thus allow n to proceed. To show that a m cannot enable n to complete, we must show that the value of X must always be *modified* between the time m completes and the time n starts. If so, then m never directly enables n to complete.

2.3.4 Multi-source lattice frameworks.

Several program analysis problems on sync graphs require us to solve data flow problems in which information arises independently from multiple sources at each node. For example, we may obtain information separately through control and synchronization. We call such problems *multi-source* problems. Some examples of multi-source problems include finding, at each node n, the nodes that always happen before n (Section 2.3.1), the semaphores that must be set at n (Section 2.3.2), and the semaphore definitions that must be killed at n (Section 8.4.5).

As a simple example, consider the "must happen before" relation (Figure 2.6) in binary semaphores. A node m must happen before n if either (1) m must happen before all control predecessors of n, or (2) m must happen before all completors of n (nodes that enable the control predecessors of n to complete). We see that a must happen before n because a must happen before all control predecessors of n; likewise, b happens before n because b happens before all completors of n.

In conventional lattice frameworks, such as the well-known reaching definitions framework, data flow information at a node n is represented as a set. However, if we were to express this "must happen before" information as an $|N|$-ary cross-product

of a single powerset lattice $P(N)$, we would lose both of these facts. Therefore, we see that information coming from control and sync predecessors must meet *separately* at n, but the *individual* meet for each type of edge must be set intersection.

Our formulation: the cross-product of homogeneous lattice frameworks.

To express a multi-source problem as a lattice framework, we use a lattice of k-tuples, where k is the number of sources of information, as the base framework for each node. The lattice framework representing the problem on the entire graph is thus the cross-product of $|N|$ *homogeneous* lattice frameworks. The homogeneity of the frameworks is important, since it makes it easier to prove correctness, accuracy, and convergence time properties in the cross-product lattice than if the node frameworks were dissimilar.

Each information source is assigned a position in the k-tuples. If the base lattice L represents the information we are trying to express, then each element of the tuple is a member of L, so that the form of the lattice representing a node is L^k. The meet in L^k is element-wise meet in L; $\sqsubseteq$ is defined accordingly. The edge function f_e, for an edge e of type i ($1 \leq i \leq k$), is a tuple with the "top" element $\top_L$ in all positions except the i'th; the function defining the i'th position may be defined in any way consistent with the monotonicity of f_e. More specifically, we require that, if $a[i] \sqsupseteq b[i]$, then $f_e(a)[i] \sqsupseteq f_e(b)[i]$.

Continuing the example of Figure 2.6, we have $k = 2$, since there are two edge types. A *completor edge* (m, n) exists if completion of m can enable n to be reached in control flow; more specific structural definitions are particular to the sync graph structure for the synchronization primitive. Position 1 in the 2-tuple is reserved for control edge information; position 2 expresses information for completor edge information.

The edge functions are defined as follows. For one of the positions in the resultant 2-tuple, the edge function returns the union of the sets in both positions of the argument 2-tuple. For the other position, the edge function returns the set of all nodes. The positions depend upon the type of the edge: for control edges, the set union is returned in the first position; for completor edges, the second.

Continuing with our example, the meet of incoming tuples at node n is the tuple $(\{a\},\{b\})$, which preserves the fact that both a and b must happen before n. The edge function for the control edge leaving n expresses the same information as $(\{a,b\},N)$; likewise, the completor edge function expresses it as $(N,\{a,b\})$.[6]

Most of the frameworks we have used have inflationary edge functions. This property implies 1-semiboundedness and other properties useful for analyzing the convergence rates of the frameworks. However, in the analysis problems we have investigated, we have found that the useful frameworks of this type are generally not distributive, i.e., $\exists a,b \in L^k : f(a \sqcap b) \neq f(a) \sqcap f(b)$. Thus, the frameworks are not *rapid* [KU76].

One advantage of using homogeneous lattice frameworks is that they are easier to analyze than the cross-product of heterogeneous lattice frameworks (described below.) The edge functions from one node lattice framework have the same properties in each of the other node lattice spaces, since all these spaces are identical. In the cross-product framework, proofs of properties of the function space (e.g., monotonicity, closure under meet and function composition, k-boundedness) often can be derived directly from proofs of these properties for the node lattices. This is often not the case for the cross product of heterogeneous frameworks, in which the lattice representing the domain and range of the edge function may differ.

---

[6]If there is some node that must not have any incoming edges of some type i, we may (figuratively) insert "dummy edges" into the sync graph to assure that no position of the meet of input tuples at a node is incorrectly driven to $\top_L$. Equivalently, we can choose edge functions so that this is effectively done for such nodes. If the edge is of type j, then the i'th and j'th position of the resultant tuple of the edge function would always have the same value.

One alternative one might consider would be to define meet over an empty set ($\bigsqcap_{y\in\emptyset} y$) as $\bot$. However, doing so has undesirable results. For each element $x \in L$, x is identical to itself not met with anything, i.e., $x = x \sqcap (\bigsqcap_{y\in\emptyset} y)$. Suppose we were to define $\bigsqcap_{y\in\emptyset} y = \bot$. Then $x = x \sqcap (\bigsqcap_{y\in\emptyset} y) = x \sqcap \bot = \bot$. Thus, all elements of the lattice would be equal to bottom!

More to the point, defining $\bigsqcap_{y\in\emptyset} y$ as $\top$ produces desirable semantics for unreachable nodes in many lattice problems. Consider reaching definitions, in which the meet operator is set union. $\top$ is thus the empty set, implying that a node with no incoming control edges has *no* reaching definitions, as we would expect. Also, defining $\bigsqcap_{y\in\emptyset} y = \top$ often eliminates special cases in implementation.

Cross-product of heterogeneous lattice frameworks.

Another possible representation for some multi-source problems uses a data flow formulation like that of [Cal88, LR92]. In these formulations, the elements of the base lattice are the members of some set. The meet operator is set union for some nodes, and set intersection for others.[7] Thus, some nodes are represented by the base lattice, while others are represented by the "inverted" base lattice.

The lattices for the intersection nodes have $\emptyset$ as the bottom element; the lattices for the union nodes have the universe as the bottom element. The lattice representing the graph is the cross-product of the node frameworks. Note that the lattice frameworks representing the nodes are thus *heterogeneous*; functions do not necessarily have the same properties in each of the node lattices.

To represent a multi-source problem with k sources of information as a heterogeneous cross-product framework, we would split each node n in the graph into nodes n_1 through n_{k+1}. (See Figure 2.8.) Nodes n_1 through n_k have set intersection as their meet; node n_{k+1} has set union as its meet. We delete each edge (m, n) representing information source i, and replace it with an edge (m, n_i). Each edge (n, p) representing information source j is replaced with an edge (n_{k+1}, p_j). Edges (n_i, n_{k+1}) are inserted for $1 \leq i \leq k$. The final graph, after node splitting, thus has at most $k|N|$ more nodes and edges than the original graph.

There are certain difficulties in proving monotonicity, k-boundedness, and other properties of heterogeneous cross-product frameworks, whether the functions are attached to edges or nodes. The reason is that there is no general definition of $\sqsubseteq$ that is consistent for all edges, or for all nodes. Rather, the property would have to be proved for the functions on the cross-product lattice. These difficulties are obviated by the use of the heterogeneous cross-product frameworks described above.

---

[7] Neither of [Cal88, LR92] formulate their data flow problems as lattice frameworks; this represents one method of doing so.

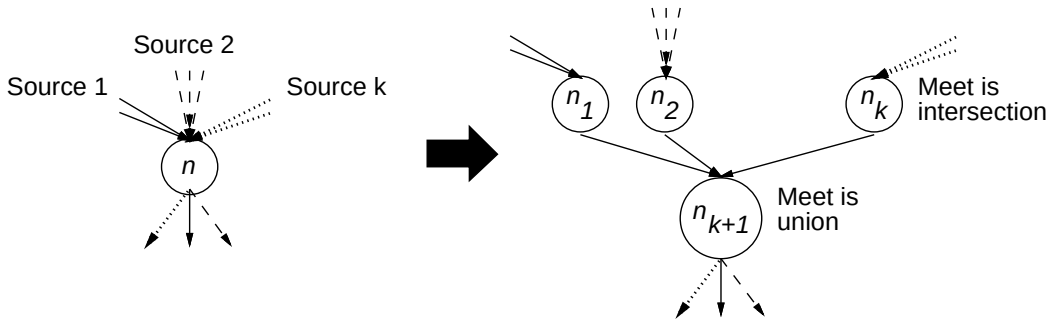

Figure 2.8: Node split operation used in representing multi-source problems in a heterogeneous cross-product framework.

2.3.5 Cycle location graphs.

The sync graph of a program may contain cycles which violate constraint 1. Any loop body containing synchronization would form such a cycle; a single undirected sync edge would also form a cycle between the two nodes it connects. Such cycles present a problem in deadlock detection, since they might appear as deadlocks.

Fortunately, for at least the synchronization primitives studied to date, it has been possible to transform the sync graph into a CLG, which has the following desirable properties for cycle detection:

- The size of the CLG (nodes plus edges) is at most a polynomial of the size of the sync graph.
- The CLG represents only those cycles which obey constraint 1, or which can be pruned using other constraints.
- If there is a valid deadlock cycle in the sync graph, then it is represented by a valid deadlock cycle in the CLG.

Each node in the CLG corresponds to some node in the sync graph; some edges correspond to control or sync edges, while other edges are added in the construction process. The transformation also builds a map from sync graph nodes and edges to the corresponding parts of the CLG.

Transforming the sync graph to the CLG involves two steps. In the first step, the parse form sync graph is transformed into a *cycle detection (CD) sync graph.* Control

loops are eliminated, while preserving all control paths that represent deadlocks. The transformation depends upon the synchronization primitive being analyzed.

An example of a CD sync graph transform is that used for Ada programs. The transform deletes loop edges, and then unrolls two copies[8] of each *outermost-level* loop. This eliminates control cycles in the program. However, if there is a control path between nodes m and n in the sync graph, then there is a control path from m either to n, or to a duplicate of n, in the unrolled graph. Thus, the unrolling preserves deadlock cycles, and eliminates control loops.

A second type of CD sync graph transform, used in Concurrent C (Section 9.5.2), may be used to prune *head-tail node pairs* from deadlock cycles. For each pair of nodes (m, n) such that (1) n can be reached in control from m and (2) n can be a tail node when m is a head node, we create duplicates of m and n, connected by control edge (m, n). The advantage of this transform is that it can eliminate some deadlock cycles; the disadvantage is that the number of nodes in the CD sync graph is potentially $\mathcal{O}(|N|^2)$ rather than $\mathcal{O}(|N|)$.

If the sync graph representation uses undirected edges, or hyperedges, a second transform, the *cycle location graph (CLG) transform* is necessary to eliminate some paths that are spurious under constraint 1. These paths enter and exit tasks without traversing control edges in the tasks. The transformation that eliminates them is to split nodes, and replace the sync edges with directed edges representing the "enable to proceed" relationship.

For an example from rendezvous synchronization, see Figure 2.9. Nodes m and n are each split into "out" and "in" nodes, and the undirected sync edge $\{m, n\}$ is replaced by a pair of directed edges. The "t" and "h" subscripts refer to the fact that the split nodes represent the tail and head nodes of deadlock cycles, respectively, after the split. The splitting operation prevents the spurious deadlock cycle (m, n, m). The CLG transform at most doubles the number of nodes and edges in the graph, and prohibits cycles that enter and exit tasks without traversing sync edges.

---

[8] By "unrolling two copies of a loop" or "unrolling a loop twice," we mean that we substitute two

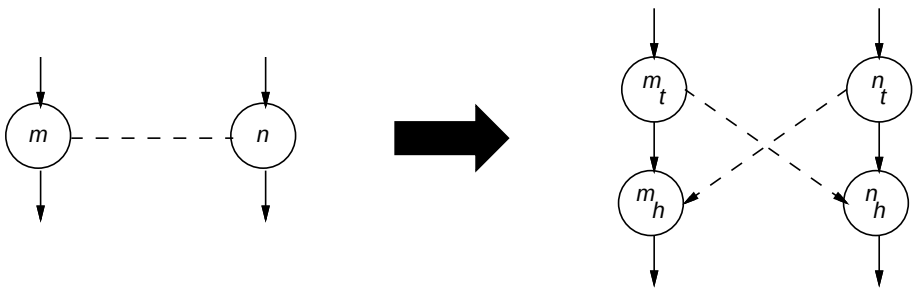

Figure 2.9: Node splitting in creating the CLG.

Note that the CLG transform does not prohibit cycles from entering single instance tasks more than once. However, it is easy to show that, if the cycle does enter a single instance task more than once, then either there is a deadlock cycle that enters only once, or the cycle contains a pair of head nodes that can't happen together [MR90a].

2.3.6 The *DeadPath* problem.

The final step in analysis is to determine, for all potential head nodes h, whether a deadlock path exists from h to itself. We do this using a *DeadPath lattice framework*. The framework is, again, particular to the synchronization primitives being analyzed, and is usually expressed as the cross product of one framework for each possible head node. The least fixed point of *DeadPath* determines, for each node n in the CLG, whether a deadlock path may exist from h to n, and what constraining information is true for all such paths. A conservative estimate of the constraining information is kept as a path summary, which is updated as new paths are discovered.

As an example, consider rendezvous. The path summary might contain a set of signal types y, such that all paths from h to m must contain at least one signaling head node of type y. Suppose that this is the case, and we wish to extend the path along a CLG edge representing sync edge $\{m, n\}$. If n is an accept head node of type y, we know that n will always have a head node to rendezvous with in any deadlock path including h and m. Therefore, the output of the edge function (m, n) will be $\top$ in this case, meaning that no such valid deadlock path exists. Other paths not involving m

copies of the loop body, in sequence, for the original loop.

may, however, result in a smaller summary at node n.

We have not included CHT information in the path summaries; though it would be possible to do so,[9] experimental work has shown that this is not a handicap in practice. Rather, we have avoided propagating path summaries for h (other that $\top$) along sync edges into nodes in $CHT(h)$. This is accomplished by a simple test in the edge functions.

2.4 Experimentation and evaluation.

Once the deadlock analyzer's algorithms have been developed, we must code, test, and evaluate a prototype analyzer. The evaluation consists of validating the polynomial analyzer against state enumeration, and determining the number, and causes, of the false alarms signaled by the polynomial-time analyzer.

It is important to use "real world" data, as well as artificial test suites, in evaluating the analyzer. There are two reasons for this. First, we wish to identify and correct as many false alarms as possible. The common false alarms might be better reflected in user programs than in a test suite. Wherever false alarms are seen, the cause should be identified, and the code, algorithm or theoretical foundation should be changed to correct it (if doing so is possible, and if the problem seems like one that will be common in practice).

Second, the analyzer finds certain common programming idioms, such as critical sections, to eliminate spurious deadlocks. The effectiveness of this analysis must be tested using data from the domain in which the analyzer is expected to operate. If an idiom turns out to be very rare, we can eliminate it from the analysis to optimize the algorithm.

It is also possible to construct the analyzer so that pruning information can be

[9] CHT information would be a conservative estimate of the nodes that couldn't happen together with some head node on the path. Thus, meet is set intersection, sync edge functions would union the CHT summary with the constant CHT set of the destination node, and control edges would have the identity function.

"turned off" in the testing process. For instance, if we never update the pruning information in the path summaries during deadlock cycle detection, then we are effectively turning off the effect of that information. Similarly, *CHT* refinements may be turned off by branching around them. If some piece of information proves ineffective in eliminating false alarms, it may be omitted from the analysis as an optimization. Our experimental work has found ineffective summary information in this way (Section 7.9).

Chapter 3

Lattice frameworks and properties.

3.1 Lattice frameworks and properties for deadlock detection.

A *lattice framework* [Hec77] is a quadruple $D = (G, L, F, M)$ where $G = (N, E)$ is a graph, L is a lattice, F is a set of monotone functions $L \rightarrow L$ such that F is closed under meet and composition, and M is a mapping function $E \rightarrow F$ from the edges of G to functions in F. Lattice frameworks are commonly used to represent iterative problems in data flow analysis, and are convenient for analyzing the time for convergence of iterative data flow algorithms. In this section, we describe some properties of lattice frameworks, which we will later use in the timing analysis of the the *B4* and *DeadPath* problems.

3.1.1 Motivating the analysis.

In the sections that follow, we analyze the 1-semiboundedness and distributivity properties of the function spaces for lattice frameworks. We are interested in these properties for the following reasons:

- *Applicability of complexity results.* The Kam-Ullman time complexity result is applicable only to lattice frameworks whose function spaces are both distributive and 1-semibounded. The combination of these two conditions implies that the framework is Kam-Ullman *rapid* (See Section 3.1.4.)
- *Solution time of path methods.* The k-boundedness properties of the function spaces are important in bounding the execution time of Tarjan's path problem methods. (See Section 3.1.3.)
- *Accuracy.* If the function spaces are distributive, then the maximum fixed point of an iterative algorithm is the meet-over-all-paths solution of the problem, i.e., the iterative algorithm is precise.

3.1.2 General lattice framework properties.

We present the following lemmas for later use in proving properties of our lattice frameworks.

Closure properties.

Lemma 5 *In a lattice, if* $a \sqsubseteq b$ *and* $c \sqsubseteq d$, *then* $a \sqcap c \sqsubseteq b \sqcap d$.

Proof:

$$\begin{aligned} a &\sqsubseteq b \\ a \sqcap c &\sqsubseteq b \sqcap c \\ c &\sqsubseteq d \\ b \sqcap c &\sqsubseteq b \sqcap d \\ a \sqcap c &\sqsubseteq b \sqcap d. \Box \end{aligned}$$

Lemma 6 *The set of monotone functions on a lattice is closed under finite composition and meet.*

Proof: A monotone function f is such that

$$a \sqsubseteq b \Rightarrow f(a) \sqsubseteq f(b).$$

If two functions f_1 and f_2 are monotone, then

$$a \sqsubseteq b \Rightarrow f_1(a) \sqsubseteq f_1(b)$$

and

$$a \sqsubseteq b \Rightarrow f_2(a) \sqsubseteq f_2(b).$$

By Lemma 5,

$$a \sqsubseteq b \Rightarrow f_1(a) \sqcap f_2(a) \sqsubseteq f_1(b) \sqcap f_2(b).$$

By the definition of monotonicity,

$$a \sqsubseteq b \Rightarrow f_1(f_2(a)) \sqsubseteq f_1(f_2(b)). \Box$$

Lemma 7 *The set of distributive functions on a lattice is closed under finite function composition and meet.*

Proof: A function f is distributive if, for all $a, b \in L$, $f(a \sqcap b) = f(a) \sqcap f(b)$. Suppose f_1 and f_2 are distributive. Then

$$\begin{aligned} f_1(f_2(a \sqcap b)) &= f_1(f_2(a) \sqcap f_2(b)) \\ &= f_1(f_2(a)) \sqcap f_1(f_2(b)). \end{aligned}$$

Therefore, the composition of two distributive functions is also distributive, and the set of distributive functions is closed under composition.

Also, let $f = f_1 \sqcap f_2$.

$$\begin{aligned} f(a \sqcap b) &= f_1(a \sqcap b) \sqcap f_2(a \sqcap b) \\ &= f_1(a) \sqcap f_1(b) \sqcap f_2(a) \sqcap f_2(b) \\ &= f(a) \sqcap f(b). \end{aligned}$$

Therefore, the meet of two distributive functions is also distributive, and the set of distributive functions is closed under meet. □

Lemma 8 *The set of inflationary functions on a lattice is closed under finite composition and meet.*

Proof: A function $f : L \to L$ is *inflationary* iff $\forall a \in L : a \sqsubseteq f(a)$. Say f_1 and f_2 are inflationary. Then

$$a \sqsubseteq f_1(a)$$

and

$$a \sqsubseteq f_2(a).$$

Therefore,

$$a \sqsubseteq f_1(a) \sqcap f_2(a)$$

and, without loss of generality,

$$a \sqsubseteq f_2(a) \sqsubseteq f_1(f_2(a)).\square$$

Note that a function f may be inflationary but not monotone; e.g., $f(x) = 2x$ if x

is even, $3x$ if x is odd.

Lemma 9 *The set of monotone and inflationary functions on a lattice is closed under finite composition and meet.*

Proof: Direct from Lemmas 6 and 8. □

Lemma 10 *The set of inflationary and distributive functions on a lattice is closed under finite composition and meet.*

Proof: Direct from Lemmas 7 and 8. □

Lemma 11 *Inflationary functions are 1-semibounded.*

Proof: A function is 1-semibounded if, for all $x, y \in L$, $f(y) \sqsupseteq y \sqcap x \sqcap f(x)$ [MR91a]. If f is inflationary, then $f(y) \sqsupseteq y \sqsupseteq y \sqcap x \sqcap f(x)$. □

Convergence time for iterative lattice problems.

Lemma 12 *If a lattice L has a finite height h, and is used in a monotone data flow framework $D = (G, L, F, M)$ where $G = (N, E)$, then the worklist iterative algorithm [Hec77] applied to D where each node in N has an initial value of $\bot$ terminates after no more than $h|N|^2\lceil\log_2(|N|)\rceil$ meet operations and $h|N|^2$ edge function evaluations in the worst case. The cost to maintain the worklist during this time is $\mathcal{O}(h|N|\log(|N|))$ in the worst case. This result also holds where each node in N has an initial value of $\top$.*

Proof: Suppose each node in N is given an initial value of $\bot$. All nodes in N are initially placed on the worklist, and can only increase in value or remain at $\bot$. If all nodes remain at $\bot$, then the worklist algorithm terminates after $|N|$ node visits and $|E|$ edge function evaluations. If any node n increases in value, then all its children are placed on the worklist. The process continues until the worklist is empty.

If any node m has its value v_m increased, then for all edges $e = (m, n)$ leaving n, $f_e(v_m)$ must also be increased (by the definition of monotonicity for f_e.) Since v_n is the meet of $f_e(v_{m_i})$ for all incoming edges (m_i, n), increasing any m_i will thus increase v_n or leave it unchanged. Since $v_n = \bot$ initially for all $n \in N$, each v_n may thus only

increase or remain unchanged throughout the execution of the worklist algorithm.

Each node n may have its value increased a maximum of h times. In the worst case, each time a node's value increases, we may have to evaluate $|N|$ edge functions (if N is a complete graph and self-loops are allowed) for edges to n's children. We must also meet $\lceil \log_2(|N|) \rceil$ values at each child node of n to decide whether to place that child on the worklist. (Here we do incremental updates to the values of the children of n and retain partial meet values for edges entering each node in a height balanced tree structure.) Since the graph G contains a total of $|N|$ nodes, we must thus do a maximum of $h|N|^2$ edge function evaluations and $h|N|^2\lceil \log_2(|N|) \rceil$ meet operations.

Assuming that the worklist is maintained as a heap, each insert operation and each delete operation will require a maximum of $\mathcal{O}(\log(|N|))$ time. We must do an insert every time a node increases in value, and a delete every time we evaluate the edge functions out of a node whose value has increased. Thus we must do a total of $\mathcal{O}(h|N|\log(|N|))$ work during execution of the worklist algorithm to maintain the heap.

We may make the same statements if we initialize all v_n to $\top$, in which case each v_n will be deflationary during the execution of the worklist algorithm. □

3.1.3 Tarjan's path problem methods.

Tarjan [Tar81b] gives an algorithm for solving global flow analysis problems which are represented as instances of the *all pairs path expression* problem. In this representation, *path expressions* $P(u, v)$, or regular expressions representing all paths from each node u to node v in the graph $G = (V, E, s)$[1], are constructed. Then the path expressions are mapped into the global flow analysis problem of interest, which is solved by substituting $\perp$ at the source vertex s of G. The result for each vertex v of G is a solution to the data flow problem at v.

Constructing the set of path expressions in a general graph takes $\mathcal{O}(m\alpha(m, n) + t)$ time [Tar81a], where $m = |V|$, $n = |E|$, t is the time required to find the path expressions for the *dominator strong components* of G, and α is the inverse Ackermann function.

[1] We use Tarjan's notation system in this section.

Deriving the dominator tree of G takes time $\mathcal{O}(m\alpha(m,n))$ [LT79]. From these, the path expressions of the dominator strong components of G must be found by applying the algorithm recursively to each component. Assuming that no graph G of interest is such that all of its vertices are involved in the same dominator strong component and that $\alpha(m,n)$ is essentially constant, the time needed to construct the path expressions is bounded by $\mathcal{O}(m^2)$. The length of the path expression produced by this procedure is $\mathcal{O}(m\alpha(m,n)+l)$, where l is the sum of the lengths of the path expressions of the dominator strong components of G. Thus, the length of the path expression is bounded by $\mathcal{O}(m^2)$ under the same assumptions.

The mapping from the path expressions to the data flow problem may require construction of *pseudotransitive closure functions*, $f^{@}(P)$ as an approximation of the transitive closure function $f^{*}(P)$ for the path expressions P. For data flow frameworks which are *k-bounded*, we can compute an approximation to $f^{@}(P)$ in $\mathcal{O}(\log(k))$ time, or we can compute $f^{*}(P)$ itself in $\mathcal{O}(k)$ time. Since k is constant for a given lattice framework, we may regard this time as a constant multiplier on the execution time of the algorithm. The substitution step thus takes $\mathcal{O}(km^2)$ time, assuming that the time to meet is constant in the parameters of G.

Lemma 13 *Under the assumption that Tarjan's algorithm terminates, a solution to a 1-semibounded (or 2-bounded) lattice framework can be found in time*

$$\mathcal{O}(m^2(t_f + t_\sqcap))$$

in the worst case, where t_f is the time for one edge function application and $t_\sqcap$ is the time for a meet operation.

Proof: The time to construct the path expressions for a program is $\mathcal{O}(m^2)$, under the assumption that the inverse Ackermann function is essentially constant. The path expression itself is of length $\mathcal{O}(m^2)$. Therefore, mapping the data flow problem to the path expression will require $\mathcal{O}(m^2)$ meet operations and edge function applications, when the data flow problem is k-bounded and that k is constant in m and n.

1-semiboundedness implies 2-boundedness [MR91a]. Therefore, the lemma holds.

□

We note that Tarjan's methods may have high overhead costs, and are not frequently used in practice. However, they allow us to reduce the asymptotic upper bound on the complexity of the lattice problems used in deadlock detection.

3.1.4 Kam-Ullman "rapid" problems in irreducible graphs.

Kam and Ullman [KU76] describe a version of Kildall's iterative node-listing algorithm which halts in at most $d(G) + 3$ iterations over the nodes of a graph G, where $d(G)$ is the *loop connectedness* of G, or the largest number of back edges found in any cycle-free path in G (with respect to the depth-first spanning tree of G from which the node listing is derived). The functions in the lattice framework are required to be both 1-semibounded and distributive for the Kam-Ullman time complexity result to hold (and to be monotone and closed under finite composition and meet). The lattice frameworks for which the Kam-Ullman result hold are called *rapid*, and include such classical data flow problems as reaching definitions and live variables.

For reducible control flow graphs, $d(G)$ is equal to the maximum loop nesting depth of any statement in the program. However, we cannot in general assume that the sync hypergraph is reducible; in fact, if the program can deadlock, then its sync hypergraph must be irreducible (because it contains the deadlock cycle and a path from the program begin node to each head node in the cycle). For general graphs G, we conservatively assume that G may be fully connected (i.e., a clique), and that $d(G)$ is also therefore proportional to the number of nodes in G.

Lemma 14 *Given a rapid lattice framework $D = (G, L, F, M)$ where $G = (V, E)$ is a general graph, Kildall's algorithm converges in $\mathcal{O}(|V||E|)$ edge function evaluations and $\mathcal{O}(|V|^2 \log(|V|))$ meet operations.*

Proof: From Theorem 2 of [KU76], Kildall's algorithm halts in no more than $d(G)+2$ iterations on a rapid problem. In the initialization step of Kildall's algorithm, the edge functions into each node are initially evaluated and a meet of the edges into each node is taken. Thus there are $|E|$ edge function evaluations and, in the worst case,

$\mathcal{O}(|V|\log(|V|))$ meet operations. The same work is done on each iteration. Thus, a total of $\mathcal{O}(d(G)|E|)$ edge function evaluations and $\mathcal{O}(d(G)|V|\log(|V|))$ meet operations are performed. Since $d(G) = |V| - 1$ in the worst case, we have $\mathcal{O}(|V||E|)$ edge function evaluations and $\mathcal{O}(|V|^2\log(|V|))$ meet operations in the worst case on a general graph. □

Unfortunately, we cannot use the results of Lemma 14 in the timing analysis of the lattice frameworks used in deadlock analysis, since none of these frameworks turns out to use a distributive function space.

3.1.5 Ordering of monotone inflationary functions in composition.

In this section, we show that we can arbitrarily permute the order of composition of monotone, inflationary functions, without affecting the fixed point of the composed function. This result is important in showing that the result of *CHT* analysis (Section 2.3.1) is not sensitive to the ordering of the refinements, since each refinement is a monotone inflationary function.

Denote the function composition $f_1 \circ f_2 ... \circ f_k$ as F. Also, denote the fixed point of $F : L \rightarrow L$ with initial value $L_\perp$ as $F^*(L_\perp)$.

Lemma 15 *$F^*(L_\perp)$ is the least common fixed point of all f_i.*

Proof: $F^*(L_\perp)$ must be a common fixed point of the f_i; otherwise, $F^*(L_\perp)$ would not be a fixed point of F. Also, for any integer j, $F^j(L_\perp)$ is at most the least common fixed point of all f_i. Therefore, $F^*(L_\perp)$ must be the least common fixed point of the f_i. □

Theorem 2 below shows that any fair ordering of functions produces the same fixed point; furthermore, it bounds the number of times any sequence of functions will have to be applied, assuming that at least one sequence reaches the fixed point after a finite number of applications.

Theorem 2 *Let $\{f_i, 1 \leq i \leq k\}$ be a set of functions, such that each f_i is monotonic and inflationary on lattice L, and let $F = f_1 \circ f_2 ... \circ f_k$. Let G be a function $f_{g_1} \circ f_{g_2} ... \circ f_{g_h}$*

such that each function from which F is composed occurs at least once in G. If $F^*(L_\perp) = F^n(L_\perp)$ for some finite integer n, then $G^{kn}(L_\perp) = F^*(L_\perp)$.

Proof: For any $x \in L$, $G^k(x) \sqsupseteq F(x)$. Also, since $F^*(L_\perp)$ is the least common fixed point of all the f_i, $G^j(L_\perp) \sqsubseteq F^*(L_\perp)$ for all j. Thus, $G^{kn}(L_\perp) = F^n(L_\perp) = F^*(L_\perp)$. □

Note that Theorem 2 does not require the functions f_i to be commutative, and indeed does not guarantee their commutativity for elements of the lattice L other than the least fixed point.

3.1.6 Incremental edge elimination.

In this section, we show a useful special property of monotone lattice frameworks. Given that we have a fixed point L_{LFP} in a framework D, we may often wish to compute a new fixed point in a framework D', which is the same as D but with some edges eliminated from the graph. We show here that we can use L_{LFP} as an initial value for D' to yield the same least fixed point as an initial value of $L_\perp$. This will allow us to delete edges and solve lattice problems (such as *B4*) incrementally.

In the following lemma, we "eliminate" edges by assigning them a new edge function that always returns $\top$. Thus, they have no effect on the meet of incoming values at the node.

Lemma 16 *Let:*

- *$G = (N, E)$ be a graph;*
- *L be a lattice with top element $L_\top$;*
- *F be a set of monotone edge functions, including the function $f_\top : F \rightarrow \{L_\top\}$ which returns $L_\top$ for any argument;*
- *$M : E \rightarrow F$ be a mapping from the edges in G to edge functions;*
- *$M' : E \rightarrow F$ be identical to M, except that for some edges $e \in E$, $M'(e) = f_\top$.*

For all $x \in L$ and all $e \in E$, $M'(e)(x) \sqsupseteq M(e)(x)$.

Proof: Obvious. □

The cross-product function assigned to the graph G by M' is thus also greater than that assigned by M.

Lemma 17 *Let G, M, and M' be the same as in Lemma 16. Let:*

- *$D = (G, L, F, M)$ be a lattice framework;*
- *$D' = (G, L, F, M')$ be the lattice framework with M' substituted for M;*
- *L_{LFP} be the least fixed point of D;*
- *L'_{LFP} be the least fixed point of D';*
- *L'_{INCR} be the fixed point of D' with initial value L_{LFP}.*

$L'_{LFP} = L'_{INCR}$.

Proof: From Lemma 16, and Theorem 4.3 of [RMP88]. □

Chapter 4

Modeling a subset of Ada programs.

4.1 Introduction.

In this chapter, we first delineate a simple subset of the Ada language, and show a sync graph and execution wave model for the subset. We also describe the instantiation of our four constraints on deadlock cycles to programs within the subset.

In Sections 4.3 and 4.4, we augment our initial model of Ada programs with a larger set of tasking constructs. This set includes the `accept-do` construct, selective waits, and conditional entries and entry calls. We describe the impact of each of these additions upon the sync graph model, the execution model, and the constraints on deadlock. This demonstrates the evolution of a model for deadlock detection.

4.2 A basic subset of Ada.

Our initial subset of the full Ada rendezvous model includes only simple rendezvous and a static task structure. The `accept` and entry call statements as the only synchronization constructs in this subset.

4.2.1 Tasking behavior and rendezvous in the basic Ada subset.

We assume that all tasks are statically defined; all tasks start execution at the start of the program, and all end at program termination. We also assume that subroutines called from the main procedure, or from the tasks, do not initiate other tasks. Each task is labeled with a *task name*, which takes the form of an alphanumeric string.

Tasks communicate and synchronize using the *rendezvous* mechanism. The rendezvous mechanism is based on *blocking sends and receives*; no synchronization statement can complete execution until its counterpart executes in another task.

Two statement types are used in the basic Ada rendezvous: the `accept` statement and the entry call. The former has the syntax:

```
accept <signalName> (<formalParamList>)
```

while the latter is of the form:

```
{\tt <taskID>.<signalName> (<actualParamList)
```

When execution in task `T` reaches an `accept S(...)` statement, `T` blocks until some other task makes the entry call `T.S(...)`. The parameters are then passed between tasks, and both tasks continue execution. Likewise, if execution reaches the entry call statement `T.S(...)` at a time when task `T` is not executing an `accept S` statement, then the task making the entry call blocks until task `T` reaches an `accept S` statement.

Rendezvous statements may be divided into compatibility classes, according to their signal names and task identifiers they address. The *signal type* of a statement denotes its compatibility class. A signal type is a pair (t, m), where t denotes the task receiving the signal and m is the signal name. The number of signal types in the program is finite

and statically discernible. We can also speak of the *polarity* of statements; an entry call `T.S()` has a polarity opposite of that of the statement `accept S()` in task `T`.

4.2.2 Sync graphs programs in the basic Ada subset.

The *sync graph* of a program is an abstract representation, which includes sufficient control flow and synchronization information to detect infinite wait anomalies. Figure 2.3 on page 22 shows a small Ada program and its sync graph. The sync graph of a program is a triple, (N, E_C, E_S) where:

- N is the set of nodes in the sync graph, and includes one node corresponding to each *tasking statement* in the program. The set of tasking statements includes all rendezvous statements, plus the `begin` and `end` statements of the main procedure and the `end` statement of each task.

  Each node n_s, corresponding to statement s, represents all maximal sequences Q of statement instances in all executions of the containing task $Task(s)$, such that Q ends with an instance s_i of s, and Q contains no instances of tasking statements other than s_i.

  The *begin node* b, denotes the start of the program, and the *end node* e denotes the end of the program. For purposes of computing CHT, we may also include a *task end node* e_T for each task T; executing e_T signifies that task T, but not necessarily the whole program, has completed execution.
- E_C is the set of control flow edges in the sync graph; these edges represent realizable control flow between rendezvous statements in the program. A directed edge (n_r, n_s) is present in E_C if there is a control flow path from statement r to statement s in the program, such that the path includes no other rendezvous statements.
- E_S is the set of synchronization edges (or sync edges) in the sync graph. Each sync edge represents a possible rendezvous between two tasks. An undirected sync edge $\{n_r, n_s\}$ exists between every pair of nodes n_r and n_s, such that statement r is an entry call of signal type y, and statement s is an `accept` statement of signal

type y.

4.2.3 Model of program execution in the basic Ada subset.

Consider the execution of the program, and its image on the sync graph. The program begins execution at the begin node b, the fork point for all tasks. For each task T, a successor of b, denoting a statement within T, is selected nondeterministically, and starts execution.

Any two nodes which are connected by a sync edge may rendezvous whenever they execute simultaneously. Should more than one pair of nodes be able to rendezvous, a single pair is selected nondeterministically. Successors of the rendezvousing nodes are then selected nondeterministically, and start execution. Thus, simulation of a program execution may be seen as the advance of an *execution wave* through the sync graph. At any point during the execution simulation, the execution wave contains one node belonging to each task, including possibly e.

Each node in the sync graph is in one of the following states during program execution:

- OFF-WAVE. The node is not currently executing.
- READY. The node is executing, and *can rendezvous* with another node which is also currently executing.
- WAITING. The node is currently executing, but *cannot rendezvous* with any other node which is also currently executing.

All nodes on the execution wave are either in state READY or WAITING; all other nodes are in state OFF-WAVE.

We represent an execution wave by a vector E, which has one entry per task; for task T, $E[T]$ is the node r which is the chosen potentially executable node in task T (i.e., next rendezvous to be executed). The initial execution wave, E_{INIT}, corresponds to the state of the program when it starts execution, and contains, for each task T, some node r which is a control flow successor of b.[1]

[1]The abstract memory A of Section 2.2.2 is not used in the model of the basic Ada subset; we

Initially, all nodes are OFF-WAVE, except those in E_{INIT}. The execution wave advances when some pair of nodes on the wave make a rendezvous and leave the wave. Each of the two rendezvousing nodes is replaced by one of its control successors. The state of the successors in E is either READY or WAITING, depending on the signal types and polarities of the other members of E.

More formally, let E be an execution wave and E', an execution wave directly derived from E by the rendezvous of nodes r and s, where r is in task T_r and s is in task T_s. Then for each task T, $E[T]$ and $E'[T]$ are related as follows:

- For all $T \neq T_r$ and $T \neq T_s$: $E[T] = E'[T]$. The state of node $E[T]$ is the same on both waves.
- $E'[T_r] = r'$ where r' is a control flow successor of r. When E' is the current execution wave, node r' is READY if there is any task T' such that $\{E'[T'], r'\} \in E_S$; r' is WAITING otherwise.
- $E'[T_s] = s'$ where s' is a control flow successor of s. When E' is the current execution wave, node s' is READY if there is any task T' such that $\{E'[T'], s'\} \in E_S$; s' is WAITING otherwise.

There may be several different execution waves derivable from a particular execution wave E, because there may be more than one pair of READY nodes on the execution wave. The set of *feasible execution waves* achievable from an initial wave E_{INIT} describes all the synchronization states realizable by a program. Define $\tau(E)$ to be the set of execution waves derivable from wave E; τ is the transition function (Section 2.2.2) for the execution model. The transition function τ_{Set} whose domain is the set of sets of execution waves, and the set S_{Trans} of all feasible execution waves, are derived from τ and E_{INIT} in the obvious way.

4.2.4 Constraints on deadlock cycles in the basic Ada subset.

The constraints on deadlock cycles in the basic subset of Ada are quite simple. Let D be the set of head nodes of a deadlock cycle in an anomalous execution wave E. The

---

therefore omit it from discussion in this section.

following conditions are necessarily true for the nodes in D.

1. *Cycle Shape.* There is a cyclic path in the sync graph, which includes the nodes of D, and is such that, for every node $n \in D$ (where n is a member of some task T):
   (a) The path enters node n through a sync edge.
   (b) The path traverses at least one control flow edge in task T.
   (c) The path leaves T via a sync edge and does not re-enter T.
2. *Head Nodes Cannot Proceed.* No two nodes in D can rendezvous with each other. Equivalently stated for the basic Ada model, no two nodes in D are connected by a sync edge.
3. *Head Nodes Can Happen Together.* All head nodes may execute at the same time. Thus, there is no pair of nodes $m, n \in D$ such that $m \in CHT(n)$.
4. *Nodes Outside Deadlock Don't Break Infinite Wait.* The nodes of D must be contained in some feasible execution wave E, such that no node in $E - D$ is connected by a sync edge to any other node in E.

4.2.5 Cycle location graphs in the basic Ada model.

The sync graphs of Section 4.2.2 may contain cycles which violate Constraint 1. These cycles include control flow cycles, and cycles that enter and exit some task T, without traversing control edges within T.

To eliminate control flow cycles, we may unroll control loops, twice at the outermost level, and eliminate backedges, as shown in Figure 4.1. This transform creates two copies of every node contained in any loop; the number of control and sync edges is at most quadrupled from the original sync graph. Also, if there is a control path $(x...z)$ in the original sync graph, then there must be a path from some node $(x_d...z_d)$ in the unrolled sync graph, where x_d and z_d were duplicated from x and z, respectively. Thus, the unrolling transformation preserves all control paths between representative nodes, eliminates control flow cycles, and results in a graph structure whose size is linearly related to that of the original.

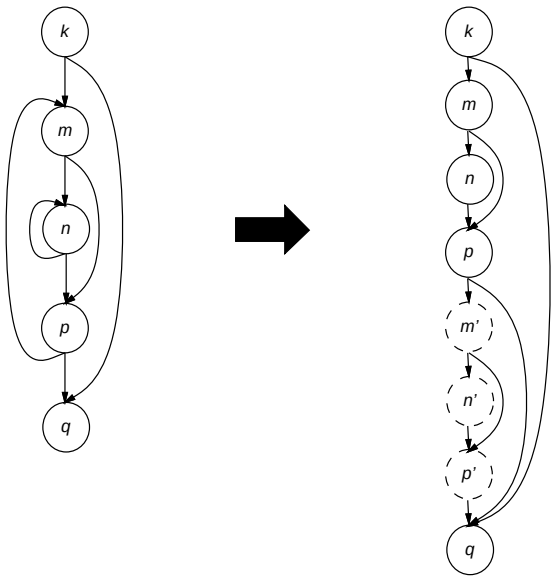

Figure 4.1: Loop unrolling transform in the basic Ada model.

For nodes that are duplicated by the loop unroll, only the copy in the first iteration need be hypothesized as a head node. This improves the efficiency of deadlock detection. In Figure 4.1, the nodes that are not hypothesized as head nodes are shown with dashed outlines.

Since sync edges are undirected, we must also perform the node-splitting operation of Figure 2.9 on the unrolled sync graph. This node-splitting operation eliminates paths that enter and exit tasks without traversing control edges. The resulting graph is a cycle location graph for the basic Ada program.

All paths in the CLG satisfy constraints 1a and 1b, but may re-enter a task once they have left. However, as Lemma 18 shows, paths that re-enter tasks are either redundant, or violate constraint 3 as well as constraint 1.

Lemma 18 *If a deadlock cycle C enters task T through more than one node, then either (1) there exists a deadlock cycle C' that enters through only one node, or (2) two of the nodes in C can't happen together.*

Proof: Suppose that the cycle C enters task T at nodes m and n. Either (a) there is a control path from one of these nodes to the other, or (b) there is not.

For case (a), suppose that there is a control path from m to n (without loss of

generality), and let m' and n' be, respectively, the nodes at which C exits T after entering at m and n. Since there are control paths from m to n and from n to n', there is also a control path from m to n'. We can thus form C' by replacing the subpath of control and sync edges $(m, ...m', ...n, ...n')$ in C with a control-only subpath $(m, ...n, ...n')$, which enters T only at m.

We can repeat this replacement process until there is no pair of nodes m and n to which case (a) applies. After this has been done, either the resultant cycle enters T at only one node (whereupon condition (1) of the lemma holds), or the cycle enters T repeatedly at some set of nodes S such that no node in S is control-reachable from any other node in S (whereupon case (b) applies).

For case (b), since m and n can't happen together, C must include two head nodes that can't happen together. So C is invalid under constraint 3, and condition (2) of the lemma holds. □

Section 4.2.6 provides some additional insight into the complexity of deadlock detection, which bears on Lemma 18. In particular, relying on constraint 3 to prune cycles that violate constraint 1 is likely to be computationally expensive. However, we can take advantage of the fact that the violations are actually special cases of constraint 3, and thus prune all cycles that violate constraint 1 in polynomial time.

4.2.6 Theoretical bounds on complexity of deadlock cycle detection.

In this section, we discuss the difficulty of detecting deadlock cycles that obey certain *combinations* of the constraints. These results refine the complexity results of [Tay83a]. First, we examine the complexity of finding cycles that obey constraint 1, but not necessarily any of the other constraints.

Theorem 3 *Detecting cycles in the sync graph that obey Constraint 1 can be done in* $\mathcal{O}(|N|(|N| + |E_S| + |E_C|))$ *time.*

Proof sketch: We detect cycles in the sync graph by detecting corresponding cycles in the CLG. By construction, the cycles in the CLG must include sync edges, and must traverse control edges in each task they enter. By Lemma 18, we can ignore violations

of Constraint 1 by a cycle that enters a task more than once, as long as one of the head nodes in the cycle is reachable in control flow from the other. All we need do is exclude cycles such that neither of the head nodes is control-reachable from the other. To do this, we use a modified version of depth-first search to detect cycles in the CLG.

In addition to the set of nodes already visited in the search, the modified search algorithm also maintain a set of nodes *forbidden* at the present time in the search. A node n in task T is forbidden if some node n' in the current path is a head node in task T, and there is no path between n and n' in control flow.

Maintaining the forbidden set costs at most $\mathcal{O}(|N|)$ bit operations per search step; time complexity for the search is thus $\mathcal{O}(|N|(|N| + |E_S| + |E_C|))$. For each node n, the set F_n of nodes forbidden in deadlock paths containing n can be determined by searching backward and forward along control edges, beginning at n to accumulate the set R_n of nodes reachable from n along forward or backward control edges. F_n is then simply $Task(n) - R_n$. All searching for this preprocessing step can be done in $\mathcal{O}(|N|(|N| + |E_S| + |E_C|))$ time; this is also the total time for the search. □

Theorem 4 *Detecting deadlock cycles which are valid under both constraints 1 and 3 in the sync graph of a program is NP-hard.*

Proof method: Reduction to 3-satisfiability.

Proof: Given an m-term conjunction $C_1C_2...C_m$, where each clause C_i is a disjunct of three literals $(L_i^1 + L_i^2 + L_i^3)$, and each literal L_i^j is a variable $v_k, 1 \leq k \leq n$ or its negation. Construct the following program[2], as shown in Figure 4.2.

- For each literal L_i^j corresponding to a positive variable v_k, construct the task of Figure 4.3 (a). Let $q = (i \mod m) + 1$. Task `L_i_j` corresponds to literal L_i^j, and contains five nodes. The first node, `accept s_i_j`, accepts a signal from a task corresponding to the previous conjunct $C_{(i-1) \bmod m}$, or from the *anti-ordering* task `a_i_j`. The anti-ordering task (and the control structure of `L_i_j`) prevents

[2]The program generated by this procedure may contain starvation anomalies. This is not of concern within the proof, since we are considering only the problem of detecting constrained deadlock cycles.

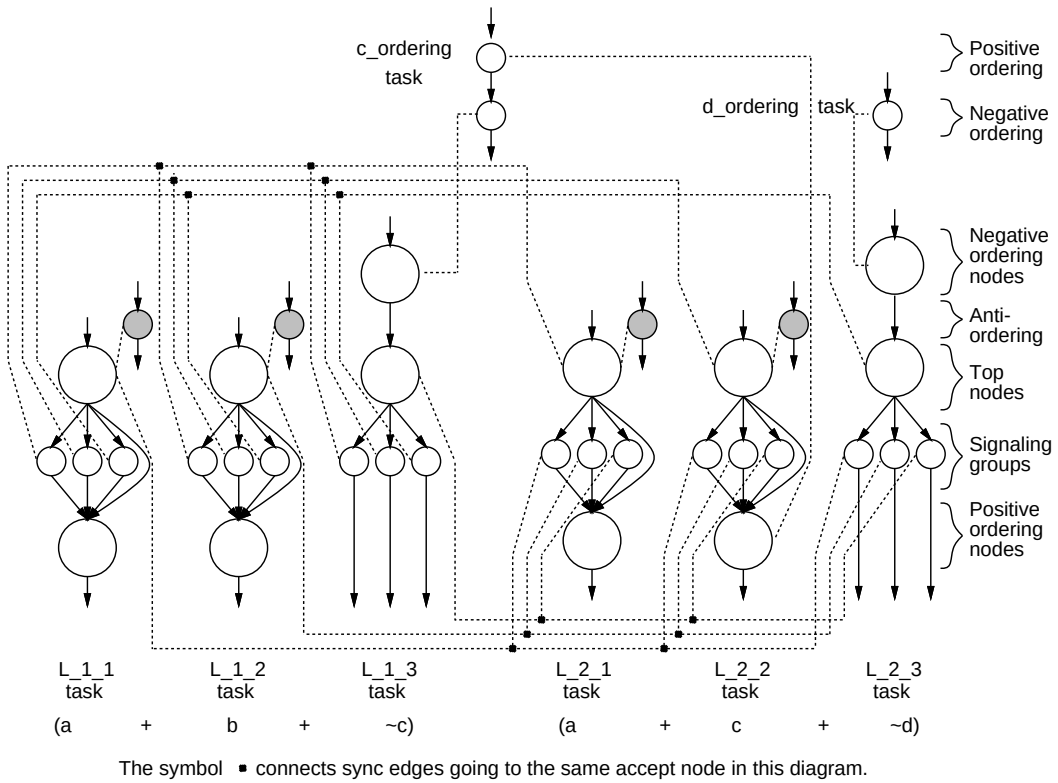

Figure 4.2: Sync graph of a program constructed corresponding to the 3-CNF expression $(a + b+ \sim c) \times (a + c+ \sim d)$.

negative literals from being ordered with respect to each other, or with respect to positive literals that do not correspond to the same variable.

The three nodes `L_q_1.s_q_1` through `L_q_3.s_q_3` form a *signaling node group*. One of these nodes may be executed, based on a random boolean value. Thus, the control flow path through `L_i_j` actually executed cannot be determined statically. The three nodes of the signaling node group create sync edges from L_i^j to each top node in the tasks created for C_q.

The last node in the task, `v_k_ordering.v_k_positive_i_j`, is used to insure that the top node for `L_i_j` is executed before the top node of any task corresponding to the negated variable $\sim v_k$. (Recall that $L_j^i = v_k$.) This node is used to establish an ordering between literal nodes corresponding to positive and negated instances of the same variable, and is called an *order-sending* node.

(a)
```
task body L_i_j is
begin
  accept s_i_j;
  if randomBoolean() then
    if randomBoolean() then
      L_q_1.s_q_1;
    else if randomBoolean() then
      L_q_2.s_q_2;
    else if randomBoolean() then
      L_q_3.s_q_3;
    end if;
  end if;
  v_k_ordering.v_k_positive_i_j;
end L_i_j;

task body a_i_j is
begin
  L_i_j.s_i_j;
end a_i_j;
```

(b)
```
task body L_i_j is
begin
  v_k_ordering.v_k_negative;
  accept s_i_j;
  if randomBoolean() then
    L_q_1.s_q_1;
  else if randomBoolean() then
    L_q_2.s_q_2;
  else if randomBoolean() then
    l_q_3.s_q_3;
  end if;
end L_i_j;
```

(c)
```
task body v_k_ordering is
begin
  accept v_k_positive_i_j;  -- Repeat for all i and j
    .                       -- where L_i_j is positive
    .                       -- variable v_k - omit if
    .                       -- no such literal
  accept v_k_negative;
end v_k_ordering;
```

Figure 4.3: Templates for positive literal and anti-ordering (a), negative literal (b), and ordering (c) tasks.

- For each literal L_i^j corresponding to a negated variable $\sim v_k$, construct the task of Figure 4.3 (b). The tasks corresponding to negated literals differ only in that the order-sending nodes are placed at the beginning of the task and send the signal `v_k_ordering.v_k_negative`.

The tasks corresponding to the literals of an entire clause form a *clause task group*. There are sync edges from the signaling node group in each task to all top nodes of the tasks in the next clause task group.

- For each variable v_k for which negative literals exist, create the *ordering task* of Figure 4.3 (c). The ordering tasks force all negative top nodes corresponding to a variable to execute after all positive top nodes for the same variable. The nodes of an ordering task are called *order-accepting* nodes.

Each literal task is connected via sync edges from the signaling node group to the top nodes of all literal tasks in the next clause task group. The top nodes of each literal task `L_i_j` are ordered with respect to other top nodes iff they correspond to positive and negated instances of the same variable. To prove this, consider the cases where two top nodes a and b may be ordered. Either both are negative, both are positive, or one is negative and the other positive.

Both positive: Top nodes of positive tasks cannot be ordered relative to each other, since all are free to execute at the start of the program.

Both negative: See Figure 4.4. The notation $a < b$ means that a always finishes before b starts, and $a = b$ if both nodes necessarily finish at the same time. $a \leq b$ if one or the other of these two cases is always true[3]. Without loss of generality, assume $a < b$. Since $a < b$, it must also be true that $a \leq b_i$ for all immediate control flow predecessors b_i of b; otherwise some predecessor would be free to complete execution and allow b to start before a finished. Node b has only one such predecessor: the order-sending node b_0. $a < b_0$ is not possible, since b_0 is free to start when its task starts. Therefore $a \leq r_b$ for all nodes r_b which make a rendezvous with b_0. There is only one such node: the negative order-accepting node for the variable v associated with b. Recall that r_b is the last node in the ordering task, since b corresponds to $\sim v$. r_b is preceded by all order-accepting nodes for positive literals for v. So some set of control flow predecessors of r_b must not be able to make a rendezvous until after a has completed. Consider such a predecessor r_c, which has a rendezvous with positive order-sending node c_1. c_1 can execute if anti-ordering node c_a makes a rendezvous with c and the control flow path from c to c_1 is taken. This sequence does not depend on any event in a's task. Therefore, there is always a way for b to start execution before a finishes, and $a \not< b$.

[3]Note: $a \not< b$ does not imply $b \leq a$, nor does $a \not\leq b$ imply $b < a$.

One negative, one positive: Let a be the positive top node and b be the negative top node. $a < b$ if they correspond to the same variable, since this is forced by the order-sending nodes and ordering task for the variable. $b \not< a$ under any conditions, since a is free to start at the beginning of the task. If a and b do not correspond to the same variable, the proof that they are not ordered is similar to that for the case of two negative top nodes.

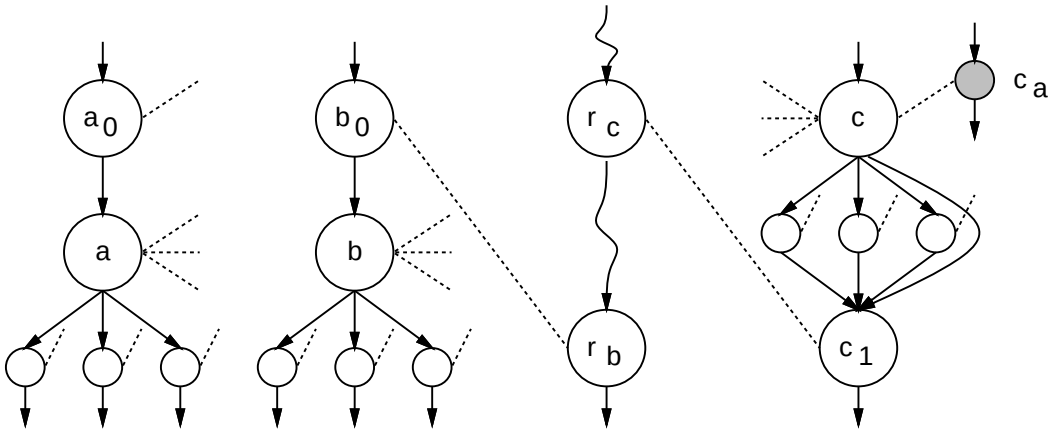

Figure 4.4: Ordered pair of negative top nodes.

NP-hard: Consider the possible cycles in the sync graph of the program. Any cycle through an ordering task must enter it at a node r and exit it to an order-sending node a_{ord}. Thus, r and a_{ord} are head nodes in the cycle and $r < a_{ord}$. Thus, any deadlock cycle involving an ordering task has a pair of ordered head nodes, and is invalid under constraint 3.

Since each anti-ordering task is connected to only one positive top node, the anti-ordering tasks do not participate in cycles. Any valid deadlock cycles must therefore involve only the sync edges between literal tasks. These sync edges go from the signaling node group of each literal task in a clause group to all top nodes of tasks in the next clause task group; sync edges similarly go from the last to the first clause task group. No sync edges go between tasks in the same clause task group. Therefore, any deadlock cycle with no sequenceable head nodes contains a top node of at least one literal task in each clause group as a head node. Further, given one literal from each clause, a cycle through the literal tasks exists corresponding to those literals.

A setting of the literal values in the conjunct will be consistent iff no pair of literal values correspond to positive and negated instances of the same variable. A cycle through the literal tasks will correspond to a feasible deadlock iff there are no two literal tasks whose top nodes are sequenceable. The top nodes of two literal tasks are sequenceable iff the literal tasks correspond to positive and negated instances of the same variable. Therefore there is a feasible deadlock in the program iff the conjunct is satisfiable. Thus, if we demand that head nodes on a deadlock execution wave not be sequenceable, approximating deadlock detection is NP-hard. □

Theorem 5 *Given an arbitrary sync graph, detecting a deadlock cycle which is valid under constraints 1 and 2 is NP-complete.*

Proof outline: Similar to Theorem 4. Given a 3-CNF expression, construct a sync graph such that a deadlock cycle without head nodes connected by sync edges exists iff the expression is satisfiable. Construct a single task for each literal as before; the ordering nodes (and control flow edges from the top node to them) are omitted. Insert a sync edge between the top nodes of any pair of tasks for positive and negative literals for the same variable. These edges cannot add new deadlock cycles; any cycle using such an edge would have to enter and exit a top node through sync edges, which is prohibited under constraint 1.

A deadlock cycle without connected top nodes corresponds to an instantiation of the variables which satisfies the expression. Such a cycle exists if the 3-CNF expression is satisfiable.

Given a deadlock cycle proposed by some algorithm, its head nodes can be readily identified and checked to see if any pair is connected by a sync edge. The problem is therefore in class NP. □

Note that the sync graph constructed in this proof cannot in general correspond to an actual program. In particular, consider chains of sync edges in the sync graph. If a program is translated into a sync graph, one end of each sync edge in such a chain must be in the accepting task for the edge's signal type. The sync graph of the proof does not observe this restriction.

Discussion.

The complexity results of Theorems 4 and 5 also hold for most useful tasking and synchronization paradigms. In particular, they hold for all paradigms in which a program P in the basic Ada subset can be emulated by a program P', such that the number of statements in P' is polynomially related to the number of statements in P.

4.3 `Accept...do` construct.

Ada [Ame83] has features that allow the rendezvous to act as a type of "remote procedure call". The task entry (or `accept`) acts as the body of the called procedure, and the signaling task acts as the caller. The syntax of a remotely callable procedure is as follows:

```
accept <signal>[(<parameter list>)] [do <statement>];
```

The code of the remote procedure is contained in the `<statement>` portion of the task entry shown. When a signaling task T sends a signal that coordinates with this task entry, T suspends execution until the body of the `<statement>` is completed. The `<statement>` can itself contain signaling and `accept` statements. Thus, task entries can be nested within each other. For obvious reasons, inner and outer nested entries cannot rendezvous with the same task.

Note that, when the `[do <statement>]` part is omitted, the `accept` construct behaves exactly as described in Section 4.2. Thus, the behavior of the `accept...do` construct actually includes the behavior of the `accept` without its `do` part.

4.3.1 Sync hypergraphs.

We utilize a representation of the program called a *sync hypergraph* to detect deadlock in programs containing `accept-do` constructs. The graph incorporates *sync hyperedges*, which represent the interactions of a call point with a called procedure. A program fragment and its corresponding sync hypergraph are shown in Figure 4.5.

If the remote procedure contains rendezvous nodes, these must be represented in the sync hypergraph of the program. In the program fragment of Figure 4.5, the `accept` statement has an attached procedure block x containing rendezvous statements. We represent a as a pair of nodes a_i and a_o corresponding to the entry ("in") and exit ("out") parts of the remote procedure respectively.

```
task T1 is
begin
  T2.sig;                          -- (s)
end T1;

task T2 is
begin
  accept sig do                    -- (a_i)
    <Body of procedure block x>    -- (x)
  end sig;                         -- (a_o)
end T2;
```

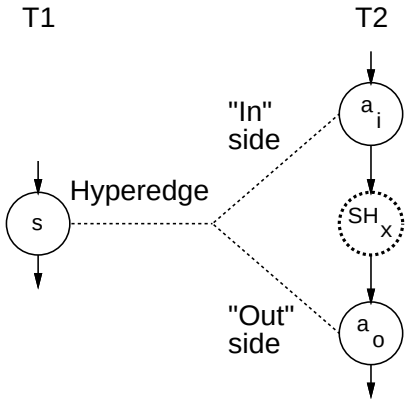

Figure 4.5: A program fragment and its sync hypergraph.

Representing control flow.

For the block x of Figure 4.5, we form a *sync subgraph* SH_x, including distinguished nodes b_x and e_x corresponding to x's beginning and end. To include SH_x in the complete sync graph, we simply replace b_x with a_i and e_x with a_o and draw sync edges appropriately, as described below. This gives us a way to recursively construct the nodes and control edges where embedded synchronization is present.

During deadlock detection, the program is initially represented by a *parse form Ada (PADA) sync hypergraph*, which includes control loops. Later steps, detailed in Section 4.3.4, transform the PADA sync hypergraph into a CLG for the program.

Representing synchronization.

Synchronization is represented by hyperedges, or edges with three endpoints. The general format of a hyperedge is (s, a_i, a_o). Such hyperedges are placed between every signaling statement s and the in and out nodes formed from each `accept` statement a of the same signal type. (A signal type y is a pair (t_y, n_y) where t_y is the receiving task and n_y is the entry name.)

For convenience, we call (s, a_i) the "in" side of the hyperedge and (s, a_o) the "out" side.

4.3.2 Execution model.

An entry call statement s can coordinate with an `accept` statement a containing a procedure block x. If x contains synchronization, then execution can advance through x without advancing past s. The previous execution model cannot represent this type of execution wave motion.

We augment the execution model of Section 4.2 with a new ENGAGED state. The model of rendezvous changes as follows:

- Initially, the execution wave contains one control successor x_T of the beginning node b for each task T.

- When nodes s and a_i are both are on the execution wave, and are connected by the "in" side of a hyperedge, and s is not in the ENGAGED state, then both are in the READY state.

  Any node on the execution wave which is not ENGAGED and is not connected to another node on the wave by a hyperedge is in state WAITING.

- Any pair of READY nodes connected by the "in" part of a hyperedge may be selected nondeterministically for firing.

- When s and a_i are selected for firing, a control successor of a_i is chosen nondeterministically and placed on the execution wave. The state of a_i changes to OFF-WAVE; the state of s changes to ENGAGED(a_o).

  Technically, the abstract memory A of Section 2.2.2 maps s to either some accept-out node a_o when s is in the ENGAGED state, or to a null value when s is not ENGAGED. In fact, we could eliminate the ENGAGED state altogether by explicitly incorporating the abstract memory in the execution model. For convenience of explanation, however, we retain the ENGAGED(a_o) notation.

- If s is in state ENGAGED(a_o), and a_o has been chosen for execution, then both s and a_o are taken off the wave, and their control successors are chosen for execution. (If s is one of its own control successors, it may be immediately placed back on the wave.)

Nodes on the execution wave can be in READY or WAITING states as before, and signaling nodes on the execution wave can also be in the ENGAGED state. A signaling node cannot exit the ENGAGED state (or the execution wave) until the out node of the accept which engaged it is also on the wave, i.e., until a control path from a_i to a_o has been executed.

4.3.3 Infinite wait anomalies in the `Accept...do` model.

An execution wave which contains no READY nodes, but which contains some node other than the distinguished program end node e, has an infinite wait anomaly. All nodes on the wave are thus WAITING or ENGAGED, or are the distinguished node e.

Lemma 19 *An anomalous execution wave must include at least one node in the WAITING state.*

Proof: Suppose that some execution wave contained only copies of node e and nodes in state ENGAGED. If all nodes are e, then the program has terminated successfully, so at least one must be ENGAGED. Let s be the signaling node on the execution wave which was the last one to become ENGAGED. Let a_i be the accept node s coordinated with when s became ENGAGED. By construction, any control path from a_i to e must include a_o. By the transition rules above, a_o cannot leave the execution wave until s does. Thus, either a_o is on the wave or some node n on a control path from a_i to a_o is. By the transition rules, accept nodes cannot enter the ENGAGED state, so a_o is not on the execution wave. For the same reason, n must be a signaling node in the ENGAGED state. By construction, n cannot be a control predecessor of a_i, so n must have entered the execution wave after s and a_i coordinated. Before n entered the wave, its state was OFF-WAVE. Therefore, n entered the ENGAGED state after s, and s cannot have been the last node to become ENGAGED. □

Characteristics of anomalies.

A *deadlock anomaly* is characterized by a group of tasks on an anomalous execution wave, such that each is waiting for some other task in the group to proceed. A *starvation*

anomaly is indicated by a node $n \neq e$ on the wave, such that n cannot coordinate with any other rendezvous node that can be reached in control flow from nodes currently on the wave. Tasks not directly deadlocked or starved may have to wait for deadlocked or starved tasks to complete execution, and thus be unable to proceed.

Naturally, the question arises: Given the new execution model, are deadlocks and starvation still the only possible types of infinite wait anomalies? The answer is yes; the proof is identical to that of Theorem 1 of [MR90a].

Deadlocks involving hyperedges.

A deadlock is characterized by a group of tasks that mutually wait on each other. Consider a subset D of the head nodes of deadlocked tasks in an execution wave. Every member of D must be a signaling node s or an accept-in node a_i. (If any accept-out node a_o appears on the execution wave, it can rendezvous with some signaling node s which is ENGAGED to a_o's corresponding accept-in node.)

In the deadlock cycle D, there are only three ways in which one task $T1$ can wait on another task $T2$. These *wait modes* are shown in Figure 4.6.

In the first mode (Figure 4.6(a)), node s of task $T1$ is WAITING, and can rendezvous with a_i and a_o of task $T2$. Node a_i is OFF-WAVE, since it cannot execute until t (and possibly other intervening nodes in $T2$) have executed. Here, a_i is a tail node and s is a head node.

The second mode (Figure 4.6(b)) is really the reverse of mode (a). Node a_i in task T1 is WAITING, and can rendezvous with s in T2, but s is OFF-WAVE. Modes (a) and (b) correspond to modes which would occur if a_i and a_o had no intervening rendezvous nodes, i.e., the modes of the simplified model in which `accept` statements have no associated `do` blocks. Here, a_i is a head node and s is a tail node.

The third mode (Figure 4.6(c)) can occur only when `accept-do` statements are permitted. Here, s is ENGAGED to a_i, and therefore cannot complete until its corresponding accept-out node a_o executes. However, a_o is prevented from completing because some intervening node t in task T2 is WAITING on or ENGAGED to some

Wait Mode	m	n	Resulting Edge
(a)	s	a_o	(a_o, s)
(b)	a_i	s	(s, a_i)
(c)	s	a_o	(a_o, s)

Table 4.1: Modes of the *Waits* relation.

OFF-WAVE node. Note that mode (c) includes cases where s is embedded in an `accept-do`, or where there is an `accept-do` surrounding a, or one inside a and surrounding t. Here, a_o is a tail node and s is a head node.

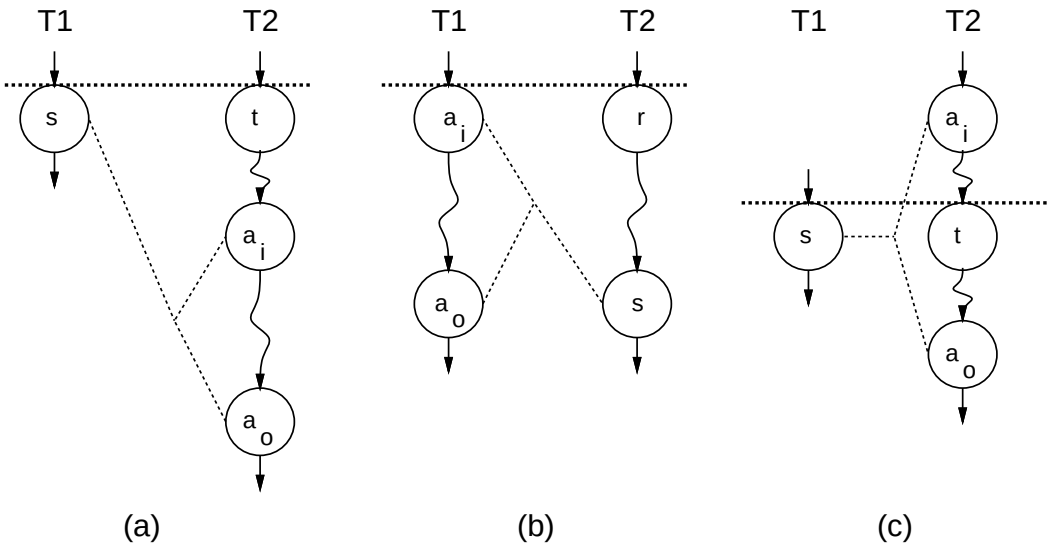

Figure 4.6: Deadlocks involving hyperedges.

"Waits on" relationship and hyperedges.

We define a relation $Waits(E, m, n)$ which means, "m cannot complete execution in execution wave E, but m could complete execution if n were on E, and n is a control successor of some node in E other than m." Informally, m waits on n in E. This relationship allows us to determine which transitions along sync hyperedges are valid in a deadlock cycle.

Table 4.3.3 summarizes, for each wait mode, which node waits on which other node.

In tracing deadlock cycles, we trace forward on control edges and from tail to head

nodes (n to m) wherever m waits on n. Thus, the edges in the "Resulting Edge" column represent the permissible transitions over hyperedge (s, a_i, a_o) when tracing a deadlock cycle.

4.3.4 Hyperedge cycle location graphs.

In cycle detection, we use a *hyperedge cycle location graph*, or HCLG, corresponding to the sync hypergraph. The HCLG for the sync hypergraph is a particularization of the cycle location graphs of Section 2.3.5. The HCLG is constructed so that any cycles must satisfy constraint 1 of [MR90a], i.e., that any cycle that enters a task should traverse at least one control edge in that task. Such a construction eliminates cycles which are spurious under constraint 1.

The first step in constructing the HCLG is to create the *cycle detection form Ada (CDADA) sync hypergraph*, from the initial PADA sync graph emitted by the parser. This involves unrolling loops, outermost twice, and eliminating control backedges, as outlined in Section 2.3.5. The CDADA sync graph is then transformed into an HCLG, via a node-splitting operation.

Figure 4.7 shows the transformation for signaling nodes and `accept-do` structures. Node s is split into engage and disengage nodes, s_e and s_d respectively. Edges between tasks correspond to sync hyperedges. Note that any path entering and then exiting a task in the HCLG must traverse at least one edge in that task, such that the edge corresponding to a control edge in the sync hypergraph. Subgraph x corresponds to an embedded `do` block, and is recursively expanded into subgraph x' in the HCLG.

Deadlock task interactions in modes (a) and (b) correspond to edges (a_o, s_d) and (s_e, a_i) respectively. Interactions in mode (c) have some node in x' as a head node, and traverse (a_o, s_d).

4.3.5 Scope depth and feasible deadlocks.

In [MR90a], constraint 2 states that no two head nodes of a deadlock cycle may rendezvous with each other. Constraint 4 says that there must be some way to execute

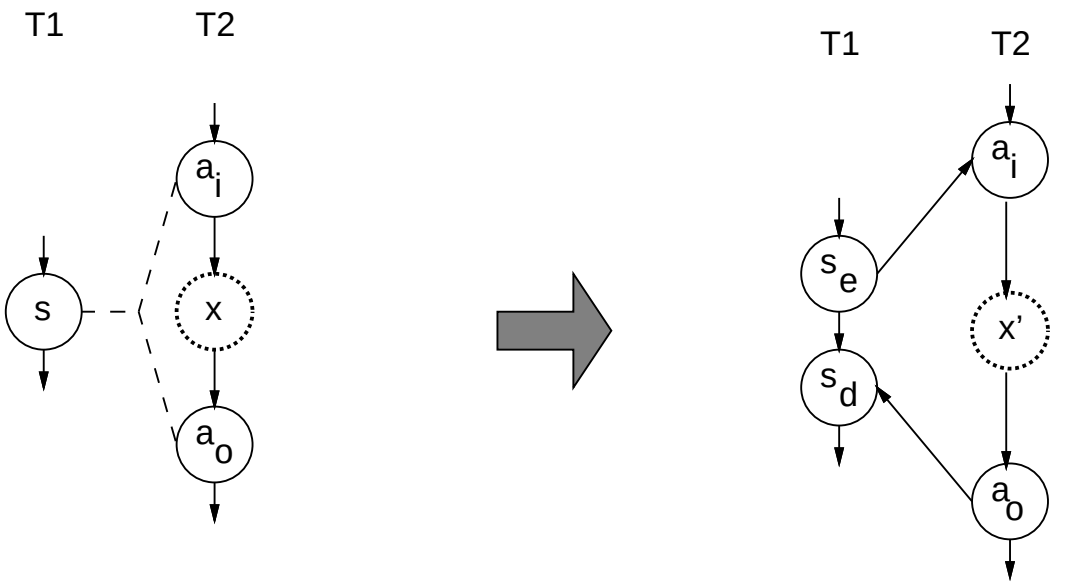

Figure 4.7: Hyperedge cycle location graph (HCLG) transform.

the nodes of the deadlock so that no node outside the deadlock can break the deadlock. The structure of the `accept-do` construct weakens both of these constraints. We outline a reinterpretation of constraints 2 and 4 in the presence of `accept-do` statements containing rendezvous nodes.

Constraint 2 must change because of mode (c) coupling. In Figure 4.6, node s may have the same signal type as node t. No rendezvous will occur, because s is already ENGAGED to a_o.

Similarly, 4 must change when `accept-do` is used. There may be no way in which to execute a deadlock such that some pair of nodes which can break the deadlock may be of the right signal type to rendezvous — but the sending node might always be in state ENGAGED, and thus may be unable to break the deadlock.

Let the *scope depth* (denoted $Scope(n, y)$) of a node n be the depth to which it is embedded in `accept-do` constructs of signal type y. $Scope(n, y)$ can easily be computed during the parse of the program. For any feasible execution wave E, the number of ENGAGED signaling nodes of type y equals the sum of the scope depths $Scope(n, y)$ of all nodes n on the wave. Call this summation $Scope_{set}(E, y)$; call the number of signaling nodes of type y in E, $SigN_{set}(E, y)$. Let predicate $Accept?(E, y)$ where $y = (t_y, m_y)$ be

true if $E[t_y]$ is an accept node of signal name m_y.

In any execution wave E, for all signal types y,

$$Scope_{set}(E, y) \leq SigN_{set}(E, y) \tag{4.1}$$

since all signaling nodes required to increase scope depth must still be ENGAGED and therefore present on the execution wave. Also, if E is anomalous, then

$$Accept?(E, y) \Rightarrow Scope_{set}(E, y) = SigN_{set}(E, y) \tag{4.2}$$

since otherwise some accept-in node of type y in E could rendezvous with a non-ENGAGED signaling node. Since t_y is the only task that can contribute to $Accept?(E, y)$, we know that $Scope_{set}(E, y) = Scope(E[t_y], y)$. Therefore,

$$Accept?(E, y) \Rightarrow Scope(E[t_y], y) = SigN_{set}(E, y). \tag{4.3}$$

For the special case where $Scope_{set}(E, y) = 0$ for all E and y (i.e., no embedded rendezvous are permitted), then we substitute in Equation (4.1) to obtain

$$SigN_{set}(E, y) \geq 0. \tag{4.4}$$

This is clearly true for all execution waves E. If E is anomalous, then we can substitute in Equation (4.2) to obtain

$$Accept?(E, y) \Rightarrow SigN_{set}(E, y) = 0. \tag{4.5}$$

This implies the original constraint 2 of [MR90a], that no pair of head nodes in a deadlocked execution wave (and hence a deadlock cycle) can rendezvous.

We would like to derive from Equation (4.1) and Equation (4.3) those conditions which can be checked given *only* knowledge of the deadlock cycle and the total number of active tasks. Let D be the set of head nodes of a deadlocked execution wave E.

Clearly,

$$Scope_{set}(D, y) \leq SigN_{set}(D, y) + |E| - |D| \tag{4.6}$$

since otherwise the total number of signaling nodes of type y which could possibly be on E would not be sufficient to drive D to the given scope depth. A more restrictive condition is

$$\sum_{y \in Sigs} [Scope_{set}(D, y) - SigN_{set}(D, y)] \leq (|E| - |D|), \tag{4.7}$$

(where *Sigs* is the set of signal types used in the program), since there must be sufficient signaling nodes on E to drive all of the nodes in D to their scope depth. This is a condition on the feasibility of D.

For the condition on accept nodes of Equation (4.3), we have that

$$Accept?(D, y) \Rightarrow Scope(D[t_y], y) = SigN_{set}(E, y). \tag{4.8}$$

Therefore,

$$Accept?(D, y) \Rightarrow Scope(D[t_y], y) - SigN_{set}(D, y) = SigN_{set}(E - D, y) \tag{4.9}$$

where $0 \leq SigN_{set}(E - D, y) \leq |E| - |D|$. Therefore, it must also be true that

$$Accept?(D, y) \Rightarrow 0 \leq Scope(D[t_y], y) - SigN_{set}(D, y). \tag{4.10}$$

If Equation (4.10) is false for some D, then two nodes of D can rendezvous and D is thus not a deadlock.

4.3.6 Impact on constraints.

We examine the constraints of Section 4.2.4 one by one to determine whether they hold, and how they are changed, by the new model of synchronization.[4]

1. *Each node in D is transitively coupled to all others in D.*

[4]The original constraints are in italics.

We have shown how each of the wait modes is modeled by hyperedges in the sync hypergraph.

Thus, there is a cycle in the sync graph, which includes the nodes of D, such that, for every node $r \in D$ in task T:

(a) *The path enters node r through a sync edge.*

Paths in the sync hypergraph can only go between tasks by traversing sync hyperedges. In a valid deadlock cycle, the hyperedges are taken from a signaling node to an accept-in node, or from an accept-out node to a signaling node.

(b) *The path traverses at least one control flow edge in task T.*

When a path enters a task in the HCLG, it cannot exit the task without traversing at least one edge corresponding to a control edge in the sync hypergraph.

(c) *The path leaves T via a sync edge and does not re-enter T.*

In straight-line code without conditional branching, any cycle which enters a task more than once corresponds to one which enters the task only once. In code with conditional branching (but no loops), any path that enters a task T more than once either corresponds to some other path which enters T only once, or includes a pair of head nodes which cannot happen together.

The above observation is based on reachability in control flow; if head node n can be reached from head node m, then any path which enters the task at both m and n also corresponds to one which enters only at m. If m and n are mutually unreachable in control flow, then they are not co-executable. This observation is unchanged in the sync hypergraph.

In Section 4.3.4, we showed how the HCLG transformation is used to eliminate control loops while retaining all possible deadlock paths. After this transform, our observation above holds on the HCLG.

2. *No two head nodes of the deadlock cycle can rendezvous with each other.*

This constraint is on the head nodes of a deadlock cycle alone. The nodes can rendezvous with each other only if the signaling node is not always ENGAGED. Equation (4.10) expresses a subset of this condition which is determinable from only the proposed deadlock and the number of tasks; it is a necessary, but not sufficient, condition for the constraint. We repeat Equation (4.10) here:

$$Accept?(D, y) \Rightarrow 0 \leq Scope(D[t_y], y) - SigN_{set}(D, y).$$

3. *All head nodes may execute at the same time.*

   (a) *The nodes in D must represent statements whose instances can happen together.*

   (b) *All nodes in D are co-executable in the sense of Callahan and Subhlok [CS88].*

   In the presence of `accept-do` statements, we have new information on co-executability of the nodes in D. It must be possible to have enough ENGAGED nodes of each signal type on the execution wave to drive the nodes in D to their scope depth with respect to all signal types. This implies Equation (4.7), which we repeat here:

$$\sum_{y \in Sigs} [Scope_{set}(D, y) - SigN_{set}(D, y)] \leq (|E| - |D|)$$

   .

4. *The nodes of D must be able to execute concurrently as part of an execution wave E such that none of the nodes in $E - D$ can rendezvous with each other or with any member of D.*

   Equation 4.7 is also relevant to this constraint, since it determines whether D can execute, given the size of $E - D$.

4.4 Select construct.

Ada has a *selective wait* mechanism that allows a task to select from a number of available `accept` statements. The selective wait mechanism is available through use of the `select` construct, whose syntax is:

```
select     [when <guard>] <accept-alt> [<sequence of statements>]
       [or [when <guard>] <accept-alt> [<sequence of statements>]]...
       [else <sequence of statements>]
end select;
```

where `<accept-alt>` is of one of the following forms:

```
accept    <signal> [do <sequence of statements>] -- or --
delay     <expression>                           -- or --
terminate
```

The selective wait may contain an `else` clause, a set of `delay` alternatives, or a `terminate` alternative, but no pair of these constructs are permitted in the same selective wait statement.

The *guard condition* is a Boolean expression. Any of the `accept` clauses without a guard, or whose guard is *open* (true) may rendezvous when execution reaches the selective wait and some other task has made the proper entry call. If all guards are *closed* (false), and there is no `else` clause, then a runtime exception is raised.

We refer to the `accept` statements of a conditional entry construct as *sisters* of each other. The optional `else` statement, if present, would execute if none of the sister accepts was available for a rendezvous. Thus, the `accept` clauses of a `select...else` construct cannot participate as the head nodes of a deadlock, since the `else` clause could always be taken even when none of the sisters can rendezvous. (A rendezvous inside the `else` clause may be a head node in a deadlock cycle, but only if none of the open `accept` statements can rendezvous.)

Similarly, an open or unguarded `delay` alternative, if present, would execute if none of the open sisters could rendezvous before the specified delay period had expired.

Thus, no set of sisters with an unguarded `delay` alternative can be the head nodes of a deadlock. However, the sequence of statements following a `delay` statement may contain rendezvous, and may ordinarily participate in a deadlock as either head or tail nodes.

4.4.1 Representing conditional entries.

Figure 4.8 shows the translation of a selective wait statement into a sync hypergraph. The `accept` clauses `u1` through `um` are all unguarded, while `accept` clauses `g1` through `gn` have guard clauses.

A pair of nodes is constructed for each entry. The nodes of each `accept` clause are marked as not co-executable with those of its sisters and with the nodes of the `else` clause and `delay` alternatives. Other markings, which are described below, denote limits on the number of signaling nodes of various types in any valid deadlock cycle containing nodes in the `select` statement.

Sections 4.4.2 and 4.4.3 explain the reasons for the construction of the sync hypergraphs for `select` statements, and detail some assumptions made during deadlock analysis.

4.4.2 Conditional entries without `else` or `delay`.

The three modes of interaction previously outlined for `accept-do` statements also apply to `select` statements. In mode (a), the `select` statement has not been reached in execution flow, but some node s on the wave might be able to complete if it were. In mode (b), the `select` statement has been reached in execution, but none of its open `accept` clauses can rendezvous. In mode (c), one of the open `accept` clauses $a_i ... a_o$ has been engaged, but a node t contained in that clause cannot proceed.

Guard clauses.

Guard clauses for the `accept` statements are optional. The unguarded `accept` statements are always open.

```
-- Control predecessors
select
  -- Unguarded entries
  accept u1
    do <seq_u1>
  end u1;
  <seq_v1>
or
  ...
or
  accept um
    do <seq_um>
  end um;
  <seq_vm>
or
  -- Guarded entries
  when <guard_1>
    accept g1
      do <seq_g1>
    end g1;
    <seq_h1>
or
  ...
or
  when <guard_n>
    accept gn
      do <seq_gn>
    end gn;
    <seq_hn>
or
  when <guard_d1>
    delay <time>;
    <seq_delay1>
or
  ...
or
  when <guard_dk>
    delay <time>;
    <seq_delayk>
else
  <seq_else>
end select;
-- Control successors
```

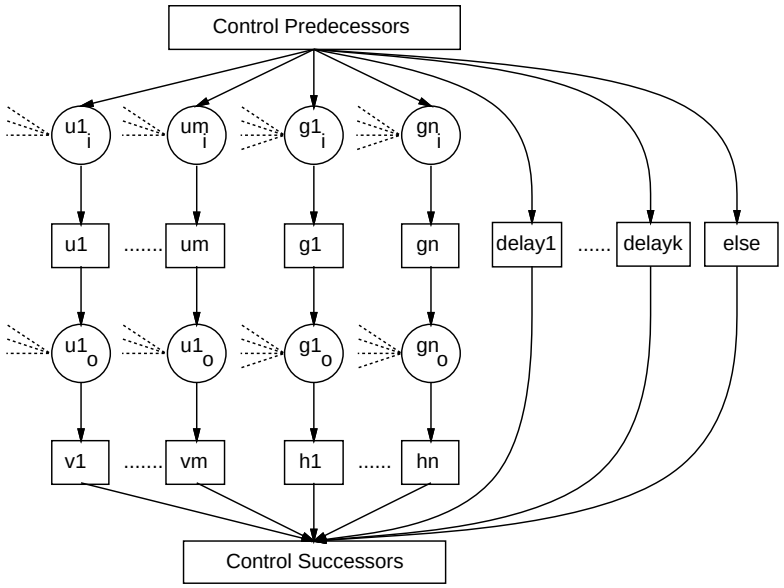

Figure 4.8: Representing conditional entries in the sync hypergraph.

In special cases, it may be possible to perform some kind of case analysis on the guard clauses, to determine which may be open and which may be closed at any given time in program execution. In general, performing an exact analysis of this sort is an intractable problem, so we must make the most conservative assumptions possible about which guards are open.

The most conservative assumption here is the one which gives the greatest possibility of deadlock, i.e., that the minimum possible number of sisters of each hypothesized head node are available for rendezvous. We assume that all unguarded sisters of a hypothesized head node are always open, but that the only open guarded sister is the hypothesized one (if the hypothesized head node is guarded).

Commitment and sync hyperedge paths.

We examine possible sync hyperedge paths according to task interaction mode, to determine what assumptions about open guards are precise and conservative. In this analysis, node a_i represents one of several `accept` alternatives and a_o represents the end of the `do` block of the *one* alternative selected for execution (if any are). Each of the alternative `accept` statements may be either guarded or unguarded.

In mode (a) task interactions, a deadlock path may go from any of the alternatives of a_i to s. To be conservative, we must assure that the maximum number of possible cycles will be followed. We therefore assume that any of the `select` alternatives of a_i would be open, if the `select` could be reached in execution. That is, any of the `select` alternatives may be used as a tail node.

In mode (b) task interactions, if any of the `accept` alternatives is open and READY to coordinate, there cannot be a deadlock. Constraint 2 leads us to discard any proposed deadlock cycle in which some signaling node can rendezvous with any of the unguarded alternatives. For conservativeness, we should not discard proposed deadlock cycles in which a signaling node can rendezvous with a guarded alternative, since the alternative may be closed by the guard. (See Section 4.4.2.)

In mode (c) task interactions, we have marked a signaling node ENGAGED and

partially executed the `do` clause of an `accept` statement. In the `select` construct, we need to make sure that the `do` clause executed corresponds to the `accept` statement which was actually engaged. We must therefore construct the sync hypergraph so that this will always be the case.

To represent the possible control edge paths in deadlock cycles, we insert control edges from each control predecessor of the `select` statement to the accept-in node of each `accept` alternative. Likewise, we insert control edges from the last nodes of each `accept` alternative to each control successor of the `select` statement. (See Figure 4.8.)

Scope depth and conditional entries.

If any of the "in" nodes of any of the sister entries of a selective wait statement are on the execution wave, then the "in" nodes of *all* sisters are effectively on the wave. Thus, no valid deadlock cycle can contain both the "in" node of a `select` clause and a signaling node that can rendezvous with one of its unguarded sisters.

Say that we have hypothesized a deadlock cycle D including accept-in node a_i, which has an unguarded sister node u_i of type y. If $Scope(u_i, y) < SigN_{set}(D, y)$, then u_i can rendezvous and D is not a true deadlock. Note that $Scope(u_i, y) = Scope(a_i, y)$, since both nodes are nested inside the same set of `accept-do` constructs.

Let $UngSis?(D, y)$ be a a predicate which is true iff D contains an accept-in node with type y, or a sister node of any unguarded accept-in node of type y. Remembering that task t_y contains all accept nodes of type y and all sisters of such nodes, we can rephrase Equation (4.10), restricting it to hold for the unguarded sisters of nodes in D, as well as D itself:

$$\forall y \overset{\text{such that}}{} UngSis?(D, y) : 0 \leq Scope(D[t_y], y) - SigN_{set}(D, y) \tag{4.11}$$

4.4.3 Conditional entries with `else` clauses or `delay` alternatives.

The `else` clause will typically not include any coordinations with other tasks. However, the ANSI Ada specification [Ame83] does not rule out this possibility, so it must be

considered for the sake of completeness. We first examine the impact of `else` clauses without rendezvous on possible deadlocks, and then look at the effects of including rendezvous statements in the `else` clause.

`Else` clauses and `delay` alternatives without rendezvous statements inside the clause.

Consider a conditional entry's accept node a, as shown in Figure 4.6. In mode (a) interactions, s can rendezvous with one of the `accept` clauses of the unexecuted conditional entry. In mode (c), one of the `accept` clauses has already been committed to execution, so s would still wait on a_o.

When a conditional entry with an `else` clause is present on the execution wave, mode (b) task interactions do not lead to deadlock. In such a case, execution will fall through to the `else` part. Thus, the hyperedge need never be traversed from s to a_i.

To represent `select` statements with an `else` clause or `delay` alternative with no rendezvous, we insert a control edge from each predecessor of the `select` statement to each successor. We also note that the accept-in node of each `accept` alternative cannot be a head node. (The sync hypergraph of Figure 4.8 has this construction if the `else` clause body contains no rendezvous.)

`Else` clauses and `delay` alternatives containing head nodes.

A *frontier node* in the sync hypergraph of a clause is defined as a node which is reached in control flow only from the beginning of the clause. A frontier node of an `else` clause or `delay` alternative may be the head node of a deadlock cycle only when none of the open `accept` clauses is ready to rendezvous when the `select` statement is executed (or before the delay period has expired). This may happen only when none of the open `accept` clauses can rendezvous with any of the signaling nodes on the execution wave at the time the `select` clause was executed, or within the delay period following the start of execution.

We might be tempted to conclude from this that the frontier nodes of an `else`

clause or `delay` alternative cannot be in a deadlock cycle along with any node that could rendezvous with an unguarded `accept` alternative. However, this is not true. It is possible, though not intuitive, that a frontier node may be co-executable with a signaling node that can rendezvous with one of the unguarded sister `accept` statements. For an example, see Figure 4.9. Suppose that, at the time the `accept` statement in task T1 started execution, node n in task T2 was waiting to rendezvous. At that time, no rendezvous to any of the open sisters was available, so the `else` statement started execution, and control flow reached f_1. After that, in T2, n completed its rendezvous and s started execution. Even though s would have prevented f_1 from executing if it had been executing before the `accept` statement started, s can execute simultaneously with f_1.

It is possible that there may be a deadlock with a head node in an `else` or `delay` clause and a tail node that is not in the same clause. Therefore, we insert control edges from the last nodes in the `else` or `delay` clause to the control successors of the `select` statement. (See Figure 4.8.)

The `else` clause or `delay` alternative of a conditional entry is not co-executable with any of the conditional entries themselves. We mark the possible head nodes of the `else` clause or `delay` alternative in this way.

`Else` clauses and `delay` alternatives containing tail nodes.

A tail node represents a rendezvous node which could break a deadlock, if that tail node were present on the execution wave. From this viewpoint, and also from the conservative viewpoint of forming the greatest number of deadlock cycles, any rendezvous nodes inside `else` clauses or `delay` alternatives should be permitted to act as tail nodes wherever it is feasible that they can do so, even if the head node is not contained within the `else` clause. Thus, we should construct the sync hypergraph so that there are control paths from the control predecessors of the conditional entry to the nodes of the `else` clause or `delay` alternative. (See Figure 4.8.)

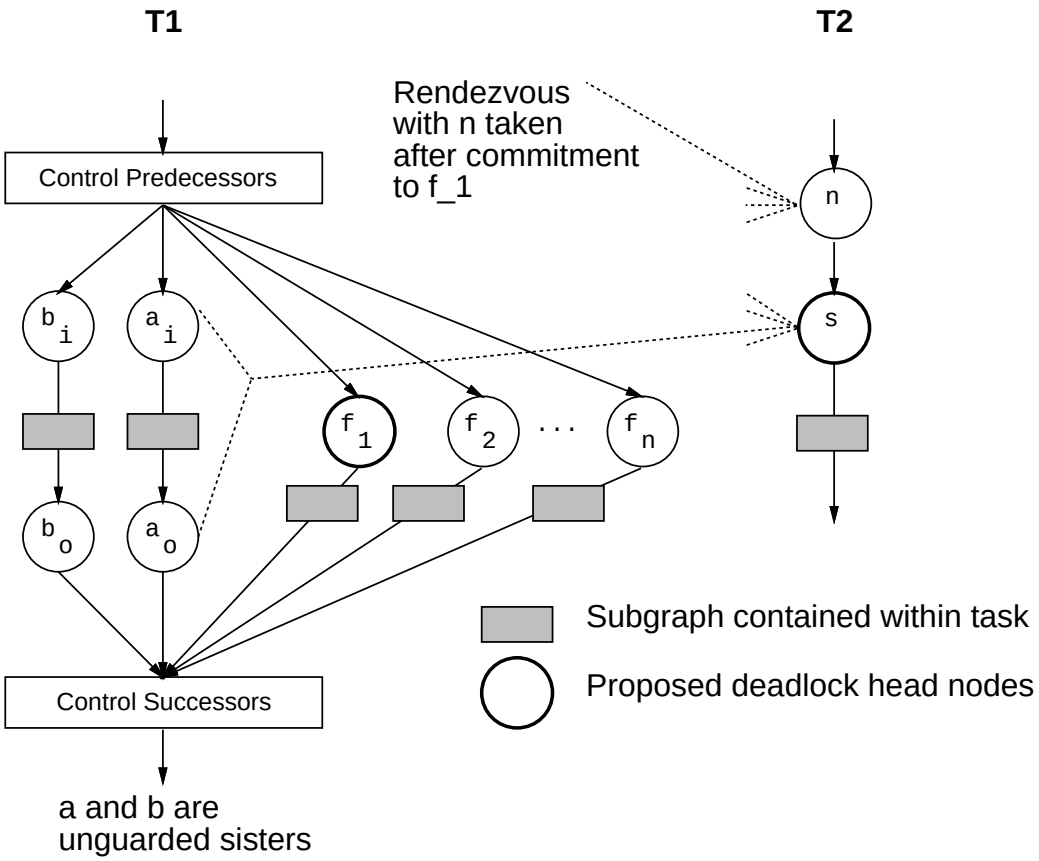

Figure 4.9: Co-executability and `else` clauses of selective wait statements.

4.4.4 Conditional and timed entry calls.

A *conditional entry call* mechanism also exists in Ada. It allows an entry call to be executed, depending on whether an `accept` is available. Its syntax is:

```
select <entry call> [<sequence of statements>]
       else <sequence of statements>
end select;
```

The *timed entry call* is a similar construct of the form:

```
select <entry call> [<sequence of statements>]
       or delay <expression> <sequence of statements>
end select;
```

Representing conditional entry calls.

Figure 4.10 shows how a conditional or timed entry call is represented in the sync hypergraph. The `else` part is treated as a conditional statement which may be executed only if the entry call cannot be accepted. Thus, the entry call and subsequent nodes are marked as not co-executable with the nodes of the `else` clause. The conditional entry call is marked as such, so that it will not be included as a head node in deadlock cycles.

Conditional entry calls and deadlock.

If the entry call cannot rendezvous, then execution will proceed to the `else` part of the conditional call, or the delayed sequence of statements in the timed entry call. The entry call thus cannot possibly be a head node in a deadlock cycle. However, rendezvous nodes embedded in the `else` or `delay` part might be head nodes, if they do not themselves have `else` clauses. The same is true for rendezvous embedded within the sequence of statements following the call. Therefore, the construction of conditional entry calls is similar to that of conditional entries.

The nodes of the `else` clause are not co-executable with the conditional entry call

```
-- Control predecessors
select
  t.s;              -- Node s
  <seq_cond_call>;
else
  <seq_else>;
end select;
-- Control successors

        -- or --

-- Control predecessors
select
  t.s;              -- Node s
  <seq_cond_call>;
or delay <expr>
  <seq_else>;
end select;
-- Control successors
```

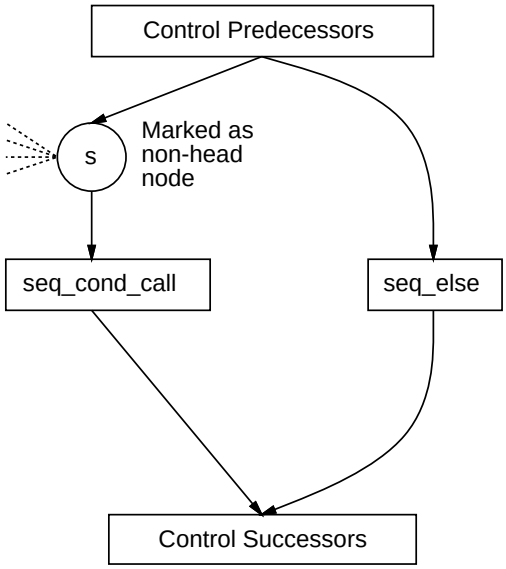

Figure 4.10: Representing conditional entry calls in the sync hypergraph.

or any nodes in the sequence of statements that follow it. However, for reasons similar to those of 4.4.3, a frontier node of an `else` clause in a conditional entry call may be co-executable with an `accept` statement which can rendezvous with the conditional call. Therefore, the construction of the sync hypergraph for conditional entry calls is similar to that for conditional entries.

4.4.5 Impact to constraints.

If a proposed deadlock cycle D is a valid deadlock cycle, then:

1. *Each node in D is transitively coupled to all others in D. Thus, there is a cycle in the sync graph, which includes the nodes of D, and such that, for every node $r \in D$ in task u:*

   (a) *The path enters node r through a sync hyperedge, going along a sync hyperedge as follows:*

      i. *From an accept-out node to a signaling node.*

         Since conditional entry calls are never deadlock head nodes, the signaling node in this case may not be a conditional entry call.

ii. *From a signaling node to an accept-in node.*

The accept-in node may not have a `else` clause. If it does, it cannot be a head node in a deadlock cycle.

(b) *The path traverses at least one control flow edge in task u.*

(c) *The path leaves u via a sync hyperedge (as above) and does not re-enter u.*

2. *No two head nodes of the deadlock cycle can rendezvous with each other. This implies Equation (4.10):*

$$Accept?(D, y) \Rightarrow 0 \leq Scope(D[t_y], y) - SigN_{set}(D, y).$$

Equation (4.11) generalizes Equation (4.10) as follows:

$$\forall y \overset{\text{such that}}{} UngSis?(D, y) : 0 \leq Scope(D[t_y], y) - SigN_{set}(D, y)$$

3. *All head nodes may execute at the same time.*

(a) *The nodes in D must represent statements whose instances can happen together.*

(b) *All nodes in D are* co-executable *in the sense of Callahan and Subhlok [CS88]. This implies Equation (4.7):*

$$\sum_{y \in Sigs} [Scope_{set}(D, y) - SigN_{set}(D, y)] \leq (|E| - |D|).$$

4. *The nodes of D must be able to execute concurrently with some set of remaining nodes $E - D$ such that none of the nodes in $E - D$ can rendezvous with each other or with any member of D.*

If any node a_i of D is a conditional entry, then all conditional entries in its `select` statement, which are unguarded or have open guards, are available for rendezvous. If one of these is an unguarded entry $u_i = (t, y)$, and $SigN_{set}(D, y) >$

$Scope(u_i, y)$, then D is always broken by a rendezvous with u_i, even though u_i does not necessarily appear in D.

Chapter 5

The *CHT* relation in Ada programs.

5.1 Computing *CHT* in the Ada model.

In this section, we discuss the refinements which yield *CHT* information for the Ada subset. These include *B4* analysis, pinning analysis, critical section analysis, and remote procedure call analysis.

5.1.1 *B4* analysis.

Given a pair of statements in the program, it may be possible to show that all instances of one statement must always happen before any instance of the other. We do this by establishing that there must always be a chain of control edges and/or synchronizations from one to the other, if both statements occur. This form of analysis is called *B4 analysis*, and is closely related to the *SCPreserved* analysis in loopless programs performed by Callahan and Subhlok [CS88].

More recently, Duesterwald and Soffa [DS91], [Due91] formulated a *before* analysis of Ada-like programs; our analysis is similar to theirs in principle, although they use a different graph representation and execution model. Our sync hypergraphs extend their representation by including the `accept-do` construct, selective wait, and conditional entry calls[1]. Though [Due91] proposes to eliminate spurious sync edges using their ordering relation, they do not mention doing so iteratively, or in combination with other refinements. However, [DS91] represents recursion and process creation, while we require that subroutines be fully inlined and all processes created at the start of the program. While we find little or no use of recursive process creation in practice, we hope to eliminate these restrictions in future research.

Lastly, while [CS88] and [DS91] presented their analyses as sets of data flow equations, we present *B4* more formally here as a lattice framework, and derive its convergence time and accuracy properties based on lattice-theoretic arguments.

---

[1] While the authors refer to selective wait, they do not provide examples of its representation.

B4 lattice framework.

A *B4 summary* is defined as a pair (C, S) where C and S are sets of nodes in a parse form Ada sync hypergraph. To give the mnemonic, the nodes in C propagate (in the lattice framework) from control predecessors, and those in S propagate from nodes that can synchronize with control ancestors.[2]

Thus the lattice space L_{B4} is:

$$L_{B4} = 2^N \times 2^N.$$

Intuitively, in a *B4* summary (C, S) for node x, C represents the *B4* information propagating from x's control predecessors, while S represents *B4* information from the synchronizations immediately preceding the start of any instance of x. The meet operation is element-wise set intersection; thus, the $\sqsubseteq$ operator is element-wise set containment. For convenience, denote $(\emptyset, \emptyset)$ as $B4_{\perp}$ and (N, N) as $B4_{\top}$.

Lemma 20 *The height of the L_{B4} lattice space is* $2|N| + 1$.

Proof: The $\sqsubseteq$ operator is defined for L_{B4} as element-wise set containment. If $a = (C_a, S_a), b = (C_b, S_b) \in L_{B4}$ are such that $a \sqsubset b$, then either $C_a \subset C_b$ and $S_a \subseteq S_b$, or $C_a \subseteq C_b$ and $S_a \subset S_b$. Thus, there must be at least one element in S_b not contained in S_a, or at least one element in C_b not contained in C_a. The longest possible ordered chain of elements of L_{B4} would thus start with $B4_{\perp}$, and be such that either C_i or S_i (but not both) has one more node than C_{i-1} (respectively S_{i-1}). The length of such a chain is thus the total maximum cardinality of C and S plus one. □

Completor edges.

Before defining the function space for the *B4* lattice framework, it is useful to discuss the propagation of *B4* information along sync hyperedges. Consider node x in Figure 5.1. We might expect to obtain some information on the synchronizations that precede x by

[2]By abuse of notation, we sometimes say $B4(n) = X$ to mean that, where $B4(m) = (C, S)$, $X = C \cup S$.

examining nodes connected to the sync hyperedges reaching its control predecessors, a_i^4 and s^5.

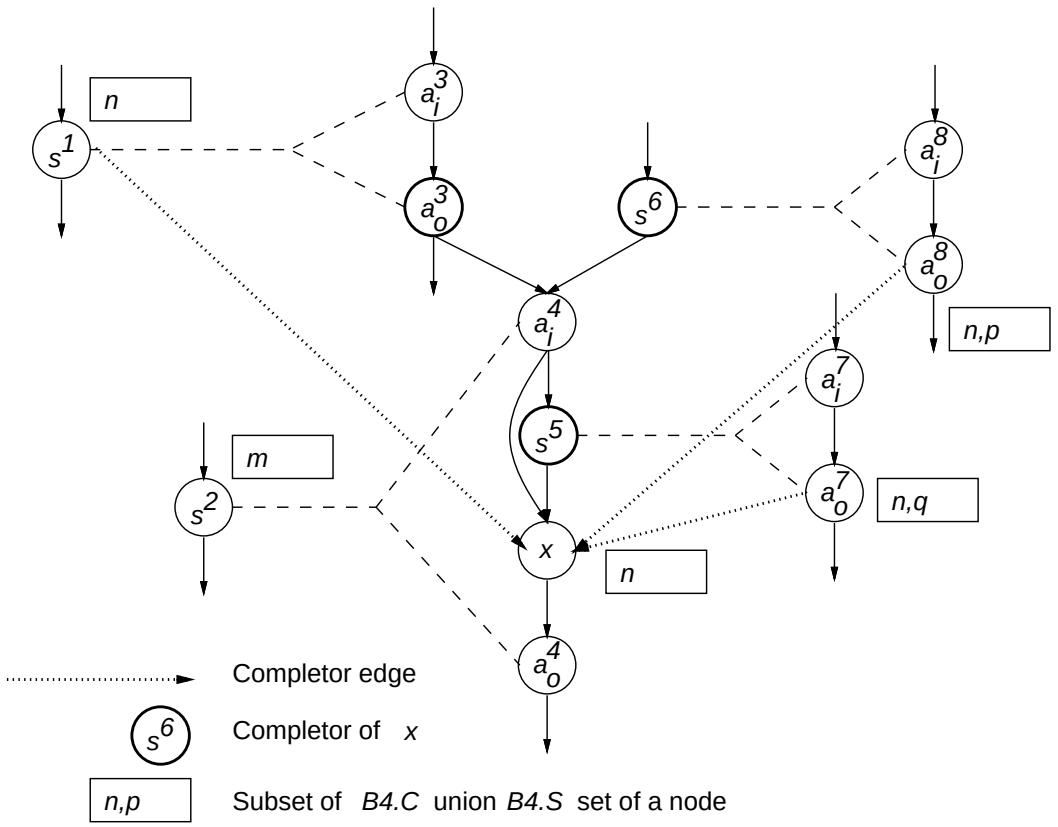

Figure 5.1: Completors of node x and completor edges to x.

In doing so, we can derive more information from s^5 than we can from a_i^4. This is because x is nested inside the accept statement corresponding to a_i^4. When x is reached in control flow, some such signaling node (say s^2) is in state ENGAGED(a_o^4), and would thus actually be *concurrent* with x. On the other hand, s^5, and the accept node pair it does rendezvous with, must complete execution before x starts.

For the sake of a simple and uniform treatment of rendezvous completion, we choose to examine only the most immediate control ancestors of x which have completed their rendezvous before x is reached in control flow. These control ancestors are called the completors of x, denoted $Completors(x)$. In Figure 5.1, $Completors(x) = \{s^5, s^6, a_o^3\}$. One of these must have completed rendezvous before x is reached. Therefore, we know that $n \in B4(x)$, since n is in the $B4$ sets (in boxes) of all the nodes which complete

rendezvous along with the completors of x.

For each node $v \in Completors(x)$, and each accept-out or signaling node w which may complete a rendezvous with v, we can imagine a directed completor edge (w, x). The presence of such an edge indicates that w may be a "synchronization ancestor" of x; the intersection of the $B4$ sets of all such synchronization ancestors must be in $B4(x)$. We denote the set of completor edges as $E_{Completors}$.

If $b \in Completors(x)$, then there is a path from the start of the program to x along which no rendezvous is completed. In this special case, we establish a completor edge (b, x) to propagate an empty $B4$ set to x through the completor edges. Thus, no nodes are known to complete before x because of rendezvous.

We may, of course, be able to derive further information from the nodes which rendezvous with the `accept` statements within which x is nested. For instance, we know that m must be in $B4(x)$, since m is in $B4(s^2)$ and s^2 is the only node that can rendezvous with a_i^4. However, making use of this information would require introducing a potentially unbounded group of "engagor edge" types, one for each scope level. We feel that doing so would complicate the lattice beyond reason. This potential source of information may be a subject for later investigation.

As the estimate of CHT is refined, we can refine the set of completor edges using new CHT information. If $m \in Completors(n)$ and $p \in CHT(m)$, then $(p, n) \notin E_{Completors}$.

Function space.

Let $CReach(m)$ denote the set of nodes q such that there is a control path from q to m, possibly of length zero. We map a control or completor edge $e = (m, n)$ to a function f_e as follows:

$$f_e((C, S)) = (C \cup S \cup K_e, N)$$

if e is a control edge, or

$$f_e((C, S)) = (N, C \cup S \cup K_e),$$

if e is a completor edge. For either edge type,

$$K_e = CReach(m) \cup \{m\} - \{p : m \in CReach(p)\},$$

i.e., the set of nodes q such that there is a control path from q to m (possibly of zero length), but no control path of length > 0 from m to q.

Lemma 21 *The edge functions f_e are inflationary and monotone.*

Proof: By the definition of the $\sqsubseteq$ operator as element-wise set containment, each f_e is clearly inflationary. Also, for any $a = (C_a, S_a), b = (C_b, S_b) \in L_{B4}$ such that $a \sqsubseteq b$, by definition of $\sqsubseteq$, $C_a \subseteq C_b$ and $S_a \subseteq S_b$. Therefore, for f_e of the form

$$f_e((C, S)) = (C \cup S \cup K_e, N),$$

we have

$$f_e(a) = (C_a \cup S_a \cup K_e, N)$$

and

$$f_e(b) = (C_b \cup S_b \cup K_e, N).$$

We know that, since $C_a \subseteq C_b$ and $S_a \subseteq S_b$,

$$C_a \cup S_a \cup K_e \subseteq C_b \cup S_b \cup K_e.$$

Therefore, $f_e(a) \sqsubseteq f_e(b)$ and so f_e is monotone. An analogous argument applies for f_e of the form for completor edges, $f_e((C, S) = (N, C \cup S \cup K_e)$. $\Box$

Lemma 22 below shows that the $B4$ lattice framework is not rapid. We therefore cannot use the time complexity results of Lemma 14 for the $B4$ problem.

Lemma 22 *The functions f_e are not distributive in general.*

Proof: Consider $a = (\emptyset, \{n\})$ and $b = (\{n\}, \emptyset)$. Let $f_e((C, S)) = (C \cup S \cup K_e, N)$ where $n \notin K_e$. Thus, $a \sqcap b = (\emptyset, \emptyset)$ and $f_e(a \sqcap b) = (K_e, N)$. But $f_e(a) = f_e(b) = (K_e \cup \{n\}, N) = f_e(a) \sqcap f_e(b)$. In this instance, $f_e(a \sqcap b) \sqsubset f_e(a) \sqcap f_e(b)$.

A similar argument holds for edge functions of the form $f_e((C,S)) = (N, C \cup S \cup K_e)$.

□

Lemma 23 below shows that the $B4$ lattice framework correctly models the semantic notion we wish to capture.

Lemma 23 *Let $B4_i$ be the i'th distinct value of $B4$, during execution of an iterative algorithm based on the lattice framework for $B4$ (with $B4_0(n) = (\emptyset, \emptyset)$ for all n). If $B4_i(n) = (C,S)$ and $m \in C \cup S$, then all instances of m must complete before any instance of n can start.*

Proof: By induction on i. As a notational convenience, let $UnionB4((C,S)) = C \cup S$.

Basis: Since $B4_0(n) = (\emptyset, \emptyset)$, the condition of the lemma is trivially satisfied.

Induction step: Suppose the lemma holds for $B4_{i-1}$. Let $B4_i(n) = (C,S)$, and suppose that $m \in C \cup S$.

If $m \in UnionB4(B4_{i-i}(n))$ then the condition of the lemma is satisfied by the induction hypothesis.

If $m \notin UnionB4(B4_{i-1}(n))$, then at least one of the following must be true:

1. $m \in C$. Thus,

$$m \in \bigcap_{p \in CPreds(n)} \left[UnionB4(B4_{i-1}(p)) \cup k_{(p,n)} \right],$$

where $CPreds(n)$ denotes the set of all control predecessors of node n.

For all control predecessors p of n, either $m \in UnionB4(B4_{i-1}(p))$, or $m \in k_{(p,n)}$, or both. If $m \in UnionB4(B4_{i-1}(p))$ for a particular predecessor p, then all instances of m must complete before any instance of p starts, by the induction hypothesis. Therefore, for any instance of n entered through p, all instances of m must have completed before any instance of n starts.

If, for a particular predecessor p, $m \in k_{(p,n)}$, then there is a control path from m to p, but no path of length > 0 from p to m. Therefore, either $m = p$ and m has only one instance, or all instances of m must complete before any instance of p start. In either case, for any instance of n entered through p, all instances of m

must complete before any instance of n starts.

Thus, if $m \in C$, then all instances of m must complete before any instance of n can start. If this is true, then obviously $m \in CHT_{perf}(n)$ and $n \in CHT_{perf}(m)$.

2. $m \in S$. Thus,

$$m \in \bigcap_{(p,n) \in E_{Completors}} \left[UnionB4(B4_{i-1}(p)) \cup k_{(p,n)} \right].$$

For all completor edges (p, n), either all instances of m complete before any instance of p starts, or $m = p$ and m has a single instance. Thus, n cannot be entered in control flow until after all instances of m have terminated.

Thus, if $m \in S$, then all instances of m must complete before any instance of n can start.

3. n either has no control edges entering it, or has no completor edges entering it. This only occurs if n is unreachable, i.e., no instances of n ever execute. Regardless of whether m executes or not, the conditions of the lemma are vacuously satisfied.

Therefore, if $m \in C \cup S$, then all instances of m must complete before any instance of n can start. □

The function space $F_{B4} = L_{B4} \rightarrow L_{B4}$ is the transitive closure of all possible edge functions under meet and function composition.

Lemma 24 *F_{B4} is closed under meet and function composition, and is 1-semibounded.*

Proof: Any set of monotone and inflationary functions is closed under meet and function composition, by Lemma 9, and is 1-semibounded by Lemma 11. □

Lemma 25 *The time for either a meet operation in L_{B4} or an edge function evaluation for $f_e \in F_{B4}$ is $\mathcal{O}(|N|)$.*

Proof: If sets of nodes are represented as bit maps of length $|N|$, then set union or intersection takes $\mathcal{O}(|N|)$ time. Evaluation of f_e takes two set union operations; meet requires two set intersection operations. □

Lemma 26 *The total time for convergence of a B4 lattice problem using the worklist iterative algorithm is* $\mathcal{O}(|N|^3 \log |N|)$ *in the worst case.*

Proof: Direct from Lemmas 12, 20, and 25. □

Lemma 27 *The total time for solution of a B4 lattice problem using Tarjan's path expression algorithm is* $\mathcal{O}(|N|^3)$ *in the worst case.*

Proof: Direct from Lemmas 13 and 25. □

Using *CHT* information to eliminate completor edges.

Recall from Section 5.1.1 that completor edges (w, x) exist from nodes w which might complete a rendezvous with any of the completors of x. Suppose w' is a completor of x, and w and w' are have a compatible signal type and polarity to rendezvous. If w' can't happen together with w, then w' will never able w to complete. Thus, no rendezvous between w and w' can ever directly enable x to start execution. If this is also true for all other completors of x which are compatible for rendezvous with w, then w will never directly enable x to start execution. We may then eliminate the completor edge (w, x). Doing so will improve the accuracy of the *B4* solution.

Incremental update of *B4* sets.

B4 sets may be updated incrementally after eliminating completor edges, as Lemma 17 shows. The lemma can be applied to by considering the control and completor edges as one single set of edges.

Augmenting *CHT* information using *B4* analysis.

Once we have found a solution to the *B4* lattice problem, we can immediately extract *CHT* information from it. Given the *B4* and *CHT* sets of a program, we form

$$CHT' = CHT \cup B4 \cup B4^T.$$

The update operation thus requires two array union operations and one array transpose.

Algorithm 2 *Augment CHT information using B4 information.*

Input:

- The PADA sync graph $G = (N, E_C, E_S)$.
- The set *CompleteRend*. *CompleteRend*(n) is the set of nodes which can complete a rendezvous with n. If n is a signaling node, then *CompleteRend*(n) is the set of accept-out nodes of the same signal type; if n is an accept-out node, then *CompleteRend*(n) is the set of signaling nodes of the same signal type. Otherwise, *CompleteRend*(n) is empty.
- $CHT \subseteq CHT_{perf}$.

Output:

- CHT' such that $CHT \subseteq CHT' \subseteq CHT_{perf}$.

Procedure:

1. Using G and CHT, construct $E_{Completors} = Completors \times (CompleteRend - CHT)$.

   The matrix notation used here, and subsequently in this chapter, is described in Appendix A.

2. Let $G' = (N, E_C, E_{Completors})$.

3. Find the least fixed point $B4$ of the lattice framework $(G', L_{B4}, F_{B4}, M_{B4})$. M_{B4} is the mapping function described in Section 5.1.1.

4. $CHT' = CHT \cup B4 \cup B4^T$.

Lemma 28 (Correctness of $B4$.) *If $CHT \subseteq CHT_{perf}$, then $CHT \subseteq CHT' \subseteq CHT_{perf}$ following Algorithm 2.*

Proof: $CHT' = CHT \cup B4 \cup B4^T$. From Lemma 23, $B4 \cup B4^T \subseteq CHT_{perf}$. Thus, $CHT \subseteq CHT_{perf} \Rightarrow CHT' \subseteq CHT_{perf}$. Obviously, $CHT \subseteq CHT'$. Therefore, $CHT \subseteq CHT' \subseteq CHT_{perf}$. □

Lemma 29 (Time bounds for *B4* analysis.) *The total time required for B4 analysis using Algorithm 2 is* $\mathcal{O}(|N|^3 \log(|N|))$ *in the worst case using the worklist iterative algorithm to solve the lattice problem, or* $\mathcal{O}(|N|^3)$ *using Tarjan's algorithm.*

Proof: The set of completors for all nodes may be found in $\mathcal{O}(|N|^3 \log |N|)$ time, using a classical "generate-preserve set" lattice framework, assuming that the flow graph has maximal loop-connectedness. The set of completor edges $E_{Completors}$ can be constructed at a cost of $\mathcal{O}(|N|^3)$ time, by multiplying Boolean arrays which represent the "completor" and "rendezvous partner" relations. From Lemmas 26 and 27, the time to solve the *B4* lattice problem is $\mathcal{O}(|N|^3 \log(|N|))$ using the worklist algorithm, or $\mathcal{O}(|N|^3)$ using Tarjan's algorithm. The time to perform the Boolean array transpose and union operations to form CHT' is $\mathcal{O}(|N|^2)$.

Thus, the total time required for *B4* analysis is $\mathcal{O}(|N|^3 \log(|N|))$ in the worst case using the worklist iterative algorithm, or $\mathcal{O}(|N|^3)$ using Tarjan's algorithm. □

5.1.2 Pinning analysis on successors of a rendezvous.

Figure 5.2 shows the sync graph of a simple program. Here, m can rendezvous only with p, and n can rendezvous only with q. For the sake of a simple explanation, we show each accept clause as a single node in the present discussion.

Suppose that we have hypothesized n as a head node. $CPreds(n) = \{m\}$, so some successor of a node that rendezvoused with m must have been placed on the execution wave after m completed its rendezvous. Such a node is called a *partner* of n.

The distinguished program begin node b is not in $CPreds(n)$. Let the set $Partners(n)$ be the set of signaling and accept-in nodes which are the most immediate successors of nodes which can rendezvous with $CPreds(n)$. In Figure 5.2, $Partners(n) = \{q\}$. Since no node in the sync graph other than n can rendezvous with q, and since q has no `else` clause, unguarded `delay` alternative, or sister which can rendezvous with nodes other than n, we know that q must remain on the execution wave until it does rendezvous with n. Furthermore, since q can rendezvous with n, we know that n cannot be a head node of a valid deadlock cycle.

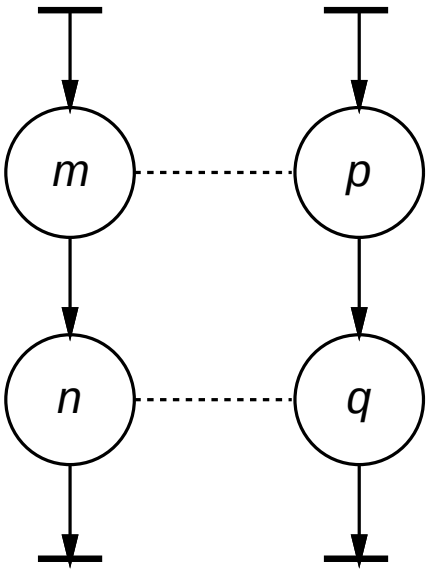

Figure 5.2: A simple sync graph. The thick horizontal bars represent the bounds of loop bodies.

Formalizing the above arguments, we have the following:

Lemma 30 *Whenever a node n is on the execution wave, at least one node $p \in$ Partners(n) must be placed on the wave at the same time as n, in the execution model.*

Note that the partner need not remain on the wave, as Figure 5.3 shows. Here, although q appears on the execution wave concurrently with n, it may rendezvous with r and leave the wave. Therefore, q need not be present at all times n is. Node q is thus a "vanishing partner" of n.

The reasons an accept-in or signaling node n may leave the wave include the following:

- n completes a rendezvous.
- n has a sister that completes a rendezvous.
- n has a `delay` alternative, for which the guard (if any) is open, and the delay period expires.
- n has an `else` clause which is taken. This may happen even if n can rendezvous with another node m on the wave. To see why this may happen in an actual

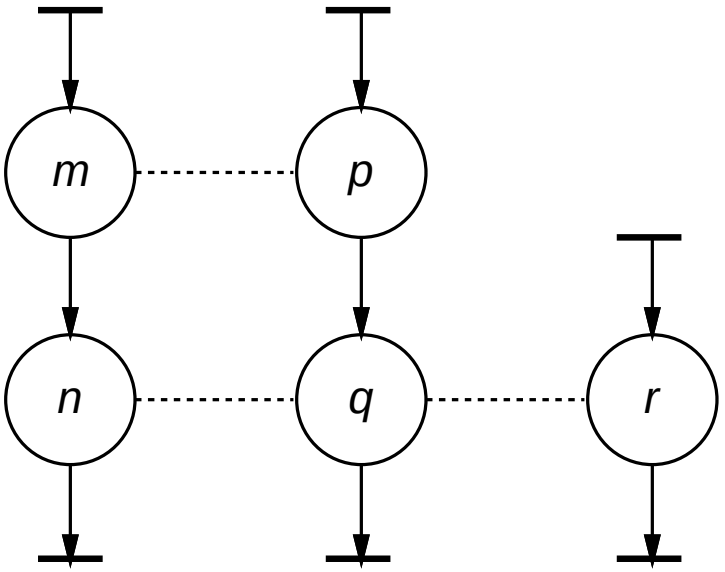

Figure 5.3: A "vanishing partner" of n.

program, consider what would happen if there was a long interval between the time n is reached in control flow and the time m is reached.

Nodes that have a `delay` alternative or an `else` clause are called "slippery" nodes, since they can slip off the execution wave without rendezvousing. In contrast, nodes that cannot slip off the execution wave without a rendezvous are called "sticky." In the construction of the sync hypergraph, if m can slip off the execution wave and allow n to execute, then there are control edges from m's predecessors to n.

Lemma 31 *If a node n is on the execution wave, at least one member of Partners(n) must appear on the wave until the first rendezvous occurs with n or a node in Partners(n).*

This, plus information about which nodes can rendezvous, gives us some information which can be used to iteratively add information to CHT:

Lemma 32 *If all nodes in Partners(n) can rendezvous only with n or with sisters of n, or with nodes in $CHT(n)$, then some member of Partners(n) is on the execution wave at all times that n is.*

If the above condition is true, we say that *n pins its partners to the wave.* Thus,

Lemma 33 *If n pins its partners to the wave and*

$$p \in \bigcap_{q \in Partners(n)} CHT(q)$$

then $p \in CHT(n)$.

We can further derive information about possible head nodes:

Lemma 34 *If n (or one of its unguarded sisters) can rendezvous with all nodes $q \in Partners(n)$ (or with an unguarded sister of each q), and if n pins its partners to the wave, then n cannot be a head node.*

Proof: If such is the case, then at any time n is on the wave, then so is another node that can rendezvous with n. □

Example of pinning analysis.

Figure 5.4 shows a sync graph for which we will conduct pinning analysis. Table 5.1 shows the results of the first iteration of pinning analysis for Figure 5.4. Initially, the CHT set for every node contains the other nodes in its task.

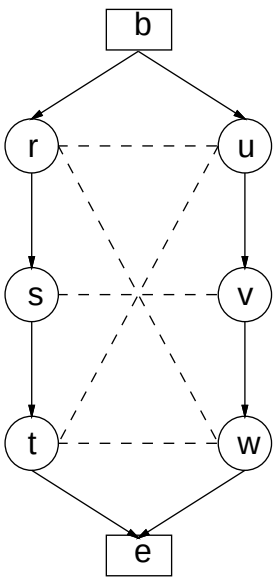

Figure 5.4: Pinning analysis example.

After the first iteration, we notice that $w \in CHT(r)$ and $u \in CHT(t)$. This gives

Node	*CPreds*	*Partners*	Head node?	*CHT*
b	$\emptyset$	$\emptyset$	No	r, s, t, u, v, w
r	b	u	No	b, s, t, v, w
s	r	v, e	Yes	b, r, t
t	s	w	No	b, r, s, u, v
u	b	r	No	b, s, t, v, w
v	u	s, e	Yes	b, u, w
w	v	t	No	b, r, s, u, v

Table 5.1: Pinning analysis - iteration 1. The "Head node?" column for a node n contains "No" if n's partners all do rendezvous only with n and members of $CHT(n)$, "Yes" otherwise.

us the opportunity to eliminate two sync edges which correspond to those rendezvous that are never taken. Figure 5.5 shows the resulting sync graph, and Table 5.2 shows the results of the second iteration of pinning analysis.

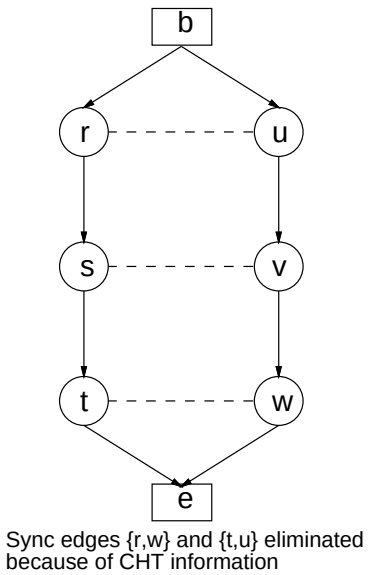

Figure 5.5: Pinning analysis example after second iteration.

We could have stopped pinning analysis after the first iteration, since at that time, only s and v were possible head nodes, and no valid deadlock cycles could have been found. In general, though, it seems more prudent to continue pinning analysis until stabilization, before attempting to find deadlock cycles.

Node	*CPreds*	*Partners*	Head node?	*CHT*
b	$\emptyset$	$\emptyset$	No	r, s, t, u, v, w
r	b	u	No	b, s, t, v, w
s	r	v	No	b, r, t, u, w
t	s	w	No	b, r, s, u, v
u	b	r	No	b, s, t, v, w
v	u	s	No	b, r, t, u, w
w	v	t	No	b, r, s, u, v

Table 5.2: Pinning analysis - iteration 2.

Pinning analysis when the begin node is in $CPreds(n)$.

One special case of pinning analysis occurs when we are analyzing some node n, and the distinguished begin node b is in $CPreds(n)$. For example, see Figure 5.6. Here, $CPreds(n) = \{b, m\}$. If we regard b as a barrier synchronization between all tasks, then we might say that $Partners(n) = \{m, p, r, s\}$. Since r and s can rendezvous with each other, and can happen together with n, we can conclude that n does not pin its partners to the wave.

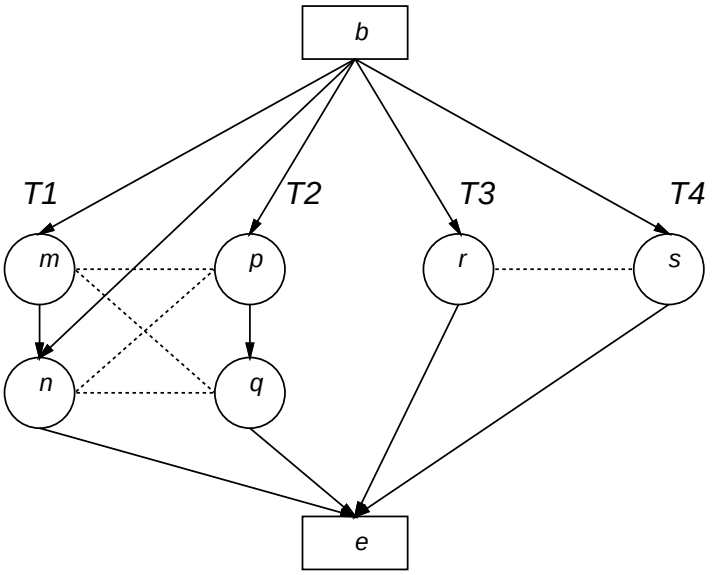

Figure 5.6: Pinning analysis where the distinguished begin node b is in $CPreds(n)$.

This, however, is too liberal a definition of the term "partner," and consequently we lose useful information. In this case, we do not find out that either p or q must be

on the wave whenever n is. Since both p and q can rendezvous with n, n cannot be a head node.

We can solve this problem by partitioning the *Partners* set by task, as in the following definition:

$$Partners(n, T \neq Task(n)) = \left[\bigcup_{\substack{(m,n)\in E_C, \\ \{m,p\}\in E_S, \\ (p,q)\in E_C}} q \right] \cup \left[\bigcup_{\substack{(b,n)\in E_C, \\ \{b,q\}\in E_S, \\ q\in T}} q \right]$$

where $Task(n)$ denotes the set of nodes in the task containing n. The union over an empty set is taken to be empty.

Semantically, $Partners(n, T)$ is the set of nodes which either:

- May start together with n when n is the control successor of a rendezvous node; or,
- Are in task T, and may start together with n when n is the control successor of b.

Lemma 35 *If the members of $Partners(n, T)$ can rendezvous only with n or with nodes that can't happen together with n, then n pins $Partners(n, T)$ to the wave.*

Proof: Whenever n starts execution, it must follow either the begin node or a rendezvous by one of its predecessors. In either case, some member of $Partners(n, T)$ is always placed on the wave whenever n starts. If all of these partners must rendezvous either with n or with some node that can't happen together with n, then any partner that starts with n must remain on the execution wave until n leaves it. □

Pinning and the `accept-do` and `select` constructs.

To this point, we have considered pinning analysis without reference to the `accept-do` or *select* constructs. In particular, we have used single accept nodes in our explanation, instead of accept in/out pairs. In this section, we examine these constructs in detail.

Predecessors of a node.

If a node n has a signaling node s as its predecessor, the successor of an accept-out node a_o which could have been engaged to s must be a partner of n. Similarly, if n has an accept-out node a_o as a predecessor, then its partners must be the successors of signaling nodes s which could have been engaged to a_o.

However, if n has an accept-in node a_i as its predecessor, i.e., n is an accept-out node or is the first node executed in an accept-do clause, then the signaling node s which a_i engaged with must still be on the wave. In a sense, s is the partner of n; although s did not start at the same time n did, s is on the wave at the same time n is.

Although this last case is more explicitly handled by remote procedure call analysis, it has been seen experimentally that including s as n's partner, rather than the nodes which can rendezvous with $Completors(n)$, often has some accuracy benefits in pinning analysis of nodes inside accept-do clauses. In particular, if a_i is the only predecessor of n, then more accurate CHT information may be propagated from s to n than from $(Completors \times CompleteRend)(n)$ to n.

Sisters.

If node n pins an accept-in node a_i with any sisters, the sisters (by definition) must be able to happen together with n. By construction of the sync hypergraph, the sisters of a_i must also be included as partners of n, since the sisters of a_i have the same control predecessors as a_i. Thus, we need not pay special attention to the sisters of partners when we perform pinning analysis.

Algorithm for pinning analysis.

Algorithm 3 *Pinning analysis.*

Input:

- The following arrays, of dimensions $|N| \times |N|$:
  - *Starters*, where $Starters(n)$ is the set of *starters* of n, i.e., the most immediate control descendants of n which can start a rendezvous.
  - *CPreds*.
  - *TakeOff*. *TakeOff* is, for each node n, the set of nodes which may take n off the execution wave (without necessarily completing a rendezvous). For signaling nodes, $TakeOff(n)$ includes only accept-out nodes of the same signal type; for accept-in or accept-out nodes, $TakeOff(n)$ includes signaling nodes of the same type as n. $TakeOff(n) \supseteq CompleteRend(n)$.
  - *CompleteRend*.
  - *InClause*. $InClause(n)$ is, for each accept-in node n, the set of nodes contained within the do clause of n (including n's accept-out node). If n is not an accept-in node, then $InClause(n)$ is empty.
  - *Sisters*. $Sisters(n)$ is the set of accept-in nodes in the conditional entry construct containing n, if n is an accept-in node. Otherwise, $Sisters(n)$ is empty. Note that $n \in Sisters(n)$ if n is an accept node.
  - *StartRendSig*. $StartRendSig(n)$ is the set of signaling nodes that can start a rendezvous with n. If n is an accept-in node, $StartRendSig(n)$ is the set of signaling nodes of the same signal type as n. If n is not an accept-in node, then $StartRendSig(n)$ is empty.
- The following arrays, of dimensions $|Tasks| \times |N|$:
  - *TaskFrontNodes*. $TaskFrontNodes(t)$ is the set of nodes in task t which may be placed on the execution wave at the start of the program.

Output: The array $CHT' \supseteq CHT$, of dimension $|N| \times |N|$.

Procedure:

1. $CHT' \Leftarrow CHT$.

2. For each task t do:

   (a) $Partners \Leftarrow (CPreds \times (CompleteRend - CHT') \times Starters) \cup (CPreds \times (StartRendSig - CHT')) - CHT'$.

   The term $(CPreds \times (StartRendSig - CHT'))$ is a refinement that allows accept-out nodes to pin the signaling nodes which engage the accept clause.

   (b) $Partners_t \Leftarrow Partners$.

   (c) For each node n such that $b \in CPreds(n)$:

      i. $Partners_t(n) \Leftarrow Partners(n) \bigcup TaskFrontNodes(t)$.

      If n may be the front node of a task, its sets of partners include the front nodes of each other task. For each task t, we refine $Partners_t$ this way.

   (d) $Pins \Leftarrow -(((Partners_t \times (TakeOff - CHT')) - I - CHT') \times ONE) \cap (Partners_t \times ONE)$.

   $(Partners_t \times (TakeOff - CHT)) - I - CHT$ is, for each node n, the set of nodes which can happen together with both n and the partners of n, and which can take n's partners off the wave. The final result is an array where n's entire row is set if n has partners, and if no node other than n, which can happen together with n and any partner q of n, can take q off the wave.

   (e) $PartnerClause_t = ((Partners_t \cap AInSameType) \times InClause) \cup Partners_t$.

   That is, $PartnerClause_t$ contains, for each node n, all partners of n, and all nodes nested in any accept-do clause of any of n's partners that n can rendezvous with. $AInSameType$ is an array such that $a_i \in AInSameType(s)$ for signaling nodes s which may start a rendezvous with accept-in nodes a_i.

   (f) $Partners_{Pinned} = (Pins \cap PartnerClause_t)$.

   If n pins its partners, $Partners_{Pinned}(n)$ is the set of nodes in the partner's clause.

(g) $CHT' \Leftarrow Widen(CHT', Partners_{Pinned}, t)$.

Lemma 36 (Correctness of pinning analysis.) *If $CHT \subseteq CHT_{perf}$ and the other parameters of Algorithm 3 are correct, then $CHT \subseteq CHT' \subseteq CHT_{perf}$.*

Proof: From the discussion in the algorithm definition, each node n such that $Partners_{Pinned}(n) \neq \emptyset$ must execute concurrently with some member of $Partners_{Pinned}(n)$. This defines a set S which can be widened using Algorithm 1. The parameter T of Algorithm 1 is an arbitrary task. Therefore, we have correctly specified the parameters S and T of Algorithm 1.

From Lemma 2, if $CHT \subseteq CHT_{perf}$ and the parameters S and T are correctly specified for Algorithm 1, then $Widen(CHT, S, T) = CHT' \subseteq CHT_{perf}$. Therefore, if $CHT \subseteq CHT_{perf}$ and the other parameters of Algorithm 3 are correct, then $CHT' \subseteq CHT_{perf}$. □

Lemma 37 (Time bounds for pinning analysis.) *One application of pinning analysis requires $\mathcal{O}(|Tasks||N|^3)$ time.*

Proof: Algorithm 3 executes a loop which iterates $|Tasks|$ times. Within this loop are a fixed number of array copies, unions, subtractions, transposes, negations, and multiplications, and one call to *Widen*. Most of these operations require $\mathcal{O}(|N|^2)$ time; multiplication and the call to *Widen* are the exceptions, requiring $\mathcal{O}(|N|^3)$ time.

Within the task loop is an iteration over $|N|$ nodes, to create $Partners_t$. Each iteration of the inner loop requires at most $|N|$ time; the inner loop total time is thus $|N|^2$.

The total time per iteration of the task loop is thus $\mathcal{O}(|N|^3)$ time. An application of pinning analysis thus requires $\mathcal{O}(|Tasks||N|^3)$ time. □

Propagating pinning information.

It is possible, under certain circumstances, to propagate pinning information from a node to at least some of its successors. While we do not currently make use of this technique, we describe it here by means of an example. The effectiveness of propagating pinning information is a subject for future study.

Consider the situation of Figure 5.7. Here, node m pins a set of its partners $\{a, b, c\}$ to the wave, and does not rendezvous with any of them or their sisters. Thus, at least one member of $\{a, b, c\}$ must remain on the wave when m completes execution.

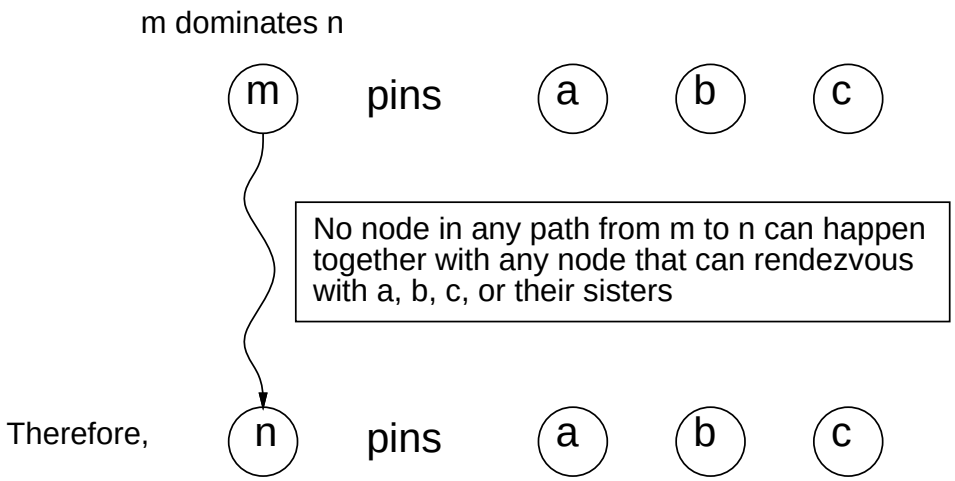

Figure 5.7: Propagation of pinning information to the successors of a node m

Node n is dominated by m, and there is no node on any path from m to n (including m but not necessarily n) that can happen together with any node that can rendezvous with a, b, or c. Thus, at least one member of $\{a, b, c\}$ must be on the wave when n starts to execute. If n can't happen together with any node that can rendezvous with a, b, c, or their sisters (except possibly n itself or one of n's sisters), then n pins a, b, and c to the wave. Therefore, we can add the same set of nodes to $CHT(n)$ which were added to $CHT(m)$ as a result of pinning analysis of m.

5.1.3 Strict intervals.

We define a *strict interval* as a single entry, single exit region in control flow with additional properties which limit control flow within the region. Strict intervals are the structural basis of critical sections and regions of mutual exclusion in several synchronization paradigms, including the Ada rendezvous. Thus, detecting strict intervals is an important part of deriving CHT from a program.

Suppose that G is a directed graph, and $\mathcal{P}$ is a subset of some partition of the nodes in G (i.e., $\mathcal{P} = \{P_1, ...P_k\}$ such that each $P_i \subseteq N$ and $\forall i \forall j \neq i : P_i \cap P_j = \emptyset$).

Define $StrictInt(G, \mathcal{P})$ to be the set of all strict intervals in G relative to the partition $\mathcal{P}$. A strict interval is denoted by the pair (n_n, n_x), where $n_n \in P_n$, $n_x \in P_x \neq P_n$, $P_x, P_n \in \mathcal{P}$. The strict interval represented by (n_n, n_x) is the set of nodes which can be reached from n_n without passing through n_x (i.e., the union of all $n_i, 1 \leq i \leq k$, such that a path $(n_n, n_1, ..., n_k = n_x)$ exists in G). The following conditions are also required of strict intervals:

- Local conditions on each strict interval:
  - n_n and n_x form a single-entry, single-exit region in G, with n_n as the entry node and n_x as the exit node.
  - $\{n_n\}$ cuts[3] $(\{n_x\}, CPreds(n_x))$ in G. I.e., there are no cycles including n_x that omit n_n.
  - $\{n_x\}$ cuts $(\{n_n\}, CPreds(n_n))$ in G.
  - $\{n_x\}$ cuts $(\{n_n\}, (P_n \cup P_x) - \{n_n, n_x\})$ in G. I.e., no node in $P_n \cup P_x$ lies within the strict interval, except for n_x.
- Global conditions on all strict intervals:
  - $\forall(n_n \in P_n)\exists(n_x \in P_x) : (n_n, n_x) \in StrictInt(G, \mathcal{P})$. Every node in P_n is the entry of a strict interval that exits at a node in P_x.
  - $\forall(n_x \in P_x)\exists(n_n \in P_n) : (n_n, n_x) \in StrictInt(G, \mathcal{P})$. Every node in P_x is the exit of a strict interval that enters at a node in P_n.

Suppose that the nodes within P_n and P_x form the entry and exit nodes, respectively, of a set S of strict intervals. Then we have the following useful properties:

- Strict intervals in S can only be entered through a node in P_n.
- When any node in P_n completes execution, control must enter a strict interval in S.

---

[3] M cuts (N, P) iff, for all $n \in N - M$ and $p \in P - M$, any directed path from n to p contains some $m \in M$.

- No node in P_n can be within a strict interval in S.
- All nodes in P_x lie inside of a strict interval in S.
- Once control has entered a strict interval in S, it can only leave the strict interval by executing a node in P_x.
- Once a node in P_x has completed inside a strict interval in S, control must leave the strict interval.
- Strict intervals in S do not overlap.

Algorithm to find strict intervals.

Algorithm 4 produces a set of strict intervals, given a graph and a partition of its nodes.

Algorithm 4 *Identify strict intervals.*

Input:

- A directed graph $G = (N, E)$.
- A partition $\mathcal{P}$ of the nodes of N.

Output: The set $StrictInt(G, \mathcal{P})$.

Procedure:

1. Construct the control dominator and postdominator trees for G.
2. Identify partitions $P_n, P_x \in \mathcal{P}$ and pairs of nodes $(n_n \in P_n, n_x \in P_x)$ such that:
   - $P_n \neq P_x$;
   - $n_n \in DOM(n_x)$, where $DOM(n_x)$ is the set of nodes that dominate n_x in control flow;
   - $\forall n' \overset{\text{such that}}{} (n' \in DOM(n_x) \cap (P_n \cup P_x)) \wedge (n' \neq n_n) : \{n_n\}$ cuts $(\{n'\}, \{n_x\})$;
   - $n_x \in POSTDOM(n_n)$, where $POSTDOM(n_n)$ is the set of nodes that post-dominate n_n in control flow;

- $\forall n' \overset{\text{such that}}{} (n' \in POSTDOM(n_n) \cap (P_n \cup P_x)) \wedge (n' \neq n_x)$: $\{n_x\}$ cuts $(\{n_n\}, \{n'\})$.

Place these pairs (n_n, n_x) in S. S is the set of *candidate strict intervals.*

3. If there is any node $n_n \in P_n$ such that there is no pair $(n_n, n_x \in P_x) \in S$, then remove all pairs (n'_n, n'_x) from S such that $n'_n \in P_n, n'_x \in P_x$.

4. If there is any node $n_x \in P_x$ such that there is no pair $(n_n \in P_n, n_x) \in S$, then remove all pairs from S (n'_n, n'_x) such that $n'_n \in P_n, n'_x \in P_x$.

5. Return S.

Lemma 38 (Correctness of the strict interval algorithm.) *The set StrictInt$(G, \mathcal{P})$ returned by Algorithm 4 is the set of those strict intervals that satisfies the definition given in Section 5.1.3.*

Proof: Step 2 places in set S exactly those node pairs which satisfy the local conditions on strict intervals for graph G and partition $\mathcal{P}$. Steps 3 and 4 eliminate from S those candidate strict intervals that do not satisfy the global conditions for G and $\mathcal{P}$. Neither of the two elimination steps cause any valid strict intervals to be eliminated, nor does any step cause any candidate strict intervals which were valid under the previous steps to become invalid under them. Therefore, upon completion of the algorithm, S contains exactly those strict intervals that satisfy the definition of Section 5.1.3. □

Lemma 39 *Using Algorithm 4, the worst-case time to find all strict intervals for a given graph and node partition is $\mathcal{O}(|N|^3)$.*

Proof: The dominator and postdominator trees of a flow graph can be found in time $\mathcal{O}(|E| \log(|N|))$ by the simpler algorithm of [LT79]. From this, the initial pairs $(n_n, n_x) \in S$ may be constructed by traversing these trees. For any particular node n, traverse the (post)dominator subtree rooted at n to find the nodes it (post)dominates; this process can be conducted in $\mathcal{O}(|N|)$ time per node, for a total of $\mathcal{O}(|N|^2)$. Finding the node pairs then requires one set intersection per node. Again, this operation takes

$\mathcal{O}(|N|)$ time per node, again for $\mathcal{O}(|N|^2)$ total time.[4] We ignore pairs of nodes which are in the same partition.

At this point, we can eliminate unsatisfactory candidate strict intervals (n_n, n_x) by performing depth-first search along forward control edges from n_n and along backward control edges from n_x. We eliminate the interval from S if there is a forward path from n_n to any node $n' \neq n_x \in P_n \cup P_x$ that does not include n_x. Likewise, we eliminate the interval if there is a backward path from n_x to any node $n' \neq n_n \in P_n \cup P_x$. Doing these searches takes $\mathcal{O}(|N|)$ time per pair in S. There may be up to $\mathcal{O}(|N|^2)$ elements in S (or, more accurately, $\mathcal{O}(|N||\mathcal{P}|)$, since each node in P_n may be in an interval with only one node in P_x, or vice versa). Step 2 thus takes a total of $\mathcal{O}(|N|^3)$ time.

Finally, for each remaining pair $(n \in P_n, x \in P_x)$, we examine P_n to see if it contains any nodes n' which is not a member of a pair $(n', x' \in P_x)$. We similarly examine P_x. This examination may be done in $\mathcal{O}(|N|)$ time per pair, using bit vectors, for a total of $\mathcal{O}(|N|^3)$ (again more accurately $\mathcal{O}(|N||\mathcal{P}|^2)$). The total execution time of Algorithm 4 is thus $\mathcal{O}(|N|^3)$. □

5.1.4 Critical section analysis.

A *critical section*, for our purposes, is a set of regions of different tasks, such that the nodes of at most one region can execute simultaneously. Critical sections may be used, for example, to enforce mutual exclusion of access to a shared variable or other resource.

There are many ways of implementing this type of mutual exclusion, some of which involve the use of memory state information, and some of which do not. We examine a fairly popular instance of the latter in detail in this section. We develop a method of identifying a particular form of critical section structures in a sync hypergraph, and of using these to obtain *CHT* information. We assume that critical section calls may be nested, that critical section bodies may call other critical sections, etc.

---

[4] A somewhat more clever way to do this is to construct a single graph, with two types of edges, corresponding to edges in the dominator and postdominator trees of signaling and accept-out nodes only. A candidate strict interval would then be found by finding a cycle consisting of a dominator and a postdominator edge. However, doing this does not decrease the order-of-magnitude complexity of Step 2.

An example of a critical section.

We present here a simple example of a critical section. A more general form of critical sections is presented in Section 5.1.4.

Consider the sticky node n of Figure 5.8, which is reachable only after a rendezvous between its predecessor m and the predecessor of p has been executed. Suppose further that there are successors s_i of p, such that, along all control paths from p to s_i, all nodes are known to be in $CHT(x)$ for any x that can rendezvous with n or any of its sisters. In this case, n *must happen together* with each s_i. The contrapositive is of interest, because any node that is in $CHT(n)$ is also in $CHT(s_i)$. This is generalized and stated more formally below:

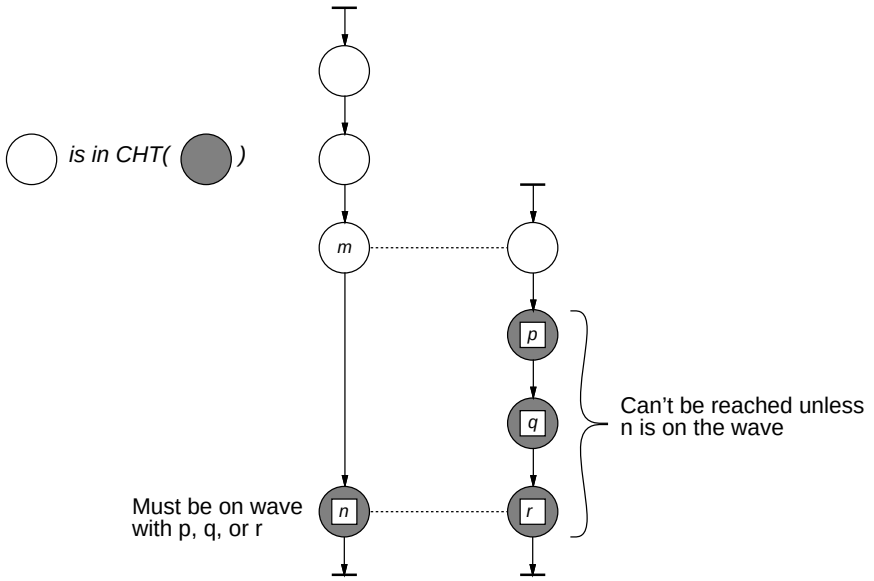

Figure 5.8: A critical section.

Lemma 40 *Assume that n is a sticky node with a partner p, and let q be a node reachable in control flow from p. If the following conditions hold:*

- *n postdominates $Completors(n)$.*
- *For all $x \neq n$, $p \notin Partners(x)$.*
- *On all control paths $s_1 = p, ..., s_k = q$:*

- *No node n' that can rendezvous with n or with any sister of n can happen together with any s_i.*
- *Each s_i is dominated by Partners(n).*

Then n must happen together with all members of all paths from p to s.

Proof: Since n is a sticky node, it must remain on the wave until some node that can rendezvous with n reaches the wave. Since $p \notin Partners(x)$ for $x \neq n$, p can only be on the wave as a result of a rendezvous between some member of $Completors(p)$ and some member of $Completors(n)$. Since n postdominates $Completors(n)$, p can only be placed on the wave concurrently with n.

Assume that p is placed on the wave, and that the members of some chain of successors $s_1 = p, ..., s_k = q$ of p such that each $s_i \in CHT(x)$ for any x that can rendezvous with n or any of n's sisters. Then, whenever the entire chain executes, n must be on the wave at all times during the execution of the chain. Since $Partners(n)$ dominates each s_i, each s_i can only execute as a result of executing the entire chain. Therefore, n must remain on the wave at least until q completes execution. □

Corollary 4 *If there is a sticky node r such that:*

- *r is a child of some node q as above,*
- *r is dominated by members of Partners(n),*
- *r can only be reached from members of Partners(n) along paths of nodes that cannot rendezvous with n or its sisters,*
- *r can only rendezvous with n or its sisters, or with nodes in $CHT(n)$,*

then r must happen together with n.

Proof: When r reaches the wave, n must be on the wave. r can only leave the wave by rendezvousing with n or one of its sisters. □

Critical section structures.

A *critical section structure* is a set of specialized single entry-single exit regions which generalize the above simple case, as in Figure 5.9. (The term "critical section structure" comes from the fact that such structures are often used to implement critical sections in the rendezvous paradigm.)

Critical section structures are denoted by the pair (C, B) of sets of node pairs. C is the set of *call bodies* of the critical section, i.e., pairs (c, r), corresponding to the calls to, and returns from, a critical section. B is the set of of *critical section bodies* of the critical section, i.e., node pairs (n, x), corresponding to the *entry* and *exit* nodes of each critical section body. Because of its structural characteristics, the critical section structure has the properties that:

- Only one of its critical section bodies may execute at any time;
- Only one of its call bodies may execute at any time;
- Any call body of the structure must execute simultaneously with a critical section body of the same structure;
- Any critical section body of the structure must execute simultaneously with a call body of the same structure.

We will initially assume that $|B| = 1$; later, in Section 5.1.4, we expand this to $|B| \geq 1$. Also note that a node may be contained within many critical section structures, and that the structures are not required to overlap in any given way, e.g., by nesting.

The pair of nodes $(n, x \neq n)$ delineates a critical section structure $(C, B = \{(n, x)\})$ if:

- n and x are either signaling or accept-out nodes.
- $Scope(n, y_n) = Scope(x, y_x) = Scope(x, y_n) = Scope(x, y_x) = 0$, where y_n and y_x are the signal types of n and x respectively.
- n dominates x.

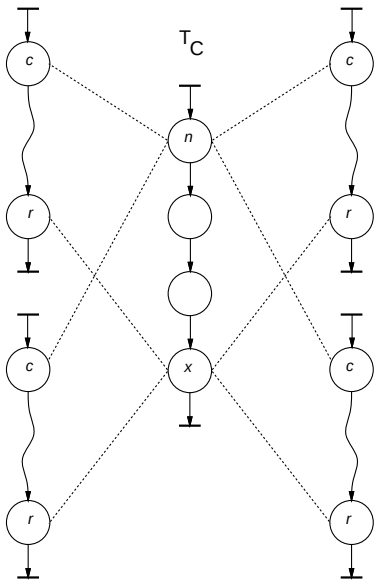

Figure 5.9: A critical section structure.

- x postdominates n.
- n and x are of different signal types.
- n and x are sticky.
- For all $(c, r) \in C$, c and r are either signaling or accept-out nodes.
- For all $(c, r) \in C$, $Scope(c, y_n) = Scope(c, y_x) = Scope(r, y_n) = Scope(r, y_x) = 0$.
- For all $(c, r) \in C$, c and r are sticky.
- For all $(c, r) \in C$, c can rendezvous only with n.
- For all $(c, r) \in C$, r can rendezvous only with x.
- If n can rendezvous with some c, then $\exists r \overset{\text{such}}{\underset{\text{that}}{}} (c, r) \in C$.
- If x can rendezvous with some r, then $\exists c \overset{\text{such}}{\underset{\text{that}}{}} (c, r) \in C$.
- Any cycle in control flow that includes n must also include x, and vice versa.

  Equivalently,

$$\{x\} \text{ cuts } (\{n\}, CPreds(n)) \wedge \{n\} \text{ cuts } (\{x\}, CPreds(x)).$$

- For all $(c, r) \in C$, any cycle in control flow that includes c must also include r, and vice versa. Equivalently,

$$\forall (c, r) \in C : \{c\} \text{ cuts } (\{r\}, \mathit{CPreds}(r)) \wedge \{r\} \text{ cuts } (\{c\}, \mathit{CPreds}(c)).$$

- For every pair $(c, r) \in C$, c dominates, and is postdominated by, r. Equivalently,

$$\forall (c, r) \in C : \{c\} \text{ cuts } (\{b\}, \{r\}) \wedge \{r\} \text{ cuts } (\{c\}, \{e\}).$$

- All control paths from any $c_i \in N_C$ to any $c_j \in N_C$ must contain some $r \in N_R$. This prevents any task from issuing a call to a critical section while control is within a call body for the same critical section. It likewise prevents nesting or overlapping call bodies to the same critical section.

  Equivalently, N_R cuts $(\{c_i\}, N_C)$.

- All control paths from any $r_i \in N_R$ to any $r_j \in N_R$ must contain some $c \in N_C$. This prevents any task from issuing a return from a critical section while control is outside any call body for the same critical section. It also prohibits nesting or overlapping call bodies to the same critical section.

  Equivalently, N_C cuts $(\{r_i\}, N_R)$.

For $(c, r) \in C$, define the *call body* $\mathit{CallB}(c, r)$ as the strict interval (c, r) minus c. Also, define the *critical section body* $\mathit{CSB}(n, x)$ as the strict interval (n, x) minus n. The set

$$\left[\bigcup_{(c,r) \in C} \{c, r\} \right] \cup \left[\bigcup_{(n,x) \in B} \{n, x\} \right]$$

is called the set of *bracket nodes* for the critical section, since these are the nodes that delimit the call bodies and critical section bodies in control flow.

Lemma 41 *If m dominates n, then m dominates each q in the induced region (m, n) (i.e., all nodes contained in paths $(m...n)$ that include n exactly once).*

Proof: Suppose some node q is reachable from a path from m which does not include

n, but m does not dominate q. Then there is a path from some predecessor of m to n (through q) that does not include m. Therefore, m does not dominate n, contrary to the assumption. □

Lemma 42 *If n postdominates m, then n postdominates all nodes q in the induced region (m, n).*

Proof: Suppose some node q is reachable from a path from m which does not include n, but n does not postdominate q. Then there is a path from m to some successor of n (through q) that does not include n. Therefore, n does not postdominate m, contrary to the assumption. □

Lemma 43 *All nodes in any critical section call body $CallB(c, r)$ are dominated by c and postdominated by r.*

Proof: Direct from the definition of the call body, and from Lemmas 41 and 42. □

Lemma 44 *All nodes in any critical section body $CSB(n, x)$ are dominated by n and postdominated by x.*

Proof: Identical to Lemma 43. □

Theorem 6 (Critical section structures.) *Each call body $CallB(c, r)$ must happen together with critical section body $CSB(n, x)$, and $CSB(n, x)$ must happen together with exactly one call body at any time.*

Proof: We will consider these contradictory possibilities in order:

1. $CallB(c, r)$ can start at some time when $CSB(n, x)$ does not,
2. $CSB(n, x)$ can start at some time when no call body does,
3. Two (or more) call bodies can execute simultaneously,
4. $CallB(c, r)$ may terminate while $CSB(n, x)$ does not,
5. $CSB(n, x)$ may terminate when no call body does.

Possibility 1.

Consider first the possibility that when $CallB(c, r)$ is entered, $CSB(n, x)$ is not. $CallB(c, r)$ cannot be entered except through c, since c dominates all nodes in $CallB(c, r)$. Node c must rendezvous with n for control to enter $CallB(c, r)$. All successors of n are members of $CSB(n, x)$, and all successors of c are members of $CallB(c, r)$. Therefore, if control enters $CallB(c, r)$, it will also enter $CSB(n, x)$ at the same time. So the first possibility cannot hold.

Possibility 2.

Similarly, $CSB(n, x)$ cannot be entered except through n, which must rendezvous with some $c \in N_C$ to enter $CSB(n, r)$. Each c dominates a call body $CallB(c, r)$. So if control enters $CSB(n, x)$, it will also enter some call body at the same time. So the second possibility cannot hold.

Possibility 3.

Now consider the third possibility, i.e., that two call bodies $CallB(c, r)$ and $CallB(c', r')$, execute simultaneously, and are (without loss of generality) the first pair of call bodies that do. Each of the call nodes for these bodies must rendezvous with n, in order for the bodies to start execution. Since n can rendezvous only with one node at any given state transition, no two call bodies can start executing simultaneously. Therefore, $CallB(c, r)$ and $CallB(c', r')$ must start at different times. Suppose, without loss of generality, that $CallB(c, r)$ starts before $CallB(c', r')$ does. $CallB(c', r')$ cannot start until c' does rendezvous with n. But n cannot be reached from within $CSB(n, x)$, and x postdominates $CSB(n, x)$ and can rendezvous only with members of N_R. Therefore, while $CallB(c, r)$ continues to execute, some $r'' \in N_R$ must rendezvous with x to allow n to be reached in control flow, before any other $CallB(c', r')$ can start.

Suppose that $r'' = r$. Since r postdominates all members of $CallB(c, r)$, and there

are no cycles inside $CallB(c, r)$ that contain r, this means that $CallB(c, r)$ stops, contrary to the hypothesis that it continues executing until $CallB(c', r')$ starts. Therefore, $r'' \neq r$. But r'' is a member of $CallB(c'', r'')$. Thus, at least one member of $CallB(c'', r'')$ executes simultaneously with $CallB(c, r)$, and must have started executing before $CallB(c', r')$ did. This violates our hypothesis that $CallB(c', r')$ was the first call body to executes simultaneously with $CallB(c, r)$. Thus, at most one call body may execute simultaneously with $CSB(n, x)$, and the third possibility cannot hold.

Possibility 4.

Now, suppose that $CallB(c, r)$ exits before $CSB(n, x)$ does. $CallB(c, r)$ is postdominated by node r, so $CallB(c, r)$ must be exited following completion of r, which must rendezvous with x. (Note that r must be sticky, or it could not postdominate the call body, by construction of the sync hypergraph.) x postdominates $CSB(n, x)$, and there is no cycle within $CSB(n, x)$ that includes x. So after the rendezvous between r and x, control is no longer in $CSB(n, x)$. So $CallB(c, r)$ cannot exit before $CSB(n, x)$ does, and the fourth possibility cannot hold.

Possibility 5.

Finally, consider the possibility that $CSB(n, x)$ exits before $CallB(c, r)$ does. $CSB(n, x)$ cannot be exited except through node x, since x postdominates all nodes in $CSB(n, x)$. x must rendezvous with some node in N_R to allow control to leave $CSB(n, x)$. Since no two call bodies can execute simultaneously, x can only rendezvous with r when control leaves $CSB(n, x)$; thus, control will leave $CallB(c, r)$ simultaneously. Therefore, the fifth possibility cannot hold. □

Corollary 5 *If $CallB_i \neq CallB_j$ are call bodies for the same critical section, then no node in $CallB_i$ can happen together with any node in $CallB_j$.*

Proof: Direct from Theorem 6. □

Corollary 6 *If $CallB_i$ is a call body and CSB_j is its critical section body, then no node in $Task(CSB_j) - CSB_j$ can happen together with any node in $CallB_i$.*

Proof: Direct from Lemma 6. □

Relaxing the structural requirements on critical sections.

In this section, we show how some of the requirements listed in the definition of critical section structure in Section 5.1.4 can be relaxed.

Eliminating the stickiness check.

The call and return nodes of a critical section, and the entry and exit nodes of a critical section body, were previously required to be sticky. This was done to enforce the requirement that call bodies be entered in execution only through the rendezvous with the call nodes, and left only through the rendezvous with the return nodes. Similar requirements restrict the entry and exit nodes of a critical section body. However, it turns out that the reason for the stickiness requirement is actually enforced by the control dominance characteristics of the sync hypergraph.

Lemma 45 *If x dominates y in control flow in the sync hypergraph and x is slippery, then y can only be entered following a rendezvous between x and some other node.*

Proof: By construction of the sync hypergraph, if x is slippery, there are then control edges from all control predecessors of x to the front nodes of the `else` clause or unguarded `delay` alternative of the clause C containing x, and/or to the control successors of x if either of these clauses are empty. In this case, x does not dominate any of the successors of C, because there is a path from the begin node b which reaches them without going through x. x does not dominate any nodes in the `else` clause or unguarded `delay` alternative, for the same reason.

The only nodes which x may dominate are those in the sequence of statements following x, or those embedded within x if x is an accept-in node of an `accept-do` construct. These can only be executed following the initiation of a rendezvous with x. □

Lemma 46 *If y is slippery, then y cannot postdominate any node in control flow in the sync hypergraph.*

Proof: By construction, if y is slippery, there are edges from the control predecessors of y to the front nodes of the `else` clause or unguarded `delay` alternative of the clause C containing y, and/or to the control successors of y if either of these clauses are empty. Thus, there are paths from the predecessors of y to the successors of y which do not include y, and y cannot postdominate its predecessors. Therefore, y cannot postdominate any node in the sync hypergraph. □

So, if x and y delineate a single entry, single exit region, then the nodes in the region (other than x itself) can only be reached through a rendezvous with x, and the region can only be left through a rendezvous with y. The stickiness checks for the entry and exit nodes are not necessary.

Multiple critical section bodies.

It is possible, though uncommon, for more than one critical section body to receive the same calls. Fortuitously, such cases are easy to identify in Ada.

Lemma 47 *If two different critical section bodies, $CSB(n, x)$ and $CSB(n', x')$, are such that both n and n' can rendezvous with $c_1, c_2 \in N_C$, and both x and x' can rendezvous with $r_1, r_2 \in N_R$, then either n and n' (and therefore x and x') are in the same task, or c_1 and c_2 (and therefore r_1 and r_2) are in the same task.*

Proof: Only accept nodes can rendezvous with nodes from more than one task, because entry calls are restricted (by the semantics of Ada) to call only the entries of one task. This implies that either the two critical section bodies or the two call bodies are in the same task. □

Lemma 48 *No two critical section bodies $CSB(n, x)$ and $CSB(n', x')$ in which n and n' (and therefore x and x') have the same signal type can happen together.*

Proof: Two such bodies must be contained in the same task, and are therefore disjoint. □

Theorem 7 *If a critical section structure contains multiple critical section bodies, then each call body must happen together with one of the critical section bodies, and each critical section body must happen together with exactly one call body at any time.*

Proof: Very similar to Theorem 6. □

Corollary 7 *If $CallB_i$ is a call body and $CSB_1, CSB_2, ...CSB_k$ are its critical section bodies, then, for all $j \in [1...k]$, no node in $Task(CSB_j) - CSB_1 - CSB_2 ... - CSB_k$ can happen together with any node in $CallB_i$.*

Proof: Direct from Theorem 7. □

Due to these results, we can treat critical sections with multiple bodies the same way that we would treat critical sections with a single body. We can therefore compute *CHT* for critical section structures with multiple critical section bodies, as long as we are careful to insure that all the critical section bodies are contained within a single task.

Scope depth and critical sections.

In Section 5.1.4, the bracket nodes of a critical section structure were forbidden to have a scope depth > 0 with respect to the signal type of any of the bracket nodes. If this restriction is relaxed, then it becomes possible for multiple critical section bodies to execute at once.

Lemma 49 *Suppose $(n, x) \in B$ for some candidate critical section structure (C, B). If $Scope(x, y_x) > 0$, then either multiple call bodies may execute simultaneously, or (C, B) does not correspond to the definition of a critical section structure.*

Proof: Under these assumptions, some accept-do construct of type y_x encloses x. Either the same construct encloses n, or it does not.

If the former is the case, then let x' be the accept-out node of the structure enclosing n and x. In order for n to execute, some signaling node must be ENGAGED to x' when n executes. This signaling node must be some r such that $(c, r) \in C$. If n makes a rendezvous with c' such that $(c', r') \in C$, then some member of $CallB(c', r')$ executes simultaneously with r. Thus, nodes in two call bodies may execute simultaneously.

If the latter is the case, then there must be some accept-in node of the same signal type as x, lying on an x-free path from n to some predecessor of x. In this case, (C, B) is not properly a critical section structure. □

Other forms of nesting have even worse implications:

Lemma 50 *Suppose* $(n, x) \in B$ *for some candidate critical section structure* (C, B). *If* n *is an accept-out node of type* y_n *and* $Scope(n, y_n) > 0$, *then either* (C, B) *must deadlock with one call body active, or* (C, B) *does not properly define a critical section structure (ignoring rules of scope depth).*

Proof: See Figure 5.10(a). Let n' be the accept-out node of the `accept-do` construct of type y_n most immediately enclosing n. There is at least one path from n to n', by construction of the sync hypergraph. If there is a path from n to n' which includes no exit nodes for that critical section structure (such as x in Figure 5.10), then the candidate critical section structure (C, B) does not meet the definition of a critical section structure.

On the other hand, suppose any path from n to n' includes some exit node in N_X (Figure 5.10(b)). Then n is postdominated by some such node x, and all paths from n to n' include x. Whenever x executes, the node c which rendezvoused with n' is still ENGAGED to n', and no node from that task (or any other) is available to rendezvous with x. Thus, the structure deadlocks with one call body active. □

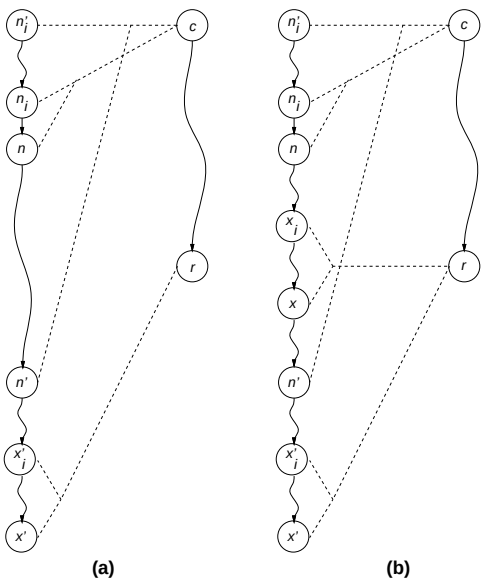

Figure 5.10: Example from Lemma 50.

We could conceivably relax the rules about critical section structures to allow nesting

of bracket nodes inside `accept-do` constructs of other bracket nodes. However, since this construct never appeared in the test data, the value of doing so is probably low.

Identifying critical sections.

The following algorithm can be used to identify critical sections. It relies on doing inexpensive tests first.

Algorithm 5 *Identify critical section structures.*

Input: The PADA sync hypergraph $G = (N, E_C E_S)$.

Output: A set of critical section structures, CRITSTRUCTS.

Procedure:

1. Create a partition $\mathcal{P}$ such that each $P_i \in \mathcal{P}$ contains all nodes of a particular signal type and polarity in G.
2. Create the control flow graph $G' = (N, E_C)$.
3. Set CANDBODIES to $StrictInt(G', \mathcal{P})$.
4. For each pair $(a, b) \in$ CANDBODIES found in step 3, form a *candidate critical section structure* (M, N) such that:
   - M is the set of pairs $(a', b') \in$ CANDBODIES, such that:
     - $SigType(a) = SigType(a')$, where $SigType(n)$ is the signal type of node n;
     - $NodeType(a) = NodeType(a')$, where $NodeType(n)$ is the node type (accept-in or signaling) of node n;
     - $SigType(b) = SigType(b')$;
     - $NodeType(b) = NodeType(b')$.
   - N is the set of pairs $(a'', b'') \in$ CANDBODIES, such that:
     - $SigType(a) = SigType(a'')$;
     - $NodeType(a) \neq NodeType(a'')$;

- $SigType(b) = SigType(b'')$;
- $NodeType(b) \neq NodeType(b'')$.

In short, for all pairs $(a', b') \in M$ and $(a'', b'') \in N$, a' can rendezvous with a'' and b' can rendezvous with b''.

Place all the candidate critical section structures into the set CANDSTRUCTS.

5. Eliminate from CANDSTRUCTS all pairs (M, N) such that $|M| = 0$ or $|N| = 0$.

6. Eliminate from CANDSTRUCTS all pairs (M, N) such that $M \cup N$ contains a pair (m, n) such that:

   - $Scope(m, y_m) > 0$;
   - $Scope(m, y_n) > 0$;
   - $Scope(n, y_m) > 0$; or,
   - $Scope(n, y_n) > 0$.

7. For each pair (M, N) in CANDSTRUCTS, let B be whichever of M or N has all nodes of its pairs as members of the same task, and let C be the other set. (If both sets meet this qualification, choose arbitrarily.) Place the resulting pair (B, C) in CRITSTRUCTS.

Lemma 51 (Correctness of critical section identification.) *Algorithm 5 correctly identifies the critical section structures of a program.*

Proof: From the discussion, following step 6, CANDSTRUCTS contains only pairs (M, N) such that either (M, N) or (N, M) meets the definition of a critical section structure (as defined in Section 5.1.4 and elaborated in Sections 5.1.4 and 5.1.4). Step 7 identifies which of these two alternatives enforces the requirement that the critical section bodies be in each others' CHT sets. All steps maintain validity of the results of the previous steps. Thus, at the termination of the algorithm, CRITSTRUCTS is the set of critical section structures of the program. □

Lemma 52 *Algorithm 5 uses $\mathcal{O}(|N|^3)$ time in the worst case.*

Proof: The partitioning of step 1 can be done in $\mathcal{O}(|N|)$ time; creating G' from G can be done in constant time. Using Algorithm 4, the CANDBODIES set of step 3 can be created in $\mathcal{O}(|N|^3)$ time.

Step 4 can be accomplished by sorting the node pairs in CANDBODIES, keyed on the signal and node types of a and b. The sort will take $\mathcal{O}(|N|^2 \log(|N|))$ time. The result of the sorting will be a list with at most $4|Sigs|(|Sigs| - 1)$ members. For each member M of this list, we find its corresponding N by looking up the pair with the same signal types and opposite polarity. The lookup process thus takes $\mathcal{O}(|Sigs|^2 \log(|Sigs|))$ time. So the total time for Step 4 is $\mathcal{O}(|N|^2 \log(|N|))$ time, since the number of nodes is at least the number of signals used by all nodes in the program. During the construction of the CANDSTRUCTS set, we can also eliminate the candidate structures with a zero-size set (Step 5).

Therefore, the total time to construct CRITSTRUCTS is $\mathcal{O}(|N|^3)$ in the worst case. □

Expanding *CHT* information using critical section structures.

Algorithm 6 *Expanding CHT information using critical section structures.*

Input:

- The CRITSTRUCTS set, as generated by Algorithm 5.
- CHT.

Output:

- $CHT' \supseteq CHT$.

Procedure:

1. $CHT' \Leftarrow CHT$.
2. For each critical section structure (B, C) in CRITSTRUCTS do:

(a) $CSBNodes \Leftarrow \bigcup_{CSB(n,x) \in B} CSB(n,x)$.

(b) $CallNodes \Leftarrow \bigcup_{CallB(c,r) \in C} CallB(c,r)$.

(c) $CHT' \Leftarrow CHT' \cup ((CallNodes^2 \cup CSBNodes^2) - I)$.

Nodes in call bodies or critical section bodies of the same structure can't happen together. If n is a call node of (B,C), then $CallNodes^2(n)$ is the set of all call nodes of (B,C); otherwise, $CallNodes^2(n)$ is empty.

(d) $MustHT \Leftarrow ((CSBNodes \times CallNodes) \cup (CallNodes \times CSBNodes))$.

If n is a node in a call body for (B,C), then $MustHT(n)$ is the set of all nodes in critical section bodies for (B,C). Likewise, if n is in a critical section body, $MustHT(n)$ contains the set of all nodes in all call bodies. If n is not in a call or critical section body for (B,C), then $MustHT(n)$ is empty. If $MustHT(n)$ is non-empty, then each instance of n must happen together with instances of nodes in $MustHT(n)$.

(e) $CHT' \Leftarrow Widen(CHT', MustHT, N)$.

Lemma 53 (Correctness of Algorithm 6.) *If $CHT \subseteq CHT_{perf}$ and the other input parameters of Algorithm 6 are correctly specified, then $CHT \subseteq CHT' \subseteq CHT_{perf}$.*

Proof: From Theorems 1 and 6. □

Lemma 54 *Algorithm 6 requires $\mathcal{O}(|N|^4)$ setup time and $\mathcal{O}(|N|^5)$ time per iterative refinement, in the worst case.*

Proof: Computing *CSBNodes* and *CallNodes* for each critical section structure can be done while the critical section structure is being constructed. Computing *MustHT* for each critical section structure can be done in $\mathcal{O}(|N|^2)$ time, and need only be done once in the life of the program. Step 2c likewise requires $\mathcal{O}(|N|^2)$ time, and need only be done once in the life of the program.

The call to *Widen* of step 2e requires $\mathcal{O}(|N|^3)$ time per critical section structure; since it is the step that uses widening to refine CHT, it must be done every time the refinement is done.

There may be up to $\mathcal{O}(|N|^2)$ critical sections, each containing $\mathcal{O}(|N|)$ nodes. Such program structures are realizable. Thus, Algorithm 6 requires $\mathcal{O}(|N|^4)$ setup time and $\mathcal{O}(|N|^5)$ time per iterative refinement, in the worst case. □

Note that the worst case time assumes a pathological case which is not likely to occur. In practice, we expect only a small number of critical section structures. However, due to the large asymptotic time bound on widening, we may wish to restrict the number of times iterative refinement is done on critical sections to a constant number, and pay some penalty in loss of *CHT* information. We will assume here that we wish to do full iterative refinement of *CHT* using critical sections, and will investigate the effects of limiting iterative refinement in our experimental work.

Lemma 55 (Time bounds for the critical section algorithm.) *The time to find critical section structures and use them to expand CHT information is* $\mathcal{O}(|N|^5)$ *in the worst case.*

Proof: From Lemmas 52 and 54. □

However, we may do better than this by using an incremental version of Algorithm 1 (the widening algorithm):

Lemma 56 (Incremental time bounds for the critical section algorithm.) *The time to incrementally expand CHT information is* $\mathcal{O}(|N|^3)$ *per bit change in CHT in the worst case.*

Proof: Each call to the incremental version of *Widen* requires $\mathcal{O}(|N|)$ time. If there are $\mathcal{O}(|N|^2)$ critical section structures, a total of $\mathcal{O}(|N|^3)$ will be required by the incremental expansion. □

Deadlocks with critical section nodes as head nodes.

Example.

Figure 5.11 shows a case in which two tasks each use nested calls to critical sections. The program here has two critical section bodies, (e, f) and (g, h). (e, f) is called by (b, c) and (k, l), while (g, h) is called by (a, d) and (j, m). The system cannot deadlock.

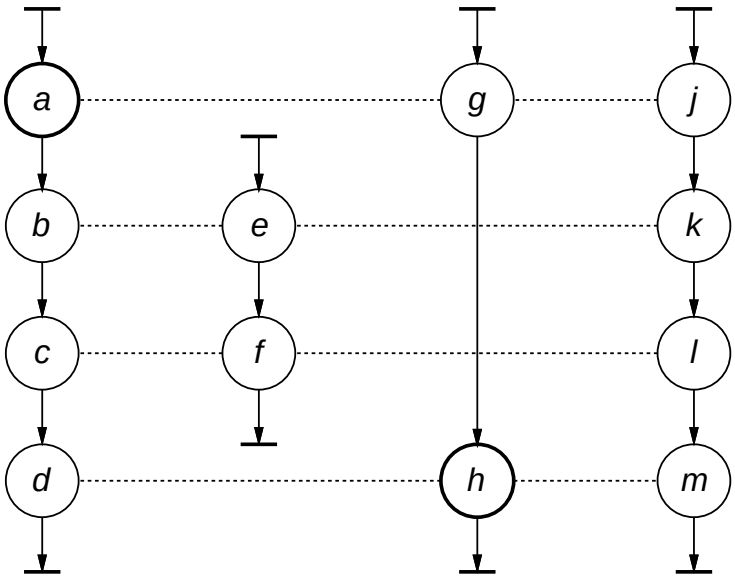

Figure 5.11: A program with nested calls to critical sections.

Table 5.1.4 shows the *CHT* relation computed between nodes of the program of Figure 5.11.

After unrolling the loops, we have a possible cycle (b, c, f, e', b), since b and f can happen together. (The fact that they can is a true *CHT* relation rather than an approximation, since the execution wave $[b, f, h, l]$ is feasible. The cycle (b, c, f, e', b) happens to violate Constraint 4, because whenever head nodes b and f execute together, l must also be available to rendezvous with f.) However, if we do pinning analysis, we find that f must always happen together with some node that can rendezvous with f. Therefore, f cannot be a head node, and (b, c, f, e', b) is not a valid deadlock cycle. For the same reason, (k, l, f, e', k) is not a valid deadlock cycle.

We notice a more serious problem with the cycle (a, b, c, d, h, g', a), which also violates Constraint 4. The violation occurs because one of k, l, or m must be executing, and could rendezvous with e, f, or h. Note that one of e or f must always be executing.

In the cycle (a, b, c, d, h, g', a), a and h are the head nodes, and they also can happen together (in waves $[a, e, h, m]$ and $[a, f, h, l]$). Pinning analysis does not show that either a or h must have a rendezvous available. Without further information, therefore, deadlock cycle (a, b, c, d, h, g', a) must be considered valid.

Node	*CHT* Set
a	b,c,d
b	a,c,d,f,g,k,l,m
c	a,b,d,e,g,k,l,m
d	a,b,c,f,g,k,l,m
e	c,f,l
f	b,d,e,g,k,m
g	b,c,d,f,h,k,l,m
h	g
j	k,l,m
k	b,c,d,f,g,j,l,m
l	b,c,d,e,g,j,k,m
m	b,c,d,f,g,j,k,l

Table 5.3: *CHT* Relation for the program of Figure 5.11.

Bracket node propagation and deadlock cycle detection.

The following theorems and lemmas allow us to eliminate (a, b, c, d, h, g', a) in Figure 5.11 as a spurious deadlock cycle, without constructing a critical section call graph.

Theorem 8 (Bracket nodes in a deadlock.) *In a deadlock, the nodes on the execution wave for those tasks which participate in a critical section structure cannot consist solely of the bracket nodes of the structure.*

Proof: Suppose the opposite, i.e., that there is a deadlock such that, for all tasks participating in a critical section structure, the nodes on the execution wave are all bracket nodes of the structure. Either one of the critical section bodies is executing or none are.

If no critical section body is executing, then, by hypothesis, the entry node n of one of the critical section bodies is on the execution wave. Since any call body must happen together with some critical section body, no call body can be executing. Therefore, the call node c of some call body must be on the execution wave. In that case, the wave cannot be deadlocked, because n and c can rendezvous.

Now suppose that some critical section body is executing. By hypothesis, its exit node x must be on the execution wave. The critical section body must happen together with some call body. Also by hypothesis, the call body's return node r must be on

the wave. But x and r can rendezvous, so the wave cannot be deadlocked in this case, either. □

In connection with critical sections, we must be especially careful about the term "valid deadlock cycle," and the nature of the "waits-on" relationship. Consider the cycle (c, r, x, n', c) of Figure 5.12. The critical section exit node x could be said to wait on either r or r'. However, it more directly waits on r', since r' belongs to the call body which is executing concurrently with the critical section. Thus, if there is a valid deadlock cycle in this case, it must contain q as a head node, where q is a member of the call body containing the next $r \in R$ that can possibly execute. The cycle (c, r, x, n', c) is *irrelevant*, since it would exist in any program with two call bodies, regardless of whether the program deadlocked or not.

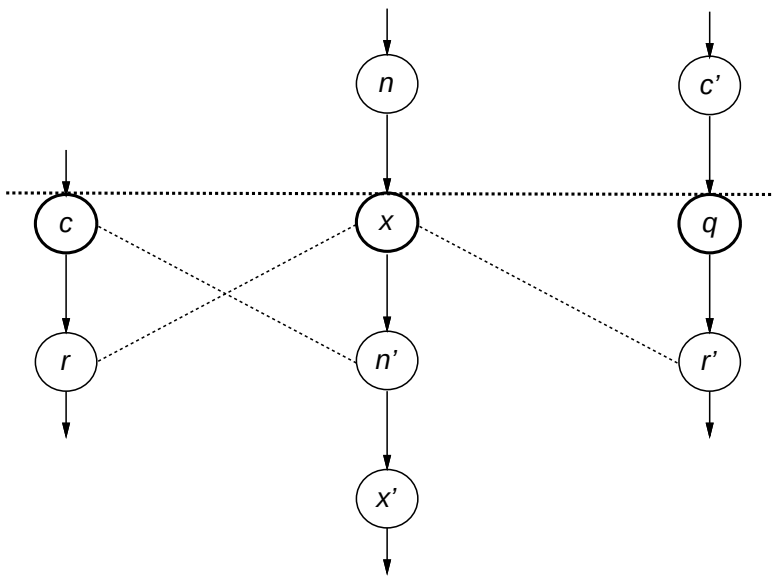

Figure 5.12: Spurious deadlock cycle which does not indicate a real "waits-on" relationship.

In Lemmas 57 through 60, we justify our assertion that deadlock cycles having only the bracket nodes of a single critical section structure as their head nodes are irrelevant.

Lemma 57 *If any valid deadlock cycle includes the exit node of a critical section body as a head node, there must also be an infinite wait anomaly, such that either (a) the anomaly includes a member of a call body for that critical section (other than the return*

node) as a head node of a deadlock or starved task, or (b) in the anomaly, a call body member (other than the return node) is transitively connected to a deadlock or starved task.

Proof: See Figure 5.13. The exit node x is a member of a valid deadlock cycle, by hypothesis. Therefore, it must be a member of an anomalous execution wave. x may only be present on the wave if a node q in a call body for x's critical section is also present. Furthermore, q cannot be the return node r (which could rendezvous with x). By Theorem 1 of [MR90a], q must be a starved node, or a head node of a deadlock cycle, or transitively connected to a starved task or a deadlock. □

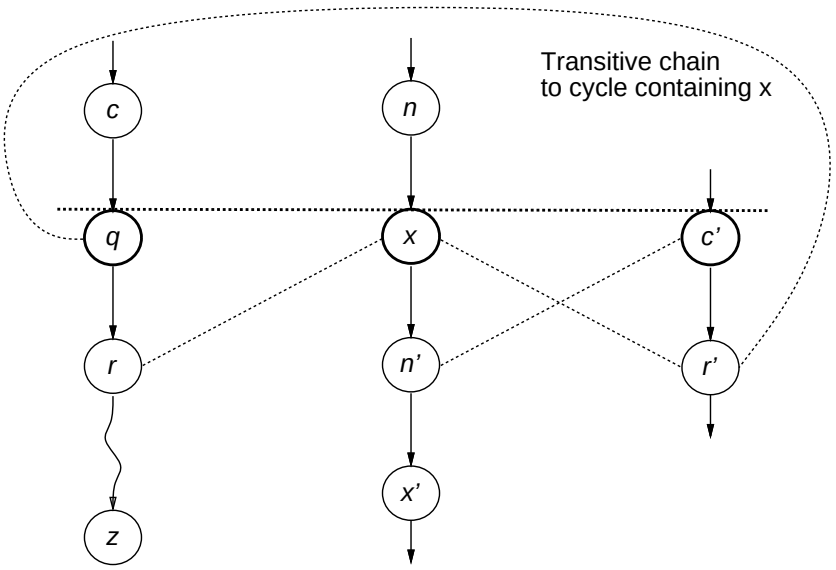

Figure 5.13: Deadlock cycle with a critical section exit node as head node.

Lemma 58 *If any valid deadlock cycle includes the return node of a call body as a head node, then the cycle must also include a member of the critical section body (other than the exit node) as a head node.*

Proof: See Figure 5.14. The return node r is a member of a valid deadlock cycle, by hypothesis. Therefore, some member node q of a corresponding critical section body is executing along with r. $q \neq x$, for otherwise q and r could rendezvous. r can only rendezvous with nodes in $Task(q)$, by the definition of a critical section structure. Therefore, the deadlock cycle must include q as well as r as head nodes. □

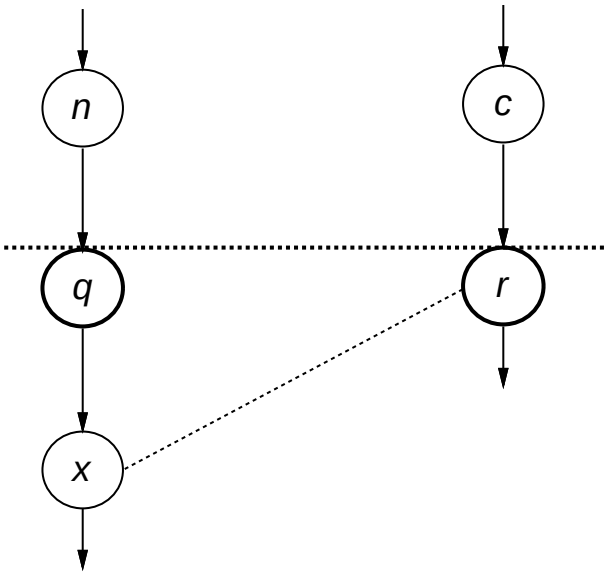

Figure 5.14: Deadlock cycle with a call body return node as head node.

Lemma 59 *If any valid deadlock cycle includes the entry node of a critical section body as a head node, then the deadlock cycle cannot include the call or return node of any call body for that critical section.*

Proof: If the deadlock cycle included the entry node and any call node, then the two could rendezvous. The deadlock cycle cannot include the entry node and any return node, since these can't happen together. □

Lemma 60 *If any valid deadlock cycle includes the call node of a call body as a head node, then the deadlock cycle cannot include the entry node of any critical section body for that critical section.*

Proof: If the deadlock cycle included the call node and any entry node, then the two could rendezvous. □

Theorem 9 (Bracket nodes and deadlock cycles.) *Any deadlock cycle which contains any bracket nodes of a critical section as head nodes must either contain some node other than the bracket node as a head node, or must be transitively coupled to another deadlock cycle or starved task.*

Proof: By Lemma 57, if the deadlock cycle contains an exit node x, it must contain,

or be coupled to, a non-bracket head node q. If the cycle contains q, then the assertion is proved. If the cycle does not contain q, then either there is a path, along forward control edges and sync edges, from q to a node z which is reachable along forward control edges from a head node of the deadlock cycle and which exits q and enters z via sync edges, or there is no such path. (See Figure 5.15.) If there is such a path, then there is a deadlock cycle containing q. If there is no such path, then q is not waiting on any head node of the cycle, and there must thus be another anomaly with q as a head node or one on which q waits transitively. Thus, if the cycle contains an exit node x, then either there is a cycle containing a non-bracket node q or there is a separate anomaly.

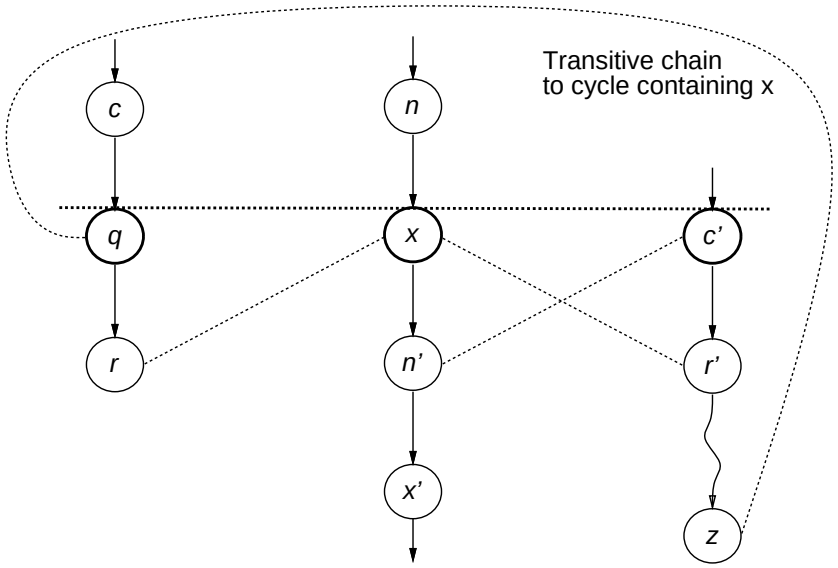

Figure 5.15: Cycles involving exit nodes.

By Lemma 58, if the deadlock cycle contains a return node, it must contain a non-bracket node.

By Lemma 59, if the deadlock cycle contains an entry node as a head node, it cannot contain any call or return nodes. Furthermore, no valid cycle can contain two entry nodes, or an entry and an exit node, since these are all in the same task, and thus can't happen together. So a deadlock cycle which contains an entry node must contain other nodes which are not in the bracket node set for the critical section.

By Lemma 60, if the deadlock cycle contains a call node, it cannot contain any entry nodes. If any call node c in a cycle is a head node, then an entry node n must be the tail node in the cycle which is connected to c via a sync edge. Thus, no single cycle contains two call nodes. (See Figure 5.16.) If the cycle contains c and no other members of the bracket set, then the assertion is trivially proved. If the cycle contains c and a return node r, it must contain a non-bracket node by Lemma 58. If the cycle contains c and an exit node x, then there must either be a cycle which contains a non-bracket node q or an infinite wait anomaly separate from the cycle. So a deadlock cycle which contains a call node must contain head nodes not in the bracket set, or must be transitively coupled to an anomaly containing head nodes not in the bracket set. □

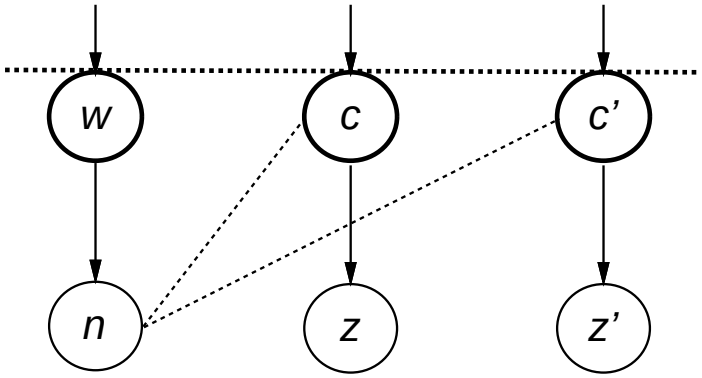

Figure 5.16: Cycles involving multiple call nodes.

Theorem 9 does not exclude the possibility of deadlock cycles containing call and exit head nodes, as the cycle (x, n', c', r', x) in Figure 5.13 shows. In fact, such cycles will exist in any program with more than one call body to a critical section, such that the call bodies can happen together with each other and with a critical section body. However, in such a case, the lack of a rendezvous for c' is not the immediate reason why x cannot leave the execution wave; rather, the more immediate reason is that q cannot rendezvous and allow r to reach the wave. The cycle (x, n', c', r', x) is at most symptomatic of another problem, more directly involving q.

If the problem that causes q to have no rendezvous is that there is a deadlock cycle including q, then we can find the cycle by hypothesizing q or c' as a head node. If the problem is a deadlock cycle on which q waits transitively, then the deadlock can be

found by tracing cycles which do not include the bracket nodes as head nodes.

We conclude that it is not necessary to propagate bracket nodes of a critical section to other bracket nodes of the same critical section in deadlock cycle tracing. Doing so will increase the false alarm rate, and will provide no additional deadlock information. Such tracing can easily be inhibited by preventing propagation of non-top summaries for a bracket node of a critical section along sync edges to bracket nodes of the same critical section. With this step, it is possible to eliminate the spurious deadlock cycles containing nodes a and h in Figure 5.11, and in other programs in which critical section bodies and call bodies contain non-bracket nodes.

Signaling head node propagation and critical section structures.

As an optimization step in deadlock cycle detection, we avoid tracing the same cycles starting at both signaling and accept-in head nodes. Signaling nodes are thus ordinarily assumed as head nodes only to trace cycles whose head nodes are all signaling nodes. To accomplish this, signaling node summaries are ordinarily propagated only to other signaling head nodes, in cases where the analysis would remain conservative.

If we eliminate propagation of bracket head node summaries to other bracket head nodes of the same critical section structure, we must allow signaling bracket node summaries to propagate to accept-in head nodes. For instance, Figure 5.17, a reduced example of the widely-known "dining philosophers" problem, does indeed deadlock, and all nodes in the program are bracket nodes. The deadlock cycle contains head nodes a, f, k, and h, in that (or some equivalent) order. Nodes f and h cannot propagate to other head nodes of the cycle, since they are bracket nodes of the same critical section structure containing k and a respectively. Nodes a and k are signaling nodes; their summaries must propagate to accept-in head nodes if the cycle is to be detected.

Thus, we allow the summaries of signaling head nodes to propagate to accept-in head nodes only when the signaling nodes are bracket nodes of a critical section structure.

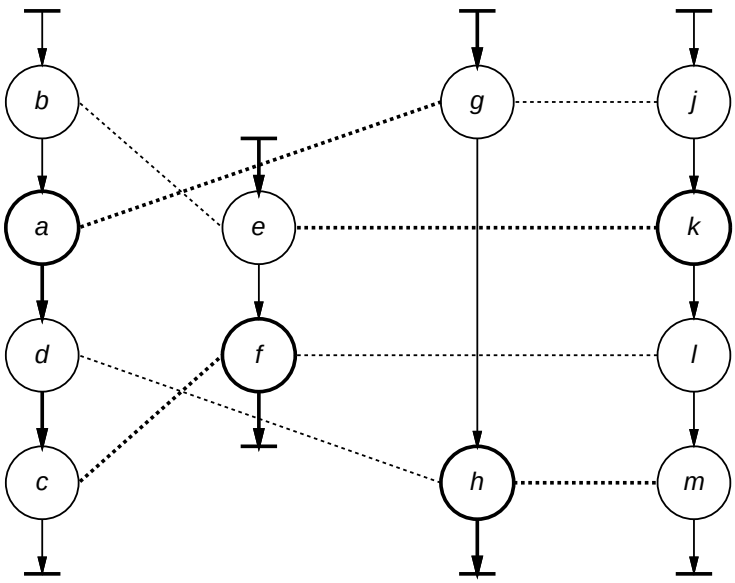

Figure 5.17: A simplified version of the "dining philosophers" problem with two philosopher tasks.

Critical sections which call other critical sections.

Here we show that neither CHT information, nor the propagation restriction of Section 5.1.4, can eliminate some spurious deadlock cycles that occur when calls to different critical sections are nested.

Cycle (a, b, c, d, h, g', a) of Figure 5.11 is a spurious deadlock cycle whose head nodes are all bracket nodes of the same critical section structure. If we do not propagate bracket head nodes of a structure to other bracket head nodes of the same structure, we can avoid detecting this spurious deadlock.

However, other spurious cycles may involve the bracket nodes of more than one critical section structure. For instance, in Figure 5.11, cycle $(a, b, c, f, e', k, l, m, h, g', a)$ (with head nodes a, f, h, and k) is spurious, because $k \in CHT(f)$. However, if a is the hypothesized head node for this cycle, the cycle cannot be eliminated, since none of the head nodes are in $CHT(a)$. Cycle $(a, b, e, f, l, m, h, g', a)$, with head nodes a, e, l, and h, is also spurious, because $l \in CHT(e)$; again, none of the head nodes are in $CHT(a)$, so the cycle is not eliminated.

A naive solution.

We might try to eliminate these cycles by carrying, in the summaries for each hypothesized head node h, an (under)estimate of CHT for all head nodes of cycles including h. To maintain a conservative estimate, meet of CHT sets must be intersection.

However, this CHT summary information will not eliminate the spurious cycles. Going back to Figure 5.11 and Table 5.1.4, the CHT summary info for node a, propagating into node e along a sync edge, would be $CHT(a) \cup CHT(e) = \{b, c, d, f, l\}$. Similarly, propagating into f along a sync edge, we have $CHT(a) \cup CHT(f) = \{b, c, d, e, g, k, m\}$. Summaries propagate between e and f along control edges, so the summary for a at both nodes is the meet (intersection) of the two CHT sets, i.e., $\{b, c, d\}$. This summarized CHT set does not eliminate either the cycle $(a, b, c, f, e', k, l, m, h, g', a)$ (with head nodes a, f, h, and k), or the cycle $(a, b, e, f, l, m, h, g', a)$ (with head nodes a, e, l, and h).

Critical section call graphs.

To resolve the problem of nested critical section calls, we can move up one level of abstraction and examine the *critical section call graph* of the program, following a construction developed by Saxena [Sax77] for binary semaphores. The critical section call graph contains a node corresponding to each critical section, and an edge (x, y) if critical section y is called from within a call body of x at any point in the program (or if y is called from within critical section body x).

Figure 5.18 shows the critical section call graph for the program of Figure 5.11. If the critical section call graph is acyclic, then the set of critical section calls cannot deadlock.

Critical section structures may, in general, have more complicated behavior than binary semaphores. For instance, the critical section bodies may contain synchronization, or may not happen together with all the call bodies. To avoid these complications, we do not construct a call graph of all critical sections. Instead, we limit ourselves to a subset of critical section structures which mimic semaphores. For this subset, a simplified

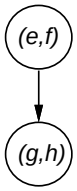

Figure 5.18: Critical section call graph for the program of Figure 5.11.

analysis is possible.

Nested calls to simple critical sections.

In this section, we elaborate on a special case corresponding to a common use of critical sections: acquiring and releasing resources by essentially using critical section structures to mimic binary semaphores. We show how spurious deadlock cycles associated with nested critical section calls may be eliminated in this special case. Although the special case has not yet been seen in experimental data gathered from outside sources, it seems to us that it might occasionally be seen in practice; we would thus like to be prepared for this eventuality.

A critical section structure is *simple* if the entry/exit node pair (n, x) for each of its critical section bodies are the only rendezvous nodes in $Task(n)$. Thus, one of n or x (or the task end node of $Task(n)$) must be the entry for $Task(n)$ in all execution waves.

Simple critical section structures have several features that make deadlock cycle pruning easier than in general critical section structures. Return nodes of simple critical sections may not participate in deadlock cycles, since an exit node of the same structure must be on the wave whenever the call body is active. Call nodes, on the other hand, may participate in deadlock cycles, but only when the cycle includes a node in a call body for the simple critical section.

Given two simple critical section structures S and S', structure S has a *nested call* to S' if any call node of S' is contained within any call body of S. We can elaborate this by transitive closure in the following recursive definition: S has a *transitively nested call* to S' if S has a nested call to S', or if S has a nested call to some structure S'' which has a transitively nested call to S'. Finally, a nested call from S to S' is an edge in a

strong component of the critical section call graph if there is also a transitively nested call from S' to S.

Lemma 61 below shows how the nesting structure of simple critical sections may be

Lemma 61 *In a proposed anomalous execution wave E, if all head nodes in deadlock cycle $D \subseteq E$ are bracket nodes of simple critical sections, and no sync edge to any head node in D represents an edge in a strong component of the critical section call graph, then either the deadlock D represents is artifactual, or E is not a possible execution wave.*

Proof: Define the *dynamic nest level* of each bracket node $d \in D$ of a simple critical section structure as the maximum length of any transitive call chain from any bracket node (of a simple critical section structure) in D, to the structure containing d (or zero, if there is no such call chain for d). By the assumption that no sync edge to any head node in D represents an edge in a strong component of the critical section call graph, all such call chains are of finite length. Thus, there is at least one node (and therefore at least one structure) whose dynamic nest level is the greatest in D.

Suppose some call node c of a structure S is at the maximum dynamic nest level, as in Figure 5.19(a). If c is on the wave, then some exit node x of S must also be on the wave; x must also be in the cycle, since otherwise the cycle would not be connected through c. x can only be on the wave when a node z in some other call body $CallB(c', r')$ for S is in E. If $z \notin D$, then D would not be a valid deadlock cycle if z could enter the wave; D is thus only symptomatic of the anomaly which keeps z from entering the wave. If $z \in D$, then z must be a call node of a simple critical section structure (since we have hypothesized this.) Since x is executing, z must also be nested inside a call body of S; x is waiting for z to complete and for r to reach the wave. Thus, c cannot be at the maximum dynamic nest level.

If some exit node $x \in D$ of structure S is at the maximum dynamic nest level, then some call node c of S must also be in D. The situation of Figure 5.19(a) still holds in this case.

If some return node r of S is at the maximum dynamic nest level, as in Figure

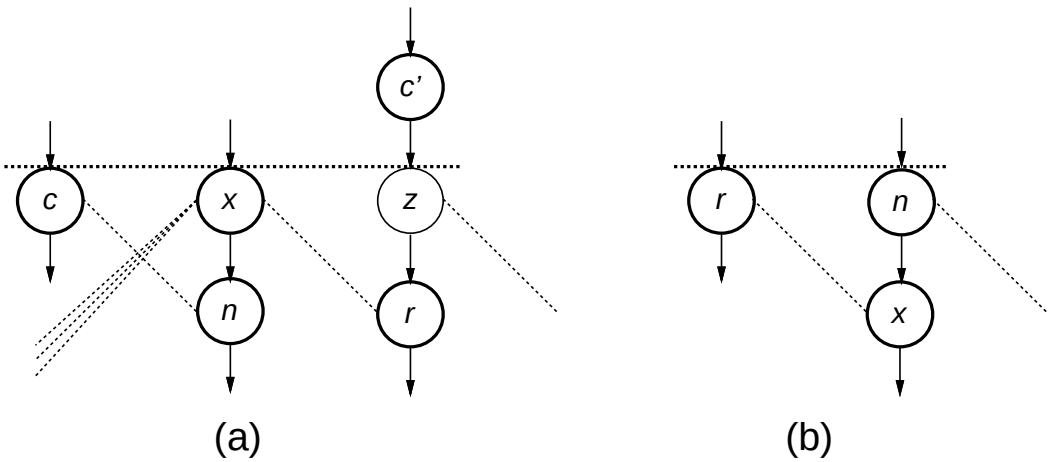

Figure 5.19: Simple critical section structure examples from Lemma 61.

5.19(b), then the entry node n of some critical section body of S must also be on the cycle. This is not possible, since $r \in CHT(n)$. Thus, E is not a possible execution wave. Finally, if an entry node n of S is at the maximum dynamic nest level, the situation of Figure 5.19(b) still applies.

Thus, either D represents a deadlock that is only symptomatic of another anomaly, or E is not a valid execution wave. □.

From Lemma 5.19, we conclude that we may safely avoid propagating summary information for assumed head nodes out of their tasks, along sync edges corresponding to acyclic regions of the call graph of simple critical sections. This will eliminate the spurious deadlock cycles involving calls between properly nested simple critical sections.

Algorithm for constructing the simple critical section call graph.

We present here an algorithm for identifying simple critical sections and building the simple critical section call graph. Because experimental evidence has not yet suggested that nested simple critical sections are common, we have not integrated it into the rest of deadlock detection; we present it here only to demonstrate that the general polynomial time bound of the algorithm would be maintained if it were added.

Algorithm 7 *Construct the simple critical section call graph; identify bracket nodes in the sync graph corresponding to calls in acyclic regions of the simple critical section*

call graph.

Input:

- The PADA sync hypergraph, $G = (N, E_C, E_S)$.
- The set of critical section structures, CRITSTRUCTS.

Output:

- The array *Acyclic*, such that $n \in Acyclic(m)$ if m and n are bracket nodes of simple critical section structures S and S' respectively, such that n is contained in the call body whose call node is m, and the edge (S, S') is in an acyclic region of the simple critical section call graph.

Procedure:

1. Let *SimpleCSS* be the set of critical section structures in CRITSTRUCTS whose critical section bodies contain all nodes in their respective tasks.
2. Let E_{call} be the set of ordered pairs $(S, S'), S, S' \in SimpleCSS$ such that some call body in S contains a call node of S'.
3. Construct the simple critical section call graph $CallG = (SimpleCSS, E_{call})$.
4. Find the strong components of *CallG*.
5. Remove from E_{call} all edges between nodes in any strong component of *CallG*.
6. For each pair $(S, S') \in E_{call}$, add the bracket nodes of S' to $Acyclic(S)$.

Lemma 62 (Time bounds for the simple critical section algorithm.) *Algorithm 7 requires $\mathcal{O}(|Tasks|^2|N|^2)$ time to execute, in the worst case.*

Proof: There are at most $\mathcal{O}(|N|^2)$ critical section bodies in the program. For each of these bodies, we can look up the task containing the body in constant time (assuming that we have precomputed this information during construction of the sync graph) to check the number of nodes in the the task. Step 1 thus requires $\mathcal{O}(|N|^2)$ time in the worst case.

Step 2 can be done by checking the set of signal types of the nodes within the call bodies of each simple structure against the set of signal types of call nodes of the other simple structures. This may be accomplished by traversing the sync graph along its control edges. If the traversal enters a call body, we look up, in constant time, the simple critical section structure we have entered, using a precomputed map from task names (which are available as part of the signal type of the traversed node) to simple critical section structures. (The map requires $\mathcal{O}(|Sigs|)$ preprocessing time to set up.) If we are in a call body for a simple critical section S, and encounter a call node for another simple critical section S', then we add an edge (S, S') to E_{call}. The traversal thus requires $\mathcal{O}(|N| + |Sigs|)$ total time, including setup (or $\mathcal{O}(|N|)$, since $—Sigs—= \mathcal{O}(|N|)$.)

One upper bound on the size of E_{call} is $\mathcal{O}(|N|)$, since each node adds at most one edge to the call graph. Another upper bound is $\mathcal{O}(|Tasks|^2)$, since the call graph has at most $\mathcal{O}(|Tasks|)$ nodes.

The strong components of $CallG$ may be found in time proportional to the number of nodes and edges of $CallG$ [ASU86]; step 4 thus takes $\mathcal{O}(|Tasks|^2)$ in the worst case. Step 5 can then be done via a simple scan of E_{call}, in $\mathcal{O}(|Tasks|^2)$ time. Following this, we can do a chase on the remaining call graph edges in $\mathcal{O}(|Tasks|^2)$ time, to form the transitive closure of the edge set. We replace E_{call} by the transitive closure.

Finally, for each pair (S, S') remaining in E_{call}, we add the bracket nodes of S' to the set $Acyclic(s)$ for each bracket node s of S. This step takes at most $\mathcal{O}(|Tasks|^2|N|^2)$ time.

Thus, the total time to construct the $Acyclic$ array is $\mathcal{O}(|Tasks|^2|N|^2)$ time. □

5.1.5 A more general definition of critical sections.

The critical section structures of Section 5.1.4 require that the critical section and call bodies be delineated as single-entry, single-exit regions. In this section, we show how this restriction can be weakened, and show how a multiple-entry, multiple-exit region can enforce CHT between a pair of nodes. We emphasize that the analysis of this

section has not yet been integrated into the deadlock algorithm; we show only some examples in which they might be used in a future algorithm.

Two-task mutual exclusion.

To insure that $m \in CHT(n)$, we must be able to show the following:

- No instances of m and n can start at the same time.
- Whenever any new instance of m starts, no instance of n is executing.
- Whenever any new instance of n starts, no instance of m is executing.

To demonstrate this, it is sufficient to show:

- For each instance m_i, there is a sequence of instances $(x(1) = d_i, ..., x(k) = n_j)$, where d_i is a descendant of m_i within $L_i(m)$. We will call this sequence an *ordering chain* from m_i to n_j. We further require that ordering chains have the following properties:
  - Each $x(h+1)$ is either a proper control successor of $x(h)$, or does rendezvous with $x(h)$;
  - n_j is a proper control successor of $x(k-1)$;
  - If $x(g)$ does rendezvous with $x(g+1)$ and $x(h \neq g)$ does rendezvous with $x(h+1)$, then $Task(g) \neq Task(h+1)$.
  - When the chain enters a task, it leaves the task in the same loop body in which it entered.

    More formally, for each subchain

    $$(x(h+1), x(h+2), ..., x(h+g))$$

    such that the subchain from $x(h+1)$ to $x(h+g)$ contains all elements of the chain that are in a particular task, instance $x(h+1)$ is in the same iteration as $x(h+g)$.

The subchain from $x(h+1)$ to $x(h+g)$ contains all elements of the chain that are in a particular task if:

* $x(h+1) = d_i$ and $x(h+g) = n_j$, and each node $x(h+e), 2 \leq e \leq g$ is the control successor of $x(h+e-1)$; or,
* $x(h+1) = d_i$ and $x(h+g)$ does rendezvous with $x(h+g+1)$, and each node $x(h+e), 2 \leq e \leq g$ is the control successor of $x(h+e-1)$; or,
* $x(h+g) = n_j$, and $x(h)$ does rendezvous with $x(h+1)$, and each node $x(h+e), 2 \leq e \leq g$ is the control successor of $x(h+e-1)$; or,
* $x(h)$ does rendezvous with $x(h+1)$, and $x(h+g)$ does rendezvous with $x(h+g+1)$, and each node $x(h+e), 2 \leq e \leq g$ in the subsequence is the control successor of $x(h+e-1)$.

The existence of an ordering chain establishes that, for instance n_j, there exists some instance m_i (or its descendant within iteration i of its loop body) such that m_i is ordered before n_j.

- No new instance $m_{i'}, i' > i$ can start execution until n_j finishes.
- When the next instance $m_{i'}, i' > i$ starts, no instance of n can be executing.
- When the next instance $n_{j'}, j' > j$ starts, no instance of m can be executing.

Theorem 10 gives specific conditions on the sync hypergraph for which we can prove $m \in CHT(n)$. We introduce the notation:

- $L(n)$ denotes the loop body most immediately containing n.
- $LS(n)$ denotes the start of the loop body most immediately containing n.
- $LE(n)$ denotes the end of the loop body most immediately containing n.

Theorem 10 *Let m and n be nodes in the sync hypergraph. Let X be a subset of proper ancestors of n within $L(n)$ which can rendezvous only with descendants of m within $L(m)$, and let W be the set of nodes with which X may rendezvous.*

Let Z be a set of proper descendants of m within $L(m)$ which can only rendezvous with descendants of n within $L(n)$, and let Y be the set of nodes with which the members of Z may rendezvous. If the following conditions hold:

- *X cuts $(\{LS(n)\}, \{n\})$;*
- *Z cuts $(\{m\}, \{LE(m)\})$;*
- *Z cuts $(W, \{LE(m)\})$.*

Then no instance of m can happen together with any instance of n.

Proof: See Figure 5.20. Without loss of generality, m and n are in different tasks. Since X cuts $(\{LS(n)\}, \{n\})$, (and, presumably, there is a control path from $LS(n)$ to n), X is non-empty. Since Z cuts $(\{m\}, \{LE(m)\})$, Z is non-empty.

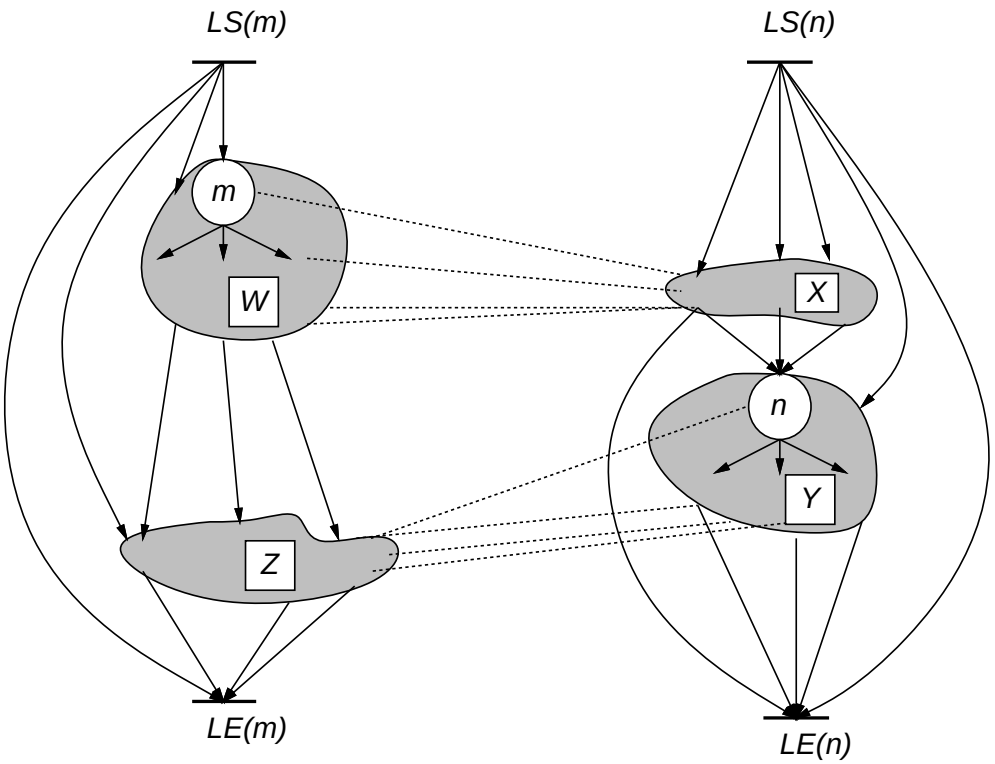

Figure 5.20: *CHT* example from Theorem 10.

Before each instance n_j of n, a rendezvous must occur between instance i of some node in W and instance j of some node in X. Suppose that n_j executes, and that

$x_j \in X_j$ has a rendezvous with instance $w_i \in W_i$. If m_i executes, then either $m_i = w_i$ or m_i precedes w_i. Thus, if m_i and n_j both execute, then m_i precedes n_j.

Next we show that when the next instance $n_{j'}, j' > j$ executes, then any instance of m which started after n_j must be off the wave when $n_{j'}$ starts. Suppose that instances $m_{i+1}...m_{i'}$ start between n_j and $n_{j'}$. In order for $n_{j'}$ to start, some node instance in $X_{j'}$ must complete a rendezvous with an instance of a node in W. If this instance is in $W_{i'}$, then $m_{i'}$ must complete before $n_{j'}$ starts. If this node is in $W_{i''}, i < i'' < i'$, then $m_{i'}$ cannot start execution until some instance in $Z_{i''}$ has rendezvoused with an instance in $Y_{j'}$, i.e., not until $n_{j'}$ is off the wave. This contradicts the assumption that $m_{i'}$ starts before $n_{j'}$ does.

Finally, we show that when the next instance $m_{i'}, i' > i$ executes, then any instance of n which starts after m_i must be off the wave when $m_{i'}$ starts. Preceding any instance of n, a node in X must rendezvous with a node in W. Therefore, any instance $n_{j'}, j' > j$ which starts between the end of m_i and the start of $m_{i'}$ must do so following a rendezvous between a node instance in X_j and a node instance in $W_i \cup W_{i+1} ... \cup W_{i'-1}$. But, in order for a new instance $m_{i'}$ to start after a node in W is executed, control in $Task(m)$ must leave W and pass through Z. In order for control to pass through Z, a node in Z must rendezvous with one in Y. Y includes only descendants of n. Therefore, the last instance of n which started before $m_{i'}$ started must have completed by the time $m_{i'}$ starts.

Therefore, no instance of m can happen together with any instance of n. □

Note that the sets X and Z (and therefore W and Y) satisfying the preconditions of Theorem 10 need not be unique. In particular, there may be X and Z sets that satisfy some of the preconditions of Theorem 10, but not others. For instance, the maximal X (the set of all proper ancestors of n which can rendezvous only with descendants of m) may induce some members in W that prevent Z from cutting $(W, LE(m))$. (See Figure 5.21. Since X is too large, Z no longer cuts $(W, \{LE(m)\})$.)

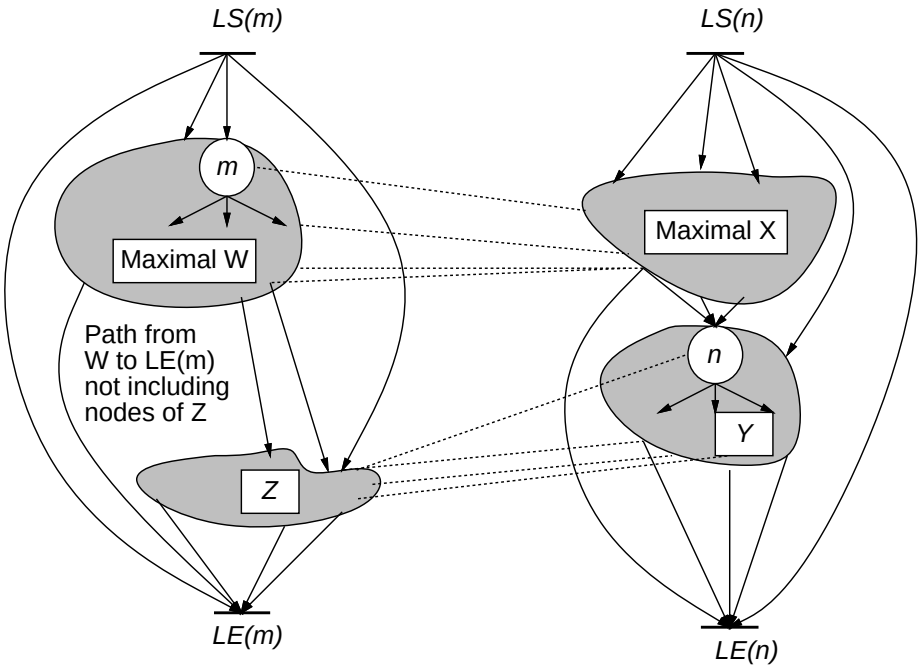

Figure 5.21: Maximal X set that induces a W that fails to satisfy the preconditions of Theorem 10.

An example of CHT with three tasks.

In this section, we present an example in which CHT is enforced by multiple-entry, multiple-exit regions in three tasks. We present this mainly to show the problems involved in generalizing the results of Section 5.1.5 to mutual exclusion in an arbitrary number of tasks.

Figure 5.22 shows an example in which the constraints of Theorem 10 are not obeyed, yet $m \in CHT(m)$. The following are true of the regions shown in Figure 5.22:

- Z postdominates m and W;
- X' dominates W';
- X' must rendezvous with W;
- X dominates n and Y;
- X must rendezvous with W or W';
- Y must rendezvous with Z';

- Y' must rendezvous with Z;
- Z' postdominates W' and dominates Y'.

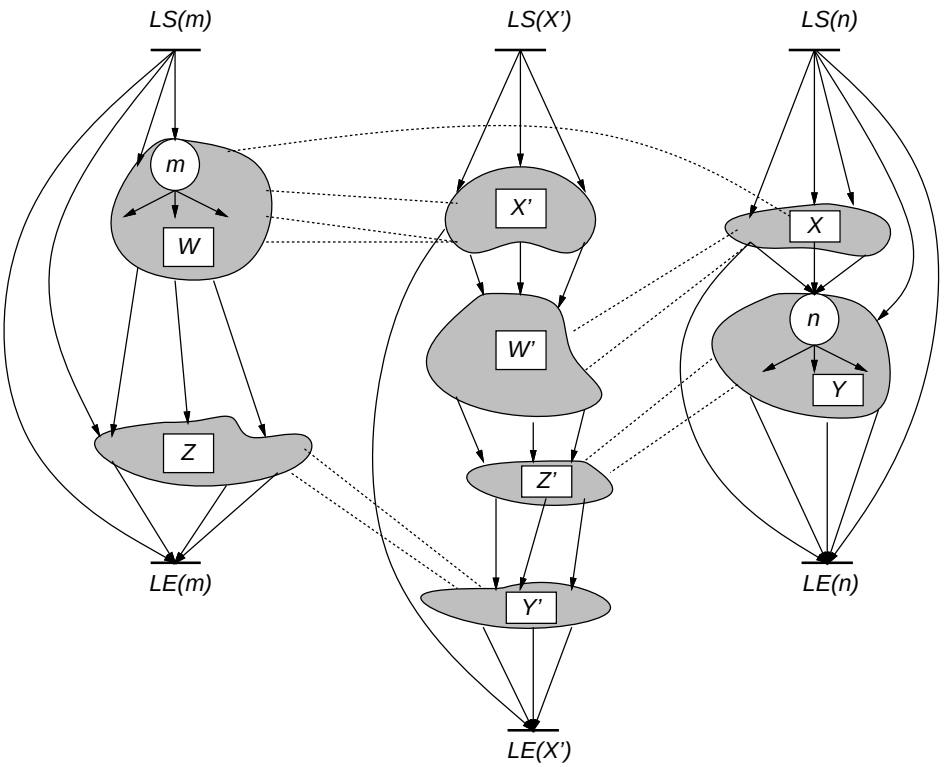

Figure 5.22: A program in which $m \in CHT(m)$ is enforced by three tasks.

Since X dominates n, some instance of a node in X must complete before any instance n_j can start. Nodes in X can rendezvous only with nodes in W and W'. So some node in W or W' must complete before any n_j can start.

Since X' dominates W', some instance in X' must complete before any instance in W' can start. Since X' can only rendezvous with W, and the nodes in W are descendants of m, some descendant of m must complete before any node in X' can complete, and thus before any node in W' can start. Therefore, some instance in W must complete before any instance of n can start. There is thus an ordering chain from a descendant $d_i \in W_i$ of $m_i \in W_i$ to each instance of n_j, even if m_i does not execute.

Now we will show that no instance $m_{i'}, i' > i$ can start while n_j is still executing. After n_j starts and while it is still executing, control in $Task(m)$ must be in W or Z,

or between the two. Control in $Task(X')$ must be in some ancestor of Z'. In order for control to pass to m again, there must be a rendezvous between Z and Y'. But Y' cannot be reached until control passes through Z', and this will only happen if there is a rendezvous between Y and Z'. When this last rendezvous occurs, n_j will be taken off the wave. Therefore, instance $m_{i'}$ must start after n_j finishes.

Next, we will show that no instance of m can start while any instance of n is executing. Suppose that $m_{i'}$ is the next instance of m to start after m_i finishes. Suppose also that, in the period from the end of m_i to the start of $m_{i'}$, instance $n_{j'}, j' > j$ is the last instance of n to start. For $n_{j'}$ to start, control must have entered W or W', so that a rendezvous with some member of $X_{j'}$ could occur. Following this, for $m_{i'}$ to start, control must have passed through Z, implying that control had reached Y' at the same time. For control to reach Y', a rendezvous between Z' and Y must have occurred, implying that $n_{j'}$ finished. So $n_{j'}$ cannot still be executing when $m_{i'}$ starts.

Last, we will show that the next instance of n cannot start while any instance of m is executing. Suppose the next instance of n is $n_{j'}, j' > j$, and $m_{i'}, i' > i$ is the last instance of m to start before $n_{j'}$ starts. In order for $n_{j'}$ to start, a rendezvous must have occurred between X and either W or W'. If this rendezvous was between X and W, then $m_{i'}$ cannot be executing when $n_{j'}$ starts. If the rendezvous was between X and W', then no instance of m could have been executing when the rendezvous occurred, and no further instance of m could start until Y had been entered. Thus, $m_{i'}$ could not have been executing when $n_{j'}$ started.

5.1.6 Remote procedure call analysis.

If some set of nodes S is contained within an `accept-do` construct of signal type t, then the nodes of S must happen together with a signaling node of type t. This leads to the following:

Lemma 63 *If some set of nodes S is contained within an* `accept-do` *construct of signal type t, and Q is the set of signaling nodes that can call S, and $CHT_{perf} \supseteq CHT$, then, for each $s \in S$, $CHT_{perf}(s) \supseteq \bigcap_{q \in Q} CHT(q)$.*

Proof: The set $\bigcap_{q \in Q} CHT(q)$ is a set of nodes that can't happen together with any node in Q. Since only the nodes of q can call the `accept-do` construct in which S is contained, any node that can't happen together with all members of Q can't happen together with any member of S. □

The following algorithm may be used to augment CHT information using the embedded `accept-do` construct. Algorithm 8 does not call *Widen*; instead, it performs a less expensive set of vector operations to achieve the same widening effect on the CHT sets of nodes embedded in remote procedure calls.

Algorithm 8 *Remote procedure call analysis.*

Inputs:

- The CHT sets for each node.
- For each `accept-do` construct A containing embedded synchronization:
  - The set of nodes $N(A)$ contained within the construct A.
  - The accept-in node $AIN(A)$ of the construct.
  - The signaling nodes $S(A)$ which call A.

Output: The CHT set for each node, augmented by remote procedure call information. Denoted CHT'.

Procedure:

1. For each `accept-do` construct A do:
   (a) $ADDED \Leftarrow N$.
   (b) For each $s \in S(A)$ such that $s \notin CHT(AIN(a))$ do:
      i. $ADDED \Leftarrow ADDED \cap CHT(s)$.
   (c) For each $n \in N(A)$ do:
      i. $CHT'(n) \Leftarrow CHT(n) \cup ADDED$.

Lemma 64 (Correctness of Algorithm 8.) *If $CHT \subseteq CHT_{perf}$ and the other parameters of Algorithm 8 are correctly specified, then $CHT \subseteq CHT' \subseteq CHT_{perf}$.*

Proof: From Theorem 1 and Lemma 63. □

Lemma 65 (Time bounds for the remote procedure call algorithm.) *The time to execute Algorithm 8 is $\mathcal{O}(|N|^3)$ bit operations in the worst case.*

Proof: A program may have $\mathcal{O}(|N|)$ `accept-do` constructs, each called by $\mathcal{O}(|N|)$ signaling nodes. Thus, the first inner loop (statement 1b) may have to execute $\mathcal{O}(|N|^2)$ times. Similarly, in pathological cases, an `accept-do` construct may contain $\mathcal{O}(|N|)$ nodes, so the second inner loop (statement 1c) may have to execute $\mathcal{O}(|N|^2)$ times. Assuming sets of nodes are represented as bit vectors of length $|N|$, the total time for Algorithm 8 is $\mathcal{O}(|N|^3)$ bit operations in the worst case. □

In practice, we have found that `accept-do` statements with embedded rendezvous statements are quite rare, so the pathological worst-case time of Lemma 65 should not occur frequently in practice.

5.1.7 Iterative *CHT* analysis.

We may use any combination of *B4* analysis, pinning analysis, critical section analysis, and remote procedure call analysis, to derive a *CHT* relation for a program. Furthermore, we may use any of these techniques to iteratively refine *CHT* information, in the following ways.

First, pseudotransitivity allows *CHT* information to propagate from node to node in pinning, critical section, or remote procedure call analysis. Thus, *CHT* information found in one analysis phase may be used to derive further *CHT* information in another.

Second, if any signaling and accept-in nodes that can't happen together are connected by a sync hyperedge, the rendezvous corresponding to that hyperedge can never occur. Therefore, the hyperedge in question may be eliminated safely from the sync hypergraph. With the hyperedge eliminated, it is possible that further analysis may be able to uncover more *CHT* information.

We base the following algorithm on this form of iterative analysis. Note that all four component *CHT* algorithms intrinsically eliminate sync edges between nodes that can't happen together.

Algorithm 9 *Find the CHT sets of each node in a program.*

Input: The sync hypergraph of the program.

Output: A *CHT* set for each node in the program.

Procedure: The functions *Pinning*, *RemoteProc*, *CriticalSect*, and *B4Analysis* correspond to the obvious refinements; parameters which are constant within a single problem are omitted here for clarity.

1. For each node n, $CHT(n) \Leftarrow Task(n) - \{n\}$.
2. $CHT_{old} \Leftarrow \emptyset$.
3. While $CHT_{old} \neq CHT$ do:
   (a) $CHT_{old} \Leftarrow CHT$.
   (b) $CHT \Leftarrow Pinning(CHT)$.
   (c) $CHT \Leftarrow RemoteProc(CHT)$.
   (d) $CHT \Leftarrow CriticalSect(CHT)$.
   (e) $CHT \Leftarrow B4Analysis(CHT)$.

Lemma 66 (Correctness of *CHT* analysis.) *Algorithm 9 terminates with* $CHT_{perf} \supseteq CHT$.

Proof: By induction on the number of refinements applied. Let CHT_i be the *CHT* set following the i'th refinement.

Basis: $CHT_0(n)$ is $Task(n) - \{n\}$ for each node n. Obviously, $CHT_{perf}(n) \supseteq CHT_0(n)$ for all n.

Induction step: Suppose $CHT_{perf} \supseteq CHT_{i-1}$, and we apply the i'th refinement step to CHT_{i-1} to get CHT_i.

If the i'th refinement is *B4* analysis, then Lemma 28 shows that $CHT_{perf} \supseteq CHT_i \supseteq CHT_{i-1}$.

If the i'th refinement is pinning analysis, then $CHT_{perf} \supseteq CHT_i \supseteq CHT_{i-1}$ by Lemma 36.

Analysis	Time	Lemma
Pinning	$\mathcal{O}(\|Tasks\|\|N\|^3)$	37
Remote procedure	$\mathcal{O}(\|N\|^3)$	65
Critical section setup	$\mathcal{O}(\|N\|^3)$	55
Critical section update	$\mathcal{O}(\|N\|^3)$ per bit	56
B4 (worklist)	$\mathcal{O}(\|N\|^3 \log \|N\|)$	29
B4 (Tarjan)	$\mathcal{O}(\|N\|^3)$	29

Table 5.4: Worst-case time for each form of *CHT* analysis.

If the i'th refinement is critical section analysis, then $CHT_{perf} \supseteq CHT_i \supseteq CHT_{i-1}$ by Lemma 53.

If the i'th refinement is remote procedure call analysis, then $CHT_{perf} \supseteq CHT_i \supseteq CHT_{i-1}$ by Lemma 64.

Thus, $CHT_{perf} \supseteq CHT_i$. Also, since $CHT_{i-1} \subseteq CHT_i$ for $i > 0$, $CHT_{i-1} = CHT_i$ for $i > 4|CHT_{perf}|$ (where we take $|CHT_{perf}| = \sum_{n \in N} |CHT_{perf}(n)|$). (This is because we are doing four *CHT* refinements in a loop; each time we do the block of four refinements, either one pair is added to *CHT* or the loop terminates.) Thus, at most $4|CHT_{perf}|$ refinements can be done before the algorithm terminates. □

Lemma 67 (*CHT* analysis insensitive to ordering of refinements.) *CHT analysis will produce the same final result at termination, regardless of the way the refinements are ordered.*

Proof: From Theorem 2. □

5.1.8 Total time for *CHT* analysis.

Table 5.4 shows the worst-case time for each technique used in *CHT* analysis.

Lemma 68 (Time bounds for the *CHT* algorithm.) *The total time for Algorithm 9 is* $\mathcal{O}(|Tasks||N|^5 \log(|N|))$ *when B4 is solved using the worklist algorithm, and* $\mathcal{O}(|Tasks||N|^5)$ *when B4 is solved using Tarjan's algorithm.*

Proof: From Table 5.4, each iteration (or bit change) of *CHT* will require

$$\mathcal{O}(|Tasks||N|^3 \log(|N|))$$

time using the worklist algorithm. At most $|N|^2$ iterations must be done, if one bit of CHT information is added per iteration. The total time for Algorithm 9 is thus

$$\mathcal{O}(|Tasks||N|^5 \log(|N|))$$

when $B4$ is solved using the worklist algorithm.

Similarly, each iteration (or bit change) of CHT will require $\mathcal{O}(|Tasks||N|^3)$ time using Tarjan's algorithm, for a total of $\mathcal{O}(|Tasks||N|^5)$ time. □

5.1.9 Additional uses of the CHT relation.

In addition to static detection of deadlocks, the CHT relation has several other notable uses.

The first, and simplest, additional use of CHT is in the detection of unexecutable statements. If $CHT(n) = N$, then we know that n cannot execute. This may indicate either an infinite wait anomaly involving the control ancestors of n, or the situation that n is unreachable in control flow.

The second additional use of CHT is in data flow analysis involving communications between tasks [LC91]. For example, consider reaching definitions analysis. If two tasks do not share memory, then definitions can propagate directly between them only through the parameters of a rendezvous. The rendezvous cannot occur if the corresponding signaling and accept-in nodes can't happen together. If we know this, we can eliminate the corresponding sync hyperedge, thus improving the accuracy of the solution.

Finally, we may be able to use CHT information for optimization of rendezvous code. Given that an entry call can't happen together with an entry of the same signal type, we can eliminate from the code for the entry any specialized code or data structures (semaphores, case statements, etc.) corresponding to the call. This may permit both speed and space optimization, depending on the implementation of the rendezvous.

Chapter 6

Detecting deadlock in Ada programs.

6.1 Signaling count lattice framework.

In this section, we define a lattice framework for cycle detection in the Ada subset. The framework maintains, for all nodes in the HCLG, lower bounds on the number of signaling head nodes of each signal type for all paths from a single hypothesized head node.

This section serves as a simplified introduction to Section 6.2. There, we will expand this framework into one which retains further cycle pruning information, and which we have used in experimentation.

The base lattice framework for cycle detection is

$$D_{SigCnt} = (G, L_{SigCnt}, F_{SigCnt}, M_{SigCnt}),$$

where G is the HCLG for the program. Each framework detects possible deadlock paths involving a single head node; to solve the *DeadPath* problem, we form the cross product of the D_{SigCnt} frameworks for all head nodes.

A *signaling node count summary SigCnt* is either:

- A vector of nonnegative integers of length $|Sigs|$, such that the y'th element of the vector is not more than some constant k_y, or;
- The distinguished top element $SigCnt_\top$.

For a node n, *SigCnt* represents a lower bound on the number of signaling head nodes of each signal type encountered on valid paths from a particular head node h to n. k_y above is the total number of signaling nodes of a given signal type y in the sync graph. $SigCnt_i \sqsubseteq SigCnt_j$ if either:

- $SigCnt_j = SigCnt_\top$, or;
- $SigCnt_i \neq SigCnt_\top$ and $SigCnt_j \neq SigCnt_\top$ and $SigCnt_i[y] \leq SigCnt_j[y]$ for all $y \in Sigs$.

The meet operation is thus an element-wise minimum. The lattice subspace L_{SigCnt} is:

$$L_{SigCnt} \subset \mathcal{N}^{|Sigs|} \cup \{SigCnt_\top\}$$

$$SigCnt_\top \in L_{SigCnt}$$

where $\mathcal{N}$ represents the set of nonnegative integers. For convenience, let $SigCnt_\bot$ denote a vector of all zero elements of length $|Sigs|$.

Lemma 69 *The height of the L_{SigCnt} lattice is at most $|N| + 2$.*

Proof: The $\sqsubseteq$ relation is defined for L_{SigCnt} as element-wise $\leq$. The maximum value other than $SigCnt_\top$ that any element of L_{SigCnt} may have is a vector $(k_1, k_2, ..., k_{|Sigs|})$, where $\sum_{y \in Sigs} k_y \leq |N|$ (since each k_y is the number of signaling nodes of type y in G.) Thus, $SigCnt_\bot$ can be increased a maximum of $|N|$ times before reaching the maximum possible value (other than $SigCnt_\top$). □

6.1.1 Edge functions.

The function class $F_{SigCnt} = \{f : L_{SigCnt} \rightarrow L_{SigCnt}\}$ is the transitive closure of the the set of possible edge functions under meet and function composition. A particular edge e will be represented by one of the following function types:

- The *identity function* $f_\iota(x) = x$.
- The *top function* $f_\top(x) = SigCnt_\top$.
- The *bottom function* $f_\bot(x) = SigCnt_\bot$.
- A *conditional identity* function $f_e(x)$ such that, if $x[y] \leq V_e[y]$ for some constant vector of integers V_e and for $y \in Y_e$ where Y_e is a constant set of signal types, then $f_e(x) = x$. Otherwise $f_e(x) = SigCnt_\top$.
- A *conditional increment* function $f_e(x)$ such that, if $x[y] \leq V_e[y]$ for some constant vector of integers V_e and for $y \in Y_e$ where Y_e is a constant set of signal types, then:

- $f_e(x)[y] = x[y], y \in Sigs, y \neq y_e$, where y_e is a constant signal type;
- $f_e(x)[y_e] = \min(x[y_e] + 1, k_{y_e})$.

Otherwise $f_e(x) = SigCnt_\top$.

Note that the identity and top functions are special cases of conditional identity functions.

Lemma 70 *The set of edge functions is monotonic. Furthermore, with the exception of $f_\perp$, the set of edge functions is inflationary.*

Proof: Clearly, f_ι and $f_\top$ are monotonic and inflationary, and $f_\perp$ is monotonic.

Since a conditional identity function $f_e(x)$ has a value of either $SigCnt_\top$ or x, it is inflationary. To show that f_e is monotonic, assume some $a, b \in L_{SigCnt}$ such that $a \neq SigCnt_\top, b \neq SigCnt_\top$, and $a \sqsubseteq b$. If neither $f_e(a)$ nor $f_e(b) = SigCnt_\top$, then $f_e(a) = a$ and $f_e(b) = b$; thus $f_e(a) \sqsubseteq f_e(b)$. If $f_e(b) = SigCnt_\top$, then $f_e(a) \sqsubseteq f_e(b)$.

Now suppose that $f_e(a) = SigCnt_\top$ and $f_e(b) \neq SigCnt_\top$. This would imply that, for some $y \in Sigs$, $a[y] > V_e[y] \geq b[y]$. By the assumption that $a \sqsubseteq b$, though, $a[y] \leq b[y]$, so this case cannot occur. Therefore, the conditional identity functions must be monotonic.

A very similar argument will show that the conditional increment functions are also monotonic and inflationary. □

Let F^-_{SigCnt} be the transitive closure of the set of edge functions excluding $f_\perp$ under meet and function composition.

Lemma 71 *The functions of F^-_{SigCnt} are monotonic, inflationary, and 1-semibounded.*

Proof: By Lemma 9, the set of all monotonic inflationary functions on L_{SigCnt} is closed under composition and meet. Since F^-_{SigCnt} is a closure of a set of monotonic and inflationary functions on L_{SigCnt}, the functions of F_{SigCnt} must also be monotonic and inflationary. Lemma 11 shows that F^-_{SigCnt} is 1-semibounded. □

Lemma 72 *F^-_{SigCnt} contains functions that are not distributive.*

Proof: Consider a conditional identity function f_e such that $Y_e = \{y_1, y_2\}$, and two summaries $a, b \in L_{SigCnt}$ such that:

$$\begin{aligned} a[y_1] &< V_e[y_1] \\ a[y_2] &> V_e[y_2] \\ b[y_1] &> V_e[y_1] \\ b[y_2] &< V_e[y_2]. \end{aligned}$$

Thus, $f_e(a) = f_e(b) = SigCnt_\top$. But $(a \sqcap b)[y_1] < V_e[y_1]$ and $(a \sqcap b)[y_2] < V_e[y_2]$, so $f_e(a \sqcap b) = a \sqcap b \sqsubset SigCnt_\top$.

A similar argument holds for the conditional increment functions. □

Lemma 72 implies that the time complexity results of Lemma 14 cannot be used in the D_{SigCnt} framework.

Lemma 73 *The meet operation in L_{SigCnt} requires $\mathcal{O}(|Sigs|)$ time to compute.*

Proof: From the definition of the meet operation and the L_{SigCnt} lattice. □

Lemma 74 *An edge function application requires $\mathcal{O}(|Sigs|)$ time to compute.*

Proof: The conditional identity and conditional increment functions require a scan of some elements of their argument against corresponding elements of a constant vector. This takes $\mathcal{O}(|Sigs|)$ time. Creating the result also requires $\mathcal{O}(|Sigs|)$ time for initialization. □

Lemma 75 *A worklist iterative algorithm using a lattice framework $D_{SigCnt} = (G, L_{SigCnt}, F^-_{SigCnt}, M_{SigCnt})$ will converge from an initialization of all elements at $SigCnt_\top$ in time $\mathcal{O}(|Sigs||N|^3 \log(|N|))$ in the worst case.*

Proof: From Lemmas 12 and 69, the worklist algorithm will require no more than $(|N|+2)|N|^2 \lceil \log_2(|N|) \rceil$ meet operations, $(|N|+2)|N|^2$ edge function evaluations, and $\mathcal{O}(|N|^2 \log(|N|))$ time to maintain the worklist in the worst case. From Lemmas 74 and 73, this will result in a worst case time of $\mathcal{O}(|Sigs||N|^3 \log(|N|))$. □

6.1.2 Signal counts and 1-semiboundedness.

The only edge functions above which are not inflationary are the $f_\perp$ functions, and these are mapped only to control edges leaving the HCLG "out" node h' corresponding to the hypothesized head node h. With a simple transformation peculiar to each h, it is possible to eliminate the need for the $f_\perp$ functions, and thereby to have a 1-semibounded framework. However, we can also show that the presence of the $f_\perp$ functions does not affect the speed of convergence.

To eliminate the need for $f_\perp$ edges, we simply split h' into two nodes, h_s and h_d. (See Figure 6.1.) h_s acts as the source of a $SigCnt_\perp$ signal. All control edges formerly leaving h' would leave h_s after the split, and are mapped to identity edge functions. (Note that no HCLG edges corresponding to sync hyperedges leave h', so none leave h_s either.) h_s has no edges entering it.

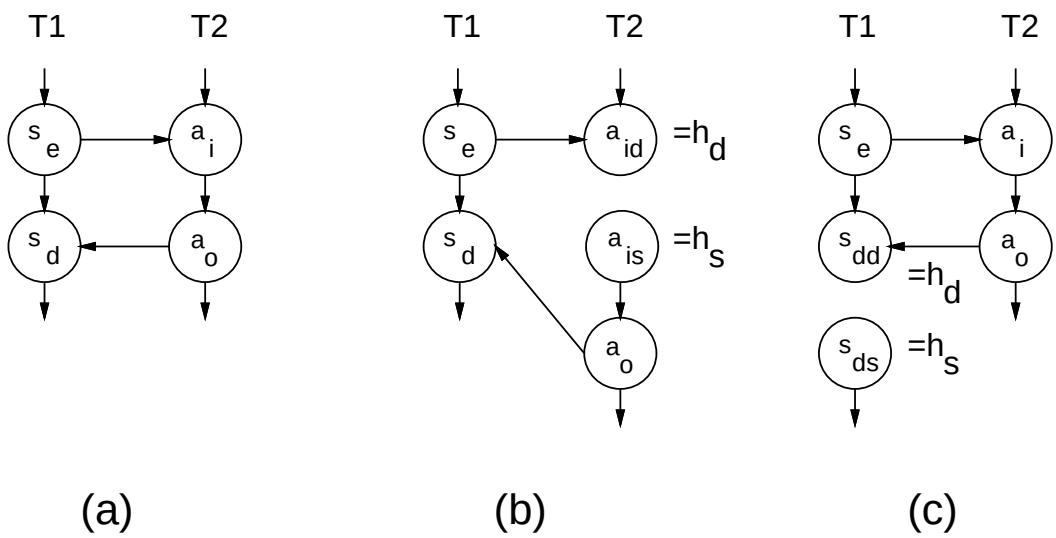

Figure 6.1: Splitting a hypothesized head node. (a) Original HCLG segment. (b) Node a_i split. (c) Node s_d split.

Node h_d acts as the destination node of propagation; a possible deadlock cycle involving h happens if h_d has a value other than $SigCnt_\top$ after convergence. All edges which formerly entered h' would enter h_d after the node split, and retain their original function mappings. h_d has no edges leaving it.

Because the edge functions in the HCLG are all inflationary after the node split,

Lemma 11 asserts that the resulting function space is 1-semibounded. We would solve the deadlock path problem for h by simply setting the value of h_s initially to $SigCnt_{\perp}$, and all other node values to $SigCnt_{\top}$, and iterating to convergence. However, we may effectively obtain the same convergence time without transforming the HCLG by simply not evaluating the edge functions for edges out of h' after the first evaluation, since their values will never change. Thus, through a slight trick, the *SigCnt* problem can be seen to converge in the same time as a 1-semibounded problem.

Lemma 76 *The total time for solution of a D_{SigCnt} lattice problem using Tarjan's path expression algorithm is $\mathcal{O}(|N|^2|Sigs|)$ in the worst case.*

Proof: Direct from Lemmas 13, 71, 73 and 74. □

A similar node-splitting operation is used by Zadeck [Zad84] to prove that a class of lattice frameworks $\mathcal{O}(|N| + |E|)$, assuming that the time to perform a meet operation is constant. Zadeck's node functions (and therefore his edge functions) are either constant valued or non*increasing* and distributive; nodes with constant valued outputs are split, as in our model. In addition to the time bound, Zadeck's model yields an exact solution (the meet over all paths equals the maximum fixed point.) Since our edge functions are not distributive, we do not have these convergence time and exactness results.

6.1.3 Mapping function.

This section describes the mapping function, M_{SigCnt}. Figure 6.2 shows the six HCLG edge types that are covered by the mapping function.

To represent an edge $e = (m, n)$ in HCLG G in this framework for a hypothesized head node h, select $f_e \in F_{SigCnt}$ such that:

- If e represents a control edge (m', n') in the sync hypergraph:
  - **Case (1).** If $m' \neq h$, then $f_e = f_\iota$.

    This allows unimpeded propagation along control edges within tasks, unless the edge is used for initialization.
  - **Cases (2) and (3).** If $m' = h$, then $f_e = f_\perp$.

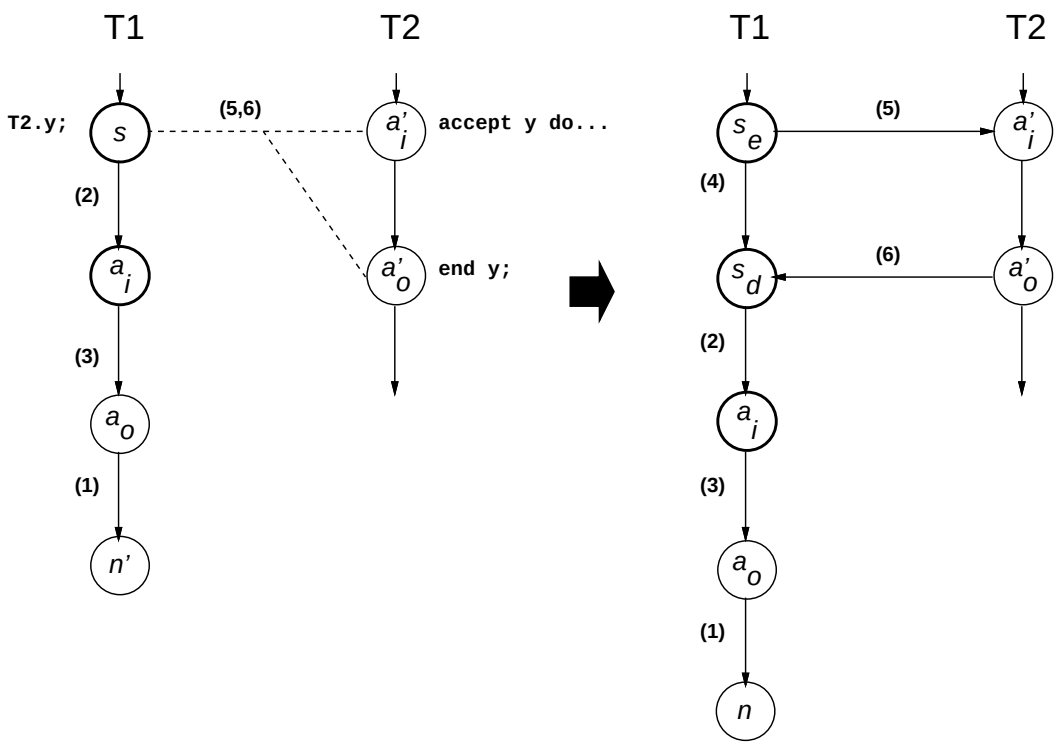

Figure 6.2: Types of edges in the HCLG, and the classes of edge functions to which they correspond.

This starts propagation along a control edge leaving h.

- **Case (4).** If e is an edge (s_e, s_d) in the HCLG for some signaling node s in the sync hypergraph, then $f_e = f_\iota$.

  Propagation is unimpeded along "pseudo-control" edges added between split copies of signaling nodes.

- **Case (5).** If e is an HCLG edge (s_e, a_i) representing a sync edge (i.e., representing mode (b) propagation):

  - If h is a signaling node, and is not the bracket node of a critical section structure, then $f_e = f_\top$.

    Signaling node propagation of non-bracket head nodes along sync hyperedges is only to other signaling nodes. This is done for efficiency reasons; if the deadlock contains an accept node, then the information found by propagating signaling nodes around the deadlock cycle is redundant. Section 5.1.4 explains why this condition must be relaxed for bracket nodes of critical section structures.

  - If a_i cannot be a head node, or is in the task of h and $a_i \neq h$, then $f_e = f_\top$.

  - If a_i is not co-executable with h or is in $CHT(h)$, then then $f_e = f_\top$.

  - If $s \in Task(h)$ and a_i and h are bracket nodes of the same critical section structure, then then $f_e = f_\top$.

    This step prevents h from propagating from its own task to bracket nodes of the same critical section structure, eliminating cycles which contain only the bracket nodes of the same critical section structure as head nodes.

  - If none of the above is true, then f_e is a conditional identity function such that:

    * Y_e is the set containing the signal types of a_i and h, and all of their unguarded sisters;
    * For each $y \in Y_e$, $V_e[y]$ is the minimum scope depth of all unguarded sisters of a_i or h of type y, with respect to signal type y. a_i and h are

also included in this minimum if either is of signal type y.

Here, propagation is allowed only if we cannot guarantee that a_i or h can rendezvous with some signaling head node on the deadlock path.

- **Case (6).** If e is an HCLG edge (a_o, s_d) representing mode (a) or (c) propagation along a sync edge, and s_d represents node s in the sync hypergraph:
  - If s cannot be a head node, or is in the task of h and $s \neq h$, then $f_e = f_\top$.
  - If s is not co-executable with h or is in $CHT(h)$, then then $f_e = f_\top$.
  - If $a_o \in Task(h)$ and s and h are bracket nodes of the same critical section structure, then then $f_e = f_\top$.
  - If none of the above is true, then f_e is a conditional increment function such that:
    * Y_e is the set containing the signal types of h and all of its unguarded sisters;
    * y_e is the signal type of node s;
    * For each $y \in Y_e, y \neq y_e$, $V_e[y]$ is the minimum scope depth of all unguarded sisters of h of type y, with respect to signal type y. The scope depth of h with respect to signal type y is also included in this minimum if h is of type y.
    * $V_e[y_e]$ is the minimum scope depth of all unguarded sisters of h of type y_e, with respect to signal type y_e, *minus one*. The scope depth of h is also included in this minimum if h is of signal type y_e. If h is not of type y_e and has no unguarded sisters of type y_e, then $V_e[y_e]$ may be set arbitrarily.

    Here, we allow propagation only if we cannot guarantee that h can rendezvous with s or some other signaling node along the deadlock path. If propagation is allowed, we increase the count of signaling head nodes of s's type by one, since s is added to the path as a head node.

6.1.4 Time for the total deadlock path algorithm.

To solve the *DeadPath* problem for all head nodes, one D_{SigCnt} problem must be solved for each potential head node. Thus, the worst-case total time for solution of the entire deadlock path problem is $\mathcal{O}(|Sigs||N_{Head}||N|^3 \log(|N|))$ using the worklist algorithm, and $\mathcal{O}(|Sigs||N_{Head}||N|^3)$ using Tarjan's path expression algorithm, where N_{Head} is the set of possible deadlock head nodes.

It is not likely that we could economize by re-using the same path expression for different assumed head nodes h in Tarjan's algorithm. We would wish to make h the origin node of the program, and use the node splitting approach of Section 6.1.2. This would require constructing a separate path expression for each h.

6.2 Signaling node count/accept scope depth lattice framework.

We hypothesized that it might be beneficial to maintain, for each hypothesized head node h, an upper bound with respect to each signal type y on the scope depth of all accept-in head nodes of type y on the paths from h to other program points. This could help us to use constraint 2 to eliminate some paths which could include a rendezvous between head nodes other than h.

We extend the D_{SigCnt} lattice framework of Section 6.1 to maintain upper bounds on scope depth for accept-in head nodes (and their unguarded sisters) for deadlock paths, as well as lower bounds on the number of signaling nodes of each signal type. If the lower bound on the number of signaling nodes of type y is at least as great as the upper bound on scope depth for signals of type y, then the deadlock path must include a pair of head nodes that can rendezvous. Such a path violates constraint 2, and is thus not part of a deadlock cycle.

The base lattice framework for cycle detection is

$$D_{SACnt} = (G, L_{SACnt}, F_{SACnt}, M_{SACnt}).$$

As for D_{SigCnt}, the base framework detects deadlock paths starting at a single head node. To solve the *DeadPath* problem, we form the cross product of the D_{SigCnt} frameworks for all head nodes.

A *signaling node/accept scope depth count summary SACnt* is either:

- A pair of vectors (S, A), where S is a signaling node vector as defined in Section 6.1, and A is a vector of integers of length $|Sigs|$, such that the y'th element of the vector lies between 0 and some constant j_y, or;
- The distinguished top element $SACnt_{\top}$.

For a node n, $SACnt$ represents both a lower bound on the number of signaling head nodes of each signal type, and an upper bound on the scope depth of any accept head node of each signal type, encountered on valid paths from a particular head node h to n. j_y above is the greater of:

- The total number of signaling nodes of type y in the sync graph (i.e., k_y), plus one, or;
- The maximum scope depth of any accept node of type y in the sync hypergraph with respect to signal type y, plus one.

No deadlock path can include either j_y signaling nodes or an accept node of type y with a scope depth of j_y with respect to signal type y. We can thus use $A[y] = j_y$ as a summary for paths which contain no accept head nodes of type y, i.e., in the bottom element of L_{SACnt}.

$SACnt_i \sqsubseteq SACnt_j$ if either:

- $SACnt_j = SACnt_\top$, or;
- $SACnt_i = (S_i, A_i) \neq SACnt_\top$ and $SACnt_j = (S_j, A_j) \neq SACnt_\top$ and $S_i[y] \leq S_j[y]$ and $A_i[y] \geq A_j[y]$ for all $y \in Sigs$.

For notational convenience, $SACnt_\bot$ is the pair (S, A), with $S[y] = 0$ and $A[y] = j_y$ for all $y \in Sigs$.

The meet operation is thus an element-wise minimum over the S vectors and an element-wise maximum over the A vectors. The lattice subspace L_{SACnt} is:

$$L_{SACnt} \subset \mathcal{N}^{2|Sigs|} \cup SACnt_\top$$
$$SACnt_\top \in L_{SACnt}$$

where $\mathcal{N}$ represents the set of nonnegative integers.

Lemma 77

$$\sum_{y \in Sigs} j_y \leq |N| + |Sigs|.$$

Proof: For each $y \in Sigs$, j_y is one more than either the number of signaling nodes of type y or the maximum scope depth of any accept node of type y with respect to y,

whichever is greater. If the former is greater, then j_y cannot be greater than one plus the number of signaling nodes of type y in the sync hypergraph.

Suppose the latter is greater and that node a is an accept-in (or accept-out) node with signal type y and scope depth $j_y - 1$. Then a must be nested in $j_y - 1$ `accept y do...` constructs to achieve this scope depth. The value of j_y is maximized when a is nested inside *all* other `accept y do...` constructs in the program. For each such construct, there is one accept-in node (and one accept-out node) of type y in the sync hypergraph. Then j_y cannot be greater than the number of accept-in nodes of type y in the sync hypergraph.

Therefore, j_y must be at most one more than the number of nodes of type y in the sync hypergraph, and

$$\sum_{y \in Sigs} j_y \leq |N| + |Sigs|. \square$$

Lemma 78 *The height of the L_{SACnt} lattice is at most $2|N| + |Sigs| + 2$.*

Proof: From Lemma 69, the elements of S can be increased a maximum of $|N|$ times before reaching the maximum value other than $SACnt_\top$. From Lemma 77, the elements of A can be decreased at most $|N| + |Sigs|$ times before reaching their minimum value other than $SACnt_\top$. Therefore, $SACnt_\bot$ can be increased at most $2|N| + |Sigs| + 2$ times before reaching $SACnt_\top$. $\square$

6.2.1 Edge functions.

The function class $F_{SACnt} = \{f : L_{SACnt} \rightarrow L_{SACnt}\}$ is, as before, the transitive closure of the set of possible *edge functions* under meet and function composition. A particular edge e will be represented by one of the following function types:

- The *identity function* $f_\iota(x) = x$.
- The *top function* $f_\top(x) = SACnt_\top$.
- A *constant function* $f_e(x) = k_e$ where $k_e \in L_{SACnt}$ is a constant. Note that $f_\top$ and $f_\bot$ (the function which always returns $SACnt_\bot$) are constant functions.
- An *accept head node function* $f_e((S_x, A_x))$ such that, if:

- $S_x[y] < A_x[y]$ for $y \in Y$;
- $S_x[y] \leq V_e[y]$ for some constant vector of integers V_e and for $y \in Y_e$ where Y_e is a constant set of signal types; and,
- $A_x[y] = j_y$ for $y \in Z_e$ where Z_e is a constant set of signal types;

then $f_e(x) = (S_x, A')$ where:

- $A'[y] = A_x[y]$ for $y \in Sigs, y \notin Z_e$, and;
- $A'[y] = A_e[y]$ for $y \in Z_e$, where A_e is a constant vector such that $0 \leq A_e[y] < j_y$ for $y \in Sigs$.

Otherwise $f_e(x) = SACnt_\top$.

- A *signaling head node function* $f_e((S_x, A_x))$ such that, if:
  - $S_x[y] < A_x[y]$ for $y \in Y$; and,
  - $\min(k_{y_e}, S_x[y_e] + 1) < A_x[y_e]$ where y_e is a constant signal type and k_{y_e} is a constant integer;

then $f_e((S_x, A_x)) = (S', A_x)$ where:

- $S'[y] = S_x[y]$ where $y \in Sigs, y \neq y_e$;
- $S'[y_e] = \min(S_x[y_e] + 1, k_{y_e})$.

Otherwise $f_e(x) = SACnt_\top$.

Lemma 79 *The set of edge functions is monotonic. Furthermore, with the exception of the constant functions, the set of edge functions is inflationary.*

Proof: The identity and top functions are clearly both monotonic and inflationary. The constant functions are, likewise, monotonic.

In the case of the accept head node functions $f_e((S_x, A_x)) = (S_x, A')$, the only elements of A' which are different from those of A_x are those for signal types y where $A_x[y] = j_y$, the maximum permissible value. Thus, $A'[y] \leq A_x[y]$ for $y \in Sigs$, and the accept head node functions are inflationary (since the lattice ordering $\sqsubseteq$ on A is element-wise $\geq$). A similar argument holds for the signaling head node functions.

To show monotonicity for the accept head node functions, suppose that $(S_x, A_x) \sqsubseteq (S_z, A_z)$. Thus, for each signal type y,

$$S_x[y] \leq S_z[y]$$

and

$$A_x[y] \geq A_z[y].$$

Suppose that

$$f_e((S_x, A_x)) = (S_x, A'_x)$$

and

$$f_e((S_z, A_z)) = (S_z, A'_z).$$

Thus, neither $f_e((S_x, A_x))$ nor $f_e((S_z, A_z))$ equals $SACnt_\top$. This can only happen if

$$\forall y \in Z_e : A_x[y] = A_z[y] = j_y.$$

Therefore,

$$\forall y \in Z_e : A'_x[y] = A'_z[y] = j_e[y],$$

and

$$\forall y \in Sigs \overset{\text{such}}{\underset{\text{that}}{}} y \notin Z_e : A'_x[y] = A_x[y] \geq A_z[y] = A'_z[y].$$

In this case,

$$f_e((S_x, A_x)) \sqsubseteq f_e((S_z, A_z)).$$

In the case where $f_e((S_z, A_z)) = SACnt_\top$, monotonicity follows automatically. Suppose, though, that we have $(S_x, A_x) \sqsubseteq (S_z, A_z)$, $f_e((S_x, A_x)) = SACnt_\top$, and $f_e((S_z, A_z)) = (S_z, A'_z) \neq SACnt_\top$. Since

$$\forall y \in Z_e : A_x[y] \geq A_z[y] = j_y$$

this must be because there is some $y \notin Z_e$ such that $S_x[y] > V_e[y]$ for some $y \in Y_e$. But

$$S_x[y] \leq S_z[y],$$

so

$$S_z[y] > V_e[y]$$

also, in which case

$$f_e((S_z, A_z)) = SACnt_\top.$$

Thus, the last case cannot exist and f_e is monotonic.

The monotonicity arguments for signaling head node functions are nearly identical to those for accept head node functions, except for the case where $(S_x, A_x) \sqsubseteq (S_z, A_z)$, $f_e((S_x, A_x)) = SACnt_\top$, and $f_e((S_z, A_z)) = (S_z, A'_z)$. Since

$$\forall y \in Z_e : A_x[y] \geq A_z[y],$$

$f_e((S_x, A_x)) = SACnt_\top$ either because

$$\exists y \in Y_e : y \notin Z_e \wedge S_x[y] > A_x[y],$$

or because

$$\min(k_{y_e}, S_x[y_e] + 1) > A_x[y_e].$$

But, for all $y \in Sigs$,

$$S_x[y] \leq S_z[y]$$

and

$$A_x[y] \geq A_z[y],$$

so we must have the same situation in (S_z, A_z), which would imply that $f_e((S_z, A_z)) = SACnt_\top$. So the situation described cannot exist, and the signaling head node functions are monotonic. □

Lemma 80 *F_{SACnt} includes functions which are not distributive.*

Proof: Consider an accept head node function f_e such that $Z_e = \{z_1, z_2\}$, and two summaries $a = (S, A_a), b = (S, A_b) \in L_{SACnt}$ such that, for all $y \in Sigs$, $S[y] = 0$, and:

$$\begin{aligned} A_a[z_1] &< j_{z_1} \\ A_a[z_2] &= j_{z_2} \\ A_b[z_1] &= j_{z_1} \\ A_b[z_2] &< j_{z_2}. \end{aligned}$$

Thus,

$$f_e(a) = f_e(b) = SACnt_\top = f_e(a) \sqcap f_e(b),$$

while

$$f_e(a \sqcap b) \sqsubseteq SACnt_\top.$$

Thus, f_e is not distributive. □

As before, we cannot use the time complexity results of Lemma 14 for the *SACnt* problem because the members of the function space are not distributive.

Lemma 81 *The meet operation in L_{SACnt} requires $\mathcal{O}(|Sigs|)$ time to compute.*

Proof: From the definition of the meet operation and the L_{SACnt} lattice. □

Lemma 82 *An edge function application requires $\mathcal{O}(|Sigs|)$ time to compute.*

Proof: The accept head node and signaling head node edge functions require a scan of some elements of their argument against corresponding elements of a constant vector. This takes $\mathcal{O}(|Sigs|)$ time. Creating the result also requires $\mathcal{O}(|Sigs|)$ time for initialization. □

Let F^-_{SACnt} be the transitive closure of the set of edge functions (excluding the constant functions) under meet and function composition.

Lemma 83 *The functions of F^-_{SACnt} are monotonic, inflationary, and 1-semibounded.*

Proof: Identical to that of Lemma 71. □

As in the case of the *SigCnt* problems, the constant edge functions occur only on control edges leaving a hypothesized head node. We can therefore split each hypothesized head node h into h_d and h_s, initializing h_s to the proper constant value and all other nodes to $SACnt_\top$ before iteration. The *SACnt* problem will thus converge in the same time as a 1-semibounded problem which uses only the F^-_{SACnt} functions on a graph that has at most one more node and $|N|$ more edges than the HCLG.

Lemma 84 *A worklist iterative algorithm using a lattice framework* $D_{SACnt} = (G, L_{SACnt}, F^-_{SACnt}, M)$ *will converge from an initialization of all elements at* $SACnt_\top$ *in time* $\mathcal{O}(|Sigs||N|^3\log(|N|) + |Sigs|^2|N|^2\log(|N|))$ *in the worst case.*

Proof: From Lemmas 12 and 78, the worklist algorithm will require no more than $(2|N| + |Sigs| + 2)|N|^2\lceil\log_2(|N|)\rceil$ meet operations, $(2|N| + |Sigs| + 2)|N|^2$ edge function evaluations, and $\mathcal{O}(|N|^2\log(|N|))$ time to maintain the worklist in the worst case. From Lemmas 73 and 74, this will result in a worst case time of $\mathcal{O}(|Sigs||N|^3\log(|N|) + |Sigs|^2|N|^2\log(|N|))$. □

Lemma 85 *The total time for solution of* $D_{SACnt} = (G, L_{SACnt}, F^-_{SACnt}, M)$ *using Tarjan's path expression algorithm is* $\mathcal{O}(|N|^2|Sigs|)$ *in the worst case.*

Proof: Direct from Lemmas 13, 81, 82 and 83. □

6.2.2 Mapping function.

Refer to Figure 6.2. To represent an edge $e = (m, n)$ in HCLG G in this framework for a hypothesized head node h, select $f_e \in F_{SACnt}$ such that:

- If e represents a control edge (m', n') in the sync hypergraph:
  - **Case (1).** If $m' \neq h$, then $f_e = f_\iota$.

    This allows unrestricted propagation along control edges whose tail is not an assumed head node.
  - **Case (2).** If $m' = h$, and h is a signaling node of type y_h, then $f_e = (S_e, A_e)$ where:

* $S_e[y_h] = 1$ and $S_e[y] = 0$ for $y \in Sigs, y \neq y_h$.
* $A_e[y] = j_y$ for $y \in Sigs$.

This starts propagation of $SACnt$ records for h, with the number of signaling nodes of type y_h initialized to 1, all other signaling nodes to 0, and the maximum scope depth of encountered accept head nodes effectively infinite. (Actually, since h propagates only to other signaling head nodes in this case, we need not be too concerned about the initial value, as long as the initial value does not *a priori* indicate that a rendezvous may happen.)

– **Case (3).** If $m' = h$, and h is an accept-in node of type y_e, then let S_h be $\{h\}$ unioned with the set of unguarded sisters of h. Let Y_e be the set of signal types of the elements of S_h, and let d_y be the minimum scope depth of any element of S_h which has type y (with respect to y). The edge function f_e is a constant function returning (S_e, A_e), where:

* $S_e[y] = 0$ for $y \in Sigs$.
* $A_e[y] = d_y$ for $y \in Y_e$, $A_e[y] = j_y$ for $y \in Sigs, y \notin Y_e$.

This starts propagation of $SACnt$ records for h, with the number of signaling nodes of all types initialized to 0, the maximum scope depth of h and its unguarded sisters initialized to the minimum necessary to guarantee that some sister can rendezvous, and the maximum scope depth of all other accept head nodes effectively infinite.

- **Case (4).** If e is an edge (s_e, s_d) in the HCLG for some signaling node s in the sync hypergraph, then $f_e = f_\iota$.

  This allows unrestricted propagation from engage (s_e) to disengage (s_d) signaling nodes in the HCLG.

- **Case (5).** If e is an HCLG edge (s_e, a_i) representing mode (b) propagation, then the edge function f_e is a function such that:

  – If a_i cannot be a head node, or is in the task of h and $a_i \neq h$, then $f_e = f_\top$.

  – If a_i is not co-executable with h or is in $CHT(h)$, then then $f_e = f_\top$.

- If h is a signaling node, and is not the bracket node of a critical section structure, then $f_e = f_\top$.

  As in the *SigCnt* lattice, hypothesized signaling nodes propagate only to other signaling head nodes along sync hyperedges.

- If $s \in Task(h)$ and a_i and h are bracket nodes of the same critical section structure, then then $f_e = f_\top$.

- If none of the above is true, then f_e is an accept head node function such that:

  * Y_e is the set containing the signal types of a_i and all of its unguarded sisters;

  * For each $y \in Y_e$, $V_e[y]$ is the minimum scope depth of all unguarded sisters of a_i of type y, with respect to signal type y. a_i is also included in this minimum if it is of signal type y.

  This allows mode (b) propagation only along paths where a_i (or one of its sisters) is not guaranteed to rendezvous with a signaling head node already in the path, and disallows propagation along some paths which enter a_i's task more than once (i.e., those which include more than one accept-in head node or unguarded sister with the same signal type).

- **Case (6).** If e is an HCLG edge (a_o, s_d) representing mode (a) or (c) propagation, and s_d represents node s in the sync hypergraph:

  - If s cannot be a head node, or is in the task of h and $s \neq h$, then $f_e = f_\top$.

  - If s is not co-executable with h or is in $CHT(h)$, then $f_e = f_\top$.

  - If $a_o \in Task(h)$ and s and h are bracket nodes of the same critical section structure, then $f_e = f_\top$.

  - If none of the above is true, then f_e is a signaling head node function such that y_e is the signal type of node s.

    This assignment allows summaries to propagate to a signaling head node only when we cannot guarantee that some head nodes in the cycle will be

able to rendezvous.

6.3 Total time for the algorithm.

Tables 6.1 and 6.2 summarize the timing analysis for the deadlock detection algorithm using the worklist and Tarjan's algorithms respectively, with both *SigCnt* and *SACnt* lattice frameworks. Note that:

- $|N| = \mathcal{O}(|N|)$;
- $|N_{Head}| = \mathcal{O}(|N|)$;
- $|E_C| = \mathcal{O}(|N|^2)$;
- $|E_S| = \mathcal{O}(|N|^2)$.

Operation	*SigCnt* algorithm	*SACnt* algorithm
Parse	$\mathcal{O}(\|N\| + \|E_C\| + \|E_S\|)$	$\mathcal{O}(\|N\| + \|E_C\| + \|E_S\|)$
CHT sets	$\mathcal{O}(\|Tasks\|\|N\|^5)$	$\mathcal{O}(\|Tasks\|\|N\|^5)$
Loop unroll	$\mathcal{O}(\|N\| + \|E_C\| + \|E_S\|)$	$\mathcal{O}(\|N\| + \|E_C\| + \|E_S\|)$
Reachability	$\mathcal{O}(\|Sigs\|\|N_{Head}\|\|N\|^3 \log(\|N\|))$	$\mathcal{O}(\|Sigs\|\|N_{Head}\|\|N\|^3 \log(\|N\|)$ $+\|Sigs\|^2\|N_{Head}\|\|N\|^2 \log(\|N\|))$
Total	$\mathcal{O}(\|Tasks\|\|N\|^5$ $+\|Sigs\|\|N_{Head}\|\|N\|^3 \log(\|N\|))$	$\mathcal{O}(\|Tasks\|\|N\|^5$ $+\|Sigs\|\|N_{Head}\|\|N\|^3 \log(\|N\|)$ $+\|Sigs\|^2\|N_{Head}\|\|N\|^2 \log(\|N\|))$

Table 6.1: Summary of worst-case timing analysis for static deadlock detection using the worklist algorithm.

Operation	*SigCnt* algorithm	*SACnt* algorithm
Parse	$\mathcal{O}(\|N\| + \|E_C\| + \|E_S\|)$	$\mathcal{O}(\|N\| + \|E_C\| + \|E_S\|)$
CHT sets	$\mathcal{O}(\|Tasks\|\|N\|^5)$	$\mathcal{O}(\|Tasks\|\|N\|^5)$
Loop unroll	$\mathcal{O}(\|N\| + \|E_C\| + \|E_S\|)$	$\mathcal{O}(\|N\| + \|E_C\| + \|E_S\|)$
Reachability	$\mathcal{O}(\|Sigs\|\|N_{Head}\|\|N\|^2)$	$\mathcal{O}(\|Sigs\|\|N_{Head}\|\|N\|^2)$
Total	$\mathcal{O}(\|Tasks\|\|N\|^5)$	$\mathcal{O}(\|Tasks\|\|N\|^5)$

Table 6.2: Summary of worst-case timing analysis for static deadlock detection using Tarjan's path expression algorithm.

Chapter 7

Experimental results.

7.1 Introduction.

To experimentally verify and characterize polynomial time deadlock analysis, a deadlock analyzer for the Ada language was written, and a series of experiments were performed.

7.1.1 Objectives of the experiment.

The experiment had three main objectives.

Objective 1: Deadlock analysis characterization.

First, we wished to characterize the time and accuracy of polynomial deadlock analysis, both in absolute terms and with respect to all-states. Since all-states is a comparatively straightforward algorithm to implement, we expected that it might have better time performance for programs with smaller sync graphs, but explode combinatorially on larger programs.

In particular, we hoped that we could predict the "break-even" point in execution time between polynomial and all-states analysis, given only easily derivable data for a program (e.g., the number of nodes in its sync graph). We could thus tell whether to try polynomial or all-states analysis first.

Objective 2: *CHT* analysis characterization.

Second, we wished to characterize the accuracy and time requirements of our *CHT* algorithm, of each of its refinements, and of iterative refinement. We particularly wanted to investigate whether any one refinement often found *CHT* information that another refinement could use, i.e., whether there were *synergistic effects* between the refinements.

Objective 3: Pruning information effect characterization.

Third, we wished to characterize the effect that *CHT* and other pruning information had upon the accuracy of deadlock detection, i.e., the number of false alarms which each source of pruning information prevented.

Data gathering objectives.

The specific data we wished to gather included:

- Absolute time requirements of polynomial deadlock analysis.
- Characteristics of programs where polynomial deadlock analysis reported false alarms, relative to all-states.
- Characteristics of programs where polynomial deadlock analysis took less time than all-states, and vice versa.
- Amount of *CHT* information obtained, relative to the size of the program, by both the exhaustive and polynomial algorithms.
- Accuracy of *CHT* information, relative to *CHT* from all-states, especially as a function of the program's characteristics.
- Amount of *CHT* information obtained from each of the following sources:
  - Pinning analysis.
  - Critical section analysis.
  - Remote procedure call analysis.
  - *B4* analysis.
  - Iterative refinement.
- Effect of deleting *CHT* information from each of the above sources on the accuracy of deadlock detection.

A note on terminology.

In our previous work, polynomial analysis was sometimes referred to as "reachability" analysis, and all-states analysis was referred to as "wavegen" analysis. These terms have been revised to avoid confusion between the polynomial analysis and "reachable states" analysis; the latter is a synonym for all-states, and is widely used in the literature on concurrency analysis. While these older terms have been eliminated from the text, the names of variables and programs may occasionally reflect this obsolete terminology.

7.2 Architecture of the deadlock analyzer software.

The deadlock analyzer, named `ggen`[1], is written in the Sun dialect of the C language. `ggen` contains about 19,000 source code lines. To facilitate a fair timing comparison, no special effort was made to optimize either the polynomial or all-states analysis code.

7.2.1 Overview of the analyzer.

The `ggen` program performs analysis in one of two modes, determined by command line input. The default mode is polynomial deadlock analysis; all-states analysis is selected using the `-w` command line option.

In either mode, the Ada source program under test is parsed and a sync hypergraph is constructed. Polynomial analysis, if selected, is done according to the algorithms of Chapters 5 and 6.

All-states analysis and deadlock detection.

In this section, we describe the way in which reachable concurrency states, or execution waves, are generated by all-states analysis, and how these states are tested for deadlock.

In all-states, we use an optimization that avoids generating an excessive number of states when guarded `select` alternatives may be on the wave. Let a_i be an accept-in node of a selective wait. We assume that, in selective wait statements, any subset of the guards may be open. If there is a deadlocked execution wave E including a_i and some set S of a_i's sister nodes, then, there is another deadlocked execution wave E' that includes only $Sisters_{Unguarded}(a_i) \cup \{a_i\} \subseteq S$, and such that a_i is a head node in E'.

Rather than generating states for all possible combinations of open guard conditions, we need only generate one state containing all unguarded sisters, and one state for each guarded sister, containing it and all unguarded sisters.

As the reachable states are generated, all states with infinite wait anomalies are

[1] `ggen` stands for "graph generator," a name dating to the initial phases of the project in which the goal was to generate the sync graphs of Ada programs.

saved. When state generation is complete, the anomalous states are tested to see whether they contain a deadlock. Deadlock detection is done by searching for deadlock cycles, starting at each node on the wave in each anomalous state. Cycles which have only the bracket nodes of a critical section structure as head nodes are rejected, since they are only symptoms of other anomalies.

We process all anomalous states, thus enumerating all deadlocks. Alternately, we might terminate and report deadlock when any deadlocked state was found, but doing so might result in early termination due to a "lucky guess" at a particular sequence of state expansions. The result would then yield inaccurate execution time estimates. Therefore, for accuracy, we need to perform exhaustive analysis.

7.2.2 Additional controls for analysis.

In addition to the control for polynomial or all-states analysis, `ggen` accepts a number of control options in the command line. Each of the four *CHT* refinements can be independently disabled, as can the signaling node count (S) and maximum accept node scope depth (A) path summary fields of the *SACnt* records. Finally, the execution time of the program can be limited by command line option.

7.2.3 Outputs and diagnostic information.

The `ggen` program can produce a variety of diagnostic outputs. Six levels of output information detail are available. Level 0 merely reports whether the program under test can deadlock. Intermediate levels report statistical information about the program and timing of the program phases. Level 5, the highest level, reports many of the intermediate data structures generated during deadlock detection.

Sync graphs and source listings of the program can also be output as alphanumerics, under command line control separate from the detail level control. Finally, `ggen` can produce a representation of the sync graph which can be interpreted by the `DAG` program [GNV88]. Thus, the user can view sync graphs in their most natural graphical form.

7.2.4 Source files.

The ggen program is built from the following source files:

- `ggen.c`. The control program and command line interpreter.
- `ggen.h`. Header file for `ggen.c`, defining global variables and symbols.
- `ada.y`. A Yacc based parser for the Ada language, with semantic attachments to generate sync hypergraphs.
- `ada-yacc.h`. Union type declarations used by the parser.
- `ada.l`. A Lex based lexical analyzer used in conjunction with `ada.y`.
- `sync-graph.c`. Code to manipulate sync hypergraphs and their components.
- `sync-graph.h`. Header file defining the structure of the sync graph component types.
- `deadhead.c`. Detects head nodes that may be involved in deadlock cycles.
- `deadhead.h`. Type definitions used in `deadhead.c`
- `wavegen.c`. Performs all-states analysis.
- `wavegen.h`. Type definitions used in `wavegen.c`.
- `cht.c`. Control code for *CHT* analysis, including iterative refinement.
- `cht.h`. Header file for `cht.c`.
- `pinning2.c`. Pinning analysis.
- `critical-sect.c`. Critical section analysis.
- `critical-sect.h`. Header file for `critical-sect.c`.
- `rpc.c`. Remote procedure call analysis.
- `b4.c`. *B4* analysis.
- `dominators.c`. Tarjan's dominator-tree algorithm. A generic directed graph structure is used; it is not specific to the sync hypergraph.
- `dominators.h`. Header file for `dominators.c`
- `misc.c`. Miscellaneous routines, including timer manipulation.
- `misc.h`. Type definitions used in `misc.c`.
- `memory.c`. Private routines used in memory allocation. Essentially, a shell around `malloc` which maintains memory usage statistics and handles allocation errors.

- `memory.h`. Macro and extern definitions used in dynamic memory manipulation.
- `sw-signals.h`. Header file defining software signals, used primarily for handling situations in which the CPU time limit and memory allocation limit are exceeded.
- `generic-table.c`. Implements generic tables, maps, and dequeues.
- `generic-table.h`. Defines structures of generic tables, maps, and dequeues.
- `bit-vector.c`. Bit array print, transpose, and multiply routines.
- `bit-vector.h`. Operations on primitive bit vectors and arrays, implemented as macros.

The generic tables use the AVL tree package by Paul Vixie, posted to `comp.sources.unix`, as their base data structure. The source files `tree.c` and `tree.h` contain code that defines and manipulates AVL trees.

7.2.5 Limitations of the analyzer implementation.

It should be noted that ggen is a research tool, based on a bare syntactic parser. As such, it lacks many of the features needed in a production quality deadlock analyzer. In particular, the semantic analysis performed by ggen is rather primitive. For instance, ggen currently cannot analyze programs contained in multiple source files, cannot recognize and instantiate generics and packages, and cannot recognize entry calls of any form other than `<name>.<entry>`. The user must therefore sometimes edit their code to a form that ggen can successfully analyze. We detail this editing in Section 7.3. This editing step could likely be omitted if ggen were based on a production quality Ada compiler.

7.3 Sources of test programs.

Ada tasking codes were obtained from the following sources:

- The STARS software repository distribution. Available from Daphne Hurrell, Science Applications International Corp., 3045 Technology Parkway, Orlando, FL 32826. Phone: (407) 282-6700. Email: `hurrelld@stars.starstech.com`.

- The Ada Software Repository. FTP address `wuarchive.wustl.edu`, in directory `/mirrors/ada` and its subdirectories.
- The Ada Compiler Validation Capability. FTP address `ftp.sei.cmu.edu`, in directory `/pub/ACVC` and its subdirectories.
- The Hartstone benchmark. FTP address `ftp.sei.cmu.edu`, in directory `/pub/hartstone` and its subdirectories.
- A public domain "Star Wars" simulation written for the IBM PC by T. Griest of LabTek Corporation, Woodbridge, CT. Available from the author.
- The DARTES fighter aircraft simulation benchmark, developed at the Software Engineering Institute. FTP address `ftp.sei.cmu.edu`, in directory `/pub/rtsia/dartes`.
- A set of demonstration programs, written by Michael Feldman of George Washington University. Programs include "dining philosophers" and "sort race" demonstrations. Available from the author.
- A "readers-writers" simulation written by LCDR Jeffrey M. Schweiger of the Naval Postgraduate School. Available from the author.
- A small Ada subset compiler validation suite written by Lori Pollock. Available from the author.
- Test programs to validate the analyzer, written by the author. Available from the author.

Some programs were edited to permit `ggen` to analyze them. This editing performed only those program transformations that could have been made automatically by a more sophisticated tool. The editing included:

- Combining separate source files that included tasking.
- Inlining of non-recursive calls to procedures and functions that contained tasking statements.
- Instantiation of generics that defined or accessed tasks.
- Conversion of entry calls through task access variables to statically defined entry calls, where possible.

- Conversion of arrays of tasks with known bounds to a corresponding set of identical tasks, where possible.

7.4 Experiment design.

The experiment was conducted on a Sun Microsystems Sparcstation-2; times measured are user CPU time on that machine. The Sparcstation-2 had a maximum available user memory of 200 megabytes.

In the data presented here, the execution time was limited to 1200 user CPU seconds. A separate experiment confirmed that, for all programs for which this limit was exceeded, the memory requirement for data structures, exceeded the Sparcstation-2's available memory. Thus, no analysis that ran for more than 1200 seconds would have run to completion, even given unlimited time.

7.4.1 Data gathered.

Tables 7.1, 7.2, and 7.3 list the types of data gathered on the performance of the polynomial and all-states deadlock analyzers. The S statistical package [BCW88] was used to analyze and plot the raw data.

The independent variables (Table 7.1) depend upon the structure of the source program under test for deadlock. The dependent variables indicate the performance of the polynomial time (Table 7.2) and all-states (Table 7.3) analyzers for a particular test program. Data marked "(boolean),", are flags indicating whether the event happened. (E.g., "Polynomial deadlock" is 1 if polynomial analysis reported deadlock, 0 otherwise.) The "field name" given to each datum refers to the variable name used within the analysis software written for the S package.

The sync graph characteristics are reported with respect to the parse form sync graph emitted by the parser, because it is a common source of data for the *CHT*, polynomial and all-states analyses.

Except for the "Program Name," each datum has a numeric value. "Report time" and "cleanup time" are, respectively, the time taken to report all (potential) deadlocks,

Datum	Field Name
Program name	`Name`
Tasks	`Tasks`
Signals	`Sigs`
Nodes	`N`
Signal nodes	`SN`
Accept in/out pairs	`AN`
Loop bodies	`Loops`
Control edges	`Ec`
Sync edges	`Es`
Non-head nodes	`Nnh`
Sisters	`Sis`
Unguarded sisters	`UGSis`
Nested accepts	`NA`

Table 7.1: Independent variables in the deadlock experiments.

Datum	Field Name
Initial *CHT* bits	`bInit`
CHT iterations	`iCHT`
Final *CHT* bits	`bCHT`
Total *CHT* time	`Tcht`
Pinning time	`tPin`
Pinning *CHT* bits	`bPin`
Remote procedure call time	`tRPC`
Remote procedure call *CHT* bits	`bRPC`
Number of critical sections	`nCS`
Critical section time	`tCS`
Critical section *CHT* bits	`bCS`
Number of *B4* nodes (outside loops)	`nB4`
B4 *CHT* bits	`bB4`
B4 time	`tB4`
B4 iterations	`iB4`
Polynomial solve time	`Trsolve`
Polynomial report time	`Trrept`
Polynomial cleanup time	`Trclean`
Polynomial total time	`Tr`
Polynomial deadlock (boolean)	`DLr`
Polynomial out of time (boolean)	`OUTr`
Number of non-top *SACnt* records at fixed point	`nSACount`

Table 7.2: Dependent variables for polynomial time analysis in the deadlock experiments.

Datum	Field Name
CHT_{perf} bits	`bCHTperf`
All-states rendezvous states	`States`
All-states solve time	`Twsolve`
All-states report time	`Twrept`
All-states cleanup time	`Twclean`
All-states total time	`Tw`
All-states deadlock (boolean)	`DLw`
All-states out of time (boolean)	`OUTw`

Table 7.3: Dependent variables for all-states in the deadlock experiments.

and the time taken to destroy the sync graph and any derived data structures.

7.5 Statistical methods.

In this section, we explain some of the statistical terms and methods used in the experimental work. This section is provided as a convenience for the reader who is not immediately familiar with the terminology. For a thorough treatment of multivariate statistical analysis, we recommend [Job91].

7.5.1 Least mean squares regression.

Some of our work involves developing methods to estimate the analysis time for programs. We do this by using the experimental data to develop *multivariate least mean square regressions*, from the independent variables to the analysis time for the program.

The S package's `lsfit` linear regression algorithm finds a linear combination of the independent variables that minimizes the mean of the square of the differences between the analysis time and the linear combination. Data points may be weighted; we take advantage of this to emphasize programs with longer analysis times. The output of the regression algorithm is a vector of weights, one for each independent variable; the best fit found is the sum of the weighted variables.

It should be noted that increasing the number of independent variables will generally increase the accuracy of the fit to the regressed data, but will not generally increase the regression's predictive accuracy. In finding a "best" regression for a data set, we must

therefore penalize regressions that use a large number of independent variables.

Regression to polynomials of independent variables.

Since the regression algorithm produces weighted sums of the independent variables, it cannot immediately handle polynomials or any functions of the variables other than linear combinations. Therefore, we must precompute any functions that we wish the regression to try. For instance, if we wish to try $\log(|N|)$ as a term, we compute $\log(|N|)$ for each data point, and regress on that as an independent variable.

Polynomials of the independent variables, or product terms in general, constitute a special case. If we regress to a product term $v_1^{e_1} v_2^{e_2} ... v_k^{e_k}$, we would generally also like to regress to its lower-order factor polynomials. Each factor must then be generated as a precomputed function.

Leaps and bounds.

The `lsfit` function of the S package will produce a single least-squares fit; however, we wish to find the *best* fit, in some measurable sense, for some combination of the independent variables. To find this, we use the `leaps` function, which finds the best regression of any subset of the independent variables by *leaps and bounds* [FW74]. Leaps and bounds is a hill-climbing algorithm which performs regressions on various combinations of independent variables, and finds the best subset of variables according to a specified goodness-of-fit metric. (We use the adjusted r^2 metric, described in Section 7.5.2.) Variables which have a very small effect on the linear combination, or whose inclusion decreases accuracy, are rejected by the regression.

Holdbacks.

The ultimate test of a regression is its ability to predict data that was not included in the regression. Therefore, we regress on only a fraction of the data points, and hold back the rest of the data to test the regression. The regression quality metrics (Section 7.5.2) are taken with respect to the regression's ability to predict the holdback data.

7.5.2 Regression quality metrics.

In this section, we describe the statistical metrics we use to measure the accuracy of regressions. It is important to note that these metrics are usually only comparable in relative terms, e.g., "a high r^2 is better than a lower one." There are no hard and fast rules saying, for example, that "if the r^2 for this regression is less than 0.5, the regression should be rejected."

Standard error.

In our experiments, the *error* of a regression at a data point is the predicted analysis time minus the actual time. The standard deviation of the errors (also known as the *standard error* gives a rough idea of the expected error at any data point. For instance, if the standard error is 20 (seconds), we know that, under the assumption that the error has a uniform Gaussian distribution, the actual analysis time will be within 20 seconds of the predicted time for about 67% of the data points.

Adjusted r^2.

The "unadjusted" r^2 metric, or *coefficient of determination* [Job91], indicates what fraction of the variation in observed data is explained by the regression. Unadjusted r^2 may falsely indicate that a regression with many independent variables, relative to the number of data points, explains noisy data better than a regression with fewer independent variables. This effect is actually due to the fact that a regression with more variables has more "degrees of freedom" to fit the data, but does not necessarily conform better to the underlying process generating the data.

In contrast, the *adjusted* r^2 metric, or *coefficient of determination adjusted for degrees of freedom* [Job91], also takes into account the sample size and the number of independent variables. Both r^2 and adjusted r^2 vary from $-\infty$ to 1, with higher values indicating a better fit. Values below zero indicate that the predicted data is actually "noisier" than the observations. For any prediction that is better than guessing the average of the observations, both of the metrics are greater than zero, and are thus

often expressed as percentages, rather than fractions.

The F ratio.

The *F ratio*, in the context of regressions, is the ratio of the mean squared variation of the regression to the mean squared error. The F ratio is a ratio of "explained" to "unexplained" variation of the sample, whereas r^2 is a ratio of "explained" to total variation. The F ratio varies from zero to infinity; higher values of F indicate a better fit for the regression.

7.5.3 Confidence intervals.

The standard error is the expected error at any arbitrary data point. However, if we choose a particular data point p, the regression may fit better, or worse, at p than it does over the entire sample. We would often like to know, for p in particular, the expected range of analysis times.

The *confidence interval* at a data point is the interval around the estimate within which the observed value is expected to fall. For example, if we predict an analysis time of 100 seconds with a 90% confidence interval of 10 seconds, then we believe that 90% of programs at that data point will have analysis times between 90 and 110 seconds. If 90% of the holdback data falls within the 90% confidence intervals, then the regression predicts the actual analysis time well.

7.5.4 Some pitfalls of statistical analysis of execution time.

As mentioned in Section 7.1.1, we wish to predict the execution time of both polynomial and all-states deadlock analysis from easily measured structural properties of the program's sync hypergraph, such as the number of nodes, edges, tasks, etc. If we could predict execution time, we would have a good, low-cost heuristic about which of the two methods to use for a given program. Sections 7.7.1 and 7.10.2 represent attempts to make such estimates. However, statistical analysis has some pitfalls in this application, which were evidenced by the experimental data.

Total programs analyzed	138	100.0%
All-states exploded	11	8.0%
All-states deadlocks	19	15.0%
Polynomial deadlocks (total)	28	20.3%
Polynomial deadlocks (where all-states completed)	24	18.9%
Polynomial false alarms (where all-states completed)	5	3.6%

Table 7.4: Overall experimental results.

In statistics, it is generally assumed that a small change in the independent variables will result in a correspondingly small change in the dependent variables. Unfortunately, this assumption is not generally true in our analysis. Programs with identical gross structural characteristics may require vastly different analysis times. For instance, one program without critical sections may have the same number of nodes, edges, etc. as a program that has critical sections; critical section analysis will be used only for the second program. The problem becomes even more pronounced in all-states; a program with an exponential number of possible execution waves may have the same gross characteristics as a program with only a few waves.

7.6 Overall experiment summary.

Table 7.4 shows the overall results of the experiment. The polynomial analysis finished for 11 large programs in which all-states analysis showed combinatorial behavior. The false alarm rate is a quite acceptable 3.6%; in fact, the only false alarms we saw were found in a set of synchronization benchmark programs, which had the troublesome n-ary CHT_{perf} relationship described in Section 2.3.1.

Only one program tested (a fighter aircraft simulation called DARTES) could not be analyzed by the polynomial-time analyzer due to memory limitations; all-states also ran out of memory for this program. However, we believe that the polynomial time analyzer would terminate on DARTES, if it were given slightly more memory, or if an easy space optimization were performed on the $SACnt$ data structures.

Figure 7.1 shows the time requirements for both polynomial and all-states analysis, versus the number of nodes in the program's sync graph. The "o" character represents polynomial analysis; "+" and "*" represent, respectively, all-states analyses that succeeded or timed out.

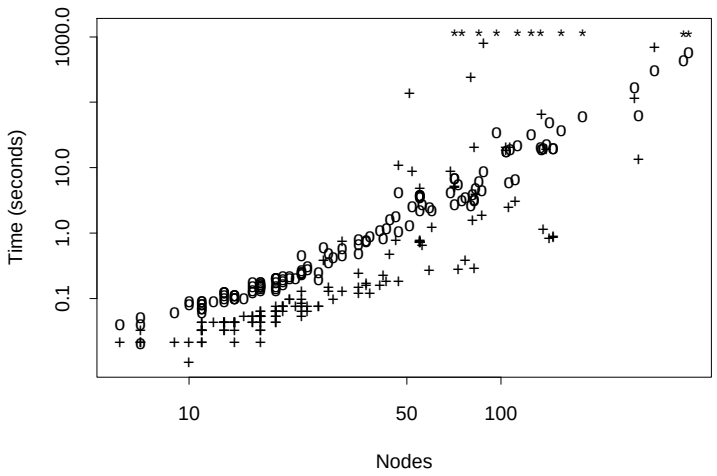

Figure 7.1: Times for polynomial and all-states analysis, versus number of nodes. Scales are log-log for clarity.

We see that, below about 50 nodes, all-states analysis is usually faster, due to set-up time requirements of the polynomial analyzer. Note, though, that the polynomial analysis takes a modest 10 CPU seconds or less for programs of 50 nodes or fewer. Above 50 nodes, all-states starts to exhibit combinatorial explosions (the "*" data points at the top of the graph, at about 1200 seconds), and the polynomial analysis wins more often. We conclude that the absolute time requirements for polynomial analysis are reasonable for all cases where all-states terminates, and are often better than all-states for larger programs.

As a rule of thumb, we would suggest trying all-states before polynomial for programs with fewer than 50 nodes, and trying polynomial only if all-states fails to terminate after some reasonable amount of CPU time (e.g., 1000 seconds). For programs with more than 50 nodes, we would reverse this ordering; however, we would not limit the CPU time for polynomial analysis, since polynomial analysis has always terminated

for any program for which all-states did.

However, no rule of thumb will be accurate all the time, as our subsequent sections show. In particular, Section 7.10 shows that it is difficult to predict the execution time of all-states for particular programs. This implies that predicting whether all-states will win for a given program is, likewise, difficult.

7.7 Polynomial deadlock analysis.

In this section, we present some observations about the experimental time complexity and predictability of the polynomial deadlock detection algorithm. To give some intuition for the factors affecting execution time, we begin by showing Figure 7.2, a scatterplot of the total time to find deadlocks using the polynomial algorithm, versus each of the twelve independent variables of Table 7.1. It shows that the best correlated factors affecting the execution time of the deadlock algorithm are, predictably, the number of nodes, sync edges, and control edges.

7.7.1 Estimated execution time of polynomial deadlock analysis.

The deadlock algorithm was shown to have a theoretical worst-case performance time of $\mathcal{O}(|Tasks||N|^5 + |Sigs||N|^5)$, in Section 6.3. We therefore posited that a polynomial of this degree might be fit to the data. Based on this hypothesis, an attempt was made to estimate the time that polynomial deadlock detection will take for any given program, as a polynomial with highest order terms of $|Tasks||N|^5$ and $|Sigs||N|^5$. Regression was done using the S package's `leaps` function to perform all-subset regression by leaps and bounds. We used the adjusted r^2 metric to measure the goodness of the fit. Since we had relatively few data points with high execution time, the fitted samples were weighted by their execution time. Analyses which failed to complete were not used in the regression.

Regression analysis produced 71 candidate regressions, with the adjusted r^2 value ranging from a maximum of 99.93199% to a minimum of 68.15962%. One third of the data were held back as a test of the predictive ability of the regression; the data were

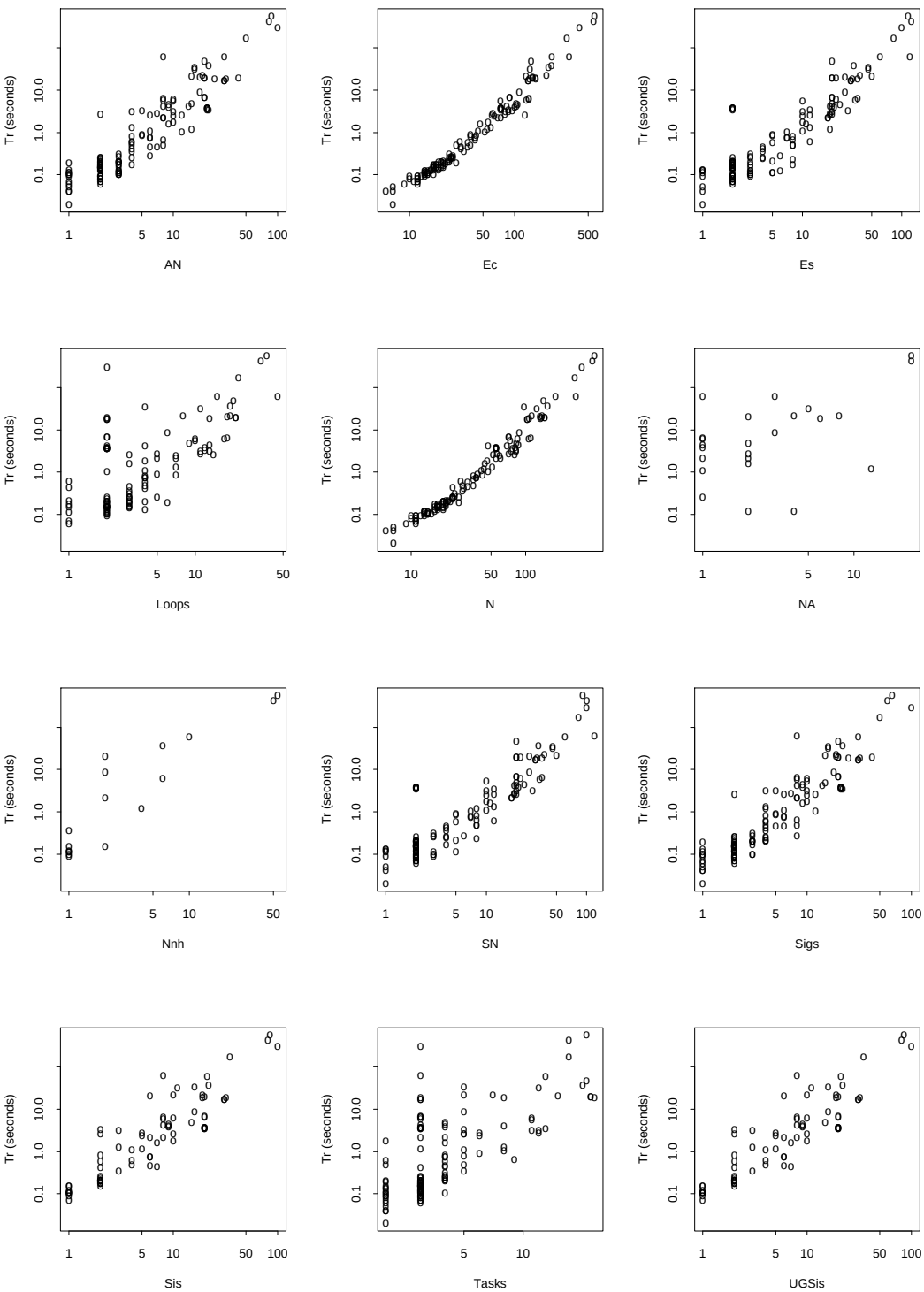

Figure 7.2: Deadlock detection time using the polynomial algorithm, versus each of the 12 independent variables. Scales are log-log for clarity.

Coefficient	Value
$\|Sigs\|$	-.689681
Intercept	-.238632
$\|N\|$	.111225
$\|N\| * \|Sigs\|$	0.0101268
$\|N\| * \|Tasks\|$	-0.00827889
$\|N\|^3 * \|Tasks\|$	2.56994e-07

Table 7.5: Coefficients for the best fit to the deadlock program execution time, in decreasing order of absolute value. Omitted variables were eliminated by the leaps-and-bounds analysis and have a value of zero.

Indicator	Value
Standard error	6.96
Adjusted r^2	99.93199%
F	1370

Table 7.6: Goodness-of-fit estimators for holdback data in the polynomial analysis.

sorted in order of total execution time, and every third item in the sorted list was held back.

The best approximation.

Table 7.5 shows the coefficients for the regression with the best adjusted r^2 value. Note that, although the coefficients for polynomials above order 1 have very small absolute values (where the leaps-and-bounds analysis did not eliminate them entirely), they may still dominate the lower-order coefficients in some cases. For instance, ignoring for the moment the number of signal types, the $|N|^3 * |Tasks|$ term, though small, will dominate in programs with more than 100 nodes.

Table 7.6 shows some of the regression metrics for the holdback data, as predicted by the polynomial of Table 7.5. While the standard error of the estimate is slightly high, the adjusted r^2 and F remain quite acceptable. Considering the probable operational use of the deadlock detection algorithm, a difference of a few seconds between actual and expected program execution time should not be a practical problem.

Figure 7.3 shows the prediction confidence limits for the holdback data for the

regression with the best adjusted r^2. All but four of the 46 predictions fall within the confidence interval; these lie in a range of execution times for which we have few data points. We would expect that 10% of the data points would fall outside the 90% confidence interval; Figure 7.3 thus bears this expectation out. Fewer points lie within the 90% confidence interval as the actual execution time increases; this is to be expected, since we have relatively few data points at the higher execution times, and thus the regression cannot accurately predict the highest execution times.

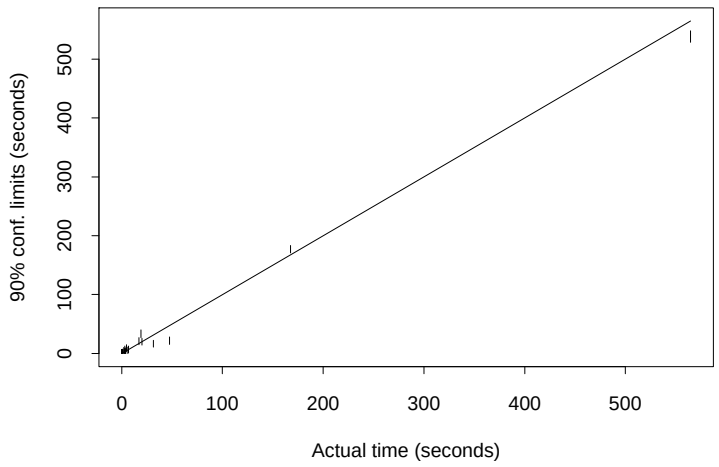

Figure 7.3: Residuals on holdback data for each of the test programs for the best adjusted-r^2 fit found. Short vertical lines are the 90% confidence bounds of each prediction.

Figure 7.4 shows the residuals for the holdback data, for all 71 best regressions found by leaps-and-bounds regression analysis. The height represents the absolute value of the residual of a given regression for a given test program in the holdback set. Each point on the x axis represents a regression; x values are sorted by increasing adjusted r^2. Points on the y axis represent the test programs, sorted by actual analysis time. Note the flatness of the surface; this indicates that any of the regressions fit the observed data well. As with Figure 7.3, the errors were more pronounced for the programs with the highest execution times; this is indicated by the ridges at the back of the plot, in the region of high actual execution time.

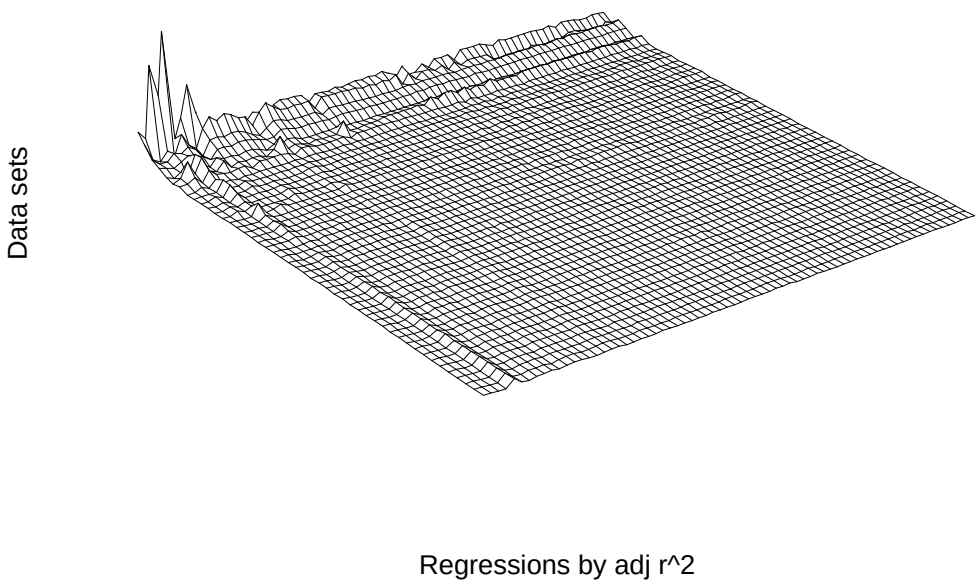

Figure 7.4: Residuals on holdback data for each of the test programs and for each of the 71 best fits found by leaps and bounds.

7.8 *CHT*.

In this section, we present results on the effectiveness and time requirements of *CHT* analysis, for each of the refinements and in total.

7.8.1 Time requirements.

Execution time. Figure 7.5 shows the time taken by *CHT* analysis, as a function of the number of nodes in the sync graph. The dashed line indicates a time bound of $\mathcal{O}(|N|^5)$, which is a factor of the theoretical time bound. We thus see that the actual time to estimate *CHT* is related to the size of the sync graph by a low-order polynomial (i.e., a polynomial of degree ≤ 5, the theoretical worst-case limit). The solid line represents an attempt to estimate the actual limit polynomial by fitting a line to the data points where $|N| \geq 50$. Its slope is 2.98, indicating a limit polynomial of this degree in the number of nodes.

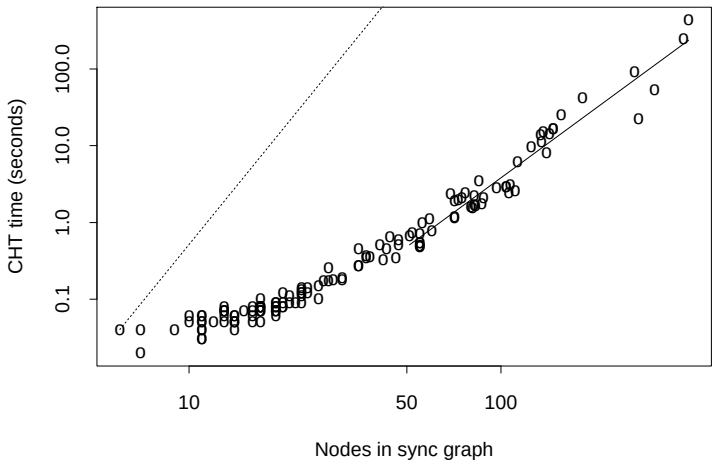

Figure 7.5: Time to estimate *CHT*. The scale is log-log for clarity.

At most five iterations of the *CHT* refinement process were needed to reach convergence. Figure 7.6 is a histogram of the number of iterations needed versus the number of programs.

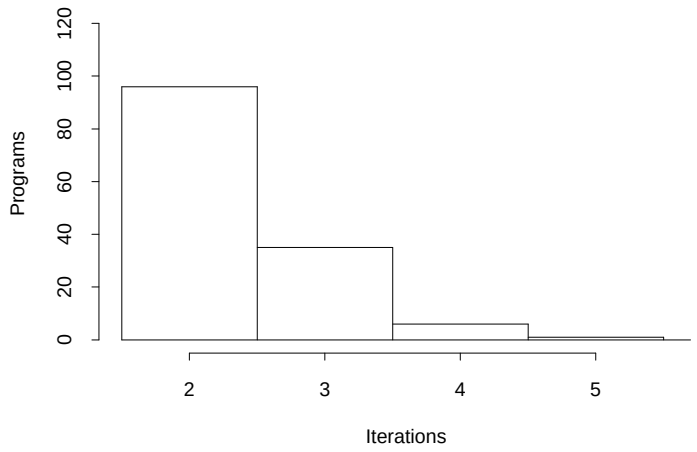

Figure 7.6: Histogram of the number of *CHT* refinement iterations, versus the number of programs needing that many iterations.

7.8.2 Overall accuracy.

Figure 7.7 shows a measure of the accuracy of the *CHT* information gathered by polynomial-time analysis. Here, the number of node pairs (m, n) such that $m \in CHT(n)$ (derived from polynomial analysis) is plotted against the number of node pairs in a "perfect" *CHT* relation (determined from all-states analysis). Data is plotted only for the 115 programs for which both polynomial and all-states analysis completed. The diagonal line represents a perfectly accurate estimated *CHT* relation; data points below or on the diagonal are conservative estimates of *CHT*. Figure 7.7 thus shows that the polynomial estimate of *CHT* is not only conservative (validating the implementation), but also quite accurate, and that accuracy is maintained over the entire range of program sizes.

Figure 7.8, a histogram of the accuracy of the *CHT* estimate, shows that the *CHT* algorithm gives a very accurate estimate for most programs. For example, it shows that the computed *CHT* finds at least 95% of all node pairs in CHT_{perf} in a total of 92 or 80.0%, of the 115 total programs for which CHT_{perf} could be found and was non-empty.

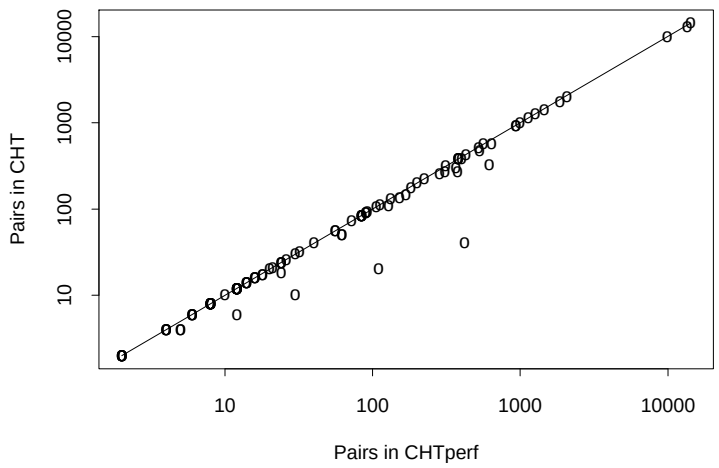

Figure 7.7: Number of pairs in calculated versus "perfect" CHT relations. Scale is log-log for clarity.

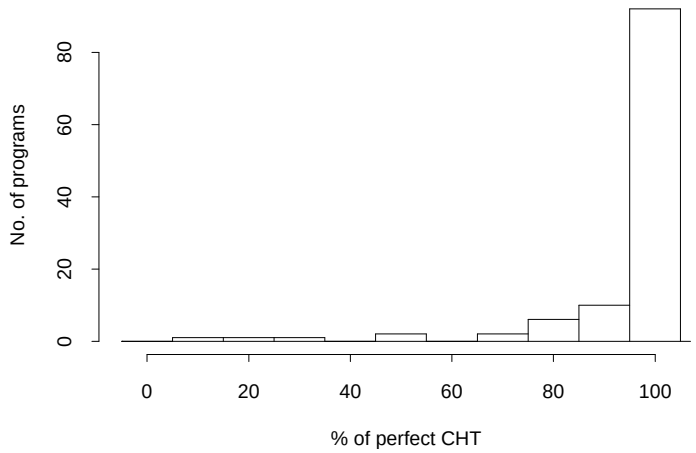

Figure 7.8: Histogram of the ratio CHT/CHT_{perf} to number of programs with that ratio.

Comparison with [DS91].

Table 7.7 and Figure 7.9 show that, while the *B4* analysis of [DS91] (and, by implication, [CS88]) are valuable as refinements, they clearly miss a large component of non-concurrency information in most programs. The other refinements are needed to find the majority of *CHT* information in a large number of programs.

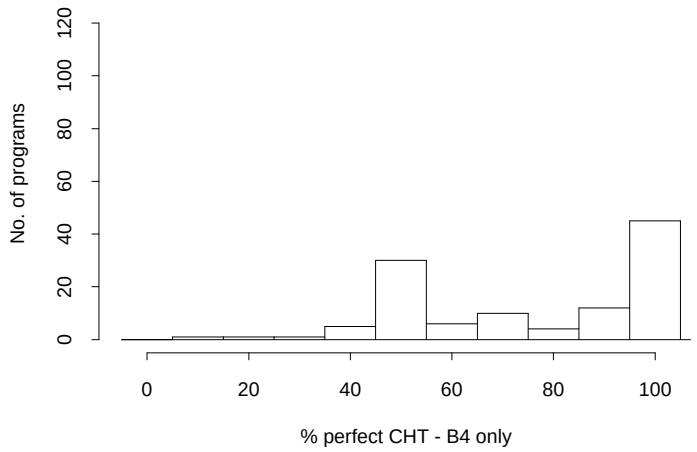

Figure 7.9: Histogram of the ratio CHT/CHT_{perf} where *CHT* was estimated using only the *B4* refinement, to number of programs.

7.8.3 Contributions of the refinements of *CHT*.

Table 7.7 shows, as a percentage, the number of programs for which each refinement to *CHT* found some information, and the number of programs for which some sync edges were pruned. In some very simple programs, any pair of rendezvous nodes in different tasks can happen together with each other, and no task contains more than one node. A total of 12 programs had this property in our experiment; they account for the difference between the "Any *CHT*" and "Total Programs" entries of the table.

Table 7.7 shows that most of the programs tested had a non-empty *CHT* relation, and that pinning analysis can be helpful in estimating *CHT* in most programs.

Figure 7.10 shows a series of histograms, of the relative number of set bits in the

Refinement	Number	Percent
Any *CHT*	126	91.3
Pinning	98	71.0
B4	24	17.4
Remote proc. calls	11	8.0
Critical sections	9	6.5
Pruned sync edges	25	18.1
Total programs	138	100.0

Table 7.7: Number of programs with *CHT* contributions from each refinement.

CHT relation contributed by initialization, pinning, *B4*, remote procedure call, and critical section analysis. In each histogram, the x axis is the fraction of bits contributed by each method, and the y axis is the number of programs for which that fraction of bits was contributed. The programs whose *CHT* sets are empty are not shown. Initialization, pinning analysis, and *B4* analysis contributed the most information to *CHT*.

B4, critical section and remote procedure call analyses contributed a relatively small fraction of the *CHT* relation. However, it should be emphasized that a small subset of *CHT* can sometimes be crucially important in eliminating false alarms, as we showed in critical section analysis. In Section 7.8.6, we explore the contribution of each of the four analysis techniques in eliminating false alarms.

7.8.4 Sync edge pruning.

Figures 7.11 and 7.12 show how effectively *CHT* information prunes extraneous sync edges. These figures show that, although the effectiveness of pruning varied widely, the percent of edges pruned tended to decrease with increasing program size.

7.8.5 Effectiveness of *CHT* in deadlock detection.

To determine the effectiveness of *CHT* in preventing false alarms, we compared the false-alarm rate with, and without, *CHT* pruning. We found that the use of *CHT* information prevented 87.2% of the false alarms that occurred without *CHT*. Thus, non-concurrency information is crucial to the practical effectiveness of our deadlock

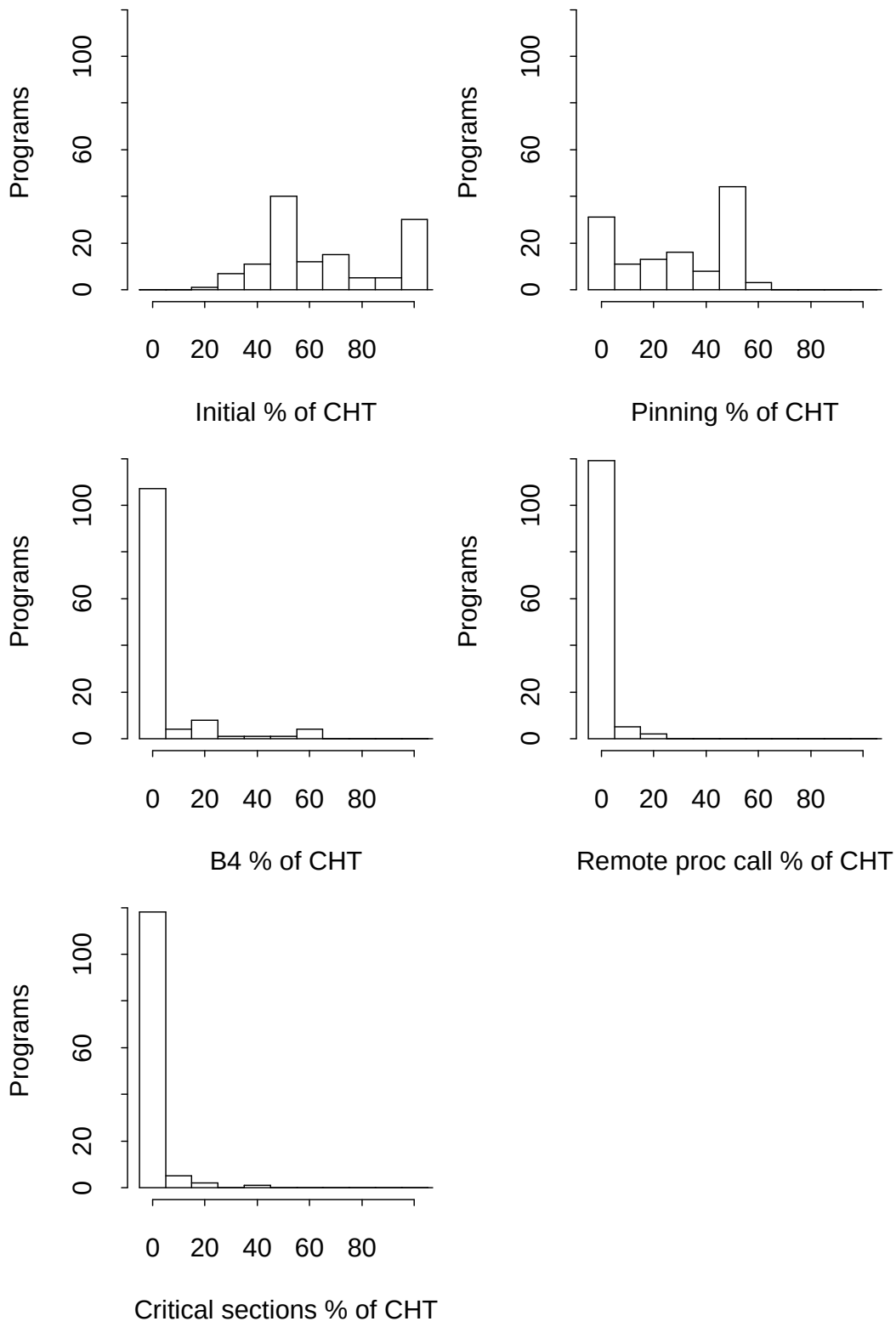

Figure 7.10: Histograms of the relative number of bits contributed by each of initialization, pinning, *B4*, remote procedure call, and critical section analyses.

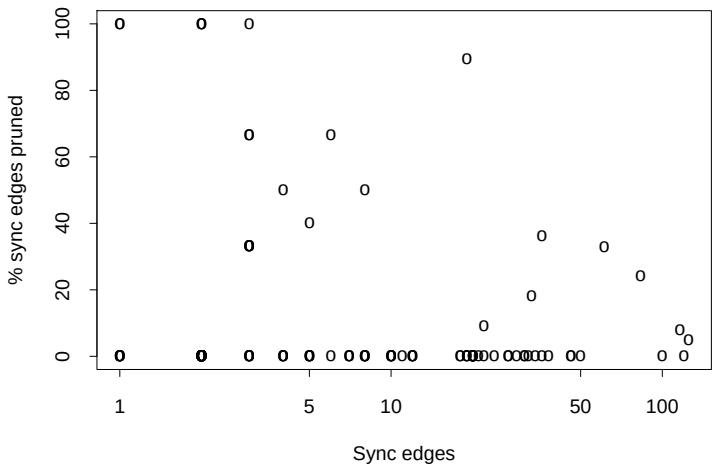

Figure 7.11: Percent of sync edges pruned, versus number of sync edges in the program. The x axis is logarithmic for clarity.

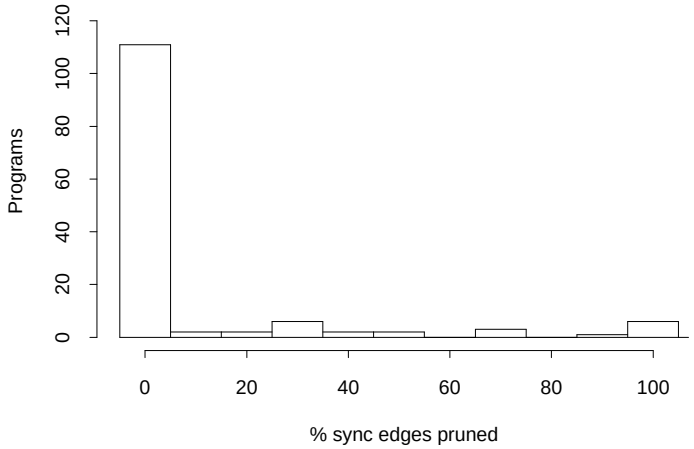

Figure 7.12: Histogram of percent of sync edges pruned, versus number of programs.

detection system.

7.8.6 Synergy effects, deadlock, and *CHT* analysis.

If some refinement to the *CHT* analysis enables other refinements to add information to the *CHT* relation, we say that it has a *synergistic* effect on the *CHT* computation. We wished to discover how often these synergistic effects occurred for each refinement to *CHT*, and how much information they added.

To estimate synergy effects for each refinement R, we conducted the following experiment. We found the set of programs for which R contributed any bits at all to *CHT*. For each of these programs, we measured the size of the resulting *CHT* relation produced with R disabled. If the difference in the size of the total *CHT* relations with and without R was greater than the number of bits contributed by R, then R must have a synergistic effect.

Some refinements had a *neutral* effect in analyzing certain programs; if eliminated, their contribution to *CHT* could be made up by the remaining refinements. However, because the refinements are inflationary, it is not possible for any refinement to have a *negative* effect on the final *CHT*, i.e., to result in smaller *CHT* sets than if it were not used.

Figure 7.13 shows the results of the synergy experiment in graphical form. The x axis is the "direct yield" of the refinement, i.e., the number of *CHT* pairs the refinement found when all refinements were enabled. The y axis is the difference between the total *CHT* bits found with and without the refinement. The curved line is the "break-even" line ($y = x$); points above this line represent programs where synergy was seen for the refinement.

Table 7.8 provides a numerical summary, and also shows whether each *CHT* analysis prevented false alarms in deadlock detection. For each refinement, percentages are relative to the total number of programs for which the refinement contributed to *CHT*. Surprisingly, pinning analysis, the major contributor to *CHT*, has almost no synergy effects at all. In fact, the reverse seems to be true; other phases of the analysis were

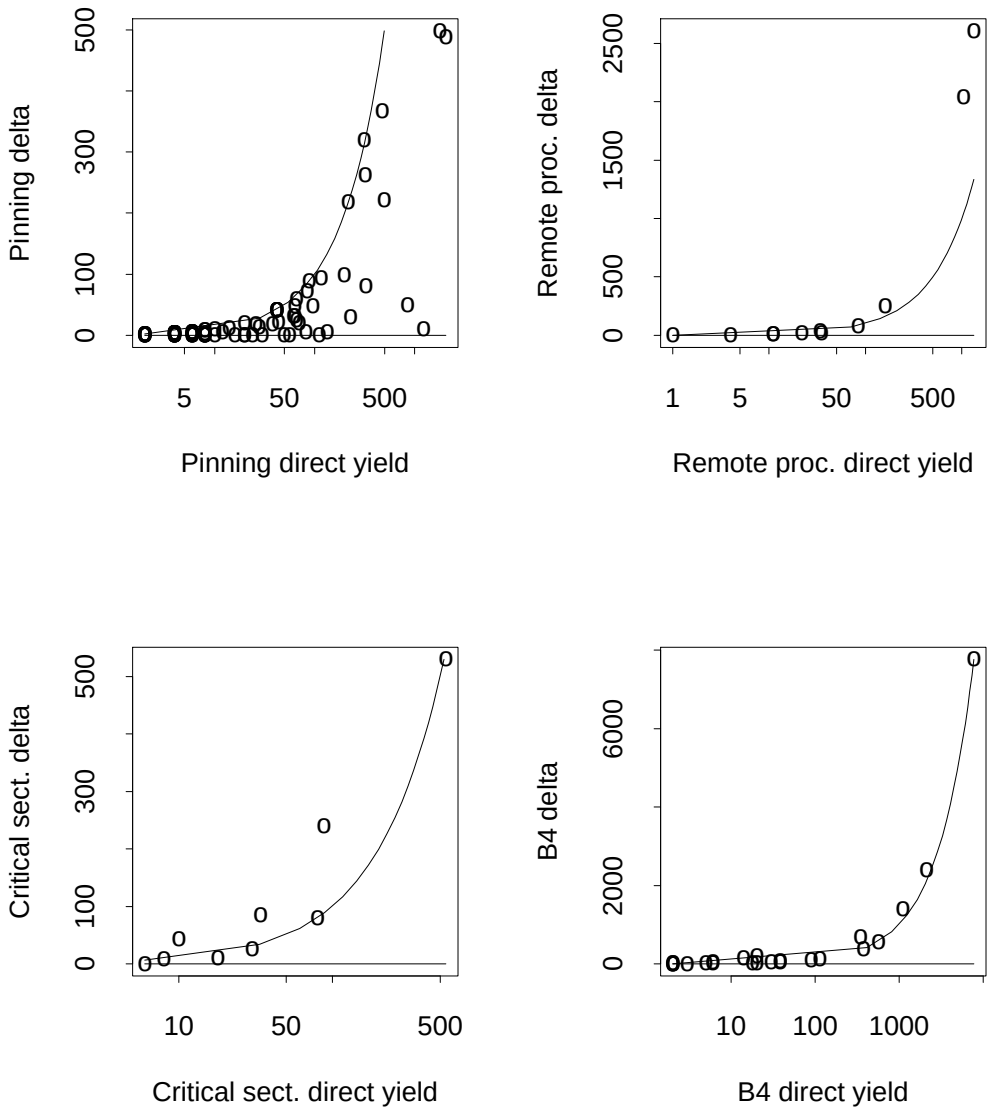

Figure 7.13: Synergy effects for each of the four refinements of *CHT* analysis. The x axis is logarithmic for clarity.

	Pinning	*B4*	Rem. Proc.	Crit. Sect.
Synergy	1.0%	37.5%	27.3%	33.3%
Neutral	28.9%	0.0%	0.0%	11.1%
Prevents false deadlock	26.8%	0.0%	0.0%	33.3%

Table 7.8: Percentages of programs for which each refinement to *CHT* had synergy or neutral effects on the total *CHT* relation, or prevented spurious deadlock reports.

often able to partly make up for the loss of pinning information.

It should be noted that the *B4* and remote procedure call refinements remain useful for two reasons. The first reason is that they may prevent false alarms in some programs not tested; the second reason is that the *CHT* data may be important, in and of itself, for purposes other than deadlock detection (Section 11.2).

Each of the other refinements of *CHT* (remote procedure calls, critical sections, and *B4* analysis) showed a few cases with synergy effects. Pinning and critical sections were the only analyses for which, for some program tested, contributions to *CHT* could entirely be made up by the other analyses (the "neutral" row in Table 7.8). More commonly, the analyses other than pinning contributed to *CHT* did not interact with each other, or with pinning.

Only pinning and critical sections analyses prevented false alarms in deadlock detection, as the "Prevents false deadlock" row of Table 7.8 shows.

7.9 Effectiveness of the *SACnt* summaries.

In this section, we examine the effectiveness of the *SACnt* summary information in eliminating false alarms. Recall that this information is used during the deadlock cycle detection stage of polynomial deadlock detection.

Given a hypothesized head node h and another node n. The *SACnt* information includes a lower bound on the number of signaling head nodes of each signal type which *do* appear on any path $(h, ..., n)$, and an upper bound on the number of signaling head nodes of each signal type which *may* appear on any such path. These bounds are known as the *signal count* and *accept depth*, respectively. To measure the effectiveness

	Signal count	Accept depth	Both
Prevented non-top summaries	8	35	35
(Percent)	5.8%	25.5%	25.5%
Prevented false alarms	0	14	14
(Percent)	0.0%	10.2%	10.2%

Table 7.9: Programs for which signal counts, accept depth, or both prevented non-top *SACnt* summaries from being propagated, and for which they prevented false alarms. Percentages are relative to the 138 total programs.

of this information in preventing false alarms, we disabled the software which collected it along paths, except at the hypothesized head node h. (Since the identity of h had to be maintained in the summaries, it seemed pointless to eliminate the bounds knowable only from h itself.)

Table 7.9 is an overview of the effectiveness of signal count and accept depth information in preventing propagation of non-top *SACnt* summaries during cycle detection, and in preventing false alarms. The table shows that, while both prevented some propagation, only accept depth prevented any false alarms. Thus, in practice, the signaling count parts of the summaries could likely be safely eliminated.

Figures 7.14 and 7.15 quantify the effect of signaling counts and accept depths in suppressing propagation of non-top *SACnt* summaries. In Figure 7.14, the x axis of each graph is the number of non-top *SACnt* summaries at the fixed point, with no suppression information maintained in the summaries. The y axis is the number of summaries suppressed, the line represents 100% suppression. Figure 7.14 shows that signaling node counts suppressed few non-top summaries, while accept depth information suppressed quite a few more. (The curved lines represent full suppression.)

Signaling counts were only slightly effective in suppressing propagation; it only had effect in 8 programs, and at best it was able to prevent only 3.8% of the non-top summaries seen at the fixed point. On the other hand, accept depth, by itself, prevented at most 58.7% of the non-top summaries from appearing at the fixed point.

In Figure 7.15, the x axis is the percent of non-top *SACnt* summaries suppressed by maintaining signaling counts, accept depths, or both forms of information. The y

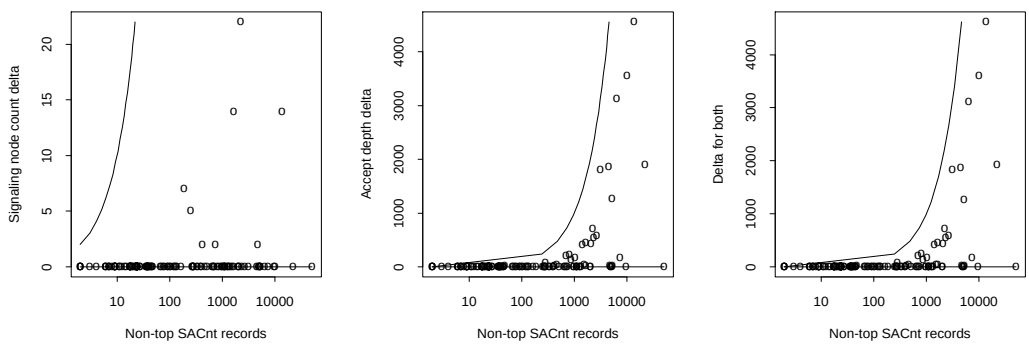

Figure 7.14: Suppression of *SACnt* summary propagation by signaling counts, accept depth, or both. The x axis is logarithmic for clarity.

axis is the number of programs for which each percentage of *SACnt* summaries was suppressed. Figure 7.15 shows that accept depth suppressed a fairly high percentage of *SACnt* summaries in quite a few test programs, while the signaling node count information did not suppress very many summaries in any test program.

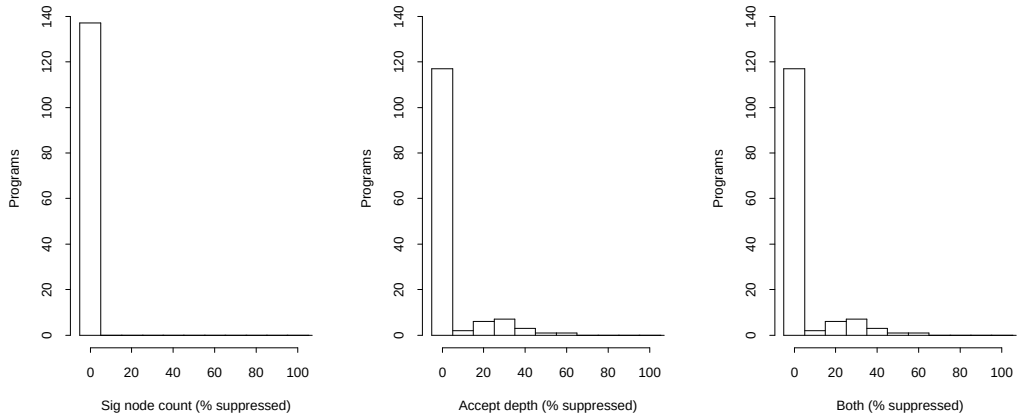

Figure 7.15: Histogram of suppression of *SACnt* summary propagation.

We conclude that accept depth information in the *SACnt* summaries prevents false alarms, but signaling node count information is not likely to do so. Unless evidence is seen to the contrary, we believe that signaling node counts may be safely eliminated from the summaries.

7.10 All-states deadlock analysis.

In this section, we examine the results of our experimentation with all-states analysis. We examine the limitations on both the accuracy of all-states analysis, and on the predictability of its execution time.

7.10.1 Accuracy of all-states analysis.

All-states deadlock analysis is more accurate than polynomial, but must necessarily report some false alarms. This section outlines the sources of this error.

The 19 programs for which all-states analysis reported deadlock were examined by hand, to determine whether the deadlocks were possible in reality. Only one program (dining philosophers) could definitely be said to deadlock. In the remaining programs, either (1) the inspection showed that the program could not deadlock, or (2) deadlocks could not reasonably be detected by inspection given the complexity of the program structure.

In all the all-states deadlocks except for the dining philosophers program, the deadlock cycles involved guarded entries, such that execution could have proceeded if the appropriate guard was open. All-states, however, made the conservative assumption that any combination of guards could be open or closed, as long as this did not violate the Ada language restriction that at least one entry must be open.

Some of the resulting spurious deadlocks would be easy to eliminate with even rudimentary global data flow analysis. For instance, in two programs (`sentest` and `test_controlled`), a flag and a `select` statement were used to implement a structure that behaved like a critical section. In the program `test_protected`, the same condition was used to guard several entries; all of these would be open whenever any of them were.

Other programs had more complicated structures, which would be less amenable to analysis. In `stars_tennis2_demo`, a complex mutual-exclusion structure was used between two tasks that represented tennis players; the structure involved intertask data and control dependences.

Several programs included tasks that allocated resources; these tasks used counters

of available resources as guard conditions. Analysis such as that of [Sax77] might be applied to insure that sufficient resources would always be available to meet demand. In at least the `demo_producer_consumer` program, though, accepted commands from users that might result in deadlock, if all produced resources were exhausted or overflowed the buffer.

7.10.2 Predicting execution time.

Figure 7.16 is a scatterplot of the total time to find deadlocks in all-states mode, versus each of the twelve measured characteristics of the sync graph. Execution time does not apparently correlate well with any of the twelve measured parameters.

It is possible that execution time might correlate well with some function of a combination of the measured independent parameters. We examine such a combination of parameters, proposed by Wampler [Wam85], in Section 7.10.2. However, it appears that number of possible execution waves cannot be reliably predicted from the sync graph statistics, since sync hypergraphs with identical parameters may have vastly different numbers of possible execution waves.

A careful examination of Figure 7.16 indicates that this is true in practice. For each of the measured parameters, many programs with similar independent values had greatly different all-states analysis times. It is therefore unlikely that we can find a good estimation function for execution time for all-states analysis.

Wampler's hypothesis.

The number of possible execution waves of a program is maximized when the nodes in each task of a program may each execute with any other combination of nodes on the wave. The total number of possible execution waves is then:

$$\prod_{t \in Tasks} |t|$$

where *Tasks* is the set of all tasks in the program, and $|t|$ denotes the number of nodes in task t. This assumes that the appearance of any node on the execution wave is not

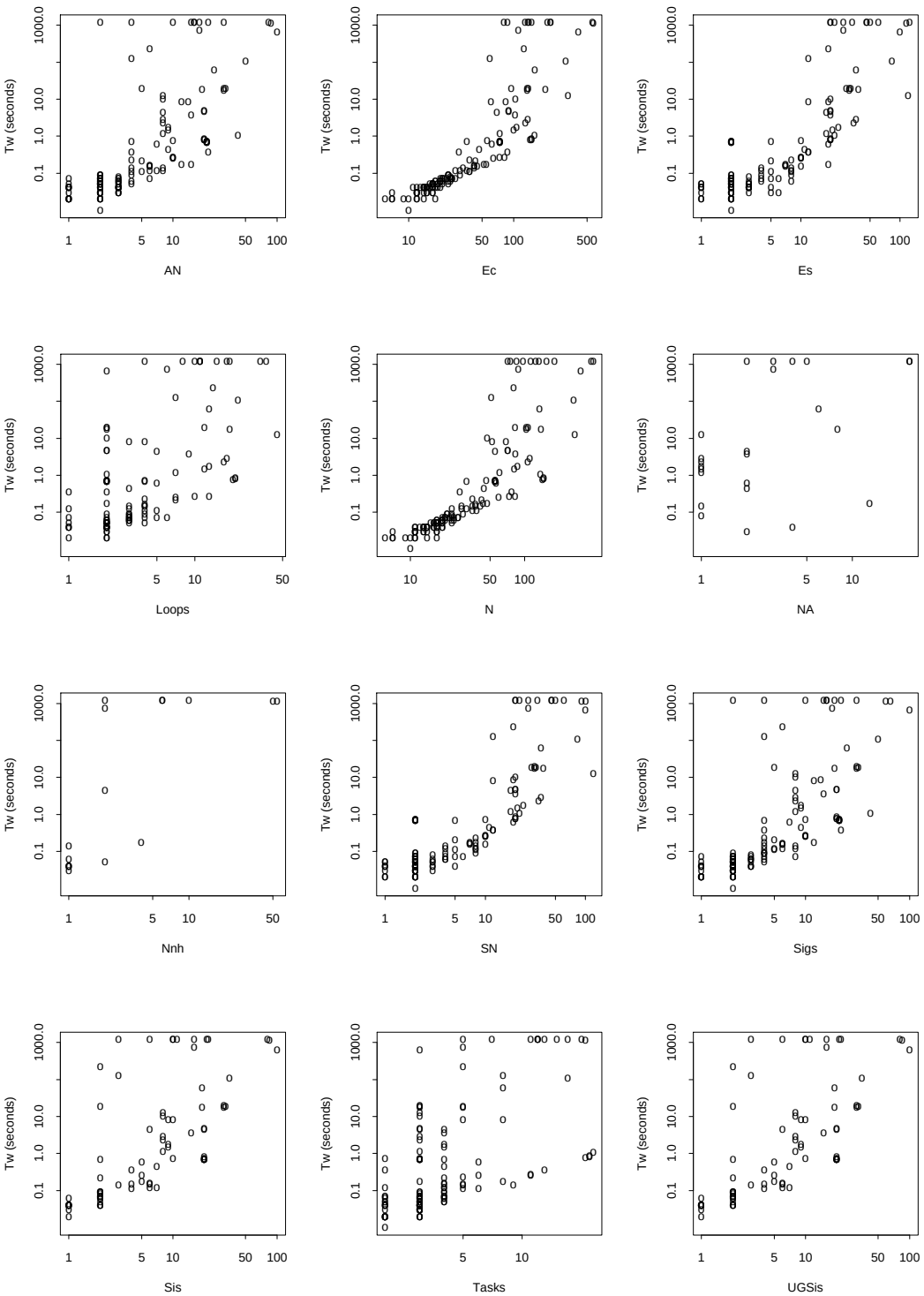

Figure 7.16: Deadlock detection time using all-states analysis, versus each of the 12 measured parameters.

correlated with the appearance of any node in other tasks.

Wampler [TLK92, Wam85] shows that

$$\left(\frac{|N|}{|Tasks|}\right)^{\frac{|Tasks|}{2}}$$

is an accurate predictor of the number of reachable states of a program, for the "dining philosophers" problem [Dij68]. By implication, Taylor et. al., in [TLK92], assume that Wampler's hypothesis applies to a wider set of programs. This assumption is reasonable if all tasks have the same number of nodes, and if nodes appear on the execution wave independently of the other nodes on the wave.

However, as Figure 7.17 shows, Wampler's formula yields wildly inaccurate predictions of state count for many programs. Each point on the graph represents a program for which the complete set of execution states was generated. The x axis represents the state count predicted by Wampler's hypothesis; the y axis represents the actual state count. Both scales are logarithmic for clarity. The r^2 metric for Wampler's hypothesis is -8990000000000, indicating that simply guessing the average of the observed number of states would provide a better prediction than Wampler's hypothesis. Clearly, the state count predicted by Wampler's hypothesis does not correlate well with that observed experimentally.

Conclusions about predicting all-states execution time.

We are thus forced to conclude that the execution time of the all-states algorithm cannot be directly predicted from the twelve measured parameters. However, it is possible that some preliminary structural analysis of the program may yield information useful in estimating the total number of states of the program.

7.11 Conclusions from the experiment.

We make the following major observations from the experimentation:

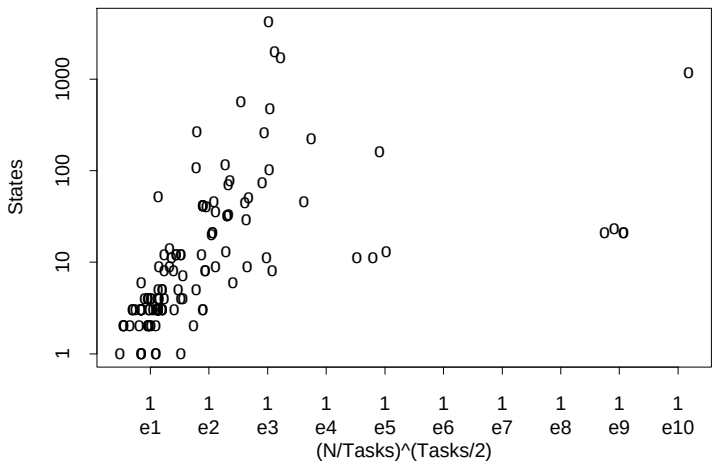

Figure 7.17: Wampler's hypothesis on the number of reachable states in a program.

- Static polynomial-time deadlock detection has acceptable absolute time performance, avoids combinatorial explosions, and has very few false alarms in practice.
- The execution time of the polynomial-time deadlock detection can be predicted with good accuracy, from easily measured features of the sync hypergraph.
- The *CHT* refinement process has excellent accuracy relative to all-states.
- The pinning analysis refinement made the largest contribution to *CHT*, in terms of both the number of programs to which it applied and the amount of *CHT* information it discovered.
- Each of *B4*, critical sections, and remote procedure call analysis showed synergistic effects in the *CHT* computation, enabling discovery of additional *CHT* information by other refinements.
- All-states analysis had exponential time behavior fairly often in larger programs.
- The execution time of state generation cannot be predicted with good accuracy from the features of the sync hypergraph we measured.

Chapter 8

Binary semaphores — an initial study.

8.1 Introduction.

A *binary semaphore* is a Boolean variable which can be accessed only through the statements p and v. A binary semaphore with a value of 1 is said to be *set;* one with a value of 0 is said to be *clear.* Executing $v(X)$ sets the value of the semaphore X to 1. A statement $p(X)$ suspends execution of the containing task until $X = 1$, then clears the value of X to 0 and proceeds. At most one task executing $p(X)$ at any given time may proceed; the task to proceed is chosen arbitrarily. Binary semaphores can be conveniently used both for mutual exclusion of access to shared resources and for event notification between tasks.

Integer semaphores.

Binary semaphores are a specialization of integer semaphores [Dij68]. Given this, static deadlock detection will likely be at least as difficult in programs using integer semaphores as it is in programs using binary semaphores.

8.2 Modeling binary semaphore synchronization for deadlock detection.

We must begin by explaining the program model we intend to use. To analyze deadlock in binary semaphore programs, we will use a tasking model very much like that used for Ada tasks, i.e., we will assume that the number of tasks is fixed, and that all tasks begin at the same time. The only difference is that rendezvous statements of the Ada model are replaced by p and v operations.

We thus make the following simplifying assumptions about the programs:

- Tasks are known statically, are initiated at the start of the program, and terminate at the end of the program.
- Each statement references at most one semaphore, and the semaphore that is referenced is knowable at compile time. There is no aliasing between semaphore references.

- All semaphores are declared to be either *initially-clear semaphores*, which have an initial value of 0, or *initially-set semaphores*, which have an initial value of 1. We adopt the following convention: initially-clear semaphores are given names starting with C, and initially-set semaphores have names starting with S.

Future work may relax these constraints, if doing so proves to be of value.

8.2.1 Mapping semaphores to the rendezvous model.

Programs which obey the simplifying assumptions of Section 8.2 can be readily converted to programs in the rendezvous model. Thus, we may simply convert binary semaphore programs to rendezvous programs with the same tasking behavior, and analyze them using exactly the machinery already developed for the rendezvous model. In this section, we explore the methodology and effectiveness of such an approach.

The set of sequences of operations which are possible to do on a single binary semaphore may be thought of as a regular language. For instance, initially-clear semaphores accept prefixes of the regular language

$$(v^+p)^*.$$

Thus, we may model an initially-clear semaphore `Ci`, by creating a task `T_Ci` which accepts the same regular language. Figure 8.1 shows the code for this task. Note that the `terminate` alternatives are present only to allow the `T_Ci` tasks to terminate when the rest of the program does. Also note that some simpler published simulations of clear semaphores, e.g. the one in Section 9.7 of [Set89], do not behave according to our model, in that they may cause $v()$ statements to wait.

Initially-set semaphores accept prefixes of the regular language

$$(v^*p)(v^+p)* = ((pv)|v)^*.$$

For each initially-set semaphore `Si`, we may similarly create a task `T_Si`, as shown in Figure 8.2.

```
task body T_Ci is begin
--
-- Loop invariant: Semaphore Ci is clear at top of loop.
-- Accept one v, then accept v's until p is called.
--
  loop
    select
      accept v;       -- [a]
    or
      terminate;
    end select;
    loop
      select
        accept v;     -- [b]
      or
        accept p;     -- [c]
        exit;
      or
        terminate;
      end select;
    end loop;
  end loop;
end T_Ci;
```

Figure 8.1: A rendezvous task to simulate the behavior of an initially-clear semaphore.

```
task body T_Si is begin
--
-- Loop invariant: Semaphore Si is set at top of loop.
--
  loop
    select
      -- Accept a p, then a v. If Si was set at the top of the loop,
      -- the "accept p" can proceed, and no further calls to p can be
      -- processed until some task calls v.
      accept p;
      select
        accept v;
      or
        terminate;
      end select;
    or
      -- Accept a v. This also maintains the invariant.
      accept v;
    or
      terminate;
    end select;
  end loop;
end T_Si;
```

Figure 8.2: A rendezvous task to simulate the behavior of an initially-set semaphore.

Although initially-clear semaphores can be simulated by rendezvous, the rendezvous deadlock detection algorithms can be "fooled" by such simulations, as the following example shows. The example is actually a simulation of rendezvous synchronization using initially-clear semaphores; the initially-clear semaphores are in turn simulated by rendezvous statements. Initially-clear semaphore tasks C1 and C2 are assumed to be defined as shown in Figure 8.1.

```
task T1 is begin                task T2 is begin
  C1.v;                           C1.p;          -- [e]
  C2.p;           -- [d]          C2.v;
  C1.v;                           C1.p;
  C2.p;                           C2.v;
end T1;                         end T2;
```

Statements [d] and [e], and statement [a] in initially-clear semaphore tasks C1 and C2 (shown in Figure 8.1), can happen together with each other, pair-wise. These four statements comprise the head nodes of a deadlock cycle, and cause a potential deadlock to be detected. However, all four statements cannot execute simultaneously; when statements [d] and [e] execute simultaneously, statements [b] and [c] in task C1 must be executing. The cycle detected is a case in which an arity-2 *CHT* relation is not sufficient to disprove the existence of a deadlock.

The need for additional state information.

Whenever statements [d] and [e] execute simultaneously, we know that semaphore C1 must be 1, and therefore that [e] can proceed. Unfortunately, the simulation of binary semaphores by rendezvous does not capture this information.

The example above thus points out the need to maintain some approximation of state information in analyzing binary semaphores, i.e., which semaphores must be 1 (or 0) at various points in the program. In Section 8.2.3, we outline the representation of state information in the rendezvous model.

8.2.2 Graph representations of binary semaphore programs.

In this section, we define a graph representation of the structure of binary semaphore programs, called the *SEM sync graph*. This representation is sufficient to capture the possible synchronization behavior of the program. We will use both the parse form (PSEM) and the cycle detection form (CDSEM) of the SEM sync graph in deadlock analysis.

SEM sync graphs are 3-tuples (N, E_C, E_S). The set N contains the nodes of the SEM sync graph, which represent p and v synchronization statements in the program. N also includes the distinguished program begin and end nodes, b and e. N may also contain a task-end node for each task.

The control edge set E_C contains pairs $(m \in N, n \in N)$ such that there is a control path $(m, ..., n)$ in the program containing no members of N except m and n.

The sync edge set E_S is a set of directed sync edges; there is a sync edge (m, n) for any $m = v(X)$ and $n = p(X)$ for the same semaphore X. For all initially-set semaphores S, there is a sync edge from b to all $p(S)$ statements, indicating that the program begin node can effectively "increment" the semaphore.

Figure 8.3 shows a program fragment and its semaphore sync graph. Since all semaphores used in this program are initially clear, there are no sync edges from b to any other node. The behavior of the semaphore program emulates rendezvous synchronization of our continuing example program, using the following convention. The calling task performs a $v(C_i)$, followed by a $p(C_{i+1})$. The accepting task performs a $p(C_i)$, followed by a $v(C_{i+1})$. The points where the "entry" is called must be mutually exclusive.

One example of an emulated rendezvous is found in nodes 1, 2, 7, and 8. To initiate a "rendezvous", the calling task `t1` executes a $v(C_1)$ operation on an initially-clear semaphore C_1, then executes a $p(C_2)$ on another initially-clear semaphore C_2. The receiving task `t2` executes a $p(C_1)$, then a $v(C_2)$. The call points are mutually exclusive because they are all members of the same task for each given "entry."

```
task t1 is begin
  v(C1);          -- 1
  p(C2);          -- 2
  p(C3);          -- 3
  v(C4);          -- 4
  v(C1);          -- 5
  p(C2);          -- 6
end t1;

task t2 is begin
  p(C1);          -- 7
  v(C2);          -- 8
  v(C3);          -- 9
  p(C4);          -- 10
  p(C1);          -- 11
  v(C2);          -- 12
end t2;
```

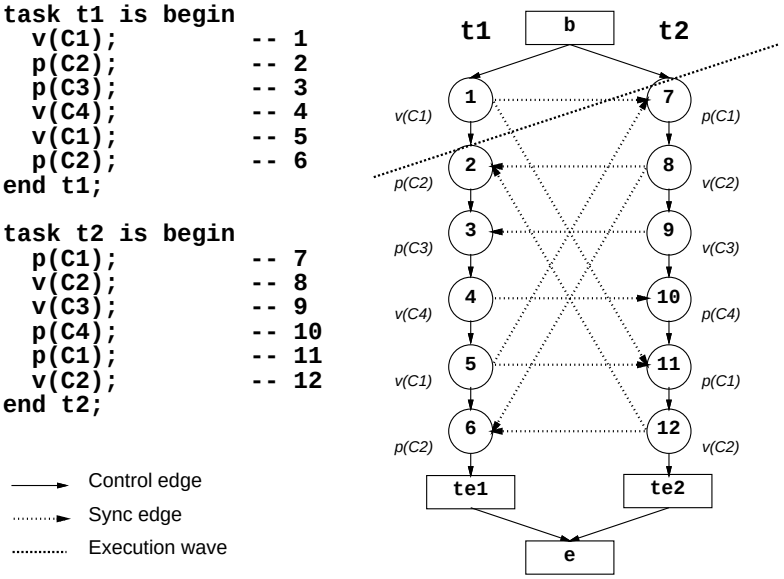

Figure 8.3: A program fragment which uses initially-clear semaphores to simulate rendezvous synchronization, and its semaphore sync graph.

8.2.3 Execution model of binary semaphore programs.

Along with the graph representation of the program, we must define an execution model on the SEM sync graph. For safe analysis, the set of possible synchronization states on the graph of a program and under the execution model must contain at least the set of possible synchronization states of the original program.

As we mentioned in Section 8.2.1, the execution wave model for semaphore programs must include information about the current state of the semaphores. Thus, a *SEM execution wave* is a pair (E, S) where E is a vector of nodes of length $|Tasks|$, and each $E[t]$ contains one node from task t. S is the set of semaphores which are 1 at any time in execution. Returning to the example, the execution wave in Figure 8.3 is $((2,7),\{C_1\})$.

During execution, each node in the SEM sync graph has state OFF-WAVE, WAITING, or READY. Exactly those nodes which are WAITING or READY are on the execution wave. The following model of synchronization and program execution is used:

- Initially, each $E[t]$ contains a node in task t which is a successor of b. All other

nodes in $N - E$ initially have state OFF-WAVE. S is initially empty.

- Any $v(X)$ nodes on the execution wave are READY. Any $p(X)$ nodes on the wave have state READY if $X \in S$; otherwise, the $p(X)$ nodes have state WAITING. Node e, if on the wave, is in state WAITING.

- From execution wave (E, S), we create a successor wave (E', S'), and update the node states as follows. Some node $E[t]$ in state READY is selected arbitrarily. A control successor $E'[t]$ is then selected arbitrarily, to replace $E[t]$ on the execution wave.

  The updated set S' of set semaphores on the successor wave is created as follows:

  - If $E[t]$ is a node $p(X)$, then $S' = S - \{X\}$.
  - If $E[t]$ is a node $v(X)$, then $S' = S \cup \{X\}$.

  The state of $E[t]$ is updated to OFF-WAVE (assuming $E[t] \neq E'[t]$), and $E'[t]$ is updated as appropriate for its type and for S'.

- The above step is repeated until all nodes in E have state WAITING.

The set of feasible execution waves for the SEM sync graph is set of execution waves generated by the above process. Each possible synchronization state of the program is represented by one of the feasible execution waves of its SEM sync graph. There may, however, be some feasible execution waves which do not represent synchronization states of the program; this may be due to the execution times of code between sync statements, or to interdependence of conditional statements.

8.2.4 Constraints on deadlocks in the binary semaphore model.

We now develop a set of constraints on deadlocked execution waves in the binary semaphore model. These constraints will aid in pruning invalid cycles during deadlock detection, and are analogous to the constraints developed for the Ada model in [MR90a].

Suppose that execution wave (E, S) is deadlocked, and that $D \subseteq N$ is the set of head nodes of the deadlock. Let S_D be the set of semaphores upon which the nodes of

D wait. The following constraints hold on D, E, and S:

1. Each node in D is a p node, and all nodes in D are transitively coupled to all others in D. Thus, there is a cycle in the SEM sync graph, which includes the nodes of D, and such that, for every node $d \in D$ in task t:
   (a) The path enters node d through a sync edge.
   (b) The path traverses at least one control flow edge in task t.
   (c) The path leaves t via a sync edge and does not re-enter t.
2. No node $d \in D$ is such that, whenever d executes, a semaphore in S_D is set.

   This constraint may be extended to *subsets* of nodes in D.
3. All nodes of D may execute at the same time. This implies:
   (a) The nodes in D must represent statements whose instances can happen together.
   (b) All nodes in D are *co-executable*.
4. There must exist some feasible execution wave (E, S), such that $D \subseteq E$, each node in E is a p node or a task-end node, and no node in E waits on a semaphore in S.

Constraints 2 and 4 involve semaphores which must be set whenever D executes. Pruning invalid cycles will thus involve safely estimating when a semaphore must be set when a particular node or pair of nodes executes. This is the *MustSet* problem; see Section 8.5.

Single-task deadlocks.

Unlike Ada, the binary semaphore model admits the possibility of deadlocks that involve only a single task. In Ada, we have assumed that no task will ever call an entry in itself; such a program would have a semantic error, and could be rejected by the compiler without further analysis. In binary semaphores, however, programs often perform p

and v operations on the same semaphore, which induces single-task cycles in the sync graph. Such cycles may or may not represent deadlocks, but in general programs which have single-task cycles cannot be rejected as semantically invalid.

Consider the binary semaphore program:

```
task T1 is begin
  p(C);          -- [S1]
  v(C);          -- [S2]
end T1;
```

When S_1 executes, it cannot proceed because semaphore C is clear. However, S_1 could proceed if S_2 were allowed to execute. The path (S_1, S_2, S_1) thus denotes a cyclic "waits-on" relationship that defines a valid deadlock cycle. However, if an initially-set semaphore S were substituted for C, then the cycle would not indicate a deadlock, because S would be set whenever S_1 executed; the cycle would thus violate constraint 2.

8.3 Overview of deadlock detection in binary semaphores.

Having constructed a program representation, an execution model, and a set of constraints on deadlock cycles, we now describe the method used to detect deadlocks in the binary semaphore model of synchronization. The method is quite similar to that for Ada: construct graph representations, derive information that allows pruning spurious cycles, and detect cycles that might not violate the constraints.

Figure 8.4 shows the process of finding deadlock cycles in a program which uses binary semaphores. The major differences between the processes for rendezvous and semaphores are:

- An additional data flow problem, *MustSet*, must be solved to prune cycles which violate Constraint 2. $MustSet(n)$ is the set of semaphores that must be set at node n; no valid deadlock cycle can contain both n and any $p(X)$ node such that $x \in MustSet(n)$.

- Pinning, per se, does not occur in binary semaphore programs; therefore, pinning analysis is omitted from the process of deriving *CHT*.
- Binary semaphore programs often use mutual exclusion to enforce non-concurrency. we make use of information about program structures that implement mutual exclusion to improve *CHT* and to eliminate some spurious cycles.
- *CHT* information is not sufficient to eliminate spurious sync edges. To do this in the binary semaphore model, we must solve an additional "semaphore kill" (*SemKill*) problem.

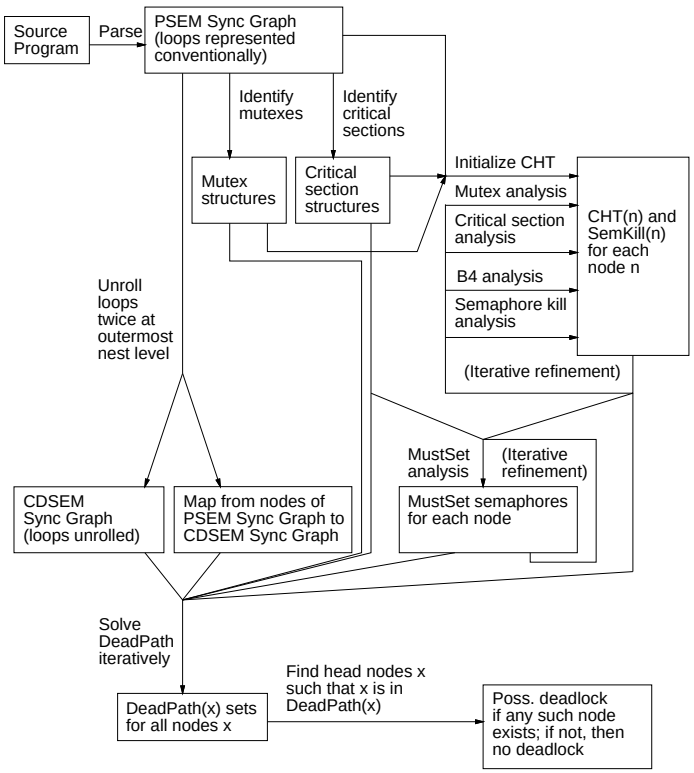

Figure 8.4: Process of detecting deadlocks in a program using binary semaphore synchronization, in polynomial time.

The first step is to parse the source program and form a *parse form SEM (PSEM) sync graph*, or *PSEM sync graph*, with control loops represented conventionally.

CHT analysis then determines a conservative (under)estimate of those nodes that can never execute concurrently with each node. As in the rendezvous model, we use

critical section analysis and limited $B4$ analysis, and refine the CHT information iteratively. Since initially-set semaphores may easily be used to implement mutual exclusion, we add *mutex analysis* as a source of CHT information.

To increase the precision of the limited $B4$ (and other) analysis, we wish to eliminate spurious sync edges. A sync edge $(v(X), p(X))$ is spurious if all instances of $p(X)$ must happen before any instance of $v(X)$, or if the semaphore X is always changed between the time the $v(X)$ statement sets it and the time the $p(X)$ statement waits for it. Thus, the execution of $v(X)$ never directly allows $p(X)$ to proceed.

The sync edge $(v(X), p(X))$ can be valid even if $v(X) \in CHT(p(X))$; therefore, CHT information alone cannot safely eliminate spurious edges by itself. Instead, we must solve a *semaphore kill* problem, *SemKill*, to determine whether the "definition" of X by statement $v(X)$ is always "killed" before statement $p(X)$ starts, i.e., if some $v(X)$ or $p(X)$ statement always executes between the two and redefines the semaphore. *SemKill* analysis is thus quite similar to reaching definitions analysis.

CHT information can be used to refine *SemKill* analysis, and vice versa. Therefore the two analyses are performed iteratively together, for more precise results. In Figure 8.4, *SemKill* is shown as a component of iterative CHT analysis.

We also wish to eliminate spurious deadlock cycles using constraint 2: no head node which waits on semaphore X may participate in a deadlock cycle with any node at which X must be set. To derive this information, we perform *MustSet analysis*, after solving CHT and *SemKill*.

Finally, we create the CDSEM sync graph, by unrolling the loops of the PSEM sync graph at the topmost nest level. The CDSEM sync graph also functions as the cycle location graph in the binary semaphore model. Using the CDSEM sync graph, we search for potential valid deadlock cycles using a D_{Sem} lattice framework for each possible head node. The framework formed from the cross product of all these D_{Sem} frameworks solves the *DeadPath* problem. We report a potential deadlock if a deadlock path found from any $p(X)$ node to itself ($p(X) \in DeadPath(p(X))$); otherwise, the program is reported to be deadlock free.

8.4 *CHT* and binary semaphores.

Deadlock detection for any synchronization primitive is most effective when coupled with an accurate (but conservative) *CHT* algorithm. One of the best ways to obtain *CHT* information in practice is to identify commonly used programming structures which prevent synchronization statements from executing simultaneously. In binary semaphores, these structures include *mutexes* (Section 8.4.1) and critical sections (Section 8.4.2).

Additionally, we can use a variant of limited *B4* analysis (Section 8.4.3). Iterative refinement can be used to increase the accuracy of limited *B4* analysis (and other analyses), by removing extraneous sync edges. To find some of these extraneous edges, we solve the "semaphore kill" (*SemKill*) problem (Section 8.4.4).

8.4.1 Mutual exclusion using semaphores.

An initially-set semaphore S can be used to provide mutually exclusive execution of code segments in different tasks. These mutually exclusive code segments, or *mutex bodies*, are bracketed by $p(S)$ entry nodes and $v(S)$ exit nodes. The set of mutex bodies for a particular set semaphore S form a *mutex structure*, or simply *mutex*, denoted M_S. Figure 8.5 shows an example of a set of mutex bodies for an initially-set semaphore.

We detect the existence of stylized mutex structures. This adds to *CHT* the information about the nodes in mutex bodies of the same structure, which cannot happen together. Additionally, we can use information about mutex structures to prune spurious cycles in deadlock cycle detection, independently of *CHT*.

Structural definition of a mutex.

In this section, we make precise the structural definition of a mutex. This definition will later be used in constructing Algorithm 10 in this section to find mutex structures in a program.

A mutex structure M_S for semaphore S is a set of mutex bodies. Each mutex body is delineated by an *entry node* n and an *exit node* x. A pair of nodes (n, x) delineate a

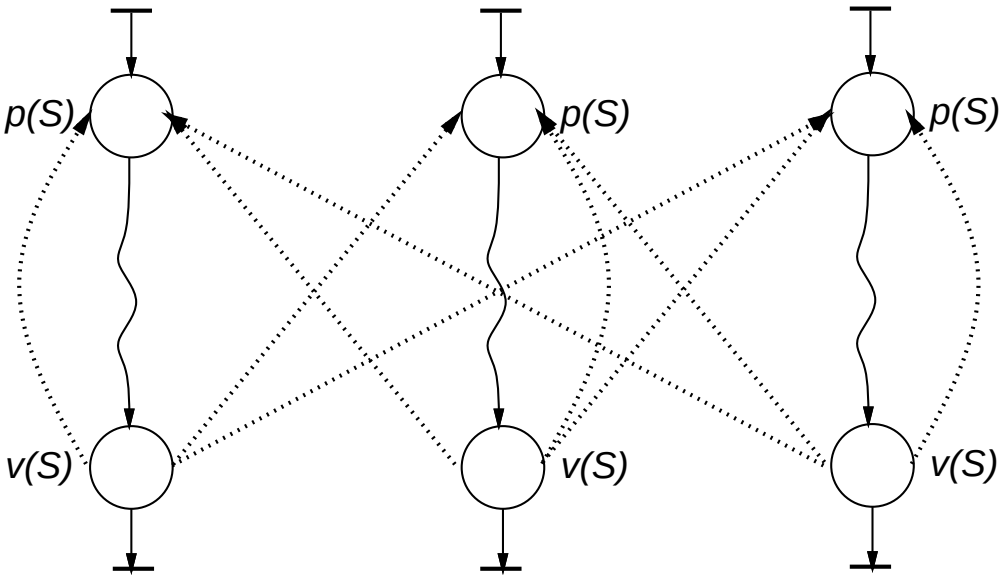

Figure 8.5: A mutex for initially-set semaphore S, with three bodies.

mutex body $MutexB(n, x)$ for semaphore S if:

- $n = p(S)$ and $x = v(S)$.
- $(n, x) \in StrictInt((N, E_C), \{\{z : z = p(S)\}, \{z : z = v(S)\}\})$.

  This requirement implies that *all* $p(S)$ and $v(S)$ nodes (including the pair (n, x)) participate in strict intervals (Section 5.1.3), which must be entered through a $p(S)$ node and exited through a $v(S)$ node, and which have other important structural constraints. The crucial point here is that once control reaches any $p(S)$ node, some $v(S)$ node in the same task must be executed before any $p(S)$ node in that task executes, and vice versa.

The *mutex body* $MutexB(n, x)$ is the set of nodes in the closed interval (n, x) minus $\{n\}$. $MutexB(n, x)$ is said to *exist* only if the pair (n, x) delineates it for some initially-set semaphore S. If $MutexB(n, x)$ exists, no node in $MutexB(n, x)$ can happen together with any node in any other mutex body for the semaphore S.

The *bracket node set* for mutex M_S is the set of entry and exit nodes for all mutex bodies of M_S, i.e., $\{z : z = v(S) \vee z = p(S)\}$. The bracket node set for M_S is denoted $N_{Bracket}(M_S)$.

Lemma 86 *Semaphore S is clear at any node in any mutex body on S.*

Proof: No mutex body may be entered without clearing S. Only one mutex body may be entered at any given time. S is set only by the exit nodes of mutex bodies. □

Note that S may be clear while some node n, outside any mutex body for S, executes. This may happen if n executes concurrently with some mutex body for S.

Lemma 87 *If semaphore S delineates a mutex, then S is set when, and only when, no mutex bodies on S are executing.*

Proof: By contrapositive of Lemma 86. □

Lemma 88 *For each pair of mutex bodies $MutexB(n,x) \in M_S, MutexB(n',x') \neq MutexB(n,x) \in M_S$, if $a \in MutexB(n,x)$ and $b \in MutexB(n',x')$, then $a \in CHT_{perf}(b)$.*

Lemma 89 below allows us to prune some spurious deadlock cycles involving only the entry nodes of a single mutex structure. Such cycles are only symptoms of other anomalies.

Lemma 89 *If a valid deadlock cycle has the entry nodes of a mutex as its only head nodes, then another anomaly exists which contains some head node which is not the entry node of the mutex.*

Proof: Suppose that a valid deadlock cycle D exists such that all its head nodes are entry nodes in M_S. Semaphore S cannot be set; otherwise, D would not be a deadlock. So, by Lemma 87, some node z in a mutex body $MutexB(n,x)$ on S must be executing. z must be a $p(X)$ node for some semaphore $X \neq S$; otherwise, z could complete execution. Therefore, there must exist an anomaly with $p(X)$ as a head node. □

Identifying mutex structures.

Algorithm 10 identifies a set of mutex structures in a program.

Algorithm 10 *Identify mutex structures.*

Input: The PSEM sync graph $G = (N, E_C, E_S)$.

Output: A set of mutex structures, MUTEXSTRUCTS.

Procedure:

1. Create a subpartition[1] $\mathcal{P}$ of N such that, for each initially-set semaphore S_i, there is a set $P_i^p \in \mathcal{P}$ containing all $p(S_i)$ nodes, and a set $P_i^v \in \mathcal{P}$ containing all $v(S_i)$ nodes.

2. Create the control flow graph $G' = (N, E_C)$.

3. Set CANDBODIES to $StrictInt(G', \mathcal{P})$. Place all such pairs (a, b) into the set CANDBODIES.

   The important point here is that CANDBODIES includes the mutex bodies of G', but may also include strict intervals delimited by p and v nodes for different semaphores, or intervals where $v(X)$ is the entry node. We filter these out in Step 4.

4. If there exists a pair $(p(S), v(S)) \in$ CANDBODIES, form a *mutex structure* M_S for semaphore S, which contains all such pairs in CANDBODIES. Let MUTEXSTRUCTS be the set of all candidate mutex structures.

Lemma 90 (Correctness of Algorithm 10.) *The set* MUTEXSTRUCTS *produced by Algorithm 10 contains only mutex structures corresponding to the definition in this section.*

Proof: The algorithm precisely follows the definition. □

Lemma 91 (Time to find mutex structures.) *Using Algorithm 10, the time to find mutex structures is* $\mathcal{O}(|N|^3)$.

Proof: The partitioning of step 1 can be done in $\mathcal{O}(|N|)$ time; G' can be obtained simply by using G and ignoring the set E_S of synchronization edges. Using Algorithm 4, the CANDBODIES set of step 3 can be created in $\mathcal{O}(|N|^3)$ time.

---

[1] By the term "subpartition", we mean a partition whose union may not be the full set N.

Step 4 can be accomplished by scanning the members of CANDSTRUCTS once in $\mathcal{O}(|N|^2)$ time. No more than $\mathcal{O}(|Sems|)$ structures will be created in this scan. ($Sems$ is the set of semaphores in the program; $|Sems| < |N|$.)

Therefore, the total time to construct the set of mutexes is $\mathcal{O}(|N|^3)$. □

8.4.2 Semaphore critical section structures.

Semaphores may also be used to form critical section structures, such that a "call body" in a calling task must execute concurrently with a "critical section body" in a different task, and such that no two call bodies may execute concurrently. We can also derive *CHT* information by "widening" *CHT* (see Section 2.3.1) between the call and critical section bodies. Information about critical section structures may also be used to eliminate spurious deadlock cycles, apart from finding *CHT*.

In this section, we examine in detail one sensible way in which a programmer might form critical section structures, and we show how to derive *CHT* information from it. There is more than one sensible construct for critical sections in the semaphore paradigm; an empirical examination of semaphore programs will be necessary to determine which constructs are most common in practice. With more complete empirical information, we can formulate algorithms to detect these common constructs, and to derive *CHT* information from them.

An example of a semaphore critical section structure.

In Figure 8.3, two pairs of initially-clear semaphores, (C_1, C_2) and (C_3, C_4), are used by tasks `t1` and `t2` to effect rendezvous-type synchronizations between the tasks. Each such semaphore pair delimits a *SEM critical section structure.* Table 8.1 shows the call bodies and critical section bodies for each initially-clear semaphore pair.

We require that the call bodies of the structure be non-concurrent with each other. In the example of Figure 8.3, call bodies $(1, 2)$ and $(5, 6)$ are in the same task, and thus cannot execute concurrently. In other cases where the call bodies are not in the same task, non-concurrency may be enforced by other means, such as embedding the call

Semaphore pair	Call bodies	Critical section bodies
(C_1, C_2)	$(1,2)$, $(5,6)$	$(7,8)$, $(11,12)$
(C_3, C_4)	$(3,4)$	$(9,10)$

Table 8.1: SEM critical section structures for Figure 8.3.

bodies in mutex bodies.

Structural definition of SEM critical section structures.

In this section, we present a formal definition of a SEM critical section structure. This definition will be used to guide the design of Algorithm 11 (of this section) to detect such structures.

A SEM critical section structure for initially-clear semaphores C_1 and $C_2 \neq C_1$ is a quadruple $CSStruct_{C_1,C_2} = (CallB_{set}, CSB_{set}, C_1, C_2)$, where $CallB_{set}$ is a set of call bodies for the pair (C_1, C_2), and CSB_{set} is a set of critical section bodies for the same initially-clear semaphore pair. Each call body is delineated by a call node $c = v(C_1)$ and a return node $r = p(C_2)$; each critical section body is delineated by a entry node $n = p(C_1)$ and an exit node $x = v(C_2)$. The set of SEM critical section structures for a program is called *Crits*.

The existence of call bodies for a pair of initially-clear semaphores depends on the existence of critical section bodies for the same pair, and vice versa. A pair of nodes (c, r) delineates a call body $CallB(c, r)$ for the pair of semaphores (C_1, C_2) if:

- c is of the form $v(C_1)$, and r is of the form $p(C_2)$.
- After executing any $v(C_1)$ node, the program must execute a $p(C_2)$ node in the same task before executing any other $v(C_1)$ nodes in that task, and vice versa. Formally,

$$(c, r) \in StrictInt((N, E_C), \{\{z : z = v(C_1)\}, \{z : z = p(C_2)\}\}).$$

- There is a non-empty set of critical section bodies on (C_1, C_2). Formally,

$$\exists \quad n = p(C_1), x = v(C_2) :$$
$$(n, x) \in \mathit{StrictInt}((N, E_C), \{\{z : z = p(C_1)\}, \{z : z = v(C_2)\}\}).$$

  This means that after executing any $p(C_1)$ node, the program must execute a $v(C_2)$ node in the same task, and vice versa. The program must also contain at least one $p(C_1)$ and $v(C_2)$ node.

- All call bodies on the same pair of semaphores, and the call nodes of those bodies, must be mutually exclusive. Formally,

$$\begin{aligned}
\forall \quad m \qquad & \in (c = v(C_1), r = p(C_2)) \\
& \in \mathit{StrictInt}(\quad (N, E_C), \\
& \qquad\qquad \{\{z : z = v(C_1)\}, \{z : z = p(C_2)\}\}) : \\
\forall \quad n \neq m \quad & \in (c' = v(C_1), r' = p(C_2)) \\
& \in \mathit{StrictInt}(\quad (N, E_C), \\
& \qquad\qquad \{\{z : z = v(C_1)\}, \{z : z = p(C_2)\}\}) : \\
& m \in \mathit{CHT}_{\mathit{perf}}(n).
\end{aligned}$$

$\mathit{CallB}(c, r)$ exists only if (c, r) delineates a call body on some pair of semaphores (C_1, C_2).

A pair of nodes (n, x) delineate an critical section body $\mathit{CSB}(n, x)$ for the pair of semaphores (C_1, C_2) if:

- n is of the form $p(C_1)$, and x is of the form $v(C_2)$.

- After executing any $p(C_1)$ node, the task must execute a $v(C_2)$ node before executing any other $p(C_1)$ nodes, and vice versa. Formally,

$$(n, x) \in \mathit{StrictInt}((N, E_C), \{\{z : z = p(C_1)\}, \{z : z = v(C_2)\}\}).$$

- There is a non-empty set of call bodies on (C_1, C_2). Formally,

$$\exists \quad c = v(C_1), r = p(C_2) :$$
$$(c, r) \in \mathit{StrictInt}((N, E_C), \{\{z : z = v(C_1)\}, \{z : z = p(C_2)\}\}).$$

$\mathit{CSB}(n, x)$ exists only if (n, x) delineates a critical section body on some pair of semaphores (C_1, C_2).

A pair of semaphores (C_1, C_2) delineate a SEM critical section structure if and only if call bodies $(v(C_1), p(C_2))$ and critical section bodies $(p(C_1), v(C_2))$ exist.

Lemma 92 *If $(\mathit{CallB}_{set}, \mathit{CSB}_{set}, C_1, C_2)$ is a SEM critical section structure, then the following repeating sequence of subsets of the set of semaphores $\{C_1, C_2\}$ is set during execution: $(\emptyset, \{C_1\}, \emptyset, \{C_2\})^*$.*

Proof: Initially, neither C_1 nor C_2 is set, and no node in any call body or critical section body is executing. At any time, exactly one node $c = v(C_1)$, where $\mathit{CallB}(c, r) \in \mathit{CallB}_{set}$ may execute, and several nodes $n = p(C_1)$, such that $\mathit{CSB}(n, x) \in \mathit{CSB}_{set}$, may be on the wave.

Suppose that, at some time, no node in any call body or critical section body is executing, and neither C_1 nor C_2 is set. As long as no call node completes, C_1 cannot be set, and control cannot reach any exit node to set C_2. So the situation will remain the same as long as no c completes.

Now suppose that a call node c completes. Semaphore C_1 is set, and control enters $\mathit{CallB}(c, r)$. Control must remain there until r completes, i.e., until some x (as above) completes. By definition of the critical section structure, all nodes in each call body for the structure (including their bracket nodes) are in the CHT_{perf} sets of all nodes in all other call bodies (including their bracket nodes). Thus, no other call body or call node can start execution until $\mathit{CallB}(c, r)$ completes, so C_1 cannot be set again until $\mathit{CallB}(c, r)$ completes.

The next entry node (or nodes) n which is on the wave when c completes (or which starts afterward) is in state READY, because C_1 is set. Exactly one of these entry nodes completes execution, and C_1 is cleared. Control reaches $\mathit{CSB}(n, x)$, and remains

there until x completes execution. Control cannot reach any other exit node $x' \neq x$ such that $CSB(n', x') \in CSB_{set}$, because n' cannot complete execution as long as C_1 is clear. Thus, while control is inside $CSB(n, x)$, both C_1 and C_2 are clear.

When x completes, C_2 is set. Control leaves $CSB(n, x)$, and cannot enter any other critical section body as long as C_1 is clear. Node r can then complete execution, clearing C_2 and allowing control to leave $CallB(c, r)$. Thus, the final state is identical to the initial state (i.e., neither semaphore is set and no call body or critical section body is executing), and the given sequence of semaphore settings repeats until the program ends. □

Lemma 93 *If* $(CallB_{set}, CSB_{set}, C_1, C_2)$ *is a SEM critical section structure, then* C_1 *and* C_2 *are never set simultaneously.*

Proof: Direct from Lemma 92. □

Lemma 94 below allows us to apply the Widening Theorem (Theorem 1) to refine CHT.

Lemma 94 *In a SEM critical section structure, each critical section body must execute concurrently with a call body.*

Proof: From the sequence of actions in Lemma 92. □

The converse of Lemma 94, however, is not true; the critical section body may terminate before the call body does.

We can use Lemma 95 below to prune some spurious deadlock cycles, independently of CHT:

Lemma 95 *Given a SEM critical section structure* $(CallB_{set}, CSB_{set}, C_1, C_2)$. *If nodes* $r = p(C_2)$ *and* $n = p(C_1)$ *are head nodes in a valid deadlock cycle, then there is some anomaly which contains some node, other than a bracket node of that critical section structure, as a head node.*

Proof: If the deadlock cycle is valid, then C_1 cannot be set; otherwise, n could proceed. Since C_1 is not set, some node r' in one of the critical section bodies in CSB_{set} is on the execution wave. r' cannot be a bracket node; the only bracket node

in any critical section body is of the form $v(C_2)$, which could proceed immediately and would thus not cause a deadlock. Thus, there is an anomaly which includes a non-bracket node r' as a head node. □

Lemma 95 implies that we need not propagate nodes in the call body of a critical section structure to the entry nodes of the same structure. From the lemma, r' is more directly the "cause" of any anomaly than is n. The deadlock involving n is only an artifact of the one involving r'.

Identifying SEM critical section structures.

In this section, we show how to identify SEM critical section structures. Since CHT information is used in identifying these structures, we may wish to identify new structures as CHT is refined.

Algorithm 11 *Identify critical section structures.*

Input:

- The PSEM sync graph $G = (N, E_C, E_S)$.
- The set $CHT(n)$ for each $n \in N$.

Output: A set of critical section structures, *Crits*.

Procedure:

1. Create a subpartition $\mathcal{P}$ of N such that, for each initially-clear semaphore C_i, there is a set $P_i^p \in \mathcal{P}$ containing all $p(C_i)$ nodes, and a set $P_i^v \in \mathcal{P}$ containing all $v(C_i)$ nodes.
2. Create the control flow graph $G' = (N, E_C)$.
3. Set CANDBODIES to $StrictInt(G', \mathcal{P})$.
4. If there exist pairs $(v(C), p(C'))$ and $(p(C), v(C')) \in$ CANDBODIES, such that semaphore $C \neq C'$, then form a *candidate critical section structure* $(CallB_{set}, CSB_{set}, C, C')$ for semaphores C and C'. Place all pairs of the form

$(v(C), p(C'))$ into the $CallB_{set}$ set for that structure, and place all pairs of the form $(p(C), v(C'))$ into the structure's CSB_{set} set.

Let CANDSTRUCTS be the set of all candidate critical section structures.

5. Eliminate from CANDSTRUCTS all critical section structures $(CallB_{set}, CSB_{set}, C, C')$ such that:

$$\begin{aligned} &\exists \quad z \in CallB(c,r) \cup \{c\} : \\ &\exists \quad z' \in CallB(c',r') \cup \{c'\}, \\ &\quad\; CallB(c,r) \neq CallB(c',r') : \quad z \notin CHT(z'). \end{aligned}$$

In other words, eliminate all structures which have at least two call bodies, $CallB(c,r)$ and $CallB(c',r')$, such that some node in the body $CallB(c,r)$ (or its call node) can happen together with a node in $CallB(c',r')$ (or its call node).

6. Set *Crits* to CANDSTRUCTS; return *Crits*.

Lemma 96 (Correctness of Algorithm 11.) *The members of the set Crits returned by Algorithm 11 correspond to the definition of critical section structures in this section.*

Proof: Following Step 4, each member of CANDSTRUCTS follows the structural definition of a critical section structure, omitting the restriction that the call bodies and their call nodes can't happen together. Step 5 filters out candidate structures that violate this restriction. Thus, following Step 6, the members of the set *Crits* correspond to the definition of critical section structures in this section. □.

Lemma 97 (Time to find critical section structures.) *Using Algorithm 11, the time to find critical section structures is* $\mathcal{O}(|N|^3 + |N|^2|Sems|^2)$.

Proof: The partitioning of step 1 can be done in $\mathcal{O}(|N|)$ time; creating G' can be done in constant time. Using Algorithm 4, the CANDBODIES set of step 3 can be created in $\mathcal{O}(|N|^3)$ time.

Step 4 can be accomplished by scanning the members of CANDSTRUCTS once in $\mathcal{O}(|N|^2)$ time. No more than $\mathcal{O}(|Sems|^2)$ structures will be created in this scan.

Finally, step 5 can be done by forming the intersection of the CHT sets for all nodes in $CallB_{set}$. Examining a critical section would then require $\mathcal{O}(|N|)$ bit vector operations for a total of $\mathcal{O}(|N|^2|Sems|^2)$ bit operations.

Therefore, the total time to construct the set of critical sections is

$$\mathcal{O}(|N|^3 + |N|^2|Sems|^2).\Box$$

8.4.3 *B4* analysis of semaphores.

This section examines the methodology of finding *B4* information in programs with binary semaphore synchronization. We formulate the problem as a lattice framework, and derive some of its properties using the framework.

We can also improve the accuracy of limited *B4* analysis by eliminating spurious sync edges that don't correspond to real synchronizations. A sync edge $(v(X), p(X))$ is spurious if the setting of semaphore X by the $v(X)$ statement never directly allows the $p(X)$ statement to proceed. The methodology of eliminating spurious sync edges is described in Section 8.4.5.

An example of limited *B4* analysis in the semaphore model.

We can use limited *B4* analysis on the program of Figure 8.3 to gain information on concurrency between statements. For instance, since statement 7 cannot complete execution until C_1 is posted, statement 8 cannot begin execution until the first $v(C_1)$ statement (1) has completed. Since statement 1 has only one instance, $1 \in B4(8)$.

Lattice framework for limited *B4* analysis in binary semaphores.

We can develop a safe underestimate of the *B4* relation, using a lattice framework. The framework, in the standard quadruple notation is $D_{B4} = (G_{B4}, L_{B4}, F_{B4}, M_{B4})$. This framework converges in a time polynomial in the number of nodes in the SEM sync graph.

A specialized graph, $G_{B4} = (N, E_C, E_{Completors})$, is used in the *B4* analysis. The

derivation of $E_{Completors}$ is presented later in this section.

Later in this section we define, and prove the properties of, the lattice and function spaces used for the $B4$ problem in binary semaphores.

Completor edges in the $B4$ sync graph.

To estimate $B4$ accurately we must know, for each node n in the graph, the nearest ancestors of n that are $p(X)$ nodes. This will give us information on what semaphores (if any) must have been set (initially or by some $v(X)$ node) before n is reached in control flow, which nodes m could have set them, and whether all instances of any nodes must have completed before all such nodes m, and is thus in $B4(n)$.

The completors of a node n are the $p(X)$ nodes which are the most immediate control predecessors of n (plus the program begin node b, if there is a control path from b to n which includes no $p(X)$ nodes). If b is not in the completors of n, then at least one semaphore must be set before n can be reached in control flow. We designate the set of completor nodes of n as $Completors(n)$.

A completor edge (m, n) is drawn from every $v(X)$ node m that sets a particular semaphore X, to every node n which has a completor $p(X)$ that waits on X. (If X is initially set, or b is in n's completors, then b may also be the tail of the edge.) We denote the set of completor edges as $E_{Completors}$.

Formally, the set of completor edges is:

$$E_{Completors} = \{(m,n) : \quad (\exists p : (m,p) \in E_S \wedge p \in Completors(n)) \\ \vee (m = b \wedge b \in Completors(n))\}.$$

Thus, for node n, at least one instance of one of the nodes $m : (m, n) \in E_{Completors}$ must complete before the first instance of n can start execution. If, for all such completor edges, there is some node q such that one of the following holds:

- All instances of node q complete before any instance of m starts; or,
- $q = m$ and q has only one instance in the program;

Then all instances of q must complete before any instance of n starts. Under these conditions, $q \in B4(n)$.

$B4$ lattice.

A $B4$ *summary* is defined as a pair (C, S) where C and S are sets of nodes in a PSEM sync graph. The lattice space L_{B4} is:

$$L_{B4} = 2^N \times 2^N.$$

The meet operation is element-wise set intersection; the $\sqsubseteq$ operator is element-wise set containment. For convenience, we denote $(\emptyset, \emptyset)$ as $B4_\perp$ and (N, N) as $B4_\top$. Note that this is exactly the same definition used for limited $B4$ analysis in the rendezvous model.

Lemma 98 *The height of the L_{B4} lattice space is $2|N| + 1$.*

Proof: Identical to Lemma 20. □

$B4$ function space and mapping for binary semaphores.

We map a control or completor edge $e = (m, n)$ to a function f_e as follows.

$$f_e((C, S)) = (C \cup S \cup K_e, N)$$

if e is a control edge, or

$$f_e((C, S)) = (N, C \cup S \cup K_e),$$

if e is a completor edge. For either edge type,

$$K_e = CReach(m) \cup \{m\} - \{p : m \in CReach(p)\},$$

i.e., the set of nodes q such that there is a control path from q to m (possibly of zero length), but no control path of length > 0 from m to q.

Lemma 99 *The functions f_e are inflationary and monotone; they are therefore closed under meet and function composition, and are 1-semibounded.*

Proof: The same as Lemma 24. □

Lemma 100 *The functions f_e are not distributive in general.*

Proof: The same as Lemma 22. □

Lemma 101 *The time for either a meet operation in L_{B4} or an edge function evaluation for $f_e \in F_{B4}$ is $\mathcal{O}(|N|)$.*

Proof: The same as Lemma 25. □

Lemma 102 (Time to solve *B4* using the worklist algorithm.) *The total time for convergence of a semaphore B4 lattice problem using the worklist iterative algorithm is $\mathcal{O}(|N|^3 \log |N|)$ in the worst case.*

Proof: Direct from Lemmas 12, 98, and 101. □

Lemma 103 (Time to solve *B4* using Tarjan's algorithm.) *The total time to solve a B4 lattice problem using Tarjan's path expression algorithm is $\mathcal{O}(|N|^3)$ in the worst case.*

Proof: Direct from Lemmas 13 and 101. □

8.4.4 Eliminating spurious sync edges.

A sync edge is *spurious* if the setting of the semaphore by the $v(X)$ statement at the tail[2] of the edge never directly allows the $p(X)$ statement at the head to proceed. It should be fairly obvious that these edges have no real significance in the execution of the program, and that they degrade the accuracy of the *B4* solution by forcing meets to be taken over more edges than necessary. Thus, it is important that we identify and eliminate spurious sync edges.

Identifying spurious sync edges is rather like identifying definition-use pairs such that the definition never reaches the use. In this case, the definition is the $v(X)$ statement at the tail, and the use is the $p(X)$ statement at the head. Intervening $p(X)$ and $v(X)$ statements also constitute definitions of X, since they may change the value of X.

[2]Nomenclature: For a directed edge (m,n), m is its tail and n is its head.

Examples of spurious sync edges.

An example of a spurious sync edge is $(1, 11)$ in Figure 8.3. The edge is spurious because C_1 is always cleared by statement 7 before 11 starts execution. The edge $(12, 2)$ in Figure 8.3 is also spurious, since the single instance of node 2 must complete before the single instance of node 12.

Formal definitions of spurious sync edges.

Lemma 104 below formally defines spurious sync edges. From this lemma, we can construct algorithms to detect spurious edges, using *B4* information. Finding spurious edges can further improve the solution to the *B4* problem.

Lemma 104 *A sync edge* $(m = v(X), n = p(X))$ *is spurious if either: (a) a* $p(X)$ *or* $v(X)$ *statement always executes after* m *finishes and before* n *starts, or (b) if no instance of* m *ever executes before any instance of* n*.*

Proof: If a $p(X)$ or $v(X)$ statement always executes after m finishes and before n starts, then X is always redefined between m and n, and thus the execution of m cannot directly allow n to complete.

If no instance of m ever executes before any instance of n, then m cannot directly allow n to complete.

Thus, if either of these conditions is satisfied, the sync edge (m, n) is spurious. □

If we have an estimate of *B4*, we can directly find cases in which no instance of m ever executes before any instance of n. Determining that a *single* $p(X)$ or $v(X)$ statement always executes between m and n may likewise be derived straightforwardly from the *B4* sets. A more difficult problem is finding when one of a *set* of $p(X)$ or $v(X)$ statements must always execute between m and n.

8.4.5 The "semaphore kill" problem.

In this section, we describe an approach to finding when a sync edge $(m = v(X), n = p(X))$ is *killed* by a set S of $p(X)$ and $v(X)$ statements, such that $m \notin S$ and $n \notin S$.

The sync edge is killed if some instance of some statement in S must always execute between the time any instance of m finishes and the time the next instance of n starts, and if m cannot re-execute while n is executing. This is the same as saying that no *definition* of the semaphore created by m is alive while n executes, i.e., m is killed at the top of n and $m \in CHT(n)$.

We find killed $v(X)$ statements by generating and propagating "semaphore kill" information through the sync graph; we define a lattice framework for this purpose. The framework retains information about which $v(x)$ (sync edge tail) nodes are killed at which other nodes in the SEM sync graph. If $v(X)$ has been killed at $p(X)$, then the sync edge $(v(X), p(X))$ is spurious.

Lemmas 105 through 108 provide a way to generate and propagate semaphore-kill information throughout the program. Propagation of semaphore-kill information via sync edges is possible, at least in some limited circumstances, as Lemma 108 shows.

Lemma 105 *Given a node $m = v(X)$, if $n \in B4(m)$, then m is killed at the bottom of n.*

Proof: Obvious. □

Lemma 106 *Given a node $m = v(X)$, and a node $n \neq m$, if $n = p(X)$ or $n = v(X)$, then m is killed at the bottom of n.*

Proof: When n completes execution, any definition of X by m, which occurs earlier than the execution of n, will be redefined by n. □

Lemma 107 *Assume a node $m = v(X)$, and a node $n = p(Y)$, where semaphore Y might or might not be the same as X. If, for all sync edges (n', n) such that n' is not killed at the top of n, either:*

- *$m \in B4(n')$ and m is killed at n'; or,*
- *$m = n'$;*

Then m is killed at the bottom of n.

Proof: Suppose that n completes after semaphore Y is last set by statement n'. If $m \in B4(n')$ and m is killed at n', then no definition of m can ever reach a successor of n. Therefore, m is killed at the bottom of n.

If $m = n'$(and thus $X = Y$), then n redefines Y after it was set by m. Thus, m is killed at the bottom of n. □

For the first condition of Lemma 107, we cannot simply require that m be in $CHT(n) \cap CHT(n')$, since m might execute between n' and n. If this were to happen, then a definition of X by m might reach a use at n. Requiring m to be in $B4(n')$ strengthens the condition allowing propagation along sync edges, thus reducing the number of circumstances to which it can be applied. Unfortunately, no safe, weaker condition allowing propagation along sync edges is evident. Finding such a condition is a subject of future research.

Lemma 108 *Given a node $m = v(X)$, if, for all control predecessors n' of n, m is killed at the bottom of n' and $m \in CHT(n) \subseteq CHT_{perf}(n)$, then m is killed at the bottom of n.*

Proof: Suppose control reaches n through predecessor n'. If m is killed at the bottom of n', then m is also killed at the top of n. Since $m \in CHT_{perf}(n)$, m cannot execute between the time n' finishes and the time n finishes. Thus, no definition of X by m can reach a use at n, and m is thus killed at the bottom of n. □

The semaphore kill (*SemKill*) lattice.

The lemmas of Section 8.4.5 suggest a lattice framework similar to the *B4* framework, but with some differences in the function space. We label this the *semaphore kill*, or *SemKill*, framework. This section describes the framework, and identifies some of its convergence time and accuracy properties.

The *SemKill* lattice framework, $D_{SemKill}$, is a quadruple,

$$(G, L_{SemKill}, F_{SemKill}, M_{SemKill}).$$

G is the PSEM sync graph. $L_{SemKill}$ is the lattice of pairs of sets $(C, S) : C, S \subseteq N$.

Intuitively, if $SemKill(n) = (C, S)$, then $C \cup S$ contains the set of $v(X)$ nodes killed at n. C represents the set of $v(X)$ nodes killed at the top of n which reach n via control flow; S represents the set which reaches n via synchronization.

The meet in $L_{SemKill}$ is element-wise set intersection; $(C_a, S_a) \sqsubseteq (C_b, S_b) \equiv C_a \subseteq C_b \wedge S_a \subseteq S_b$. The lattice space is identical to L_{B4}.

Lemma 109 *The height of the $L_{SemKill}$ lattice space is $2|N| + 1$.*

Proof: Similar to Lemma 20. □

The $F_{SemKill}$ function space.

This section defines, and presents some theoretical results for, the $F_{SemKill}$ edge function space. For a more intuitive idea about the edge functions themselves, the reader is advised to skip this section on first reading and proceed to the paragraph labeled "*SemKill* mapping function," in this section.

$F_{SemKill}$ is the set of functions of the form:

$$f((C, S)) = ((((C \cup S) \cap P_1) \cup G_1), (((C \cup S) \cap P_2) \cup G_2))$$

where P_1, P_2, G_1, and G_2 are each fixed subsets of N.

Lemma 110 *The set of edge functions in $F_{SemKill}$ is closed under meet and function composition.*

Let $f, g \in F_{SemKill}$ be:

$$f((C, S)) = ((((C \cup S) \cap P_1) \cup G_1), (((C \cup S) \cap P_2) \cup G_2))$$

and

$$g((C, S)) = ((((C \cup S) \cap P_3) \cup G_3), (((C \cup S) \cap P_4) \cup G_4))$$

So, if we let $(f \sqcap g)((C,S)) = (C',S')$, then

$$\begin{aligned} C' = \quad & (((C \cup S) \cap P_1) \cup G_1) \cap (((C \cup S) \cap P_3) \cup G_3) \\ = \quad & ((C \cup S) \cap P_1 \cap P_3) \\ \cup \quad & ((C \cup S) \cap P_1 \cap G_3) \\ \cup \quad & ((C \cup S) \cap P_3 \cap G_1) \\ \cup \quad & (G_1 \cap G_3) \\ = \quad & (C \cup S) \cap ((P_1 \cap P_3) \cup (P_1 \cap G_3) \cup (P_3 \cap G_1)) \cup (G_1 \cap G_3). \end{aligned}$$

Letting

$$P_1' = (P_1 \cap P_3) \cup (P_1 \cap G_3) \cup (P_3 \cap G_1),$$

and

$$G_1' = G_1 \cap G_3,$$

we have

$$C' = ((C \cup S) \cap P_1') \cup G_1'.$$

By symmetry, a similar result holds for S'. So $F_{SemKill}$ is closed under meet.

Now, let $(f \circ g)((C,S)) = (C'',S'')$. Then we have:

$$\begin{aligned} C'' = \quad & (((((C \cup S) \cap P_1) \cup G_1) \cup (((C \cup S) \cap P_2) \cup G_2)) \cap P_3) \cup G_3 \\ = \quad & ((((C \cup S) \cap P_1) \cup ((C \cup S) \cap P_2) \cup (G_1 \cup G_2)) \cap P_3) \cup G_3 \\ = \quad & ((((C \cup S) \cap (P_1 \cup P_2)) \cup (G_1 \cup G_2)) \cap P_3) \cup G_3 \\ = \quad & ((C \cup S) \cap ((P_1 \cup P_2) \cap P_3)) \cup (((G_1 \cup G_2) \cap P_3) \cup G_3) \end{aligned}$$

Letting

$$P_1'' = (P_1 \cup P_2) \cap P_3,$$

and

$$G_1'' = ((G_1 \cup G_2) \cap P_3) \cup G_3,$$

we have

$$C'' = ((C \cup S) \cap P_1'') \cup G_1'',$$

and similarly for S''. So $F_{SemKill}$ is also closed under function composition. □

Lemma 111 *$F_{SemKill}$ is monotone, but is not distributive or inflationary.*

Proof: Suppose that $f \in F_{SemKill}$. Then, let

$$(C, S) = f((C_a, S_a)) \sqcap f((C_b, S_b))$$

and

$$(C', S') = f((C_a, S_a) \sqcap (C_b, S_b)).$$

Expanding C,

$$\begin{aligned} C &= (((C_a \cup S_a) \cap P_1) \cup G_1) \cap (((C_b \cup S_b) \cap P_1) \cup G_1) \\ &= (((C_a \cup S_a) \cap P_1) \cap ((C_b \cup S_b) \cap P_1)) \cup G_1 \\ &= (((C_a \cup S_a) \cap (C_b \cup S_b)) \cap P_1) \cup G_1 \end{aligned}$$

Expanding C',

$$\begin{aligned} C' &= (((C_a \cap C_b) \cup (S_a \cap S_b)) \cap P_1) \cup G_1 \\ &= (((C_a \cup S_a) \cap (C_a \cup S_b) \cap (C_b \cup S_a) \cap (C_b \cup S_b)) \cap P_1) \cup G_1 \\ &\subseteq (((C_a \cup S_a) \cap (C_b \cup S_b)) \cap P_1) \cup G_1 \\ &= C. \end{aligned}$$

By symmetry, $S' \subseteq S$. Therefore, $(C', S') \sqsubseteq (C, S)$, and thus $f((C_a, S_a) \sqcap (C_b, S_b)) \sqsubseteq f((C_a, S_a)) \sqcap f((C_b, S_b))$.

Thus, the functions of $F_{SemKill}$ are monotonic. However, since it is possible to have

$$(C_a \cup S_b) \cap (C_b \cup S_a) \not\supseteq (C_a \cup S_a) \cap (C_b \cup S_b),$$

then $F_{SemKill}$ is not distributive.

To show that the functions of $F_{SemKill}$ are not, in general, inflationary, simply select $f \in F_{SemKill}$ with $P_1 = G_1 = \emptyset$. □

Lemma 112 *$F_{SemKill}$ is 2-bounded.*

Proof: Let $f \in F_{SemKill}$ be a function

$$f((C,S)) = ((((C \cup S) \cap P_1) \cup G_1), (((C \cup S) \cap P_2) \cup G_2)).$$

Therefore, if we let $f^2((C,S)) = (C',S')$, and $f^{[2]}((C,S) = (C,S) \sqcap f((C,S)) = (C'',S'')$, then

$$\begin{aligned}
C' = & \ (((((C \cup S) \cap P_1) \cup G_1) \cup (((C \cup S) \cap P_2) \cup G_2)) \cap P_1) \cup G_1 \\
= & \ ((((C \cup S) \cap (P_1 \cup P_2)) \cup G_1 \cup G_2) \cap P_1) \cup G_1 \\
= & \ (((C \cup S) \cap P_1) \cup (G_2 \cap P_1)) \cup G_1. \\
C'' = & \ (((C \cup S) \cap P_1) \cup G_1) \cap C \\
= & \ (C \cap P_1) \cup (G_1 \cap C) \\
\subseteq & \ C'.
\end{aligned}$$

By symmetry, $S'' \subseteq S'$; thus, $(C',S') \sqsupseteq (C'',S'')$ and $f^2 \sqsupseteq f^{[2]}$, satisfying the definition of 2-boundedness. So $F_{SemKill}$ is 2-bounded. □

Note that $F_{SemKill}$ is not 1-semibounded, except possibly where the G and P sets defining it are further restricted.

Finally, we discuss the time bounds of meet and function application in the *SemKill* framework in Lemma 113:

Lemma 113 *The meet of two elements of $L_{SemKill}$, or the application of any function in $F_{SemKill}$, can be found in $\mathcal{O}(|N|)$ time.*

Proof: Both the meet operation and the application of any function in $F_{SemKill}$ require a constant number of union and intersection operations, on sets whose maximum size is $|N|$. □

SemKill **mapping function.**

In the *SemKill* framework, we map a control or sync edge $e = (m,n)$ to a function $f_e \in F_{SemKill}$ as follows:

- If e is a control edge, then:

- If n is a $p(X)$ node, then $f_e((C,S)) = (((C \cup S) \cap CHT(m)) \cup K_e, N)$.

  That is, $P_1 = CHT(n)$, $G_1 = K_e$, P_2 is arbitrary, and $G_2 = N$. The $(C \cup S) \cap CHT(n)$ term allows propagation of *SemKill* information along control edges from the top of m to the top of n, using Lemma 108. K_e is a "generate" set, arising from Lemmas 105 and 106.

- Otherwise, $f_e((C,S)) = (((C \cup S) \cap CHT(n)) \cup K_e, ((C \cup S) \cap CHT(n)) \cup K_e)$.

  That is, $P_1 = P_2 = CHT(n)$, $G_1 = G_2 = K_e$. This construction prevents *SemKill* from being driven to N when n is a node of a type that can have no sync predecessors, e.g., a $v(X)$ node. It is equivalent to placing a dummy sync edge from m to n.

K_e is defined as follows:

- If m is a node $v(X)$ or $p(X)$, then $K_e = \{q : q = v(X) \wedge q \neq m\} \cup \{q : q = v(X) \wedge n \in B4(q)\}$.

  The subset $\{q : q = v(X) \wedge n \in B4(q)\}$ is permitted by Lemma 105; effectively it acts as a "generate" set for n. The subset $\{q : q = v(X) \wedge q \neq m\}$ is permitted by Lemma 106, and acts as a "generate" set for m.

- Otherwise, $K_e = \{q : q = v(X) \wedge n \in B4(q)\}$.

- If e is a sync edge, then, from Lemma 107, $f_e((C,S)) = (N, ((C \cup S) \cap B4(m)) \cup \{m\})$.

  That is, P_1 is arbitrary, $G_1 = N$, $P_2 = B4(m)$, and $G_2 = \{m\}$.

Convergence properties of the *SemKill* framework.

In this section, we derive worst-case bounds for the convergence time of a problem in the *SemKill* lattice framework. The height of the *SemKill* lattice, the meet and function application times, and the 2-boundedness of the *SemKill* function space, together give us the worst-case time.

Lemma 114 (Time to solve *SemKill*.) *The time for convergence of a SemKill problem using the B4 framework is* $\mathcal{O}(|N|^3 \log |N|)$ *using the worklist algorithm, or*

Analysis	Time	Lemma
Mutex	$\mathcal{O}(\|N\|^3)$	91
Critical section	$\mathcal{O}(\|N\|^3 + \|N\|^2\|Sems\|^2)$	97
B4 (worklist)	$\mathcal{O}(\|N\|^3 \log(\|N\|))$	102
B4 (Tarjan)	$\mathcal{O}(\|N\|^3)$	103
SemKill (worklist)	$\mathcal{O}(\|N\|^3 \log \|N\|)$	114
SemKill (Tarjan)	$\mathcal{O}(\|N\|^3)$	114

Table 8.2: Time for each component of *CHT* analysis in binary semaphore programs.

$\mathcal{O}(|N|^3)$ *using Tarjan's algorithm.*

Proof: Direct from Lemmas 12, 13, 109, and 113. □

To iteratively refine *B4*, we would first solve *B4*, then use the information to refine *SemKill* and eliminate spurious sync edges. The updated sync graph would then be mapped to a new *B4* problem, and the iteration repeated until stabilization.

Lemma 115 *The total time to find an iterative solution of a limited B4 problem, using SemKill information to refine B4, is* $\mathcal{O}(|N|^5 \log |N|)$ *using the worklist algorithm, or* $\mathcal{O}(|N|^5)$ *using Tarjan's algorithm.*

Proof: From Lemmas 102 and 103, the time to solve either a *B4* or *SemKill* problem is $\mathcal{O}(|N|^3 \log |N|)$ or $\mathcal{O}(|N|^3)$, using the worklist algorithm or Tarjan's algorithm respectively. At most $|N|$ can be in the *B4* set of any node; thus, at most $|N|^2$ iterations can be done until the *B4* solution stabilizes. Thus, the time to find a solution using *SemKill* information to refine *B4*, is $\mathcal{O}(|N|^5 \log |N|)$ using the worklist algorithm, or $\mathcal{O}(|N|^5)$ using Tarjan's algorithm. □

Because of the expense of iterative refinement, it is not likely that it should be used, except possibly in cases for which other methods fail to disprove deadlock.

8.4.6 Time complexity of *CHT* analysis.

Table 8.2 shows the time for each technique used in *CHT* analysis of binary semaphore programs.

If we wish to refine the *CHT* solution iteratively, we assume that we will repeat the

critical section analysis during refinement. In actuality, though, it is unclear how much new *CHT* information could be provided through iterative refinement of semaphore critical sections. This could be determined experimentally.

Lemma 116 (Time to solve *CHT*.) *The total time for CHT analysis with SemKill refinement of B4 information, is*

$$\mathcal{O}(|N|^5 \log(|N|) + |N|^4 |Sems|^2)$$

using the worklist algorithm, or

$$\mathcal{O}(|N|^5 + |N|^4 |Sems|^2)$$

using Tarjan's algorithm.

Proof: Similar to Lemma 68. □

8.5 Finding nodes at which semaphores must always be set.

Constraint 3 asserts that a proposed deadlock cycle that includes a head node $p(X)$ is valid only if semaphore X may be clear while the head nodes of the cycle are executing. Otherwise, if X were always set while the head nodes were executing, then $p(X)$ could always complete, and the cycle would not represent a real deadlock. We can prune any cycles containing both $p(X)$ and a head node at which X must be set.

To prune such cycles, we develop an estimate of a "semaphore must be set" relation, *MustSet*. For each node n, $MustSet(n)$ is a set of semaphores which are known to be set whenever n is executing. *MustSet* can be estimated using a lattice framework which is closely related to that of *SemKill*.

8.5.1 Theoretical background for the *MustSet* relation.

The following lemmas allow us to construct an approximation to the *MustSet* relation. Lemma 117 below provides a "generate" function for set semaphores.

Lemma 117 *Let node $n = v(X)$ for some semaphore X. Semaphore X is set at the start of any successor of n.*

Proof: Direct from the execution model. □

Lemmas 118 and 119 provide the basis for a "preserve" function for set semaphores, propagating through nodes and control edges.

Lemma 118 *If semaphore X is set at the start of statement s, and no $p(X)$ statement can execute concurrently with s, and control passes from s to a successor s', then X is set at the start of s'.*

Proof: Since no $p(X)$ can execute concurrently with s, X must be set at the conclusion of X. From the execution model, the state of X is unchanged when the successor is placed on the wave. So X must be set at the start of s's successor. □

Lemma 119 *If a semaphore X must be set at the bottom of all predecessors of a node n, then X must be set at the top of n.*

Proof: Direct from the execution model. □

Lemma 120 is a result which is closely related to widening (Theorem 1). The lemma allows propagation of set semaphore information between tasks.

Lemma 120 *If statement n must happen together with statements in some set S, such that semaphore X is always set whenever any $s \in S$ executes, then semaphore X is always set when n executes.*

Proof: Obvious. □

8.5.2 The *MustSet* lattice framework.

In this section, we construct a lattice framework to estimate the *MustSet* relation. We also investigate the convergence properties of the framework.

The *MustSet lattice framework* is a quadruple,

$$D_{MustSet} = (G_{MustSet}, L_{MustSet}, F_{MustSet}, M_{MustSet}).$$

Each of these components is explained in the sections below.

The $G_{MustSet}$ graph structure.

We use a graph structure, $G_{MustSet}$, to estimate *MustSet* information. $G_{MustSet}$ is a graph (N, E_C, E_{MayHT}) where E_{MayHT} is the set of node pairs (x, y) such that x may happen together with y (i.e., $x \neq y$ and $x \notin CHT(y)$). The E_{MayHT} set is used to propagate *MustSet* information between tasks, by widening as in Lemma 120.

Note that we need to have an accurate, though conservative, approximation of CHT in order to construct E_{MayHT}. If we overestimate the number of nodes which may happen together, we reduce the number of implied set semaphores which can be found at a given node by the method of Lemma 120. Thus, an underestimate of CHT is conservative for this purpose, as well as for pruning deadlock cycles.

The $L_{MustSet}$ lattice space.

The *MustSet* lattice space, $L_{MustSet}$, is a lattice of pairs of sets of semaphores, (C, M). Intuitively, $MustSet(n) = (C, M)$ means that C is the set of semaphores which are known to be set at the start of n by propagation along control edges (Lemma 118), and M is the set of semaphores known to be set at the start of n by widening (Lemma 120).

Meet in $L_{MustSet}$ is element-wise intersection; $(C_a, M_a) \sqsubseteq (C_b, M_b) \equiv C_a \subseteq C_b \wedge M_a \subseteq M_b$.

Lemma 121 *The height of the $L_{MustSet}$ lattice space is $2|Sems| + 1$.*

Proof: Similar to Lemma 98. □

The $F_{MustSet}$ function space.

$F_{MustSet}$ is the set of functions of the form:

$$f((C, M)) = ((((C \cup M) \cap P_1) \cup G_1), (((C \cup M) \cap P_2) \cup G_2))$$

where P_1, P_2, G_1, and G_2 are all fixed sets of semaphores. $F_{MustSet}$ is clearly quite similar to $F_{SemKill}$; we shall therefore refer to many of the lemmas proved in Section 8.5.1 in our proofs in this section.

Lemma 122 *$F_{MustSet}$ is closed under meet and function composition.*

Proof: Similar to Lemma 110. □

Lemma 123 *$F_{MustSet}$ is monotone, but is not distributive or inflationary.*

Proof: Similar to Lemma 111. □

Lemma 124 *$F_{MustSet}$ is 2-bounded.*

Proof: Similar to Lemma 112. □

As with $F_{SemKill}$, note that $F_{MustSet}$ is not generally 1-semibounded.

Lemma 125 *The meet of two elements of $L_{MustSet}$, or the application of any function in $F_{MustSet}$, can be found in $\mathcal{O}(|Sems|)$ time.*

Proof: Both the meet operation and any function application require a constant number of union and intersection operations, on sets whose maximum size is $|Sems|$. □

Lemma 126 (Time to solve *MustSet*.) *A solution to a MustSet problem can be found in time $\mathcal{O}(|N|^2 \log(|N|)|Sems|)$ by the worklist iterative algorithm, or in $\mathcal{O}(|N|^2|Sems|)$ by Tarjan's algorithm.*

Proof: Direct from Lemmas 12, 13, 121, and 125. □

The $M_{MustSet}$ mapping function.

To each edge $e = (m, n)$ of $G_{MustSet}$, we assign an edge function f_e as follows:

$$f_e((C, M)) = (((C \cup M) \cap Pres(m)) \cup Gen(m, n), Sems)$$

if $e \in E_C$, or

$$f_e((C, M)) = (Sems, ((C \cup M) \cap Pres(m))$$

if $e \in E_{MayHT}$.

For nodes $m = v(X)$ and all its control successors n, $Gen(m, n) = \{X\} - \{Y : \exists z = p(Y) \in N - CHT(n)\}$. For the program begin node b, and for all its control successors n, $Gen(b, n)$ is the set of set semaphores. For all other nodes m and n, $Gen(m, n) = \emptyset$.

For all nodes, $Pres(m) = Sems - \{Y : \exists m' = p(Y) \in N - CHT(m)\}$. X is preserved if no $p(X)$ statement can execute concurrently with m. Note that this covers the case where $m = p(X)$, since $m \notin CHT(m)$ (if m can execute at all). Further note that $Gen(m) \subseteq Pres(m)$.

By inspection, these edge functions are members of $F_{MustSet}$.

Refining *MustSet* with critical section information.

Earlier in this section, we described a method of estimating *MustSet* which used only *CHT* information. Lemma 120 implies that, if we can find any sets of nodes S such that some member of S must always happen together with a node n, then we may be able to refine $MustSet(n)$ further. If semaphore X is always set while any member of S executes, then X must be set at n.

If n is a member of a critical section body for a critical section structure, then critical section analysis will have already found such a set S, i.e., set of nodes in all call bodies of the structure. By Lemma 94, n must happen together with some member of S. Since S is a proper subset of $N - CHT(n)$, using S in propagating *MustSet* information between tasks will generally yield more accurate information than pseudotransitivity, since S will usually be smaller than $N - CHT(n)$.

Methodology and time bounds.

MustSet can be iteratively refined by propagating information from call bodies to critical section bodies for all critical section structures, during each iteration of the worklist algorithm. Thus, we will have an inner loop in the worklist algorithm, which iterates over critical section bodies. If a node n is on the worklist because it has had its *MustSet* set increased, and if n is a member of a critical section body, then we attempt to propagate *MustSet* information from n to all call bodies of critical section structures to which n belongs, in the inner loop. This multiplies the cost of finding *MustSet* by a factor of $|Crits|$, in the worst case.

As *MustSet* information is refined for one structure, it may be useful in refining

MustSet information for a second structure. This may be the case even if there are no nodes which are members of the same structure, because *MustSet* information can propagate to nodes outside of structures.

Lemma 127 (Time to solve *MustSet* with critical section refinement.) *A solution to a MustSet problem, refined with information from critical sections, can be found in time*

$$\mathcal{O}(|N|^3 \log(|N|)|Sems|^2|Crits|)$$

by the worklist iterative algorithm, or in

$$\mathcal{O}(|N|^3|Sems|^2|Crits|)$$

by Tarjan's algorithm.

Proof: From Lemma 126, a single solution to the *MustSet* problem can be found in $\mathcal{O}(|N|^2 \log(|N|)|Sems|)$ by the worklist iterative algorithm, or in $\mathcal{O}(|N|^2|Sems|)$ by Tarjan's algorithm. *MustSet* information can only be increased at most $|N||Sems|$ times before each node is assigned the top value.

To increase *MustSet* once, each critical section must be examined, in the worst case. Therefore, at most $|N||Sems||Crits|$ solutions to the *MustSet* problem may have to be found before stabilization. Total solution time is therefore $\mathcal{O}(|N|^3 \log(|N|)|Sems|^2|Crits|)$ using the worklist iterative algorithm, or $\mathcal{O}(|N|^3|Sems|^2|Crits|)$ using Tarjan's algorithm. □

Additional refinements.

Further refinement may be possible for nodes which are members of more than one critical section body. The following example shows one (admittedly farfetched) way in which this might be done. Suppose that node m is a member of two critical section bodies for different structures. Further suppose that each of these structures has one call body. Let these call bodies be denoted $CallB(c, r)$ and $CallB(c', r')$. Whenever m executes, it must execute together with nodes in both bodies. If there is a node

$m' \in CallB(c, r)$ such that X is set in every node in $CallB(c', r') - CHT(m')$, then X must be set whenever m and m' both execute. Circumstances like this, however, are probably not likely enough to justify the effort of analysis to detect them.

8.5.3 The *MustClear* relation.

We can enumerate a set of lemmas about semaphores that *must be clear*, as we did in Section 8.5.1 for *MustSet*. These lemmas allow us to construct an approximation of a *MustClear* relation, i.e., the semaphores which must be clear whenever each node executes.

Although the *MustClear* relation is not useful in eliminating spurious deadlocks, it may be useful in other ways. For example, if semaphore X must be clear at a statement $s = p(X)$, then s will *always* be the head node of an infinite wait anomaly.

We present this section for reference only, and do not develop an approximation algorithm for *MustClear* here.

Lemma 128 *Let s' be a successor of statement $s = p(X)$. Semaphore X is clear at the start of s'.*

Proof: Direct from the execution model. □

Lemma 129 *If semaphore X is clear at the start of statement $s = p(Y)$, and no $v(X)$ statement can execute concurrently with s, then X is set at the start of the successor of s.*

Lemma 130 *If statement s must happen together with statements for which semaphore X is always clear, then semaphore X is always clear at s.*

Proof: Obvious. □

8.6 Finding deadlock cycles in semaphore programs.

A program can deadlock only if it has a valid deadlock cycle in its SEM sync graph. A deadlock cycle is valid only if it obeys the four constraints of Section 8.2.4. We therefore

attempt to find cycles in the SEM sync graph that *may* obey the constraints; if no such cycle can be found, the program cannot deadlock.

To detect deadlock cycles in the SEM sync graph, we hypothesize each $p(X)$ node as a head node h of a deadlock cycle. We then, effectively, search all deadlock paths starting at h, to see if there might be a cycle containing h that obeys the constraints. If such a cycle exists, then h may participate in a deadlock; if no such cycle exists for any $p(X)$ node, then the program cannot deadlock.

We use a D_{Sem} lattice framework to perform this search, by propagating head nodes along possible deadlock paths. The following information is used to prune the impossible paths:

- The CDSEM sync graph construction does not contain cycles which violate constraint 1: that the cycle enters each task at a single node, traverses at least one control edge, and leaves the task. The construction of the the SEM sync graph enforces the rule that, when sync edges enter a node, none exit it. The CDSEM sync graph is also constructed without control cycles; all cycles must therefore obey constraint 1.

- *MustSet* information prunes paths which are invalid under constraint 2: that none of the head nodes of a deadlock cycle wait on a semaphore which is always set when the head nodes execute together. To do this pruning, we retain information on which semaphores must be set, and which semaphores are waited on by head nodes, for any deadlock path.

- *CHT* information prunes paths which are invalid under constraint 3: that all head nodes of a deadlock cycle must be able to execute concurrently. We prune paths which include any head node that can't happen together with h.

- *SemKill* information eliminates some spurious sync edges, and therefore some spurious deadlock paths.

8.6.1 The CDSEM sync graph.

The CDSEM sync graph contains only those cycles which satisfy constraint 1. We unroll the outermost nest level of any loops in the program twice, adding forward edges around iterations as appropriate. The resulting graph contains no control cycles, but does contain deadlock cycles (if the program can deadlock).

8.6.2 Lattice framework for deadlock cycle detection.

In this section, we define the lattice framework used in cycle detection on the SEM sync graph. We describe the base framework particularized for a given hypothesized head node. A framework which finds deadlock paths for all head nodes at once, may be formed by taking the cross product of all these frameworks for all possible head nodes.

The base framework for cycle detection is $D_{Sem} = (G_{Sem}, L_{Sem}, F_{Sem}, M_{Sem})$. G_{Sem} is the CDSEM sync graph, which has its outermost loop bodies unrolled twice and backedges removed. Each lattice framework is specific to a particular assumed head node, h.

On the proposed deadlock paths beginning at h, we propagate conservative information on which semaphores which must be *set* at some head node on the path, and on which semaphores must be *waited upon* by some head node on the path. If the intersection of these two sets is non-empty at some node n, then some head node on the path must be in state READY when all head nodes execute. Thus, no path containing n can correspond to an actual deadlock cycle.

Lattice space.

The value of L_{Sem} at a node n contains information about whether there is any valid deadlock path from h to n, and about which semaphores must be set, and must be waited upon, by the head nodes of all such paths.

A non-top element of L_{Sem} is a pair of sets (H, S), where:

- H is a set of semaphores which are waited upon by the head nodes in all deadlock paths from h to n; and,

- S is the set of semaphores known to be set at the head nodes along any such path.

We also include a separate top element, $Sem_\top$. In L_{Sem}, $a \sqsubseteq b$ iff:

- $b = Sem_\top$; or,
- $a = (H_a, S_a)$ and $b = (H_b, S_b)$ and $H_a \subseteq H_b$ and $S_a \subseteq S_b$.

The meet is then element-wise set intersection. For convenience, we denote $Sem_\bot = (\emptyset, \emptyset)$.

Lemma 131 *The height of the L_{Sem} lattice is $2|Sems| + 2$.*

Proof: Similar to Lemma 20. □

Function space.

In this section, we define the function space for the D_{Sem} lattice framework, and prove some theorems relating to the convergence time of the framework. We also show what these edge functions represent, in more concrete terms.

To each edge $e \in E_C \cup E_S$, we assign a function $f_e \in F_{Sem} : L_{Sem} \to L_{Sem}$. F_{Sem} is the closure under function meet and composition of the following sets of functions:

- *Constant functions*: $f_e(x) = k_e$ where $k_e \in L_{Sem}$.

  The distinguished *top function*, $f_\top(x) = k_\top$, is a constant function.

- *Conditional inflationary functions:* Let $\tau_e : L_{Sem} \to \{True, False\}$ be an arbitrary *test function* for edge e. We require that τ_e be evaluable in $\mathcal{O}(|N|)$ time.

  The edge function itself is defined as follows: if $\tau_e((H_x, S_x)) = True$, then $f_e(x) = (H_x \cup H_e, S_x \cup S_e)$. Otherwise, $f_e(x) = Sem_\top$.

  The *identity function*, f_ι, is a conditional inflationary function which has $\tau_\iota(x) = True$ for all x and $H_\iota = S_\iota = \emptyset$. Similarly, a *conditional identity function* has τ_ι set arbitrarily and $H_\iota = S_\iota = \emptyset$.

Lemma 132 *The set of edge functions is monotonic. Furthermore, with the exception of the constant functions, the set of edge functions is inflationary.*

Proof: Identical to Lemma 70. □

Define F^{-}_{Sem} as the closure of the set of the conditional inflationary functions under meet and function composition. Using arguments similar to those of Section 6.1.2, we need not consider the constant functions in deriving time bounds on convergence of the D_{Sem} framework.

Lemma 133 *The functions of F^{-}_{Sem} are monotonic, inflationary, and 1-semibounded, but are not distributive.*

Proof: Similar to Lemmas 71 and 72. □

Lemma 134 *The meet operation in L_{Sem} requires $\mathcal{O}(|Sems|)$ time.*

Proof: The meet requires the intersection of two pairs of sets of semaphores. □

Lemma 135 *The edge functions in L_{Sem} require $\mathcal{O}(|N| + |Sems|)$ time to compute.*

Proof: The constant functions require constant time to compute. The conditional inflationary functions require a test function evaluation, and possibly a join operation. Altogether, the edge function takes $\mathcal{O}(|N| + |Sems|)$ time to compute. □

Lemma 136 (Time for convergence of cycle detection problems in semaphores.) *A problem in the Sem lattice framework requires*

$$\mathcal{O}(|N|^3 \log(|N|) + |N|^2 |Sems| \log(|N|))$$

time to complete using the worklist iterative algorithm.

Proof: Direct from Lemmas 12, 134, and 135. □

Lemma 137 *A problem in the Sem lattice framework requires $\mathcal{O}(|N|^3 + |N|^2|Sems|)$ time to complete using Tarjan's algorithm.*

Proof: Direct from Lemmas 13, 134, and 135. □

Mapping function in the SEM lattice framework.

In this section, we show how edge functions are assigned to each edge in the CDSEM sync graph, given a hypothesized head node h. This is done by way of a *mapping*

function, M_{Sem}.

The mapping function $M_{Sem} : (E_S \cup E_C) \rightarrow F_{Sem}$ maps control and sync edges to the function space. It is defined for an edge $e = (m, n)$ as follows:

- If $e \in E_C$, then:
  - If $m = h = p(X)$, then $M_{Sem}(e)$ is the constant function $(\{X\}, MustSet(h))$.
  - Otherwise, $M_{Sem}(e) = f_\iota$.
- If $e \in E_S$, (and, therefore, $n = p(X')$ for some semaphore X'), then $M_{Sem}(e)$ is the conditional inflationary function f_e, with H_e, S_e, and τ_e defined as follows:
  - $H_e = \{X'\}$.
  - $S_e = MustSet(n)$.
  - $\tau_e((H, S)) = False$ if any of the following hold:
    - $h \in CHT(n) \cup (Task(n) - \{h\})$.

      This prunes cycles containing head nodes which are in $CHT(h)$, or which enter h's task at nodes other than h.
    - $(H \cup \{X'\}) \cap (S \cup MustSet(n)) \neq \emptyset$.

      This prunes cycles which contain any head node that waits on any semaphore which must be set.
    - $m \in Task(h) \wedge \{h, n\} \subseteq N_{Bracket}(M_S)$ for some mutex M_S.

      This prunes cycles which contain only bracket nodes of a single mutex structure.
    - $m \in Task(h) \wedge h \in (c, r) \in CallB_{set} \wedge (n, x) \in CSB_{set}$ for some critical section structure $(CallB_{set}, CSB_{set}, C_1, C_2)$, or vice versa.

      This prunes cycles which contain only the entry node of a critical section body, and nodes inside the call body of the same structure.

    Otherwise, $\tau_e((H, S)) = True$.

Lemma 138 below shows that the edge test function τ_e, as defined in the mapping above, can be evaluated in the time required in the definition of the F_{Sem} function space given previously.

Lemma 138 *The test function τ_e for a control edge $e = (m, n)$ can be evaluated in $\mathcal{O}(|N|)$ time.*

Proof: Assume that sets are represented as bit vectors and that sets (and indices from nodes into sets) which remain constant throughout the iteration are precomputed. Evaluating whether there is an $h' \in CHT(n) \cup (Task(n) - \{h\})$ takes $\mathcal{O}(|N|)$.

Evaluating whether $(H \cup \{X'\}) \cap (S \cup MustSet(n)) \neq \emptyset$ requires $\mathcal{O}(|Sems|)$ time. Since the number of semaphores used in the program is not more than the number of nodes, this is also $\mathcal{O}(|N|)$ time.

Evaluating whether $m \in Task(h) \wedge \{h, n\} \subseteq N_{Bracket}(M_S)$ for some mutex M_S, can be done as follows. If n is a bracket node of any mutex, then we can precompute for edge (m, n) whether m and n are bracket nodes in the same mutex. Evaluating this for τ_e thus takes constant time, as does evaluating whether $m \in Task(h)$ and h is inside a call body of a critical section structure for which n is an entry node, or vice versa.

The total time to evaluate τ_e is thus $\mathcal{O}(|N|)$. □

8.7 Complexity of deadlock analysis in binary semaphores.

Table 8.3 summarizes the time complexity of analyzing a binary semaphore program for deadlock.

Operation	Worklist	Tarjan's
Parse	$\mathcal{O}(\|N\| + \|E_C\| + \|E_S\|)$	$\mathcal{O}(\|N\| + \|E_C\| + \|E_S\|)$
CHT sets	$\mathcal{O}(\|N\|^5 \log(\|N\|)$ $+\|N\|^4 \|Sems\|^2)$	$\mathcal{O}(\|N\|^5$ $+\|N\|^4 \|Sems\|^2)$
$MustSet$ sets	$\mathcal{O}(\|N\|^3 \log(\|N\|) \|Sems\|^2 \|Crits\|)$	$\mathcal{O}(\|N\|^3 \|Sems\|^2 \|Crits\|)$
Cycle detection	$\mathcal{O}(\|N\|^3 \log(\|N\|) + \|N\|^2 \|Sems\| \log(\|N\|))$	$\mathcal{O}(\|N\|^3 + \|N\|^2 \|Sems\|)$

Table 8.3: Complexity of deadlock analysis for binary semaphore programs.

Chapter 9

Concurrent C — an initial study.

Feature	Ada	Concurrent C
Control	Tasks	Processes
Synchronization	Rendezvous	Transactions
Type definition	`task type`	`process spec`
Multiple entries	Selective wait	Selective wait
Guarded entries	Guard clauses	Guard conditions
Remote proc. call	`accept...do`	`accept...{...}`
Call aborting	`select <call> or delay...`	`within <duration>...`
Entry aborting	`select...else`	`select...otherwise`
Self-termination	`terminate`	`terminate`
Task aborting	`abort`	`c_abort`

Table 9.1: Corresponding synchronization features in Ada and Concurrent C.

9.1 Overview of Concurrent C.

Concurrent C [GR86],[GR88] is a version of the C language [KR78], augmented for use in concurrent programming. It uses a rendezvous style of synchronization, quite similar in functionality (and often in syntax) to that of Ada. Table 9.1 presents some of the features common to both the Ada and Concurrent C synchronization structures, with naming conventions or syntax for each language.

9.1.1 Differences between Concurrent C and Ada.

Despite the above similarities, there are important differences between Ada and Concurrent C which make static analysis of synchronization behavior quite difficult in the latter. These differences lie in task initiation, reference, and termination.

Tasks in Ada are generally declared statically, and are activated when their containing body (package, procedure, block, etc.) is activated. Ada does have a facility for creating tasks dynamically, but this appears to be infrequently used in practice. Experience has also shown that, in most of those Ada programs in which tasks are created dynamically, only a constant number are created, and the entry call patterns between tasks are usually fairly simple. Thus, it is fairly easy to replace dynamically created tasks by statically created ones in the program. However, in Concurrent C, process instances may be created in loops, and the number of process instances might

not be known at compile time.

Process variables.

Processes in Concurrent C are created only by an executable program statement, `create`:

```
process <processType> <label>;        /* <label> will refer to a */
                                      /* process of type */
                                      /* <processType> */
...
<processVar> = create <processType>(<params>);
                                      /* Initiate a new */
                                      /* process and save a */
                                      /* reference to it */
<processVar>.<transaction>(<params>); /* Call the new process */
```

References to a particular process or transaction can thus only be made through variables.

Transaction variables.

Variables may also have, as values, the *transactions* of processes:

```
trans <returnType> (*<transPtrID>)(<paramTypeList>);
                                /* Declare a transaction pointer */
...
<transPtrID> = <processVar>.<transaction>;
                                /* Assign the transaction pointer */
(*<transPtrID>)(<paramList>);  /* Call the transaction */
```

Other Concurrent C synchronization constructs.

In addition to the `create` operation, some further built-in procedures and functions are relevant to Concurrent C synchronization:

```
c_abort(<processVar>);     /* Terminates <processVar> */
c_mypid();                 /* Returns process variable */
                           /* for this process */
```

We will pay some attention to the `c_mypid()` function call in analyzing Concurrent C programs, but will defer the analysis of programs calling `c_abort` for later research.

Passing variables between processes.

Process and transaction variable definitions may be passed to processes at the time they are created, through the parameters of the `create` statement. Synchronization variables may also be passed between processes through the transaction mechanism.

It is important to note that the referenced process or transaction must have been created before it can be referenced. We will use this fact in Section 9.5.1, to eliminate spurious deadlock cycles.

An example of Concurrent C synchronization.

The program in Figure 9.2 duplicates our continuing task interaction example in Concurrent C. Figure 9.2 shows the *parse form Concurrent C sync graph*, or *PCC sync graph*, for this program. Here, we represent process creation by *create nodes* c_1 and c_2; process initiation is represented by *process begin* nodes *pb1*, *pb2*, and *pb3*.

```
process spec P1()
{
  trans init(process P2 p2Proc);
  trans p1t();
}

process spec P2(trans void (*p1Trans));
{
  trans p2t();
}

process body P1()
{
  accept init(p2Proc);     /* r */
  p2Proc.p2t();            /* s */
  accept p1t();            /* t */
  p2Proc.p2t();            /* u */
```

```
process body P2(p1Trans)
{
  accept p2t();                          /* v */
  p1Trans();                             /* w */
  accept p2t();                          /* x */
}

main()
{
  process p1Proc;
  process p2Proc;

  while (!done)
  {
    p1Proc = create P1();                /* c1 */
    p2Proc = create P2(p1Proc.p1t);      /* c2 */
    p1Proc.init(p2Proc);                 /* q */
  }
}
```

Figure 9.1: A Concurrent C program showing the use of process and transaction variables.

One instance each of processes `P1` and `P2` are created by each iteration of the loop

in `main`. The process variables `p1Proc` and `p2Proc`, respectively, are assigned the values of these process instances. When the instance `p2Proc` of `P2` is created, it is passed a transaction variable identifying transaction `p1Proc.p1t` through its creation parameters. A process variable identifying `p2Proc` is passed to `p1Proc` by the call to `p1Proc.init` in `main`. The remainder of the program duplicates the synchronization structure of the continuing example.

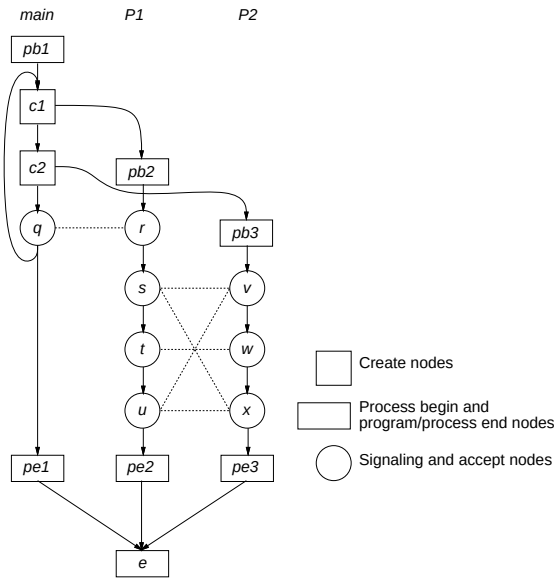

Figure 9.2: The parse-form CC sync graph of the Concurrent C program of Figure 9.1.

The CC sync graph structure shown is a fairly intuitive extension of that for the Ada rendezvous; a more complete description of CC sync graphs may be found in Section 9.2.1. We show it here mainly to compare the sync graph for the continuing example in Concurrent C with those for other constructs for synchronization and manipulation of processes.

Initial simplifying assumptions.

Our main objective here is to demonstrate that static analysis of deadlock is a viable research goal, in languages which feature dynamic process creation and process reference variables. To accomplish this, we create a model of programs in a subset of Concurrent C. This subset is large enough to pose interesting research problems, though not so

large as to make the problems too daunting for this preliminary study.

We also assume that we have conservative, though fairly accurate, estimates of some necessary data flow information about the program. This information is necessary for accurate analysis.

Reasons for the choice of language features.

The subset of language features we address was chosen for two reasons. The first reason is to avoid (for the present) complications in modeling several constructs of the full Concurrent C language which have already been successfully modeled for Ada, such as remote procedure calls and selective wait. We feel that modeling such constructs would complicate the explanation of the model, while adding little worthwhile evidence of its viability.

The second reason is that some features, such as recursive process initiation, are beyond the scope of a preliminary effort. These features, though, probably could be addressed using techniques developed for the analysis of recursively defined data structures [HPR89, HN90, CWZ90, HHN92].

Of note in this area is Duesterwald and Soffa [DS91], in which the authors perform concurrency analysis in the presence of recursive activation of *units*, i.e., tasks or procedures. They do this by forming a "must happen before" relation (similar to our $B4$) on unit activation sites, identifying sites that cannot start a recursion (e.g., call sites for a procedure P which must be inside either P or procedures called only by P), intersecting $B4$ sets over the remaining sites, and and propagating $B4$ information into the activated units.

The assumptions.

The subset of Concurrent C programs which we initially address has the following restrictions:

- Transactions consist of simple `accept` statements or transaction calls. In particular, we omit `select` statements, guards, `suchthat` qualifiers, `abort` statements,

and `delay` alternatives.

Since most of these constructs have been successfully incorporated into the Ada model, they should present few problems for the successful modeling of Concurrent C. The resulting Concurrent C model thus has much the same flavor as the early Ada model of [MR90a], with the addition of process-valued variables and explicit process creation.

- A reasonably accurate static analysis algorithm is available to estimate *reaching definitions* and *def-clear paths* of scalar variables.

  If no transactions or procedure calls ever pass or return process or transaction variables, this analysis could be done using global data flow methods. Otherwise, we would have to utilize *interprocedural* and *interprocess* analysis, assuming that process and transaction variables could propagate between procedures and processes respectively.

  The development of interprocess data flow algorithms is beyond the scope of the current research; we will simply assume, for the moment, that they can be developed. Others [LC91] have made some preliminary investigations in interprocess data flow methods. We expect that these algorithms will be improved, in both accuracy and representational completeness, within the next few years.

- Programs do not contain arrays or linked structures that refer to processes or transactions.

  Such structures introduce imprecision in identifying references. Array sectioning [CK88, BK90] or structure summarization [CWZ90, HN90, HHN92] would be needed to resolve array references; it is unlikely that any sectioning method would be as precise as scalar analysis.

- Procedure and function calls are either inlined (if they are not recursive) or transformed to iteration (if they are recursive).

- Processes do not create new instances of themselves, through direct or indirect recursion.

- There are no aliasing effects between process or transaction variables.

9.2 Modeling of Concurrent C program execution.

To analyze Concurrent C programs for deadlock, we must define both the graph representation of the program and an abstract model of program execution on the graph. Once these models are complete, we may determine the constraints on deadlock cycles in the graph, and develop algorithms for conservatively detecting such cycles.

The graph representation we develop in Section 9.2.1 is of size (i.e., nodes plus edges) polynomial in the number of synchronization statements in the (transformed) program. We also discuss an alternative graph representation in Section 9.2.2, which may yield more accurate results, but which does not necessarily achieve the goal of polynomial graph size.

Since process instances are created dynamically, we use an execution model in which the number of process instances is not fixed for any given program. In particular, as process instances are created, they are added to the execution wave.

9.2.1 CC sync graphs.

The static structures of Concurrent C programs are modeled by *CC sync graphs*. CC sync graphs may be in one of two forms, depending on the analysis to be performed on them: *parse form* or *PCC* sync graphs, and *cycle detection form* (*CDCC*) sync graphs. The PCC sync graph is the most natural to use in modeling program execution, since control loops are represented.

To improve the accuracy and speed of analysis (as explained later in Section 9.3.2), we may also wish to represent partitions[1] of a complete program. In a partition, a distinguished *root process*, with only one instance, creates a set of interacting process instances. Neither the root process instance, nor any of the created process instances, transact with instances of any process outside the partition.

[1] Technically, partition *elements*; see Section 9.3.2.

Definition of the CC sync graph.

A CC sync graph is a quadruple (N, E_C, E_S, b), where N is a set of nodes and E_C and E_S are sets of directed control edges and undirected sync edges respectively. b is the distinguished *partition begin node*, representing the beginning of the root process.

Nodes.

Each node in N is of one of the following types:

- *Process begin nodes*, representing the beginning of process instances. b is the process begin node for the single instance of the partition's root process.
- *Process end nodes*, representing process completion. Process end nodes are present only in the PCC sync graph.
- The *partition end node*, e.
- *Create nodes*, representing `create` statements.
- *Signaling nodes*, representing transaction calls. Since transaction calls may be made by reference to a process- or transaction-valued variable, signaling nodes represent either of these types of calls.
- *Accept nodes*, representing transaction entries.

The create, signaling, and accept node types represent `create` statements, transaction calls, and transaction entries respectively. Collectively, these are known as *transaction nodes* in the sync graph and *transaction statements* in the program.

Each process body is represented by a single subgraph delimited by its process begin node and the partition end node (or the process end node, in the case of PCC sync graphs). All instances of the process are represented by the process body subgraph.

Control edges.

A directed control edge (m, n) exists in E_C if:

- m is a transaction node or a process begin node, n is a transaction node, process end node, or the partition end node e, and there exists a control path from m to n which does not include any transaction statements; or,
- m is a `create P` statement and n is the process begin node for the subgraph representing process `P`.

To avoid adding an unnecessary edge type to the model, we use control edges to represent process instance creation, as well as ordinary control flow. For exactly those control edges $e = (m, n)$ which represent process instance creation, m is a create node and n is a process begin node. Control edges representing process creation can be easily distinguished from edges representing statement sequencing, by examining the types of nodes at the ends of the edges.

Signal types, sync variables, and sync edges.

Informally, sync edges are placed between each signaling node and all accept nodes it can reference. The references depend on which definitions reach the signaling node, and which signal types they represent.

We make the following definitions, to formally identify the edges in E_S:

- A *signal type* y is a pair (P_y, n_y), where P_y is a process and n_y is the name of a signal accepted by some accept node in P_y.
- The set $MayCall(y)$ is the set of signaling nodes which *may* call signals of type y.
- The set $MustCall(y)$ is, similarly, the set of signaling nodes which *must* call signals of type y. Although we do not use the *MustCall* sets in creating E_S, we define it here because of its similarity to *MayCall*.

  Note that $MayCall(y) \supseteq MustCall(y)$, but that the two sets are not necessarily identical in Concurrent C.
- The set $N_{Accept}(y)$ is the set of nodes that accept signals of type y, i.e., $\{n : (n \in P_y) \wedge (n = \texttt{accept } n_y)\}$.

- The set of *sync variables* is the set of variables whose values may be processes or transactions. This set can be identified from type declarations, assuming types are not cast.

An undirected sync edge $\{m, n\}$ exists in E_S if $m \in MayCall(y)$ and $n \in N_{Accept}(y)$.

Semantics of nodes and sync edges.

In the sync graph, each signaling or accept node may represent more than one process instance. Thus, the sync graph condenses all instances of a synchronization statement, in all instances of its containing process, into a single node.

Similarly, each sync edge $\{m, n\}$ represents a possible transaction between any single instance of m and any instance of n. This semantics is conservative with respect to that of program execution, in the sense that all synchronization states of the program may be generated from the sync graph. However, in some special cases, we may have information available which limits the set of instances of n with which m can transact. In deadlock analysis, we use information about these limitations to improve the accuracy of analysis.

Sync variables in the execution model.

In this section, we explain some structures based on the abstract memory A of the execution wave (E, A). These structures are used in the dynamic execution model, as described in in Section 9.2.3.

The *reference function* of an executing program is the set of processes and transactions to which the signaling node instances refer. In this section, we formally define the reference function, through a series of preliminary definitions.

A *sync variable instance* is a sync variable in a particular process instance. If sync variable v is defined in process P, then one instance of v exists for each instance of P. The set of sync variable instances created in the execution of the program is denoted *SyncVarInsts*.

At each step in the program's execution, each sync variable instance v_i refers to

exactly one process or transaction. However, the process or transaction instance to which v_i refers may change during program execution. During the modeled execution of the sync graph, we must thus retain information on what process or transaction that each sync variable references.

A *sync environment* is essentially a set of the values of sync variable instances, similar to the "environment" from denotational semantics. More formally, a sync environment is a mapping $M : SyncVarInsts \rightarrow (ProcInsts \times 2^{N_{Accept}}) \cup \{\perp\}$, from a sync variable instance v_i to a set of all instances of accept nodes to which a signaling node using v_i may refer.

$M(v_i) = (P_j, A)$ means that a signaling node referring to the variable v_i may refer to any instance of any node in the set A, within the process instance P_j. If v_i is a transaction variable instance referring to a signal type y, then A is the set of all instances of accept nodes of signal type y in P. If v_i is a process variable instance, then A is the set of all instances of *all* accept nodes in P. $M(v_i) = \perp$ means that v_i is uninitialized or invalid, and thus v_i does not refer to any currently executing instance of a process or transaction.

The *reference function* f_R is a function defining, for a particular step in program execution, the accept node instances to which each signaling node instance may refer. It is a mapping from a sync environment and a signaling node instance to the set of accept node instances to which the signaling node may refer in that environment. Formally, $f_R \in F_R : NodeInsts \rightarrow (ProcInsts \times 2^{N_{Accept}})$, where $NodeInsts$ is the set of node instances that may ever be created by the program, and $ProcInsts$ is likewise the set of process instances that may ever be created by the program.[2]

Suppose that node instance n_i references variable instance v_j, and that sync environment M holds at some time while v_j executes. Note that $f_R(n_i) = M(v_j)$. Thus, given a sync environment and a function g mapping node instances to the variable instances they reference, we can construct f_R as the function composition $M \circ g$.

---

[2]Since f_R effectively defines a set of accept node instances, we will also abuse the notation slightly, and use $NodeInsts \rightarrow NodeInsts$ as an alternative signature of f_R.

CC execution graphs of dynamic program executions.

Because CC sync graphs are a bounded, static representation of a potentially unbounded number of processes, it is somewhat misleading to use them to represent of program execution. We therefore introduce the concept of a *CC execution graph*, which models program execution behavior more intuitively. Figure 9.3 shows the execution graph corresponding to an execution of the program in Figure 9.2 in which the main loop executes three iterations.

In the CC execution graph, loops are unwound as many times as they are executed, and the number of processes equals the number of `create` statements in the unwound loops (plus one, for the `main` process.) Sync edges are drawn from signaling nodes to the accept nodes to which they refer.

To demonstrate the more intuitive nature of the CC execution graph, let M be the sync environment at node instance $\mathtt{s}:1$. The value of the reference function $f_R(M, \mathtt{s})$ is, then, $\{\mathtt{v}:1, \mathtt{x}:1\}$.

9.2.2 An alternate static graph representation.

We have shown one such possible static graph representation in Section 9.2.1. As we show in this section, this representation can sometimes lose information that might be of value in proving that the program cannot deadlock. Therefore, we propose an alternative representation that may preserve at least some of this lost information. The penalty for preserving this information is, of course, that the worst-case bounds on space for the sync graph are increased.

A problem with the CC sync graph.

In the CC sync graphs of Section 9.2.1, the synchronization behavior of every instance of each process P is summarized by a single representative subgraph G_P. The instances of process P are summarized by G_P under the assumption that any process instance represented by G_P may be created at any instance of any `create` node that creates an instance of P. While the sync graph for the program has a number of nodes linear in the

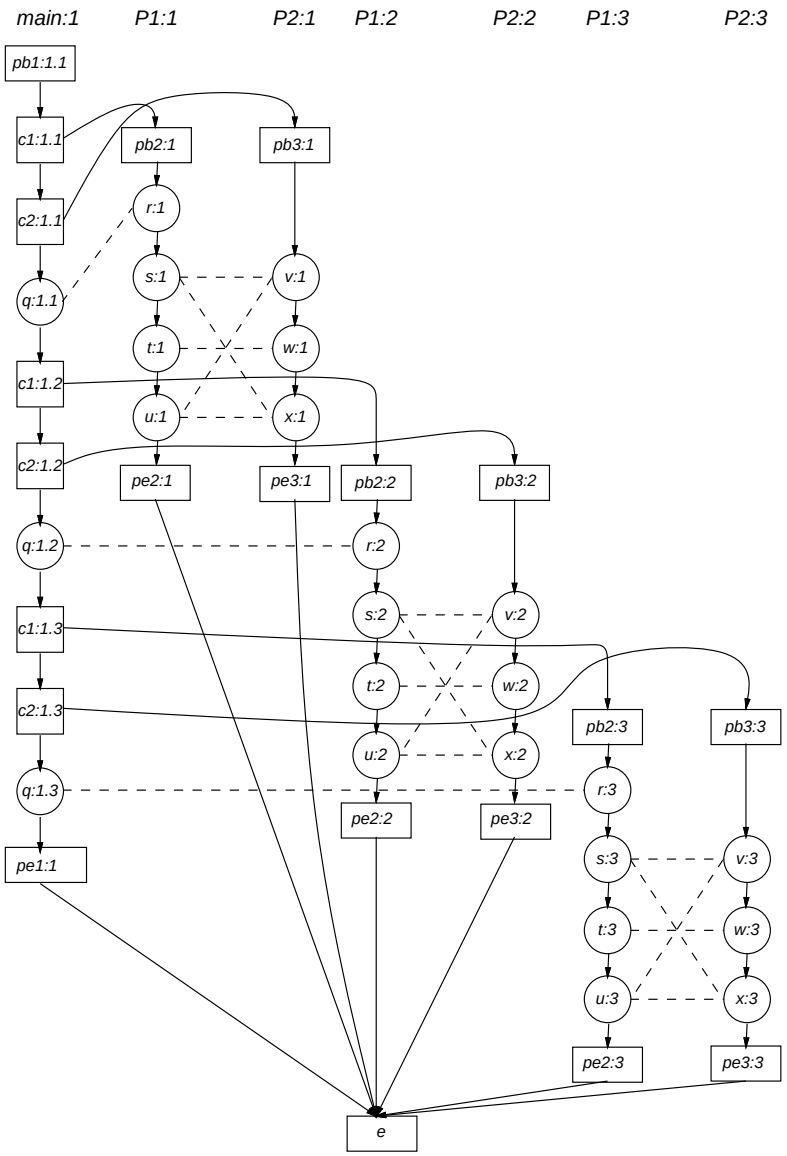

Figure 9.3: A CC execution graph for an execution of the Concurrent C program of Section 9.1.1.

number of synchronization statements, some information will be lost in the summary under such assumptions.

As an example see Figure 9.4. Suppose that we are attempting to use the "must happen before" relation to deduce which pairs of nodes have no instances that can ever happen together. Let P and Q be processes in a program, and let m and n be nodes in P and Q respectively. It is possible that, for those instances P_i of P which are created before any instance of Q is created, that all instances of m in P_i finish execution before any instance of n starts. For example, in those instances of P created by cP, node m of each newly created instance must complete a transaction with m' in `main`.

Suppose we also know that all instances of P are created either before any instance of Q is created, or after all instances of n have finished. (Again, in Figure 9.4, each instance of n must transact with an instance of n' before cP' can be reached in control flow.) Given this, we know that $m \in CHT(n)$.

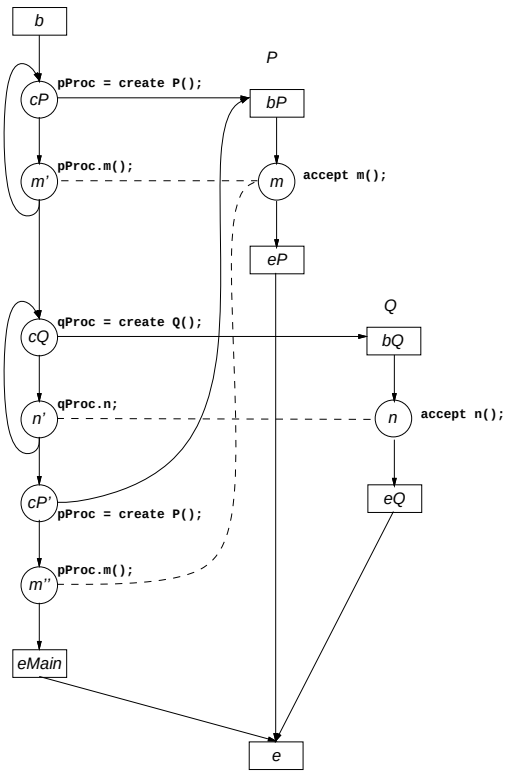

Figure 9.4: Example of a case in which collapsing all instances of a single process into a single subgraph results in loss of information.

However, we also know that, for each instance Q_j of Q, there is some instance P_i of P which is initiated *before* Q_j, and some instance $P_{i'}$ of P which is initiated *after* Q_j. If we summarize all instances of P with a single subgraph, then we must also summarize the instances of Q as being unordered with respect to the instances of P. Thus, we would lose the knowledge that $m \in CHT(n)$.

An alternative graph representation.

The CC sync graph of Section 9.2.1 collapses all instances of a process into a single subgraph. One alternative is to partition the instances of processes by enumerating the chains of `create` statements that initiated them. It might then be possible to retain the kind of information that was lost in the example presented earlier in this section. Here, we show an alternative sync graph representation which, instead, represents, as a single subgraph, all process instances created at the end of a particular *chain* of `create` statements. This expansion of the representation of Section 9.2.1 prevents some of the loss of some non-concurrency information, at the price of increased size for the sync graph. (Alternatively, we could keep the graph structure of Section 9.2.1 and enrich the meaning of CHT to include call chain information.)

A *create set* is a set of instances of processes which has been created by instances of nodes in a single, statically defined chain of `create` statements. Create sets are defined recursively as follows:

- All process instances which are initiated by instances of a single `create` statement in the main procedure (or inlined subroutines), are in the same create set.
- All process instances which are initiated by instances of a single `create` statement in the same create set, are in the same create set.

Figure 9.5 shows an example of the create sets of a program.

Create sets allow us to collapse the (unbounded) number of processes in a program to a set of representative processes whose size is polynomially related to the size of the source program, assuming that the lengths of `create` call chains are bounded by some constant. Lemma 139, unfortunately, shows that the create set representation violates

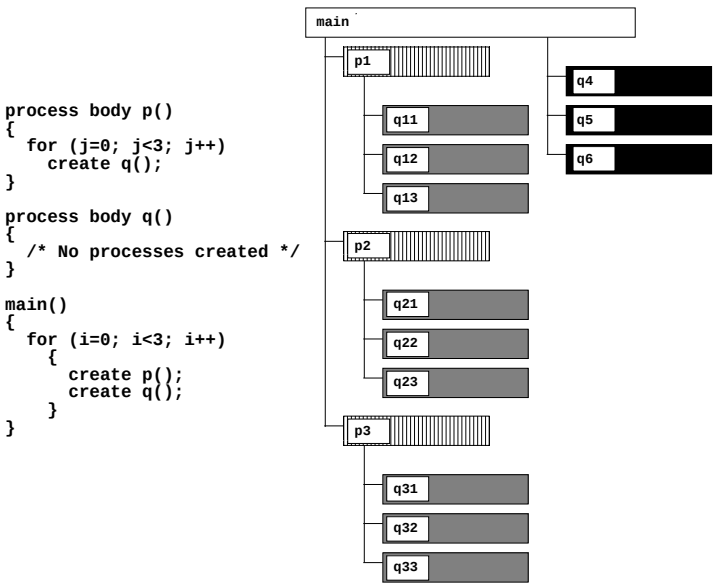

Figure 9.5: Create sets. All processes with the same coloring are in the same create set.

the polynomial size restriction on sync graphs.

Lemma 139 *Let l be the (finite) maximum length of any* `create` *call chain in a program. Let k be the maximum number of* `create` *statements in any process, and let n be the number of nodes in a sync graph representing an instance of any process. Then the maximum number of nodes in a sync graph where each create set for a process is represented by the sync graph of that process is $\mathcal{O}(nk^l)$.*

Proof: Define the *create level* of a process instance as the level of the instance in a tree of process creation. The maximum size of the sync graph for any process at level l (the bottom-most level) is n. Call this quantity s_l. We form a geometric series:

$$\begin{aligned} s_l &= n \\ s_{i-1} &\leq n + ks_i \\ s_1 &\leq n\textstyle\sum_{i=0}^{l-1} k^i \\ &= \mathcal{O}(nk^l). \end{aligned}$$

Note that a program where $s_1 = \mathcal{O}(nk^l)$ can be realized. □

If we represent each create set individually, we thus run the risk of poor behavior

resulting from over-large sync graphs. However, create chains may have finite length limits in practice, as loop nest depth in Fortran apparently does [Knu71]. We need experimental data to determine whether the create set representation will help to disprove deadlock in enough instances to make its size penalty worthwhile. For the present, we use the simpler representation of Section 9.2.1 in which every instance of a process P is represented by the single subgraph of P.

9.2.3 Execution wave.

The CC sync graph of Section 9.2.1 is an abstraction of the synchronization structure of a Concurrent C program. Having defined this abstraction, we now define an abstract execution model which operates on the CC sync graph. The abstract execution model generates a set of synchronization states which contains at least the possible synchronization states of the original program.

We include the settings of sync variables in the execution model, for greater fidelity to the original program's execution than we could obtain if we excluded them. We assume that the sync variables may be updated in some arbitrary way between transactions. Since we have assumed in Section 9.1.1 that the sync variables are not aliased, the updates of the sync variables are independent of each other.

The *CC execution wave* is a pair (E, M), where E is a vector of sync graph node instances of length at least 1, and M is a sync environment, as described in Section 9.2.1. Unlike the models for Ada and binary semaphores, the length of the execution wave may vary during execution; the wave may expand as process instances are created, but its size is nondecreasing during execution (since process end nodes remain on the wave when their process instances complete.)

We say that the CC execution wave is *normalized* if it contains only transaction nodes, and process and partition end nodes. If the wave is normalized, then the next action that advances the wave must be a transaction. Only normalized waves may contain deadlocks. Deadlock analysis, therefore, operates under the assumption that the wave is normalized at the time of a deadlock.

The program's execution is modeled by creating a set of possible normalized execution waves from an initial wave, using the following process:

- E initially has length 1. $E[1]$ is the single instance b_1 of the partition begin node b. Node b initially has state READY; all nodes other than b initially have state OFF-WAVE. All sync variables are initially uninitialized, so $M(v_i) = \perp$ for all $v_i \in SyncVarInsts$.

  Note that the initial E is thus not normalized.

- On all execution waves, any accept and signaling node instances are in state READY if they are connected by a sync edge to another node on the wave, and if the signaling node instance refers to the accept node instance. Otherwise they are in state WAITING. (Process end nodes are always in state WAITING.)

  More formally, if $\{m_i, n_j\} \subseteq E$ and $n_j \in f_R(m, M)$, then both m_i and n_j are in state READY. (From the definition of the sync graph, if $n_j \in f_R(m, M)$, then m_i and n_j are connected by a sync edge.)

- To transform a normalized execution wave (E, M), $|E| \geq 1$, to another normalized wave (E', M'), select a pair of process instances (P_1, P_2) such that $E[P_1] \in f_R(E[P_2], M)$. $|E'| = |E|$, and $E'[i] = E[i]$ for $1 \leq i \leq |E|, i \neq P_1, i \neq P_2$. $E'[P_1]$ is a control successor of $E[P_1]$; $E'[P_2]$ is a control successor of $E[P_2]$. The states of $E[P_1]$ and $E[P_2]$ are updated to OFF-WAVE (unless they are selected as their own successors). Then perform the following normalization step.

  To transform the initial execution wave (E, M), $|E| = 1$, to another normalized wave (E', M'), replace the partition begin node instance b_1 with one of its successors nondeterministically, update the state of b_1 to OFF-WAVE, and execute the following normalization step.

  Normalization step. While E' contains any create nodes, do the following:

  - Select a process t such that $E'[t]$ is a create node for some process P. $|E''| = |E'| + 1$, and $E''[i] = E'[i]$ for $1 \leq i \leq |E'|, i \neq t$. $E''[t]$ is a control successor of $E'[t]$, other than the process begin node of process P. $E''[|E'| + 1]$ is a

successor of the process begin node of P. The state of $E'[t]$ is updated to OFF-WAVE (unless $E'[t] = E''[t]$), as is that of the process begin node of P.

– Set E' to E''.

M' is updated from M to correspond to one of the possible environments of process and transaction variables for the nodes of E'. We do not give an explicit model of how to obtain M' from M, since the update would require retaining information which we do not wish the execution model to represent (i.e., assignment statements, function calls, and control branch dependences.)

- Updating continues until all nodes on the execution wave are in state WAITING.

The set of execution waves generated by this process contains at least the waves which would be generated by any execution of the program. Thus, if the original program may deadlock, then the execution model of the program will produce a deadlocked execution wave.

9.3 Constraints on deadlock cycles in the Concurrent C model.

If a Concurrent C program deadlocks, its CC sync graph must contain a deadlock cycle. In this section, we identify the constraints on valid deadlock cycles in the CC sync graph.

It is important to note that the nodes in the CC sync graph may represent arbitrarily many node instances, which in turn may be contained in arbitrarily many process instances. Therefore, a valid deadlock cycle may enter the same process subgraph in the CC sync graph several times.

We first define the following notation:

- $Process(n)$ is the process containing node (or node instance) n.
- $ProcInst(n_i)$ is the process instance containing instance i of node n.
- $ProcRef(s_i)$ is the process instance referred to by instance i of signaling node s.

Let D be the set of nodes in the CC sync graph which represent the head nodes of the deadlock. The following constraints hold on D:

1. Each node in D is a signaling or accept node, and all nodes in D are transitively coupled to all others in D. Thus, there is a cycle in the CC sync graph, which includes the nodes of D, and such that, for every node $d \in D$ in process P:

   (a) The path enters node d through a sync edge.

   (b) The path traverses at least one control flow edge in process P.

   (c) The path leaves P via a sync edge.

   (d) If P has only a single instance, then the path does not re-enter P.

   (e) For each sync edge $\{s, a\}$ in the deadlock cycle, such that s is the signaling node of the edge and a is the accept node, some instance of s must be able to refer to some instance of a.

   More generally, all instance references associated with sync edges must be simultaneously consistent. For example, suppose we find a deadlock cycle which includes sync edges $\{s, a'\}$, $\{s', a''\}$, and $\{s'', a\}$, where $s \in Process(a)$, $s' \in Process(a')$, and $s'' \in Process(a'')$. Then it must be possible to satisfy the following set of simultaneous equations for some instances of the end-points of the sync edges in the cycle:

$$\begin{aligned} ProcRef(s_i) &= ProcInst(s'_j) = ProcInst(a'_j) \\ ProcRef(s'_j) &= ProcInst(s''_k) = ProcInst(a''_k) \\ ProcRef(s''_k) &= ProcInst(s_i) = ProcInst(a_i) \end{aligned}$$

   This is a new constraint, which is made necessary by the fact that different instances of the transaction calls may refer to different instances of transaction entries.

2. If any two nodes $s \in N_{Signal}$ (the set of signaling nodes in the sync graph) and $a \in N_{Accept}$ in D are connected by a sync edge, there must exist some execution wave, containing instances of the nodes in D, such that no instance of s in the wave refers to any instance of a.

Formally, for all pairs of nodes $s, a \in D$, such that $\{s, a\}$ is a sync edge,

$$\exists i, j, E : \{s_i, a_j\} \subseteq E \wedge ProcRef(s_i) \neq ProcInst(a_j),$$

where E is a feasible execution wave. If s_i always refers to a_j, then any instance of s can always transact with the instance of a on the wave.

This constraint may be extended to *subsets* of signaling nodes in D.

The difficulty in using this constraint comes in determining when node instances can ever refer to each other. As a first approximation, we limit ourselves to the case in which the accepting process has only a single instance. Thus, the signaling process instance *must* refer to the accepting process instance.

3. Instances of all nodes of D may execute at the same time. This implies:

   (a) The nodes in D must represent statements whose instances can happen together.

   (b) All nodes in D are *co-executable.*

4. There must exist some feasible execution wave containing instances of the nodes in D, and such that no node instance outside the nodes represented by D transacts with any other node instance on the wave.

9.3.1 The use of constraint 3 in pruning deadlock cycles.

Processes in Concurrent C may have multiple instances. Thus, the "can't happen together" (CHT) relation between nodes has properties which would not be intuitive in the single-instance model. In this section, we examine these properties, and how they may be used in pruning spurious deadlock cycles.

Consider the program of Figure 9.1, and its CC sync graph (Figure 9.2). If we can guarantee that the loop in `main` iterates only once, then no process has more than one instance. In this limited circumstance, the techniques developed for Ada rendezvous may be directly applied to prove that the program cannot deadlock.

If the loop iterates more than once, however, then there would be multiple instances of processes P_1 and P_2. Under such circumstances, the methods developed for the Ada rendezvous may not yield correct results.

An example of this can be seen in the *CHT* relation between nodes. In Figure 9.1, nodes r and s are members of process P_1. If only a single instance of P_1 is created, then no instance of r could execute concurrently with any instance of s. However, if multiple instances of P_1 are created, then some instance r_i of r may execute concurrently with an instance s_j of s, where r_i and s_j are in different instances of P_1. So, technically, $s \notin CHT(r)$.

Because of this effect, we must develop special techniques for handling multiple process instances. One of these is *partitioning*, discussed in Section 9.3.2. We lay the groundwork for partitioning in this section.

Reaching definitions and *CHT*.

If we were to attempt to analyze the synchronization structure of the entire program all at once, the *CHT* relation would often be ineffective in proving freedom from deadlock, as we have illustrated. However, we can sometimes partition the sync graph so that the *CHT* relation yields more information.

Returning to Figures 9.1 and 9.2, each instance of P_1 transacts with only one instance of P_2, and vice versa. Also, in the `main` routine, only one definition each of `p1Proc` and `p2Proc` ever reaches any use, and the definition reaching the use cannot be loop-carried. Thus, any instance of q must refer to any instance of r with which it can happen together; no instances of q and r that can happen together are head node instances in a deadlock cycle.

The reasoning that allows us to draw these conclusions stems directly from analysis of reaching definitions of sync variables. For this particular example, we have:

- Only one definition of `p2Proc` reaches any use in any instance of P_1. Therefore, both of the transaction calls `p2proc.p2t` in P_1 must refer to the same process. Any deadlock containing s (or u) and v (or x) as head nodes is spurious under

Constraint 2, because the two head nodes could transact.

- Only one definition of `p1Trans` reaches any use in any instance of P_2. This definition is always set to the `p1t` entry of some P_1 process instance. By analyzing the parameter passing in `main`, we can determine that, for any instance of P_2, the P_1 process instance to which the `p1Trans` definition refers must be the same as the P_1 process instance which calls the `p2t` entries. We therefore know that any cycle which includes instances of t and w as head nodes would be spurious. (In this example, no such cycle is formed, but one might be if loops existed in P_1 and P_2.)
- Only one definition of `p1Proc` is ever used inside any iteration of the loop in *main*. This definition changes on each iteration. Therefore, two successive iterations cannot transact with the same instance of P_1.

Because each instance of P_1 can transact with only one instance of P_2 (and vice versa), we can determine that, in instances of P_1 and P_2 which may transact with each other, no instances of t and v may happen together. Thus, the cycle (t, u, v, w, t) is spurious, since it must include such a sync edge.

9.3.2 Partitioning.

In Figure 9.2, each iteration of the loop in `main`, and one instance each of the processes generated by the main loop, can be analyzed in isolation, since no deadlock cycle can be formed through two instances of any process or through two iterations of the loop. Furthermore, the analysis can be conducted as if the loop in `main`, and the created processes, had one instance each. Such a partitioning is quite useful, since it will generally result in more accurate *CHT* analysis.

Our objective in this section is to define the preconditions under which the CC sync graph can be partitioned into subgraphs which can be analyzed in isolation. We are particularly interested in partitioning the CC sync graph so that some processes created in the subgraph may be analyzed as if they had only a single instance.

Lemma 140 shows the circumstances under which a process has only a single instance, by way of a recursive, top-down definition.

Lemma 140

- *The process* `main` *has a single instance.*
- *If there exists a process P whose instances are created by a single* `create` *statement s, such that s is contained in a single-instance process and there is no control cycle containing s, then P has a single instance.*

Proof: Since `main` is the process representing the main program, it has only one instance.

If process P has only a single `create` statement s, then all instances of P are created by s. If s is not contained in a control cycle, and is in a single-instance process, then s has only a single instance. Thus, P is created by a single statement with a single instance, and P therefore has only a single instance. □

Lemma 141 shows the conditions under which we can analyze a subgraph of the sync graph in isolation. Note that the lemma does not necessarily allow us to treat any created processes in the partition as if they had single instances.

Lemma 141 *If there exists some induced connected control subgraph of the CC sync graph, which corresponds to a set of program statements S within a single-instance process such that:*

- *No statement in S and its created processes uses any sync variable definition from outside of S; and,*
- *No statement outside S and its created processes uses any sync variable defined in S; and,*
- *No statement outside S and its created processes can transact with statements inside S;*

then the CC sync graph consisting of the statements in S and the processes created in S (if any) can be analyzed in isolation to determine if S can deadlock.

Proof: If no statement in S creates process instances, then, under the assumed restrictions, S cannot contain any transactions that other processes can reference. If some statement in S does create process instances, then these process instances (and any subsidiary process instances they create) cannot refer to, or be referred to by, any process instances created outside of S. Thus, the process instances created inside S cannot transact with any process instances outside of S, and no deadlock cycle satisfying constraint 1 can include head nodes in S and tail nodes outside of S, or vice versa. Therefore, the CC sync graph consisting of S and its created processes can be safely analyzed for deadlock in isolation. □

Lemma 142 adds some power to Lemma 141, allowing us to treat parts of multiple-instance processes as if they had only a single instance.

Lemma 142 *If there exists some induced connected control subgraph of a single process in the CC sync graph, which corresponds to a set of program statements S such that, for each instance S_i of S:*

- *No statement instance in S_i uses any sync variable definition from outside of S_i and its created process instances; and,*
- *No statement instance outside S_i uses any sync variable defined in S_i; and,*
- *No statement instance outside S_i and its created process instances can transact with statements in S_i;*

then the CC sync graph consisting of S and its created processes (if any) can be analyzed in isolation to determine if S can deadlock. Furthermore, S may be treated as a single-instance process in the analysis.

Proof: Any instance S_i of S in the execution graph of the program forms a set of program statements, whose CC sync graph meets the definition of S in Lemma 141. □

Note that sync variable definitions may propagate into S_i only through control edges; other sync variables are defined locally via process instance creations or transactions within S_i. (Recall that we consider transaction parameter passing to define the local parameters of the transaction.) Any set of statements S which meets the preconditions

of Lemma 142 is said to induce a *partition* on the CC sync graph. The partition consists of two elements: one being S and the process subgraphs for which S transitively creates process instances, and the other being the rest of the sync graph. (By abuse of notation, we refer to the first element as the partition.) If S induces a partition, then we may analyze the partition for deadlock in isolation from the rest of the CC sync graph. Furthermore, Lemma 140 may allow us to treat some processes within the partition as if they had only a single instance, thus producing more accurate CHT information, and, therefore, fewer false alarms in deadlock detection.

If the partition induced by S does not show a deadlock when analyzed in isolation, then we may excise S from its containing process and analyze the containing process as part of another partition. Obviously, the `main` process induces a partition, so the partitioning procedure generalizes to the entire program.

We are particularly interested in cases where S is the innermost loop body which meets the specification of Lemma 142, and where instances of some process P are created in S, but not in any nested loop body or in any other process transitively created in S. If we can identify such a loop body, we can analyze P as if it has only a single instance. This helps us to initialize CHT information for P, since no instance of any node in P may happen together with any instance of any other node of P, within the partition induced by S. Partitioning may help speed the deadlock analysis by reducing the number of nodes that have to be analyzed at any time (though, as Section 9.6.1 shows, there are some cases where partitioning can slow the analysis).

An algorithm for partitioning.

In this section, we present an algorithm for partitioning the CC sync graph. We assume that we have created a *process creation multigraph*, analogous to a conventional call graph, but whose nodes represent processes and whose edges represent process creation. Because of our restriction in Section 9.1.1 that processes do not create themselves directly or transitively, the process creation multigraph is a DAG, with `main` as the root node.

Algorithm 12 *Identify partitions.*

Input:

1. The PCC sync graph $G = (N, E_C, E_S)$.
2. The process creation multigraph $M = (N_{PCM}, E_{PCM})$. N_{PCM} is the set of processes; the edge (m, n) is in E_{PCM} if process m contains a `create n` statement. The edges of E_{PCM} are annotated with the create and process begin nodes in N corresponding to the process creations.
3. The set R of reaching definitions information $\{(Def_v, Use_v, C)\}$ for all sync variables v.

   All definitions and uses of sync variables occur at, or between, transaction statements; as such, they may be attached to nodes or control edges in the sync graph. Def_v and Use_v are thus nodes or control edges. C is a boolean flag that is true if Def_v and Use_v reside in the same loop body, and if in addition the dependence may be loop carried. We include parameter passing in procedures, transactions, and process creations as definitions and uses of the formal or actual variables.

Output: A list L of loop bodies and process bodies which induce partitions on G, presented in reverse topsort order on M. (This ordering is useful in later analysis.)

Procedure:

1. Initialize L to the empty list.
2. For each edge $e = (m, n) \in E_{PCM}$, in reverse topsort order on n in multigraph M, do:

   Identify the loop or process body where instances of process n were created, and see if the body induces a partition.

   (a) Let B be the loop or process body containing the create node that corresponds to the tail node of e.

   (b) Let D be the set of definitions of sync variables inside B and its contained loops.

Note that D includes all process variables that can be used to access the process instances created inside B.

(c) Let U be the set of uses of sync variables inside B and its contained loops.

Likewise, note that U includes all process variables that can be used by B and its created processes to transact with nodes in $Process(B) - B$ and its creating processes.

(d) Let $RejectB = False$.

$RejectB$ is a flag which is $True$ if B does not induce a partition. We check each of the conditions for inducing a partition in turn.

(e) If there is any definition $d \in D$ such that d reaches a use in $Process(B) - B$, then let $RejectB = True$.

(f) If there is any use $u \in U$ such that a definition in $Process(B) - B$ reaches U, then let $RejectB = True$.

(g) If B is contained in a loop, and the loop containing B carries any sync variable dependences, then let $RejectB = True$.

(h) If B contains any accept statements, then for each accept statement a in B do:

Verify that the only transaction calls to each such accept statement a must come from within the process instances created in B.

i. If there are any definitions of transaction variables for a outside of B, then let $RejectB = True$.

ii. If there are any definitions outside of B of process variables for $Process(B)$, which reach uses at transaction calls for a, then let $RejectB = True$.

This rejects calls to a which are made from outside the process instances created by B.

(i) If $RejectB = False$, then add B to the end of L.

3. Return L as a list of loop bodies which induce partitions on G.

Lemma 143 (Correctness of Algorithm 12.) *Each loop body in the list L output by Algorithm 12 is valid as the set of statements S of Lemma 142.*

Proof: The tests of Algorithm 12 correspond to the preconditions of Lemma 142. □

Lemma 144 (Execution time of Algorithm 12.) *An upper bound on the total time for Algorithm 12 is $\mathcal{O}((|Loops| + |Procs|)|N|^4 \log(|N|)|SyncVars|)$, where Loops is the set of loops, Procs is the set of processes, and SyncVars is the set of sync variables in the program.*

Proof: We can preprocess the set R of def-use pairs to organize it as a pair of height-balanced trees, one tree keyed with the definitions in each pair and the other with the uses.

The size of R may be $\mathcal{O}(|N|^4|SyncVars|)$, on the conservative assumption that every sync variable may be defined and used on every control edge, that all definitions reach all uses, and that the control subgraph is complete. Thus the preprocessing will require $\mathcal{O}(|N|^4 \log(|N|)|SyncVars|)$ time. After doing this, the set of uses for a given definition, or the set of definitions for a given use, may be found in $\mathcal{O}(\log(|N|))$ time.

We may likewise preprocess each process and each loop body to find its sets D and U, and its set of nodes. We need only examine each node once to do this preprocessing. For each node n, we find def-use chains with endpoints at n by doing a lookup in the D and U trees, and union these sets into that of the process or loop body containing n. If sets of defs and uses are represented as bit vectors, their union requires $\mathcal{O}(|N|^2|SyncVars|)$ time. Thus, the total time for this preprocessing step is $\mathcal{O}(|N|^3 \log(|N|)|SyncVars|)$.

Following this, we then only have to examine each loop and each process to determine whether it meets the preconditions of Lemma 142. We look up the defs D and uses U of sync variables in each loop or process body B, in $\mathcal{O}(\log(|N|))$ time. For each loop or process, we union, over the possibly $\mathcal{O}(|N|^2)$ defs in D, the uses corresponding to each def, in a total time of $\mathcal{O}(|N|^4 \log(|N|)|SyncVars|)$. We also union, for each use in D, the defs corresponding to that use; this takes the same time.

Lastly, if B has any accept nodes, we must determine whether any transaction variables for those nodes are defined outside of B. Assuming that B has $\mathcal{O}(|N|)$ accept

nodes, each with a different transaction type, we must examine $\mathcal{O}(|N|^2)$ nodes and control edges to determine whether each of these transactions are defined outside D. Doing so requires $\mathcal{O}(|N|^3|SyncVars|)$ time. Similarly, there may be $\mathcal{O}(|N|^2)$ places at which each process variable for $Process(D)$ could be defined, and $\mathcal{O}(|N|)$ places where it might be used to call the accept nodes. Examining all these places for def-use chains for each process variable thus also takes $\mathcal{O}(|N|^3|SyncVars|)$ time. The total time per loop or process is thus $\mathcal{O}(|N|^4 \log(|N|)|SyncVars|)$.

Thus, an upper bound on the total time for Algorithm 12 is

$$\mathcal{O}((|Loops| + |Procs|)|N|^4 \log(|N|)|SyncVars|).\square$$

9.4 Computing *CHT* sets in Concurrent C.

The semantics of the CHT relation in Concurrent C is as follows: $m \in CHT(n)$ means that no instance of m in the partition under analysis may execute together with any instance of n in the same partition. Except for the fact that CHT is computed relative to the current partition, this meaning of CHT is identical to that for binary semaphores and Ada.

In this section, we discuss the initialization and refinement of the CHT relation in the Concurrent C model. Because processes may have multiple instances, the initialization is quite different from that in Ada. We use a subset of the refinements for Ada: pinning analysis, critical sections, and *B4*. (Remote procedure calls are omitted in the initial model; this refinement is thus not included here.) The refinements have been adapted to compensate for differences between the Ada and Concurrent C models.

9.4.1 Initializing *CHT*.

In the CC sync graphs, a single node may represent not only statement instances in the same process, but also statement instances in many different process instances. Thus, it is possible for more than one node in the same process to be on the execution wave at any given time. For this reason, we cannot initialize $CHT(n)$ to $Process(n) - \{n\}$,

as we did in the Ada model, unless n is contained in a single-instance process.

In computing *CHT* sets, we make use of single-instance processes, which the partitioning algorithm tries to produce. We also define a new *CHTSelf* relation, akin to *CHT*, which is useful for nodes in multiple-instance processes.

9.4.2 The *CHTSelf* relation.

The conventional *CHT* relation provides information on pairs of nodes which have no instances that ever happen together. It is also useful to retain information on nodes n such that no two instances of n (in different process instances) ever happen together. This may occur for a variety of reasons; one such example occurs when $Process(n)$ has only one instance. If n is such a node, then no valid deadlock cycle may include more than one instance of n as a head node. Such information is also of particular use in pinning analysis (Section 9.4.3) and critical section analysis (Section 9.4.4).

We capture this semantics in a *CHTSelf relation*: $m \in CHTSelf(n)$ if there are no instances m_i, n_j (of nodes m and n, respectively) such that m_i executes simultaneously with n_j and $m_i \neq n_j$.

Properties of *CHTSelf*.

For the "perfect" *CHTSelf* relation $CHTSelf_{Perf}$, and for all nodes n that can execute at all, $CHTSelf_{Perf}(n) - \{n\} = CHT_{perf}(n)$. Whereas *CHT* is antireflexive (if all code is executable), *CHTSelf* is in general neither reflexive nor antireflexive. Both *CHT* and *CHTSelf* are symmetric. If P is a single-instance process, then for all $m, n \in P$, $m \in CHTSelf(n)$. Critical section analysis can generate and propagate *CHTSelf* information, as we see in Section 9.4.4.

The Widening Theorem (Theorem 1 in Section 2.3.1) states that, if there exists a set of nodes S, such that some instance of some member of S must always execute together with any instance of node n, then the nodes in $\bigcap_{n' \in S} CHT(n')$ cannot happen together with n. The *CHTSelf* relation cannot be substituted directly for *CHT* in Theorem 1, as Lemma 145 below shows.

Lemma 145 *If we have a set S of nodes such that some instance of a node in S must always be executing together with n, and $CHTSelf(n) \subseteq CHTSelf_{Perf}(n)$, then it is possible that $\bigcap_{n' \in S} CHTSelf(n') \not\subseteq CHTSelf_{Perf}(n)$.*

Proof: $\bigcap_{n' \in S} CHTSelf(n')$ is the set of nodes m such that, if instance m_i of m executes together with instance n'_j of n', then $m_i = n'_j$. Thus, $\bigcap_{n' \in S} CHTSelf(n')$ may include nodes in S, the set of nodes which must occur together with n. An example of this would occur if n pinned a single partner p in a single-instance process; then $S = \{p\}$, and $\cap_{n' \in S} CHTSelf(n') = CHTSelf(p)$, which may include p. □

The *CHTSelf* relation can be widened, in a manner similar to that of Theorem 1. Unfortunately, the result of doing so will give no more new information than a *CHT* set, formed by subtracting the identity relation I from *CHTSelf*, would.

Theorem 11 (Widening the *CHTSelf* relation.) *If we have a set S of nodes such that some instance of a node in S must always be executing together with n, and*

$$\forall n' \in N : CHTSelf(n') \subseteq CHTSelf_{Perf}(n'),$$

then

$$\bigcap_{n' \in S} (CHTSelf(n') - \{n'\}) \subseteq CHT_{perf}(n).$$

Proof: From Theorem 1, and the observation that

$$CHTSelf_{Perf}(n) - \{n\} = CHT_{perf}(n) \square$$

Lemma 146 below shows us that *CHTSelf* is, likewise, not pseudotransitive. Note that the Boolean array notation used in this chapter is described in Appendix A.

Lemma 146 *CHTSelf is not pseudotransitive.*[3] *That is, there may exist an n such that, assuming that $N - CHTSelf(n) - \{n\} \neq \emptyset$, $CHTSelf(n) \not\supseteq \bigcap_{m \in N - CHTSelf(n) - \{n\}} CHTSelf(m)$.*

[3]See Corollary 2 in Section 2.3.1 for comparison.

Proof: Let $M = N - CHTSelf(n) - \{n\}$. M contains the set of all nodes $m \neq n$ which may happen together with n; by assumption, M is nonempty. At any time n is executing, only the nodes of M may execute together with n; n may also execute alone if n is in the main process, and no other process instances have yet been created.

In Figure 9.6, nodes m and n are in single-instance processes, and can only transact with each other. Thus, $m \in M$, since m *must* happen together with n, and vice versa. Only the two single-instance processes exist, so for all nodes $n' \neq n$ (and including m), $m \in CHTSelf(n')$; thus, $M = \{m\}$. Therefore, $m \in \bigcap_{m \in M} CHTSelf(m)$, despite the fact that m can clearly happen together with n. □

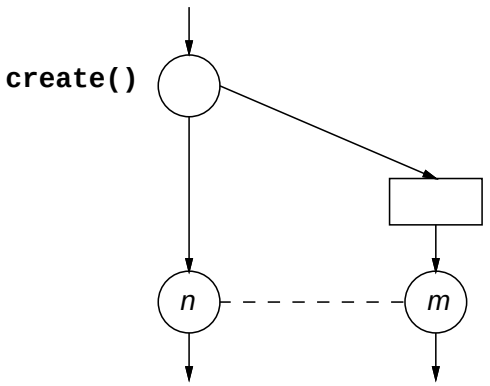

Figure 9.6: Non-pseudotransitivity of the *CHTSelf* relation.

The difficulty in widening the *CHTSelf* relation (without effectively reducing it to a *CHT* relation) makes *CHTSelf* generally no more powerful than *CHT* in non-concurrency analysis. It is possible that other variants on *CHT* may be amenable to widening, and yield useful information in tasking models with multiple process instances, such as Concurrent C. One possibility would be to somehow take reference variables into account, i.e., the relation "a cannot happen together with b unless a refers to b." Such possibilities deserve further investigation, directed by experimental evidence identifying any forms of *CHT* information which would be useful in pruning deadlock cycles.

9.4.3 Pinning analysis.

Pinning analysis in Concurrent C is closely related to that for the Ada model, but changes have been made to incorporate dynamic process creation. To develop pinning analysis for Concurrent C, we first consider the conditions for pinning for a particular node instance in a particular execution of the partition. We then generalize this to all instances of the node in all possible executions of the partition. We then develop the formalisms for pinning analysis, and show how non-head nodes that violate constraints 2 or 4 may be found as a low-cost byproduct of pinning analysis. Finally, we describe an algorithm for pinning analysis, and present proofs of its correctness and worst-case execution time.

Dynamic conditions for pinning.

This section discusses pinning in a particular program execution; we thus refer to CC execution graphs in this section. Refer to Section 9.2.1 for a discussion of CC execution graphs.

Refer to Figure 9.7. Let n_i be an instance of a transaction node; thus, n_i may be a member of a normalized execution wave. The control predecessor of n_i is node instance m_j. We define the notation $CPred(n_i)$ to represent the unique node instance which is a control predecessor of n_i.

The *transactor instance* of m_j, denoted $TransInst(m_j)$, is a node instance p_k which is known to complete at the same time that m_j does. $TransInst(m_j)$ is formally defined as follows:

- If m_j is an instance of a transaction node, then p_k is the node instance which transacted with m_j.
- If m_j is an instance of a create node, then p_k is the instance of the process begin node of the process instance that was created by m_j.
- If m_j is an instance of a process begin node, then p_k is the instance of the create node that created $ProcInst(m_j)$.

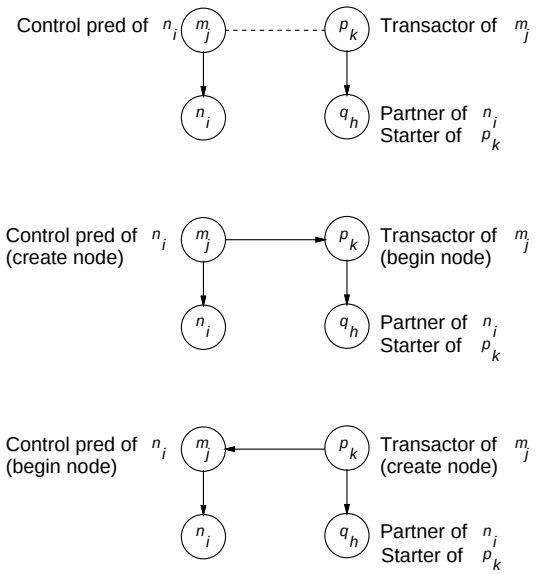

Figure 9.7: Transactor and partner instances of nodes. The three forms of transactor instances are shown.

- If m_j is the single instance of the partition begin node b, then m_j has no transactors.

A *starter instance* of p_k, denoted $StartInst(p_k)$, is a transaction node or process end node instance q_h, which is a control descendant of p_k. We require that the control path $(p_k, ..., q_h)$ contain no transaction node instances. However, q_h may be contained in a descendant process instance of $ProcInst(p_k)$; p_k may thus have more than one starter instance. Following the completion of p_k, execution cannot proceed past q_h without some synchronization with other process, i.e., either a transaction or the end of the program. q_h may be a member of a normalized execution wave.

A *partner instance* of n_i is a node instance q_h which starts on a normalized execution wave when n_i does. The set of partner instances of n_i, denoted $PartInst(n_i)$, is $StartInst(TransInst(CPred(n_i)))$. Note that n_i may have more than one partner instance, due to Concurrent C's dynamic process initiation; this is not the case in Ada.

If $q_h \in PartInst(n_i)$ is an instance of a process end node, or can only transact with n_i or with node instances that don't happen together with n_i, then n_i pins q_h to the normalized wave. q_h must execute on all normalized waves containing n_i.

Static conditions for pinning.

In this section, we use the results obtained earlier in this section to formulate a static analysis of pinning in Concurrent C programs. We assume that all paths to any instance of a node n are possible, i.e., that any of n's control predecessors may have executed before n is reached on the execution wave. Further, n's partners may include the starters of any node that may have transacted with, or been initiated by, n's completors.

The control predecessors of node n are denoted $CPreds(n)$. Before any instance of n starts executing, an instance of one of n's control predecessors must complete.

The *transactors* of node m, denoted $Trans(m)$, is a set of nodes p, one of which must complete at the same time as m. $Trans(m)$ is formally defined as follows:

- If m is a transaction node, then $Trans(m)$ is the set of nodes whose instances may transact with m.
- If m is a create node, then $Trans(m)$ is the set containing the process begin node of the process whose instances are created by m.
- If m is a process begin node, then $Trans(m)$ is the set containing the create nodes that create instances of $Process(m)$.
- If m is the partition begin node b, then $Trans(m)$ is empty.

We generalize the domain of $Trans$ to sets of nodes: $Trans(S) = \bigcup_{m \in S} Trans(m)$ if $b \notin S$, or $\emptyset$ if $b \in S$.

The starters of a node p, denoted $Starters(p)$, is the set of the closest control descendants of p which are transaction nodes or process end nodes. Note that $Starters(p)$ need not all be in $Process(p)$. We generalize the domain of $Starters$ to sets of nodes: for a set S, $Starters(S) = \bigcup_{p \in S} Starters(p)$.

The partners of n, denoted $Partners(n)$, is a set of nodes q, such that an instance of some $m \in Partners(n)$ starts when any instance of n does. $Partners(n) = Starters(Trans(CPreds(n)))$.

Finding pinning in Concurrent C programs.

In this section, we identify a special difficulty of pinning analysis, which is due to the existence of multiple instances of processes. We describe a way to alleviate this difficulty.

Suppose that, for some node n, more than one instance of $Process(n)$ may execute concurrently, and that n has a single partner q which must transact with nodes in $Process(n)$. Let n_i be an instance of n, and let q_h be its partner instance. The instance q_h might be able to transact with some instance $n_{i'} \neq n_i$ of n, which may execute concurrently with n_i. Thus, we must conservatively estimate that n does not pin q.

However, the program may be constructed so that q_h can only transact with node instances in $ProcInst(n_i)$. This can occur in at least each of the following scenarios:

- $Process(n)$ is a single-instance process.
- q, and the nodes in $Process(q)$ which transact with the control predecessors of n, are signaling nodes that access the same process instance. (Figure 9.8(a).)
- q is a signaling node, and the control predecessor of n_i passes to $ProcInst(q_h)$ a reference into $ProcInst(n_i)$, which q must use as its transaction call. That is, q_h calls the caller of its predecessor. (Figure 9.8(b).)
- m creates an instance of $Process(q)$, and then uses the process variable from the process creation to call an entry in $Process(q)$. (Figure 9.8(c).)

In any of these cases, each instance of n does pin some instance of q to the wave. We can make use of this information in determining whether n pins its partners.

We define the set $Starters_{other}(p)$ to be the set of nodes which are starters of p, but which may transact with process instances other than the one which transacts with p.[4] From the $Starters_{other}$ relation, we can define a $Partners_{other}$ relation:

$$Partners_{other} = (CPreds \times (Trans - CHT) \times Starters_{other})$$

[4]We do not specify here how this information is derived. We expect to use some form of template matching on the sync graph, e.g., against cases similar to those of Figure 9.8, to eliminate from $Starters_{other}(p)$ those nodes that must transact with the same process instance that transacts with p.

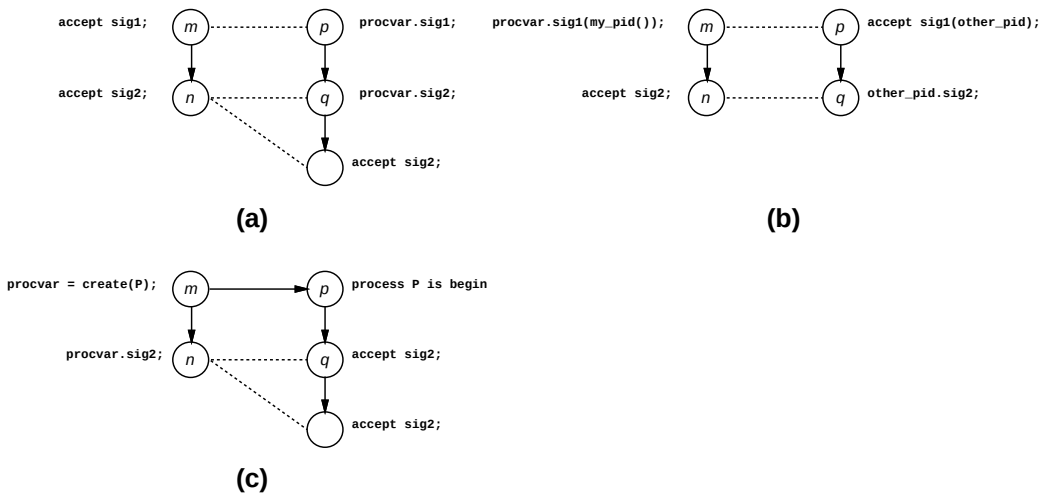

Figure 9.8: Some examples of pinning scenarios in Concurrent C.

$x \in Partners_{other}(n)$ means that x is a partner of n, and that, for instance n_i of n where x_j is a partner instance of n_i, x_j might be able to transact with a node in a process instance other than $ProcInst(n_i)$. If all nodes in $Partners_{other}(n)$ transact only with nodes in $CHT(n)$, or if $Partners_{other}(n)$ is empty and n has partners, then n pins its partners. Thus, we define the relation $Partners_{Pinned}$ as follows.

- The term $(Trans - CHT)(q)$ is the set of nodes r that might have instances r_k that can happen together with some instance q_j of q, and that might transact with q_j.

- $((Partners_{other} \times (Trans - CHT)) - CHTSelf)(n)$ is thus the set of nodes r which may have some instance r_k such that:

  - r_k may possibly transact with a partner instance q_j of some instance n_i of n; and,
  - r_k may happen together with both n_i and q_j; and,
  - r_k is not provably the same instance as n_i.

  If this set is not empty, then n does not pin its partners, and we eliminate all of n's partners from the $Partners_{Pinned}$ set.

  The second CHT relation is subtracted from this term to eliminate those nodes

which cannot happen together with n, as opposed to those which cannot happen together with the nodes with which n's partner transacts.

- $(((Partners_{other} \times (Trans - CHT)) - CHTSelf) \times ONE)(n)$ is thus the set of all nodes if n has some partner that is not pinned, or $\emptyset$ if all n's partners are pinned.
- $(Partners - (((Partners_{other} \times (Trans - CHT)) - CHTSelf) \times ONE))(n)$ is thus the set of n's pinned partners if n pins its partners, or $\emptyset$ otherwise.

To summarize,

$$Partners_{Pinned} = Partners - (((Partners_{other} \times (Trans - CHT)) - CHTSelf) \times ONE).$$

Pinning analysis and non-head nodes.

If any instance n_i of node n pins any set of partner instances that must transact with n_i, then n will always be able to transact with another node whenever it executes. Thus, n cannot be the head node of a deadlock cycle.

We can detect this case statically by examining all partners of n. If $Partners(n) \neq \emptyset$, $Partners_{other}(n) = \emptyset$, and n's partners cannot refer to nodes of a type other than n (or vice versa), then n cannot be a head node on a deadlock cycle. In this case, we place n into the set $N_{NonHead}$, which contains nodes that cannot be the head nodes of any valid deadlock cycle.

Formalism.

Let $S = Partners - (Partners_{other} \times ONE)$. For nodes n whose partners can only transact with the same process instance as n_i, $S(n)$ is the set of n's partners; for other nodes n', $S(n')$ is empty.

Let $V = ((S \times Trans) - I) \times ONE_{Vector}$. V is a set of nodes n whose partners transact only with the process instance containing n, but may transact with nodes other than n. Thus, $(S \times ONE_{Vector}) - V$ is a set of nodes whose partners all transact only with the same node instance with which they are partnered. No node n' in this set can ever be

a deadlock head node, because some partner will always be available to transact with n'.

Algorithm for pinning analysis in Concurrent C.

In this section, we present an algorithm for performing pinning analysis on Concurrent C programs, which also extends the set of non-head nodes. We prove that the algorithm has a polynomial worst-case time bound.

Algorithm 13 directly follows the discussion beginning on "Finding pinning in Concurrent C programs" (page 310.) For further explanation of the terms used in Algorithm 13, please refer to that paragraph.

Algorithm 13 *Pinning analysis in Concurrent C.*

Input:

- The following arrays, of dimensions $|N| \times |N|$:
  - CHT.
  - $CHTSelf$.
  - $CPreds$.
  - $Starters$.
  - $Trans$.
  - $Starters_{other}$.
  - $N_{NonHead}$.

Output: The following arrays of dimension $|N| \times |N|$:

- $CHT' \supseteq CHT$.
- $CHTSelf' \supseteq CHTSelf$.
- $N'_{NonHead} \supseteq N_{NonHead}$.

Procedure:

1. $Partners \Leftarrow CPreds \times (Trans - CHT) \times Starters$.

2. $Partners_{other} \Leftarrow CPreds \times (Trans - CHT) \times Starters_{other}$.

3. $Partners_{Pinned} \Leftarrow Partners - (((Partners_{other} \times (Trans - CHT)) - CHTSelf) \times ONE)$.

4. $CHT' \Leftarrow Widen(CHT, Partners_{Pinned}, N)$.

   Here we are again using the pseudotransitivity of CHT.

5. $S \Leftarrow Partners - (Partners_{other} \times ONE)$.

   The set S is as defined previously, in the paragraph labeled, "Pinning analysis and non-head nodes" (page 312.) For nodes n whose partners can only transact with the process instance containing n_i, $S(n)$ is the set of n's partners; for other nodes n', $S(n')$ is empty.

6. $CHTSelf' \Leftarrow CHTSelf \cup CHT'$.

7. $N'_{NonHead} \Leftarrow N_{NonHead} \cup ((S \times ONE_{Vector}) - (((S \times Trans) - I) \times ONE_{Vector}))$.

   $(S \times ONE_{Vector})(n) = 1$ if $S(n)$ is nonempty, i.e., if n_i's partner instances can only transact with nodes in n_i's process instance. $(((S \times Trans) - I) \times ONE_{Vector})(n) = 1$ if any of n's partners can transact with a node other than n. Subtracting the latter from the former, we get a vector in which each position n is one if n's partner instance can only transact with the instance of n for which it is a partner.

Lemma 147 (Correctness of pinning analysis in Concurrent C.) *If $CHT \subseteq CHT_{perf}$, then the output CHT' of Algorithm 13 is also a subset of CHT_{perf}. Furthermore, if the nodes in $N_{NonHead}$ cannot be head nodes of a valid deadlock cycle, then neither can the nodes in $N'_{NonHead}$.*

Proof: From the preceding discussion. □

Lemma 148 (Time bounds for pinning analysis in Concurrent C.) *One application of pinning analysis requires $\mathcal{O}(|N|^3)$ time in a Concurrent C program.*

Proof: The algorithm does not iterate; the most costly operation it contains is Boolean array multiplication of $|N| \times |N|$ arrays. □

9.4.4 Critical sections.

As in Ada and binary semaphores, critical sections may be defined in Concurrent C using stylized critical section structures. A critical section structure consists of a set of call bodies and critical section bodies; each call body must happen together with some critical section body, and vice versa.

In this section, we discuss the methods of finding critical section structures in Concurrent C programs, and the *CHT* (and other) information that can be derived from them. We also introduce a weak form of critical section structures, which occur in Concurrent C but not in the Ada or the binary semaphore critical sections defined previously.

Weak critical section structures.

Critical section structures in Concurrent C may be potentially weaker than those in Ada or binary semaphores. Because processes may have multiple instances, the critical section bodies of a critical section structure in Concurrent C may also have multiple instances; this is not true in Ada or the binary semaphore model, though the binary semaphore model permitted concurrent critical section bodies to exist in different processes. However, in the binary semaphore model, call bodies could not execute concurrently because of a necessary rule in their definition. Concurrent C, however, permits a form of critical section structure in which the call body instances must execute together with some critical section body instance, but where the call body instances need not be mutually exclusive.

Figure 9.9 shows an example of such a *weak critical section structure*. Here, both processes Q and R may call a critical section body in an instance of process P. However, since P has more than one instance (i.e., P_1 and P_2), the calls do not exclude each other. Note also that Q and R each may call either P_1 or P_2 at different times in their execution.

It is possible to propagate *CHT* information from P to Q and R, since the call bodies must happen together with some instance of a critical section body. However,

if any two (or more) instances of the critical section body may happen together, then we must assume, in the absence of other information to the contrary, that instances of the call bodies may likewise happen together.

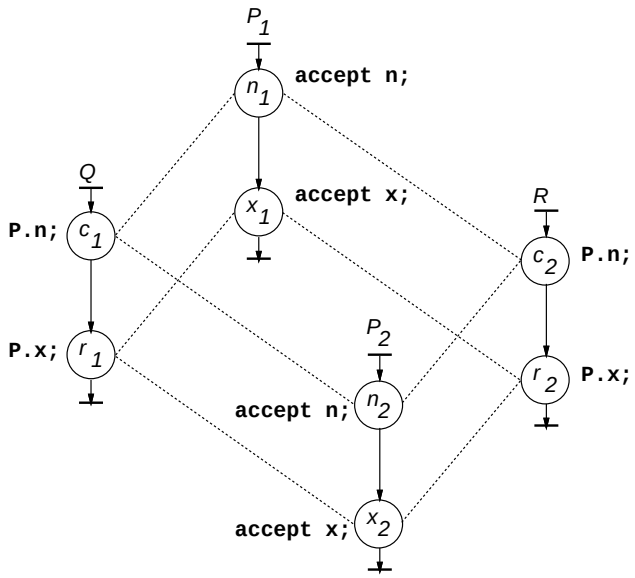

Figure 9.9: A weak critical section structure.

A deadlock whose head nodes are all bracket nodes of a single critical section structure (strong or weak) is symptomatic of an anomaly containing at least one head node which is not a bracket node of the structure. The proofs of Section 5.1.4 directly substantiate this for the subset of strong critical section structures in which the critical section bodies are all contained in a single-instance process. Where the critical section bodies may be contained in multiple-instance processes, we must also address the possibility that a call node on the wave might not address the entry node on the wave, because of sync variable settings.

For example, in Figure 9.9, suppose that c_1 and n_1 are on the wave, and are head nodes in a hypothesized deadlock cycle, but that the sync variable instance P in process instance Q addresses process instance P_2 rather than P_1. If the hypothesized cycle denotes an actual deadlock, then n_2 cannot also be on the wave. The anomaly must be more directly caused by some node instance in P_2, other than n_2, which prevents n_2 from reaching the wave. Thus, in the static sync graph, the anomaly must contain

some node in P other than n as a head node. Note that this argument holds regardless of whether the critical section structure is strong or weak.

Similar extensions may be made to the lemmas of Section 5.1.4 for the remaining three cases to show that hypothesized deadlock cycles which contain only the bracket nodes of a critical section structure either involve nodes which can't happen together, or which can transact, or where the deadlock cycle is only symptomatic of some other anomaly containing a head node that isn't a bracket node of the structure. (In Figure 9.9, these three cases are exemplified by cycles with head nodes c_1 and x_1, r_1 and n_1, or r_1 and x_1.) Thus, we can eliminate cycles whose head nodes are all bracket nodes of a particular critical section structure.

Formal definition of Concurrent C critical section structures.

The structural definition for Concurrent C critical sections presented here is one of at least two possible definitions. The definition below follows the Ada definition. We briefly discuss an alternate form in this section.

A *CC critical section structure* for signal types y_1 and y_2, $y_1 \neq y_2$ is a quadruple $CSStruct_{y_1,y_2} = (CallB_{set}, CSB_{set}, y_1, y_2)$, where $CallB_{set}$ is a set of *call bodies* for the pair of signal types (y_1, y_2), and CSB_{set} is a set of *critical section bodies* for the same pair of signal types. Each call body is delineated by a call node c and a return node r. Each critical section body is likewise delineated by a entry node n and an exit node x. A critical section structure is *strong* if no pair of critical section bodies may happen together, and is *weak* otherwise. The set of CC critical section structures for a program is called *Crits*.

A pair of nodes (c, r) delineates a call body $CallB(c, r)$ for the pair of signal types (y_1, y_2) if:

- c and r are call nodes.
- $c \in MustCall(y_1)$.
- $r \in MustCall(y_2)$.
- $(c, r) \in StrictInt((N, E_C), \{MustCall(y_1), MustCall(y_2)\})$.

- $\exists(n, x) \in StrictInt((N, E_C), \{N_{Accept}(y_1), N_{Accept}(y_2)\})$.
- $MayCall(y_1) = MustCall(y_1)$. I.e., any node that *may* call y_1, *must* call y_1.

  A later condition takes care of process variables in the call body which may refer to different instances of the critical section body's process.
- $MayCall(y_2) = MustCall(y_2)$.
- For all node pairs $(c', r') \in StrictInt((N, E_C), \{MustCall(y_1), MustCall(y_2)\})$ such that $c' \in MustCall(y_1)$ and $r' \in MustCall(y_2)$, and for all instances i of c' or r', c'_i calls exactly the same process instance as r'_i. This prevents the return node of a critical section body from transacting with a process instance other than that with which the call node transacted.

$CallB(c, r)$ exists only if (c, r) delineates a call body on some pair of signal types (y_1, y_2).

A pair of nodes (n, x) delineate an critical section body $CSB(n, x)$ for the pair of signal types (y_1, y_2) if:

- n and x are accept nodes.
- $(n, x) \in StrictInt((N, E_C), \{N_{Accept}(y_1), N_{Accept}(y_2)\})$.
- $\exists(c, r) \in StrictInt((N, E_C), \{MustCall(y_1), MustCall(y_2)\}) \overset{\text{such that}}{} \exists CallB(c, r)$.

$CSB(n, x)$ exists only if (n, x) delineates a critical section body on some pair of signal types (y_1, y_2).

A pair of signal types (y_1, y_2) delineate a CC critical section structure $CSStruct_{y_1, y_2}$ if and only if call bodies and critical section bodies for the signal pair, as defined above, exist. $CSStruct_{y_1, y_2}$ is strong if, for all critical section bodies (n, x) and (n', x') in the CSB_{set} set of $CSStruct_{y_1, y_2}$, $n \in CHTSelf(n')$. If this condition is not true, then $CSStruct_{y_1, y_2}$ is weak.

Lemma 149 provides us with two ways to establish *CHTSelf* information using critical section structures. First, we can generate *CHTSelf* information directly for strong critical sections; $m \in CHTSelf(n)$ if m and n are in critical section bodies in the same strong structure. Second, we can propagate *CHTSelf* information between critical

section bodies and call bodies (for strong or weak critical sections) using widening. If n is in a critical section body, it must happen together with a call body for the critical section; n therefore cannot happen together with any node that can't happen together with at least one node in the critical section body.

Lemma 149 *Each instance of a call body of a critical section structure must happen together with some instance of a critical section body of the same structure. Furthermore, if the critical section structure is strong, then no two instances of any call body may happen together at the same time.*

Proof: Similar to Theorem 6. □

An alternate definition of critical section structures for Concurrent C.

Previously in this section, we presented a definition of Concurrent C critical section structures, in which both bracket nodes of the critical section bodies must be accept nodes. There exists at least one alternate definition, as exemplified in Figure 9.10. The return node r of the call body is an accept node and the exit node x of the signaling body is a signaling node. Each call node instance c_i passes a reference to the return node instance r_i of the same call body instance, as a parameter to the entry (accept) node instance n_j of the critical section body. The exit node instance x_j then calls r_i upon completion of the critical section body.

This alternate form of critical section structures is so similar in structure to the one presented here, and would probably be so rarely used in practice, that we omit the details of finding it in programs.

Algorithms for critical section structures in Concurrent C programs.

The algorithm for identifying critical section structures in Concurrent C is quite similar to that used in Ada. We present it below.

Algorithm 14 *Identify critical section structures in a Concurrent C partition.*

Input:

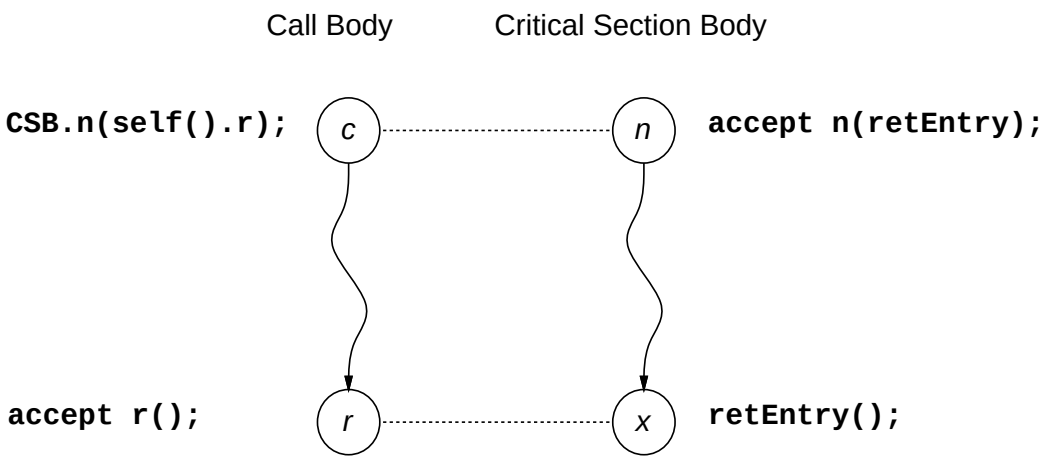

Figure 9.10: An alternative structure that may be used as a critical section.

- The PCC sync graph of the partition $G = (N, E_C E_S)$.
- *MustCall*.
- *MayCall*.
- *CHTSelf*.
- *CallSameInst*. $m \in CallSameInst(n)$ if, on all control paths from n to m that contain only one instance of n, m always calls the same process instance as n.

Output:

- A set of all critical section structures, CRITSTRUCTS.
- A set of weak critical section structures, WEAKSTRUCTS.
- A set of strong critical section structures, STRONGSTRUCTS.

Procedure:

1. Create a partition $\mathcal{P}$ of nodes. For each signal type y such that $N_{Accept}(y) \neq \emptyset$, $MustCall(y) \neq \emptyset$, and $MustCall(y) = MayCall(y)$, $\mathcal{P}$ contains the following sets of nodes:
   - $N_{Accept}(y)$;
   - $MustCall(y)$.

2. Create the control flow graph $G' = (N, E_C)$.
3. Set CANDBODIES to $StrictInt(G', \mathcal{P})$.
4. For each pair $(a \in MustCall(y), b \in MustCall(y')) \in$ CANDBODIES found in step 3, form a *candidate critical section structure* (M, N) such that:

   - M is the set of pairs $(a' \in MustCall(y), b' \in MustCall(y')) \in$ CANDBODIES.
   - N is the set of pairs $(a'' \in N_{Accept}(y), b'' \in N_{Accept}(y')) \in$ CANDBODIES.

   In short, for all pairs $(a', b') \in M$ and $(a'', b'') \in N$, a' can transact only with nodes like a'' and b' can transact only with nodes like b''.

   Place all the candidate critical section structures into the set CANDSTRUCTS.
5. Eliminate from CANDSTRUCTS all pairs (M, N) such that $|N| = 0$.
6. Eliminate from CANDSTRUCTS all pairs (M, N) such that M contains a pair (c, r) such that $r \notin CallSameInst(c)$. I.e., an instance r_i of r might call a process instance different from that of the preceding instance c_i of c.
7. For each pair (M, N) in CANDSTRUCTS, form a critical section structure $CSStruct_{y_1,y_2} = (M, N, y_1, y_2)$, where, for any pair $(m, n) \in N$, $m \in N_{Accept}(y_1)$ and $n \in N_{Accept}(y_2)$. Place $CSStruct_{y_1,y_2}$ into CRITSTRUCTS.

   If there exist any two (possibly identical) pairs $(m, n), (m', n') \in N \times N$ such that $m \notin CHTSelf(m')$, then place $CSStruct_{y_1,y_2}$ into WEAKSTRUCTS. If no two such pairs exist, place $CSStruct_{y_1,y_2}$ into STRONGSTRUCTS.

Lemma 150 *Using Algorithm 14, the time to find critical sections is*

$$\mathcal{O}(|N|^3 + |N||Sigs|^2).$$

Proof: Almost identical to Lemma 55. □

In a fairly obvious way, we can use widening on all critical section structures to improve *CHT* information, since each critical section body must execute together

with some call body. We can further use strong critical section structures to improve *CHTSelf*, since only one instance of any node in any call body of a strong critical section structure may execute at any given time.

Note that *CHTSelf* information is used to differentiate strong critical section structures from weak ones in step 6. As *CHTSelf* is augmented by analysis, it may be possible to decide that a critical section structure previously labeled as weak is, in fact, strong. This may further improve *CHTSelf* information.

9.4.5 *B4* analysis in Concurrent C programs.

In Concurrent C, as in the other models, we say that m is in $B4(n)$ if all instances of m in the partition must complete before any instance of n in the same partition can start. If $m \in B4(n)$, then $m \in CHT(n)$ and $n \in CHT(m)$.

If m is contained in a process with at most a single instance, then the computation of *B4* sets is much the same as it is in Ada. If not, then it becomes very difficult to deduce that $B4(n)$ contains m, for any node n.

Preconditions to deduce *B4*.

In this section, we outline a set of preconditions by which we may recognize that $m \in B4(n)$. In this section, we will use these preconditions to construct a lattice framework for *B4* analysis.

Lemma 151 defines a precondition which can be used to find *B4* information within a single-instance process.

Lemma 151 *If m is contained in a process which has a single instance in the current partition, and there is a control path from m to n, and no control path from n to m, then $m \in B4(n)$.*

Proof: Let n_i be the first instance of n to complete. No instance $m_j \in ProcInst(n_i)$ of m can complete after n_i, because control cannot reach m after exiting n. Since there are no instances of $Process(n)$ other than $ProcInst(n_i)$, no instance of m may start once the first instance of n has started. □

Lemma 152 defines a precondition which can be used to propagate $B4$ information along control edges.

Lemma 152 *If there exist nodes m and $n \neq m$ such that, for all control predecessors p of node n, $m \in B4(p)$, then $m \in B4(n)$.*

Proof: Note that *Process*(m) need not be the same as *Process*(n), and that neither *Process*(m) nor *Process*(n) need be a single-instance process. Suppose that, during execution, an instance n_i of n is entered by a control edge from instance p_k of p. The control edge may represent either control flow within a process instance, or the creation of a new process instance. In either case, all instances of m have completed before p_k started. Therefore, all instances of m must complete before n starts. □

The completors of a node n in the Concurrent C model, denoted *Completors*(n), is the set of the closest control ancestors of n which are transaction nodes or process begin nodes, excluding n. Since *Completors*(n) dominates n, some instance of a node in *Completors*(n) must finish before any instance of n can start. Unlike the starters of n (Section 9.4.3), the completors of n must all be in *Process*(n).

Lemma 153 defines a precondition which can be used to propagate $B4$ information along sync edges.

Lemma 153 *Let n be a node in the sync graph. If, for all nodes q that can transact with any completor of n, one of the following holds:*

- *$m \in B4(q)$; or,*
- *$m = q$, Process(m) is a single instance process, and m is a single-instance node within that process.*

Then $m \in B4(n)$.

Proof: Note that, if the process begin node for *Process*(n) is a completor of n, then q may be a create node for *Process*(n). Let n_i be an instance of n, and p_k be the instance of the completor of n_i through which n_i was reached. By assumption, p_k is not an instance of n. Let q_h be the node instance with which p_k completed a transaction before n_i was reached.

If $m \in B4(q)$, then all instances of m must have completed before q_h could start. This is true regardless of whether q is a create node or a transaction node. Thus, q_h could not have completed, and control could not have reached n, before all instances of m had completed.

Now suppose that $m = q$, $Process(m)$ is a single instance process, and m is a single-instance node within $Process(m)$. Thus, $m_h = q_h$. m_h must then complete before n_i can start. By the preconditions for this case, m_h is the only instance of m. Then the single instance of m must have finished before n_i started.

Therefore, all instances of m must have completed before any instance of n can start. □

Lattice framework for limited $B4$ analysis in Concurrent C.

The $B4$ lattice framework $D_{B4} = (G_{B4}, L_{B4}, F_{B4}, M_{B4})$ for the Concurrent C program model is quite similar to that for the Ada and binary semaphore models. The lattice L and function space F are, in fact, identical to those used in both of these models, and thus have identical convergence time properties. We will therefore present only the graph G (and mapping function M) here.

Completor edges in Concurrent C programs.

Earlier in this section, we defined the completors of a node. Here, we introduce a new edge type which is useful in codifying the $B4$ problem as a lattice framework.

A completor edge (m, n) exists if m may transact with a completor of n, or if the process begin node of $Process(n)$ is in $Completors(n)$ and m creates instances of $Process(n)$. The set of completor edges for the current partition is designated $E_{Completors}$. Let $Gen(m) = \{m\}$ if m is a single-instance node, $\emptyset$ otherwise. By Lemma 153, any nodes in $\bigcap_{(m,n)\in E_{Completors}}(B4(m) \cup Gen(m))$ must be in $B4(n)$.

Formally, the set of completor edges is:

$$\begin{aligned} E_{Completors} = \{(m,n): \quad & (\exists p: \ \{m,p\} \in E_S \\ & \wedge \ p \in Completors(n) \\ & \wedge \ m \notin CHTSelf(p)) \\ \vee \quad & (\ m = \texttt{create}(Process(n)) \\ & \wedge \ b_{Process(n)} \in Completors(n))\}. \end{aligned}$$

As we iteratively refine the *CHTSelf* sets, we may also eliminate spurious completor edges to improve the estimate of the *B4* sets.

B4 mapping function for Concurrent C.

We map a control or completor edge $e = (m,n)$ to a function $f_e \in F_{B4}$ as follows:

$$f_e((C,S)) = (C \cup S \cup K_e, N)$$

if e is a control edge, or

$$f_e((C,S)) = (N, C \cup S \cup K_e),$$

if e is a completor edge. For either edge type, if m is in a single-instance process,

$$K_e = CReach(m) \cup \{m\} - \{p : m \in CReach(p)\},$$

i.e., the set of nodes q such that there is a control path from q to m (possibly of zero length; possibly with $Process(m)$ as a descendant of $Process(q)$), but no control path of length > 0 from m to q. If m is not in a single-instance process, then $K_e = \emptyset$.

Lemma 154 *The total time for convergence of a Concurrent C B4 lattice problem using the worklist iterative algorithm is $\mathcal{O}(|N|^3 \log |N|)$ in the worst case.*

Proof: By similarity to Lemma 29. □

Lemma 155 *The total time to solve a Concurrent C B4 lattice problem using Tarjan's path expression algorithm is $\mathcal{O}(|N|^3)$ in the worst case.*

Analysis	Time	Lemma
Pinning	$\mathcal{O}(\|Procs\|\|N\|^3)$	148
Critical section	$\mathcal{O}(\|N\|^3 + \|N\|^2\|Sigs\|^2)$	150
B4 (worklist)	$\mathcal{O}(\|N\|^3 \log(\|N\|))$	154
B4 (Tarjan)	$\mathcal{O}(\|N\|^3)$	155

Table 9.2: Time for each form of *CHT* analysis in Concurrent C programs.

Proof: By similarity to Lemma 29. □

9.4.6 Time complexity of *CHT* analysis for Concurrent C.

Table 9.2 shows the time for each technique used in *CHT* analysis of Concurrent C program partitions. Again, we may use *CHT* information to eliminate spurious sync edges by iterative refinement.

Lemma 156 *The total time for CHT analysis of a partition, with iterative refinement, is*

$$\mathcal{O}(|N|^5 \log(|N|) + |N|^4|Sigs|^2)$$

using the worklist algorithm, or

$$\mathcal{O}(|N|^5 + |N|^4|Sigs|^2)$$

using Tarjan's algorithm.

Proof: Similar to Lemma 68. □

9.5 Deadlock cycle detection in Concurrent C programs.

The final step in Concurrent C deadlock detection is to find cycles which are not known to violate the four constraints of Section 9.3. If no such cycle exists, then the program cannot deadlock. This section explains how deadlock cycles are detected by an iterative approximation algorithm.

9.5.1 Invalid deadlock paths within a single process.

In Concurrent C, certain control paths within a single process are not part of any valid deadlock cycle. This section identifies two such types of paths, and identifies a transformation on the sync graph that eliminates these spurious paths.

Control paths with accept head and tail nodes of the same signal type.

If a proposed deadlock cycle enters and exits a process at accept nodes of the same signal type, the cycle must be spurious. While such paths are also spurious in Ada, the information kept in the *SACnt* lattice elements is usually sufficient to eliminate them. However, in Concurrent C, keeping similar information would not be sufficient to disprove the cycles in most cases. Lemma 157 shows why such cycles are spurious:

Lemma 157 *No valid deadlock path can enter and exit a process through accept nodes of the same signal type.*

Proof: If such a path exists in the static sync graph, then the implication is that, in the dynamic sync graph, an accept head node instance in a deadlock cycle is in the same process instance, and has the same signal type, as an accept tail node. A signaling head node that can transact with the accept tail node must also be on the wave at that time. Thus, the signaling head node can also transact with the accept head node, and the execution wave represented is not deadlocked. □

Lemma 158 presents a similar proposition, that discards paths containing certain pairs of signaling nodes as head and tail nodes.

Lemma 158 *No valid deadlock path can enter and exit a process through signaling nodes that must always refer to the same signal type and process instance.*

Proof: Similar to Lemma 157. □

Recall that *SACnt* summarizes both the number of signaling nodes and the scope depth of accept-in nodes in deadlock paths; if scope depth is not more than the number of signaling nodes, some pair of head nodes can rendezvous, and the path does not correspond to a deadlock.

In Concurrent C, though, simply counting nodes on the cycle is not sufficient to disprove deadlock. Any number of process instances may be present. While the signaling nodes of one process may reference the signal type of accept nodes of a different process, any given pair of instances of these processes might or might not refer to each other. Thus, signaling and accept nodes of the same signal type may be head nodes of the same cycle; if $SACnt$ were used, the result would not be conservative. Instead, we must rely on sync graph transforms to eliminate these paths.

Control paths that traverse create edges.

A valid deadlock path may not traverse a create edge and exit the child process at an accept node. If a proposed deadlock cycle contains such a path, then the corresponding deadlock would contain a head node that refers to a process that does not exist at the time the head nodes execute.

Figure 9.11 shows a Concurrent C program with a spurious cycle that contains such a path. The cycle (m, d, b_{P1}, p, m) exists because the `main` process transacts with two process instances it creates. However, note that m cannot refer to accept node p, because the process instance to which p belongs could not have been created at any time that m executes. Thus, since the path from m to p includes a create edge, the path cannot have any accept node as a tail node. Lemma 159 below formalizes this property:

Lemma 159 *Any proposed deadlock path which includes a create edge, and which has an accept node as a tail node of the descendant process, violates constraint 1.*

Proof: Let D be a proposed deadlock cycle in the execution graph that includes a create edge, and which has an instance a_k of an accept node a as a tail node in the descendant process. Let $h_i \in D$ be the instance of the head node such that a is the tail node, and let $s_j \in D$ be the head node instance which refers to a. h_i must complete before $ProcInst(a_k)$ can be created. So no definition of any sync variable referring to $ProcInst(a_k)$ can exist in the program while h_i is executing. Since s_j is on the wave with h_i, it cannot refer to a_k. Thus D is not a valid deadlock cycle under constraint 1.

```
process spec P1()
{
  trans s;
}

process body P1()
{
  accept s;             -- p
}

main()
{
  process P1 p1Var;

  p1Var = create();     -- c
  p1Var.s;              -- m
  p1Var = create();     -- d
  p1Var.s;              -- n
}
```

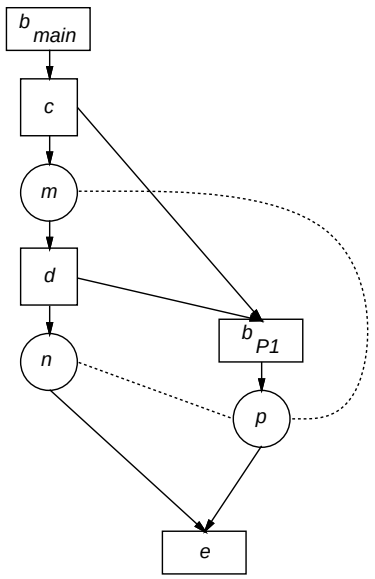

Figure 9.11: A Concurrent C program with a spurious cycle.

□

One way to avoid this problem might be to insert flags into the lattice elements, indicating that the path has traversed a create edge in its current process. However, this would result in non-monotonic edge functions. Instead, we rely on the transformation of the sync graph described next, to eliminate this case.

The cycle detection (CDCC) sync graph.

The cycle detection form of the CC sync graph, denoted the *CDCC* sync graph, is derived from the PCC sync graph. For every pair i of transaction nodes (m, n) in the PCC sync graph such that:

- There is a control path from m to n, and;
- m and n are not both accept nodes for the same signal type (in violation of Lemma 157), and;
- m and n are not both signaling nodes for the same signal type such that each one always uses the same process and transaction definition as the other (in violation of Lemma 158), and;

- Either n is a signaling node, or the path contains no create edges (and the path thus does not violate Lemma 159; if any path contains a create edge, then all paths will);

we create nodes m_i and n_i in the CDCC sync graph, with a control edge (m_i, n_i) between them, for each such pair i of nodes. If there exists a sync edge $\{m, p\}$ in the original sync graph, we insert sync edges from every m_i to every p_j. Note that such paths also have the property that all tail nodes of deadlock cycles are reachable from all head nodes. Thus, this transformation performs the same function that the double loop inlining did in Ada and binary semaphores.

Figure 9.12 shows the CDCC sync graph for the program of Figure 9.2. Note that, although there is a control path from v to x in Figure 9.2, the pair (v, x) are omitted from the CDCC sync graph, because v and x are accept nodes for the same signal type. Node q is omitted, since there is no control path between q and any other transaction node in the partition induced by the loop body containing q.

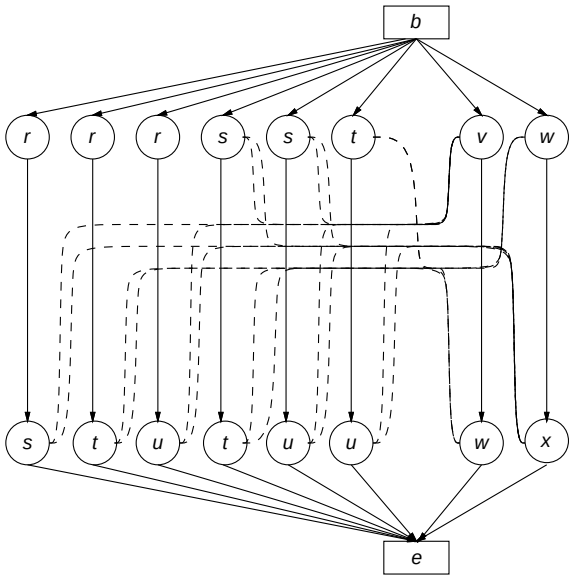

Figure 9.12: CDCC sync graph of the partition on the loop in `main` of Figure 9.2.

By this construction, the CDCC sync graph has $\mathcal{O}(|N|^2)$ nodes and $\mathcal{O}(|N|^4)$ control and sync edges, in the worst case.

9.5.2 CC cycle location graphs.

Under constraint 1, all deadlock cycles must traverse at least one control edge in each process instance they enter. To enforce this requirement, we detect cycles in a *CC cycle location graph*, rather than in the sync graph itself, much as was done for Ada in Section 4.2.5.

The transform from the CDCC sync graph to the CC cycle location graph is simply a matter of directing the sync edges from "tail" to "head" nodes; all transaction nodes in the CDCC sync graph represent one or the other, unambiguously. A transaction node n is a "tail" node if there is a control edge in the CDCC sync graph from another transaction node to n.

9.5.3 CC lattice framework for cycle detection.

To find deadlock cycles, we hypothesize a series of possible deadlock head nodes. For each such head node h, we search for cycles including h that may be valid deadlock cycles. If no such cycle is found for any possible head nodes, then the partition cannot deadlock.

We formulate this process of cycle detection as an iterative data flow problem, based on a lattice framework. The lattice elements include information used to help eliminate spurious cycles. The lattice space for global deadlock detection is a cross-product of the base lattice spaces for each hypothesized head node h. The elements of the base lattice at node n contain the following:

- A set S of signals called to single-instance processes, by head nodes on all deadlock paths from h to n.
- A set A of signals accepted by head nodes in single-instance processes, on all deadlock paths from h to n.

The two sets of signal types are used to eliminate cycles in which some signaling head node s transacts with an accept head node a, and where a is in a single-instance process. (If a were in a multiple-instance process, then the instance of s on a deadlocked

execution wave might refer to a different process instance than the instance of a. In this case, s and a might not be able to transact, and the deadlock would not be broken.)

Lattice space.

The base lattice framework for cycle detection is the quadruple $D_{CC} = (G_{CC}, L_{CC}, F_{CC}, M_{CC})$. G_{CC} is the CC cycle location graph (N_{CLG}, E_{CLG}). Each base lattice framework is specific to a particular assumed head node h. To solve the *DeadPath* problem, we form the cross product of the D_{CC} frameworks for all head nodes.

The value of L_{CC} at a node n maintains information about whether there is any valid deadlock path from h to n, and whether that path must include a process creation edge. We represent a non-top element of L_{CC} as a pair (S, A), where S and A are sets of signals. A value of (S, A) at a node n, for an assumed head node h, means:

- S is a set of signals called to single-instance processes by the head nodes on any deadlock path from h to n.
- A is a set of signals accepted by accept nodes in single-instance processes which are head nodes on any deadlock path from h to n.

A distinguished top element, $CC_\top$, is also included in L_{CC}. $a \in L_{CC} \sqsubseteq b \in L_{CC}$ iff:

- $b = CC_\top$; or,
- $a = (S_a, A_a)$ and $b = (S_b, A_b)$ and $S_a \subseteq S_b$ and $A_a \subseteq A_b$.

Thus, $(S_a, A_a) \sqcap (S_b, A_b) = (S_a \cap S_b, A_a \cap A_b)$. We denote $CC_\bot = (\emptyset, \emptyset)$.

Lemma 160 *The height of the L_{CC} lattice is $2|Sigs| + 2$.*

Proof: Similar to Lemma 69. □

Function space.

For each edge $e \in E_{CLG}$, we assign a function $f_e \in F_{CC} : L_{CC} \to L_{CC}$. F_{CC} is the closure under function meet and composition of the following sets of functions:

- *Constant functions*: $f_e(x) = k_e$ where $k_e \in L_{CC}$.

  The distinguished *top function*, $f_\top(x) = k_\top$, is a constant function.

- *Conditional inflationary functions:* If $\tau_e(x) = True$, then $f_e(x) = (S_x \cup S_e, A_x \cup A_e)$. Otherwise, $f_e(x) = CC_\top$.

  $\tau_e : L_{CC} \rightarrow \{True, False\}$ is the *test function* for edge e. We require that τ_e be evaluable in $\mathcal{O}(|Sigs|)$ time.

  The *identity function*, f_ι, is a conditional inflationary function such that $\tau_\iota(x) = True$ for all x, and $S_\iota = A_\iota = \emptyset$. Similarly, a *conditional identity function* has τ_ι set arbitrarily, and $S_\iota = A_\iota = \emptyset$.

Lemma 161 *The set of edge functions is monotonic. Furthermore, with the exception of the constant functions, the set of edge functions is inflationary.*

Proof: Identical to Lemma 70. □

Define F^-_{CC} as the closure of the set of the conditional inflationary functions under meet and function composition. Using arguments similar to those of Section 6.1.2, we need not consider the constant functions in deriving time bounds.

Lemma 162 *The functions of F^-_{CC} are monotonic, inflationary, and 1-semibounded, but are not distributive.*

Proof: Similar to Lemmas 71 and 72. □

Lemma 163 *The meet operation in L_{CC} requires $\mathcal{O}(|Sigs|)$ time.*

Proof: The meet requires the intersection of two sets of nodes. □

Lemma 164 *The edge functions in L_{CC} require $\mathcal{O}(|Sigs|)$ time to compute.*

Proof: The constant functions require constant time to compute. The conditional inflationary functions requires a comparison and possibly a set union, which takes $\mathcal{O}(|Sigs|)$ time to compute. □

Lemma 165 *A problem in the CC lattice framework requires $\mathcal{O}(|Sigs||N|^4 \log(|N|))$ time to complete using the worklist iterative algorithm.*

Proof: From Lemmas 12, 163, and 164, and considering that the number of nodes in the CDCC sync graph is $\mathcal{O}(|N|^2)$ in the worst case. □

Lemma 166 *A problem in the CC lattice framework requires $\mathcal{O}(|N|^4|Sigs|)$ time to complete using Tarjan's algorithm.*

Proof: Direct from Lemmas 13, 163, and 164, and considering that the number of nodes in the CDCC sync graph is $\mathcal{O}(|N|^2)$ in the worst case. □

Mapping function in the CC lattice framework.

The mapping function $M_{CC} : (E_S \cup E_C) \rightarrow F_{CC}$ maps control and sync edges to the function space, for a particular hypothesized head node h. It is defined for edges e as follows:

- If e is a control edge (m, n) in the CDCC sync graph, then:
  - If $m = h$, and h is an accept node in a single-instance process, then $M_{CC}(e)$ is the constant function $(\emptyset, \{SigType(h)\})$.
  - If $m = h$, and h is a signaling node that must call an accept node in a single-instance process, then $M_{CC}(e)$ is the constant function $(\{SigType(h)\}, \emptyset)$.
  - Otherwise, $M_{CC}(e) = f_\iota$.
- If e is a sync edge $\{m, n\}$ in the CDCC sync graph, then $M_{CC}(e)$ is the conditional inflationary function f_e, with S_e, A_e, and τ_e defined as follows. Intuitively, τ_e tests whether the propagation can occur over the edge for a pair (S, A); if not, it forces $f_e((S, A))$ to $CC_\top$.
  - If n is a signaling node that must call an accept node in a single instance process, then $S_e = \{SigType(n)\}$. Otherwise, $S_e = \emptyset$.
  - If n is an accept node in a single instance process, then $A_e = \{SigType(n)\}$. Otherwise, $A_e = \emptyset$.
  - $\tau_e((S, A)) = False$ if any of the following hold:
    - $n \in N_{NonHead}$.

* $n \in CHT(h)$.
* For the unique pair m', m such that (m', m) is a control edge in the CDCC sync graph, $m' \in CHT(n)$.

  Note that since each tail node corresponds to a particular head node in the CDCC sync graph, we can maintain this information for m.
* n is a node in a single-instance process, $n \neq h$, and the signal type of some accept node of $Process(n)$ is in A. (Thus, the path would enter a single-instance process more than once at a node other than the hypothesized head node.)
* n is a signaling node that calls only accept nodes of signal type t in a single-instance process, and $t \in A$.
* n is an accept node of signal type t in a single-instance process, and $t \in S$.
* (h, m) is a control edge in the CDCC sync graph, and $\{h, n\} \subseteq N_{Bracket}(CSStruct_{y_1,y_2})$ for some critical section structure $CSStruct_{y_1,y_2}$.

Otherwise, $\tau_e((S, A)) = True$.

Lemma 167 *τ_e can be evaluated in $\mathcal{O}(|Sigs|)$ time.*

Proof: We can express $N_{NonHead}$ and $CHT(h)$ as Boolean vectors, and determine whether n is contained in either of them in constant time.

By construction of the CDCC sync graph, a single head node m' is associated with each tail node m. Therefore, we can also look up n in a Boolean vector representing $CHT(m')$ in constant time.

S and A may also be represented as Boolean vectors of length $|Sigs|$. So, for a particular signal type t, we can look up whether $t \in S$ or $t \in A$ in constant time. Thus, if n is an accept node of signal type t in a single-instance process, or if n must call such a node, we can evaluate this condition of $\tau(e)$ in constant time.

We may also form a Boolean vector representing all the bracket nodes of all critical section structures to which h may belong. If h is the head node corresponding to tail node m, we can evaluate, in constant time, whether n and h are bracket nodes in the

Operation	Worklist	Tarjan's
Parse	$\mathcal{O}(\|N\| + \|E_C\| + \|E_S\|)$	$\mathcal{O}(\|N\| + \|E_C\| + \|E_S\|)$
CHT sets	$\mathcal{O}(\|N\|^5 \log(\|N\|)$ $+\|N\|^4 \|Sigs\|^2)$	$\mathcal{O}(\|N\|^5 + \|N\|^4 \|Sigs\|^2)$
Cycle detection	$\mathcal{O}(\|N\|^4 \log(\|N\|) \|Sigs\|)$	$\mathcal{O}(\|N\|^4 \|Sigs\|)$

Table 9.3: Complexity of deadlock analysis for partitions of Concurrent C programs.

same critical section structure.

Finding whether A contains any signal type of a process containing n can be done in $\mathcal{O}(|Sigs|)$ time; this is the most complex step in the evaluation of $\tau(e)$.

Thus, $\tau(e)$ may be evaluated in $\mathcal{O}(|Sigs|)$ time. □

9.6 Complexity of the entire analysis.

Table 9.3 summarizes the time complexity of analyzing a partition of a program written in the Concurrent C dialect specified in Section 9.1.1. The complexity analysis of the entire program is detailed in Section 9.6.1.

Lemma 168 *The time complexity for analyzing a partition of a Concurrent C program is* $\mathcal{O}(|N|^5 \log(|N|) + |N|^4 |Sigs|^2)$.

Proof: Direct from Table 9.3. □

9.6.1 Complexity of analyzing the entire program.

Since we are analyzing Concurrent C programs partition by partition, we must relate the time complexity for analysis of a single partition to the complexity of analyzing the entire program. Because the CC sync graphs of the partitions of a program are not necessarily disjoint, this time may be greater than the time to analyze the CC sync graph of the entire program.

As an example of a program whose partitions are not disjoint, consider a slight modification to the `main` process of Figure 9.2. Suppose the loop in `main` were copied several times. Each copy would represent a partition, and each partition would contain all the nodes in P_1 and P_2.

However, the number of partitions is bounded by the number of create nodes of the program's CC sync graph, since the root of each partition must contain at least one create node. (If the root of a partition contains no create nodes, then the partition must synchronize with, or use sync variables defined at, some program point outside the partition.) Thus, the complexity of analyzing the entire program is at most $|N_{Create}|$ times that of analyzing a single partition., where N_{Create} is the set of create nodes for the program. If the majority of the nodes are other than create nodes, the complexity of analyzing a single partition may approximate that of analyzing the entire CC sync graph.

Lemma 169 *The time complexity of analyzing a Concurrent C program for deadlock is bounded by* $\mathcal{O}(|N|^5|N_{Create}|\log(|N|) + |N|^4|N_{Create}||Sigs|^2)$.

Proof: From Lemma 168 and the above discussion. □

9.7 Conclusions.

We have demonstrated that our static deadlock analysis techniques can, in principle, be applied to programs in which processes are created dynamically, and in which process references propagate around the program as scalar variables.

The technique must be refined, and its effectiveness demonstrated, by experimental work.

Chapter 10

Related work.

10.1 Introduction.

In this chapter, we provide a survey of the current state of the art in deadlock detection and non-concurrency analysis. We show how our work extends or complements previous research in static anomaly detection in parallel programs.

We review work that is related to ours in any of four ways. First, in Section 10.2, we review the literature which theoretically characterizes the problems of static synchronization analysis. We discuss the relation of these theoretical and enumerative results to our own methods.

Second, we review the *directly related work*, whose aim is to statically analyze synchronization in some way. This is divided into three sections: synchronization anomaly detection (Section 10.3), race and non-concurrency analysis (Section 10.4), and formal verification methods (Section 10.5). The relation of the work in static synchronization analysis to our own is succinctly summarized in Section 10.6; we show how our methods extend and refine the current state of the art.

Third, we review work in intertask static data flow analysis, which *might benefit* from the analyses we have developed, in Section 10.7. For each of these, we detail the possible refinements to other research which our methods might provide.

Lastly, for the sake of completeness, we review some of the literature of *dynamic deadlock detection* in Section 10.8. The main relationship of this work to ours is that both attempt to remediate deadlock problems by detecting them automatically; there is arguably some similarity between our sync graphs and certain dynamic representations.

For the convenience of the more casual reader, a "Concise Overview," detailing the main features of the research discussed, is provided at the beginning of each section.

10.2 Problem characterization.

In this section, we discuss the literature characterizing the types and characteristics of deadlocks, the complexity of deadlock detection, and methods of resolving the problem of deadlock.

Concise overview.

First, in Section 10.2.1 Coffman and Denning [CD73] give a general overview of the phenomenon of deadlock itself. Using their general model of deadlock, we update their classification of strategies for deadlock management to include static deadlock detection.

Taylor [Tay83a] characterizes the complexity of static analysis of synchronization, including analysis of deadlock and *CHT*. In Section 10.2.2, we relate these complexity results to our own work.

Lastly, Cheng [Che90] provides a specific enumeration of types of deadlocks which may occur in Ada. In Section 10.2.3, we discuss which of these are, and are not, covered by our Ada deadlock analysis algorithm, and the problems which must be solved to cover all the deadlock types Cheng lists.

10.2.1 Coffman and Denning [CD73].

In their text, Coffman and Denning [CD73] present a formal overview of the problem of deadlock. They address the required conditions for deadlock, and the broad categories of preventive measures.

Deadlock is described in terms of a *progress space*, through which the program negotiates a route to its completion. The number of dimensions in the progress space equals the number of tasks in the system; distance in each dimension represents the "progress" that a task makes toward its completion. The program's progress is represented by a path whose length is nondecreasing in any dimension. Within the progress space, *infeasible regions* represent places where two or more tasks are mutually exclusive; the progress path cannot enter those regions. The infeasible regions may also overlap to form *unsafe regions*, i.e., cusps. If the progress path enters an unsafe region, it will never be able to leave it, and the program will deadlock. The authors identify three conditions needed for the unsafe region to exist: *mutual exclusion* between tasks, *nonpreemption* of deadlocked tasks, and *resource waiting* (i.e., tasks hold their resources while waiting for others). This model may, with effort, be adapted to synchronization such as rendezvous [SS90].

Coffman and Denning describe three main ways of dealing with deadlocks:

- *Prevention*, by removing one of the three conditions for deadlock. This is the way in which "deadlock-free" schedulers operate. However, this strategy may have other operational side effects, such as requiring tasks to recover from preemption.
- *Detection*, in the dynamic sense. As with prevention, recovery remains a problem.
- *Avoidance* of the unsafe region, using scheduling methods that steer the progress path away from the unsafe space. Habermann [Hab69] proposed such avoidance methods. However, the unsafe space may sometimes not be avoidable; if it cannot be avoided, then the program must deadlock.

None of these methods quite expresses what static deadlock detection attempts to do. Therefore, to Coffman and Denning's three methods, we add a fourth: *elimination and certification*. Elimination and certification employs automatic tools to alert the programmer that the program may deadlock. If the tool detects a potential deadlock, the programmer is required to change the progress space to *eliminate* the unsafe region in the progress space. If the program cannot deadlock, it is *certified* to be free of deadlocks. Elimination and certification includes all static deadlock detection methods.

10.2.2 Complexity of static synchronization analysis: Taylor [Tay83a].

In [Tay83a], Taylor addresses the complexity of concurrency analysis, for a variety of synchronization analysis problems, under a variety of different assumptions about program structure. Among his results are that exact solutions to many useful problems are NP-hard or NP-complete.[1] Taylor used a subset of Ada as his programming model; his results would also apply to synchronization primitives that can simulate the Ada rendezvous with a constant number of synchronization operations. (These include binary semaphores, if we omit the Ada `select` statement.)

Taylor restricts the domain of his proofs to programs whose tasks are statically defined, and which execute exactly while the main procedure is executing (i.e., the

[1] Taylor's criterion of exactness is actually that the problem solutions are *precise up to symbolic execution* [Ban78], i.e., they make the assumption that all control paths are executable. Since he analyzes concurrent programs, he also assumes that all rendezvous between statements are executable.

programs have no `abort` statements). He may further restrict the domains of particular proofs to programs obeying some combination of the following four restrictions on program structure:

1. The program has no control branches or loops.
2. The program has no `select` statements.
3. Only one task may call a given entry.
4. Synchronization statements on the branches of a conditional may differ only in order of occurrence, but not in number of occurrences.

In general, relaxing any of these conditions would only make the problems being considered more difficult, and not less.

In combination with these programming disciplines, he develops theorems about the following problems:

- *Possible rendezvous (PR):* Given an entry call and an accept statement, can they ever rendezvous?

  This problem might be useful in deadlock detection, but only in very limited circumstances (e.g., detecting that a program with only one accept statement of a given type *must* have an anomaly whenever some particular call to the accept is executed). However, the problem is used in Taylor's later proofs.
- *Parallel actions (PA):* Can two statements execute in parallel?

  PA is equivalent to the *CHT* problem. PA is NP-complete, even if PR is known, for programs obeying restriction 1 (Theorem 5). From this, determining *CHT* exactly is co-NP-complete for programs in the same domain.
- *Possible deadlock (PD):* Can the program have an infinite wait?

  PD subsumes deadlock detection itself, since Taylor's *infinite wait anomalies* include both deadlocks and starvation. PD is NP-complete for programs observing restrictions 1 and 2 (Theorem 6), or 1 and 3 (Theorem 7), so it is also at least NP-complete for general programs.

In [MR90a], we prove related theorems showing that detection of deadlock cycles that observe constraints 1 and 2, or 1 and 3, is NP-hard. Note that these constraints are not the same as Taylor's; they are constraints on the deadlock cycles, not program structure restrictions.

- *Converse of "must rendezvous" (Co-MR):* Given an entry call and an accept statement, can there be a terminating execution such that they do not rendezvous? A solution to the Co-MR problem could be used to eliminate deadlock cycles that contain entry or accept nodes that must be able to rendezvous. Co-MR is, however, NP-complete under restrictions 1 and 2 (Theorem 9), and therefore is also at least NP-complete for general programs.

The conclusions we must draw are that, except in unrealistically simple programs, exact detection of synchronization anomalies may be exponentially complex in the number of synchronization statements. We should therefore investigate safe and accurate approximations.

10.2.3 Classification of Ada deadlocks: Cheng [Che90].

In [Che90], Cheng's intent is to enumerate all possible types of deadlocks which can occur in the Ada language. To do this, Cheng presents 22 different types of deadlock, using his *task wait for graph (TWFG)* as a representation of task and subprogram interaction during *dynamic* program execution.[2] It is useful to review here the types of deadlocks listed in [Che90], and to cite which of these our present algorithm and methodology can detect.

The TWFG has nodes of two types and edges of five types. Nodes represent tasks or subprograms. The edge types represent task activation, waiting by accept or entry call statements, subprogram and block activation, and task termination. Edges are directed from the waiting task node to the node for the task which is waited upon. It should be noted that the *termination waiting* edge type arises only from accept statements that are waiting on an open terminate alternative, and not from tasks which have reached

[2] In [Che91], Cheng puts the number of types of deadlock at 18; nonetheless, [Che90] has 22 examples.

the end of their execution.

Cheng defines deadlock as a circular "waits-on" relation between tasks; this is identical to our definition. He categorizes deadlocks according to the combinations of the five edge types they contain. Of the types of deadlocks he enumerates, our current algorithm will detect the following without massage of the source code:

- Circular acceptance waiting during execution of a task (Ex. 2).
- Circular entry calling during execution of a task (Ex. 6).
- Acceptance and entry call blocking (Ex. 9).

Semantic analysis of the source code can detect deadlocks involving calls from a task to itself, without the necessity of solving static analysis problems. These include the following types of deadlocks:

- Self acceptance waiting (Ex. 3).
- Self entry calling during activation of a task (Ex. 5).
- Self entry calling during execution of a task body (Ex. 7).

Certain of the deadlock types listed in [Che90] cannot presently be detected by our *algorithm*, but we feel that, with minor extensions to our deadlock detection machinery, these could also be modeled and detected in our *methodology*. These include deadlocks which incorporate activation, dependence, or termination edges. The changes necessary to accommodate these deadlock types would include modeling nested task activations and the `terminate` alternative, and inlining task activation code as a preheader for the task. These classes of deadlock include all of the remaining types that Cheng enumerates:

- Circular entry calling during activation of a task (Ex. 4).
- Activation and entry call blocking (Ex. 8).
- Acceptance and termination blocking (Ex. 10).
- Entry call and dependence blocking (Ex. 11).
- Entry call and termination blocking (Ex. 12).
- Activation and acceptance and entry call blocking (Ex. 13).

- Activation and entry call and dependence blocking (Ex. 14).
- Activation and entry call and termination blocking (Ex. 15).
- Acceptance and entry call and dependence blocking (Ex. 16).
- Acceptance and entry call and termination blocking (Ex. 17).
- Entry call and dependence and termination blocking (Ex. 18).
- Tasking blocking without activation waiting (Ex. 19).
- Tasking blocking without acceptance waiting (Ex. 20).
- Tasking blocking without dependence waiting (Ex. 21).
- Tasking blocking without termination waiting (Ex. 22).
- Complex tasking blocking (Ex. 23).

[Che90] is an extension of Cheng, Araki and Ushijima [CAU88], in which four types of deadlock are enumerated. [CAU88] also mentions that deadlock in a program can be dynamically detected by detecting the existence of a cycle in the TWFG of the executing program (called the "digraph" in [CAU88]).

10.3 Static nontermination anomaly detection.

In this section, we will describe past research in static detection of deadlocks and other infinite wait anomalies, and show how our work extends, or avoids difficulties in, that of other researchers. We include all references to such work that we have found in the recent literature, even if infinite wait detection is not the main goal of the research.

Concise overview.

We have divided the past work in infinite wait anomaly detection according to the preferred representation. *Flow graph methods* are built around abstractions of the program's flow graph; these are described in Section 10.3.1. *Petri net methods* use Petri net [Pet81] abstractions of program behavior, possibly with modifications to the Petri net model; these are described in Section 10.3.2. Most of the flow graph methods rely on state enumeration, though two do not [Sax77, BDER79]. All the Petri-net methods we have found rely, in one way or another, on state enumeration.

Section 10.3.3 discusses the work of Karam and Buhr [KB90], in detecting livelock-induced starvation; strictly speaking, their work does not address infinite waits. Finally, Section 10.3.4 reviews, and refutes some assertions in, a survey of deadlock detection methods by Cheng [Che91].

10.3.1 Flow graph methods.

The methods described in this section are based on condensed flow graph representations of the program. These are graphs in which nodes represent synchronization operations (or phases of synchronization) and directed control edges represent control paths between synchronizations. Other edges, representing the synchronizations themselves, may or may not be present. Our sync graph representations fall into this category.

Concise overview.

Taylor [Tay83b] details a state-enumeration approach on a control flow graph of the program; variations on Taylor's work include Young and Taylor [YT88], Long and Clarke [LC89], and McDowell [McD89]. In an early work similar to our own, Saxena [Sax77] finds deadlock cycles in a graph of tasks and resource requests, and prunes spurious cycles using constraints. Although deadlock detection is not their primary goal, Bristow et. al. [BDER79] give a method of detecting some, but not all, deadlocks; more importantly, they make the first mention of iterative refinement of a "happened before" relation similar to *B4*. Finally, Zhou, Yeh, and Ng [ZYN86] propose the use of linear algebra to analyze path expressions, as a step preliminary to state enumeration.

Concurrency state enumeration: Taylor [Tay83b].

Taylor, in [Tay83b], presented one of the first published algorithms for static analysis of concurrency in languages using rendezvous. The method used is to represent the program as a set of flow graphs, and find the *concurrency history* (set of reachable concurrency states and state transitions) given this representation. Taylor also proposes

optimizations for state generation which can be applied in special cases.

Taylor's flow graphs are control flow graphs, whose nodes represent task interactions and whose edges represent control paths between them which contain no tasking statements. The node types Taylor uses include call-pending, call-engaged, awaiting-call, accept-engaged, accept-end, and select; these represent the phases of accept and entry call statements in execution. The resulting graph structure is quite similar to the sync graph representation of a single task, but uses a different set of node types.

From the set of task flow graphs for the program, Taylor develops the program's concurrency history. The set of concurrency history states is similar to the set of states produced by our all-states analysis, again taking into account the differences in node types. From the concurrency history, CHT_{perf} information can be extracted, and starvations and deadlocks can be detected and distinguished from one another. However, the length of the concurrency history may be exponential in the number of source program statements.

Section 4.5 of Taylor's paper describes some optimization techniques used in generating the concurrency history. In this section, Taylor introduces the *task graph*, in which nodes represent tasks and edges represent possible interactions between them. Taylor's optimization is *parceling*, or dividing the state generation between sets of tasks which are represented by biconnected components in the task graph. The effectiveness of parceling had not been investigated experimentally at the time [Tay83b] was published.

In conclusion, the graph representations of [Tay83b] closely resemble our sync graphs, and Taylor's analysis is analogous to our all-states analysis. However, as our experimental data shows, Taylor's analysis can be combinatorially explosive in practice.

Concurrency analysis and symbolic execution: Young and Taylor [YT88].

In [YT88], Young and Taylor attempt to address the possible combinatorial explosiveness in [Tay83b] in two ways: by using symbolic execution to constrain concurrency state generation, and by partitioning the program.

The authors propose to construct a directed graph, representing the execution of the program, and to annotate the nodes with *path expressions (PEs)* and *path conditions (PCs)*. The nodes in the graph are *symbolic execution states*; each concurrency state may correspond to one or more symbolic execution states, because different PEs and PCs are possible within a single concurrency state. If the path condition and expression of a newly-generated symbolic execution state can be reduced to false, the state is unreachable, and is therefore discarded. The authors also give a heuristic for prioritizing symbolic execution states, for expansion by marking them as "interesting" (when they represent error states, such as deadlocks) or "promising" (when they are ancestors of "interesting" nodes).

The section on program partitioning considers biconnected components (as did [Tay83b]), and *weak monitors*. A weak monitor is a part of the program with specific properties that allow it to be analyzed separately, then ignored in the analysis of the containing code. Weak monitors must be identified by the programmer; however, it is possible to check automatically that a proposed weak monitor has the required properties. Young and Taylor's concept of the weak monitor bears some resemblance to our concept of partitions in the analysis of Concurrent C.

Despite the similarities between weak monitors and our partitions, the two serve different purposes. Weak monitors are intended to avoid combinatorial explosion, while partitions are intended to improve the accuracy of analysis. Also, while manual identification of partitions can be used, our intention is that partitions be identified automatically.

Essentially, [YT88] is an "idea paper." There is no evidence that symbolic execution will reduce the total analysis time, improve the precision of the analysis, or significantly reduce the combinatorial explosions of state generation, in practice. More work is clearly needed.

Control edges as task interaction: Long and Clarke [LC89].

Long and Clarke [LC89], reacting to the combinatorial explosions in Taylor's original approach [Tay83b], proposed a concurrency analysis based upon a different graph representation for programs. They dubbed their representation a *task interaction graph*, or *TIG*; the TIG represented the concurrency behavior of a single task. Upon the TIG representation, they build a *task interaction concurrency graph*, or *TICG*, representing the behavior of the entire program.

The TIG of a task is a directed graph. Rather than representing the rendezvous statements of a task, as in [Tay83b], the nodes of the TIG represent the *regions* of the task, i.e., the code between tasking statements. Nodes are annotated with *pseudocode* for the regions they represent; the pseudocode is Ada, annotated with ENTER and EXIT *transition pseudostatements*, denoting the task interactions that cause transitions between regions. Directed edges in the TIG represent transitions between regions, and are labeled with the task interactions (e.g., entry calls) that caused the transitions. The TIG construction resembles a dual of the sync graph, in which the nodes represent synchronizations and edges represent paths between them.

Each node in the TICG of the program of k tasks is a k-tuple, in which each t'th element is a node in the TIG for task t. The nodes of TICG thus represent sets of regions which may execute concurrently; this concept which is dual to that of execution waves, which represent sets of synchronizations that may happen concurrently.

Directed edges in the TICG represent the beginning and end of rendezvous. Section 4.1 outlines an extension of the model that allows for nested task activations and early termination via the `terminate` alternative. The authors cite empirical data suggesting that, in practice, the TIG/TICG model has a lower number of reachable states than the model of [Tay83b].

Long and Clarke note that the TICG representation is usually smaller than, and is never larger than, the concurrency histories of [Tay83b]. Nonetheless, since the state space of a program may be exponential in the number of source statements, the size of the TICG (and therefore the time required to analyze concurrency) may also be

exponential.

In section 4.2 of [LC89], Long and Clarke describe a method of infinite wait anomaly detection using the TICG. It should be noted that they use the term "deadlock" in a different sense than we do; to them, deadlocks encompass all infinite wait anomalies, including starvation. An infinite wait anomaly occurs at a TICG node n if none of the regions in n is able to provide synchronization for any other region in n. This may occur even if the TICG node has successors, unlike the concurrency state graph of [Tay83b].

Also in Section 4.2, Long and Clarke develop the concept of *edge groups*, representing transitions to the alternatives of a select statement. These edge groups closely resemble the groups of sister nodes used in our analysis. [LC89] does not address the issue of guarded versus unguarded select alternatives.

In some special cases, it is possible to reduce the number of TIG nodes needed to represent a task. Section 4.3 describes such a structural optimization, which reduces the number of nodes needed to represent entry calls or entries from 3 to 2, if the entries have no nested synchronization. However, this linear optimization cannot prevent combinatorial explosion in the size of the TICG.

In summary, [LC89] should be viewed as an incremental advance over [Tay83b], but retains the undesirable, combinatorially explosive behavior of the former.

Task clans: McDowell [McD89].

In [McD89], McDowell investigates a state enumeration method in which sets of related concurrency states [Tay83b] are represented by a single *virtual state*. The idea behind this is to reduce the size of the state space to one which can be enumerated in practice; McDowell claims that "the number of virtual states is often orders of magnitude less than the number of reachable states." However, McDowell admits that "the number of concurrency states in the worst case is still greater than polynomial" in his methodology.

Although McDowell compares his work to Taylor's [Tay83b], he actually analyzes a source language quite different from Ada. His language contains both binary semaphores (`wait` and `signal`) and mutual exclusion (`lock` and `unlock`), as well as dynamic

task initiation. (The primitives apparently originated at the Los Alamos National Laboratory, where McDowell obtained a set of test programs.) Like Taylor, McDowell uses a compressed flow graph for analysis, which he calls a *synchronization graph.* He proposes an execution model containing both program state and data state information, similarly to our models for binary semaphores and Concurrent C.

Using his synchronization graph and execution model, McDowell creates a canonical *concurrency history graph*, or *CHG*. The canonical CHG is similar to the graphical representation of Taylor's concurrency histories, except that data state information is added at the nodes. McDowell then proposes to generate *reduced* CHGs that are *anomaly equivalent* to the canonical CHG: corresponding nodes in the canonical and reduced CHGs must both have (or not have) infinite wait anomalies and identical concurrency relations between source code fragments.

McDowell's reduced CHG replaces tasks by *task clans*, intended to represent the possible states of more than one task. Each task clan consists of a set of task identifiers and a set of possible states in which they may be. For the reduced CHG, McDowell defines both a new execution model and a new state enumeration algorithm.

McDowell presents experimental evidence that his technique reduces the size of the CHG, based on four test programs from Los Alamos and one "contrived" test program. However, McDowell says that, "for two of the programs the prototype (analyzer) was unable to create the CHG when more than two tasks were analyzed." This indicates that the prototype analyzer still experiences problems with combinatorial explosion.

Despite its problems, the idea of reduced state spaces is an interesting one. Future research might well be directed at exploring state spaces which are reduced *past* the point of anomaly equivalence, to polynomial size. These reduced state spaces would be required to be *anomaly safe,* in the sense that the reduced space will contain anomalous nodes if (but not necessarily *only if*) the canonical space also does. In a sense, our analyzer actually does this implicitly, if the reduced state is taken to be the cross-product of $SACnt$ values propagated along all sync edges. This represents, for a head node h, a superset of the nodes that can execute together with h, and thus a superset of the possible concurrency states of the program.

Deadlock cycle pruning in resource graphs: Saxena [Sax77].

Saxena, in [Sax77], proposes to statically detect deadlock in systems in which tasks share resources. Thus, there is no synchronization *per se*, only mutual exclusion. His methods have influenced our analysis of nested critical sections (see Section 5.1.4). His analysis resembles ours in other ways; he uses constraints 2 and 3 to prune invalid deadlock cycles (although his representation is far different from ours).

Saxena's main method is to show that resources in the program may be totally ordered, such that if a task has a resource with number r, it can never request a resource with a number less than itself. The ordering is effectively the nesting level of resource requests.

Saxena defines a *resource graph* model of resource requests in the program. Nodes represent resources; a directed edge (r_i, r_j) is drawn if any process holding resource r_i requests r_j. Note the resemblance between the resource graph and the general resource graph of [Hol72]. A program can deadlock only if its resource graph has a cycle.

Even if the resource graph does have a cycle, it may sometimes be possible to prove that the program will not deadlock. One way Saxena describes is by finding whether the cycle is part of an *exclusive component*. An exclusive component is a strongly connected component such that there is a single-unit resource that must be held by any task before the task requests other resources in the component. This situation forces mutual exclusion of the requests that might otherwise deadlock, and is akin to finding a *CHT* relation between requests for resources.

Strongly connected components in the resource graph can also be pruned from deadlock detection by proving that they are part of a *reducible component*. For a reducible component, the available resources exceed any requests that can be made by the tasks that use the resources in the component. This type of pruning uses a relation like Constraint 2.

In sum, Saxena's methodology for deadlock detection anticipates some key points of ours: polynomial-time cycle detection and cycle pruning using constraints 2 and 3. However, Saxena presents his results in terms quite specific to his shared resource

model, and does not give experimental results.

Iterative refinement: Bristow et. al. [BDER79].

Bristow, Drey, Edwards and Riddle, in [BDER79], proposed an early use of static analysis to detect races and infinite waits in concurrent programs. Their work centered around a hypothetical language they called X, which had post-wait-clear synchronization and dynamic initiation of *single* process instances. Their emphases were on both synchronization anomalies and data races.

Bristow et. al. construct an *interprocess precedence graph*, which strongly resembles the sync graph. Both control and *interprocess precedence edges (IPPEs)* (i.e., sync edges) are directed. Waits could apparently be governed by negated event variables, so sync edges go from `reset EV` statements to `wait for not EV` statements.

[BDER79] describes an algorithm to generate BEFORE and AFTER sets, based upon a form of analysis that resembles [CS88], but is formulated as an algorithm. In Section V.3, [BDER79] makes the first mention of iterative refinement of sync edges, and event variable killing, that we have found in the literature. They delete IPPEs (m, n) when either n is before m, or a node that cancels the effects of m must always execute after m and before n. Our *SemKill* problem (Section 8.4.4) in binary semaphore analysis strongly resembles their method. They refine iteratively by deleting IPPEs until the BEFORE and AFTER solutions stabilize. Thus, Bristow et. al. probably discovered iterative refinement of both *CHT* and the sync graph (in our terminology), but did not apply it to refinements of *CHT* other than *B4*.

[BDER79] gives a simple heuristic for detecting infinite wait anomalies, including deadlock. A wait node w may be involved in an infinite wait if it is possible that, between every node s and w such that s sets the event variable upon which w waits, there may be a clear node that clears the event variable. (If w waits upon a cleared event variable, then it must be possible for some s to be between c and w.) No experimental evidence is cited; we believe that this method would yield too many false alarms in practice, especially when synchronization is contained in loops.

Linear algebraic methods: Zhou, Yeh, and Ng [ZYN86].

Zhou, Yeh, and Ng [ZYN86] analyze deadlock in Ada programs in three stages. The stages are: (1) a source-level transformation, (2) comparing the number of `accept` statements and entry calls for each signal, and (3) "virtual execution" of the program. This paper essentially extends the work of Hao and Yeh [HY82] from CSP to Ada.

The source-level transformation produces a condensed version of the program's source code, reduced to synchronization statements and with procedures inlined.[3] Transformation 6 is unsafe, since it changes the control flow of the program by eliminating `goto` and `exit` statements.

In the second step, the authors form an expression for the entire program, denoting the total number of entry call instances without corresponding accept instances that can occur in the program execution. They use a notation $(a+*B)$, where a is a set of integers denoting possible contributions to the count from single-instance statements, and B is a set of possible contributions from multiple-instance statements. This amounts to a set of congruences; see [Mer92] for a more formal application of this idea.

Using the expression, a set of *discriminating equations* is formulated for the program. The discriminating equations assert that, for successful termination, the number of calls to each entry equals the number of times the entry is accepted. If the discriminating equations have no natural number solutions, then the program cannot terminate at all. Theorem 4.2 gives conditions for the system to have such solutions; theorem 4.3 gives necessary conditions for a program to be free of deadlock, in terms of the coefficients of the discriminating equations. Note that the discriminating equations may yield an infinite number of solutions. [ZYN86] states that the solutions must be identified, for use in the next step, but no indication is given of how the solutions are used.

Even if the discriminating equations have solutions satisfying Theorem 4.3, the program may still deadlock because of improper ordering of the synchronization statements. The final *virtual execution* step of detection is thus a state enumeration like that

[3] There is a serious typographical error in step 4, implying that all loops with synchronization should be eliminated. The example does not reflect the step as it is worded.

of [Tay83b], followed by a cycle detection for each of the dead states. Any efficiency gains in [ZYN86] must therefore come from the use of discriminating equations to detect programs that do not deadlock. It is difficult to see how the second step would be very effective in practice; the authors do not justify their technique experimentally.

10.3.2 Petri net methods.

A large part of the static synchronization anomaly literature is devoted to the use of the *Petri net* representation in detecting anomalies. As in the flow graph literature, the representation of the program varies from researcher to researcher. We review several of the Petri net papers here, for comparison with our work. We assume here that the reader has some familiarity with the general Petri net representation; a widely used text on the subject is [Pet81].

The idea in this research is (or should be) to use the extensive collection of Petri net analysis methods to facilitate approximate deadlock analysis. However, much of the Petri net research seems to be devoted to direct state enumeration, rather than approximation methods; [Tay83a] proves that this may be combinatorially explosive, regardless of representation. Petri nets also have the disadvantage of placing an additional level of abstraction between the program structure and the results of analysis.

Concise overview.

Murata, Shenker, and Shatz ([MSS89], Section 10.3.2) use mathematical invariants on Petri nets as a mechanism for detecting infinite waits; unfortunately, they can only analyze loopless programs, and have doubly-exponential complexity bounds. Cheng and Ushijima ([CU91], Section 10.3.2) detect deadlocks by state enumeration, in a version of the Petri net extended with "inhibit" inputs to transitions.

Petri net invariants: Murata, Shenker, and Shatz [MSS89], and related work.

Murata, Shenker, and Shatz [MSS89] use the concept of *invariants* on Petri nets to detect deadlock. Invariants are used initially to guide a phase of interactive program modification, and are then used to constrain state space generation. Unfortunately, since the authors ignore loops in state space generation, their method is unsafe; further, it is at least doubly exponentially complex in worst-case execution time.

In brief, two types of *invariants* exist on Petri nets. A *T-invariant* is a vector of transition firing counts which does not change the number of tokens in each place in the net. An *S-invariant* is a set of weights on the places of the Petri net, such that the weighted sum of the tokens in the net remains constant. The invariants can be obtained by a conceptually simple algorithm, which has the disadvantage of being exponentially complex in time [MS82].

The authors propose a representation of Ada programs as Petri nets; since the Petri net execution model is well-defined, this automatically gives them an execution model. Places represent phases of rendezvous within tasks; transitions represent synchronization occurring between tasks.

Deadlocks, in the authors' notation, include all infinite wait anomalies. The set of deadlocks of a program are divided into two parts: *inconsistency deadlocks*, representing a situation where a certain transition can never be fired in any terminating execution of the program, and *circular deadlocks*, representing the cyclic "waits on" relationship (which includes starvation, by the construction of the Petri net model of the program). Inconsistency deadlocks are artifacts of the Petri net representation; there is no direct correspondence to them in in our model. Circular deadlocks can be detected only when all inconsistency deadlocks have been removed.

The authors find the inconsistency deadlocks by finding a subset of the T-invariants that represent terminating executions of the program; these are the *Ada T-invariants*. The algorithm to do this is exponential in the number of synchronization statements in the program and in the number of Ada T-invariants. Once an inconsistency deadlock

is found, the programmer is required to remove it and re-analyze the program.

Once all inconsistency deadlocks have been removed, a second algorithm repeatedly removes vertices that are not members of any *C-type dicircuit*, i.e., deadlock cycle. Since the authors also make the (unsafe!) decision not to report deadlocks caused by loops containing synchronization, the procedure disconnects loops from the rest of the program's Petri net. Following this, the Ada T-invariants are used to generate a subset of the state space of the Petri net representing the program; again, this may be exponentially complex.

The algorithms of [MSS89] are combinatorially explosive in theory. The authors do not present experimental data, so their time performance in practice cannot be evaluated. However, unless one is willing to ignore deadlocks in loops, the safety properties of [MSS89] need improvement.

Reduction transforms.

In a later article, Tu, Shatz and Murata [TSM90] describe a set of reduction transforms on the Petri net representations of Ada programs. Each transform can be applied in time linear in the size of the Petri net. Four of these transforms are general to all Petri nets, and five are specific to the Petri nets formed by Ada programs. The transforms given are *deadlock preserving*, in that the transformed program will deadlock iff the original one does. The authors show an example of the "gas station" problem, a 3-task program whose original Petri net has 46 places, and whose reduced net has only 7 places. While the reduction does not eliminate the combinatorial explosion problems of [MSS89], it may go a long way toward alleviating them.

Software tool kits.

Shatz, Black and Tu, in [SMBT90], describe a software tool kit for analyzing Ada tasking programs. It consists of two major components: the front end (*FETS*), which parses programs and assembles their Ada nets, and the back end (*BIDS*), which returns query information about the Petri nets. Other tools (such as reduction transformers or

deadlock analyzers) can be invoked from the BIDS; the intent is to provide a flexible platform from which static analysis technology of the type described in [MSS89] and [TSM90] can be studied.

Extended Petri nets: Cheng and Ushijima [CU91].

Cheng and Ushijima [CU91] propose to statically detect both deadlocks and livelocks using an *extended Petri net (EPN)* model. In this model, transitions may be inhibited from firing if certain of their input places are marked. The need to inhibit transitions comes from the fact that events in one task may inhibit events in another. For instance, termination of a task at an accept statement with a terminate alternative may be inhibited by a call to the accept statement.

Cheng and Ushijima outline a complicated model of 24 distinct types of tasking events and 19 different task states. Based on these sets, they create translation schemas from Ada tasking statements to sub-nets of the program's extended Petri net.

In Theorem 4.2, Cheng and Ushijima note that, if a certain linear equation on the transition matrix and initial state has no positive integer solution which is a "firing vector" for the initial state, then the program has no "static tasking dead-state" (infinite wait anomaly). Similarly, Theorem 4.3 shows that, if another linear equation has no positive integer solution for X, then the program cannot livelock. [Pet81] gives procedures for testing whether a vector is a firing vector.

However, even ignoring the difficulties of disproving the existence of positive integer solutions which are firing vectors, there is a basic problem. *Any* program with synchronization in terminating loops will exhibit both infinite wait anomalies and static livelocks using this approach. Thus, the false alarm rate for the approach of [CU91] will be unacceptably high. The method, as described in [CU91], thus cannot be used in practice.

10.3.3 Livelock-induced starvation: Karam and Buhr [KB90].

Karam and Buhr [KB90] use a method similar to symbolic execution to generate a set of reachable *program state assertions (PSAs)* for Ada programs. From these, "starved" states are detected, and tested to see whether they could have been produced under "fair" scheduling. Karam and Buhr's term "starvation" would, perhaps, better be described as "starvation induced by livelock." This analysis, and the fact that they prune some false alarms by use of assumptions about scheduling, are the primary contributions of this paper. The technical description in [KB90] is quite difficult to follow.

The authors describe their *program state assertion (PSA) tree* as analogous to the tree of reachable states, but with additional associated information. According to the authors, starvation detection consists of identifying repeatable sequences of PSAs in the tree, then testing whether each sequence could have been generated by fair scheduling. The PSA tree is itself produced by a deadlock analyzer; it seems to be similar in structure to the concurrency history of [Tay83b], though the authors do not give explicit algorithms for generating the PSA tree.

Starvation detection itself is done by examining the properties of the sets *must-do* (rendezvous corresponding to open entry calls), *is-done* (rendezvous producing child states), and *option-to-do* (rendezvous producing sibling states) for each repeating path in the PSA tree. The cardinality of these sets indicates starvation induced by livelock in a fair schedule, according to Theorem 1.

The authors apparently have an implementation, and report an analysis time of twenty minutes for the more complex example (readers-writers) given in the paper. It would be exceedingly difficult, in our opinion, to reproduce or extend the authors' results, due to the opacity of the technical exposition.

10.3.4 Survey of Ada deadlock detection methods: Cheng [Che91].

In [Che91], Cheng surveys seven of the published methods for detection of deadlock in Ada programs; both static and dynamic methods are examined, and tested against "correctness" criteria that he invents. In the introduction, he asserts, "The most important

conclusion is that finding a correct tasking deadlock detection method is still an open issue." He then reviews a total of four static methods [Tay83b, ZYN86, CU91, MSS89] and two dynamic methods [HL85, CAU88] of deadlock detection, dismissing each of these in turn as as "incorrect." Since most of these articles are presented in this chapter, we will not repeat the reviews here.

The word "correct" does not describe what Cheng should rightly imply; "incomplete" would be a much less loaded term. One might think that Cheng had simply misunderstood the meaning of the word, were it not for his later assertion that no form of static deadlock detection can be "correct" by his standards, and his implication that all such work should be abandoned. Given the serious nature of Cheng's allegations, one must examine his correctness criteria in their strictest possible interpretation.

Cheng's criteria for correctness are, "(1) the method must be able to detect, in finite time, any tasking deadlock in any Ada program, and (2) the method does not report any nonexistent deadlock in any Ada program." Criterion (1) is with respect to all the types of deadlock enumerated in [Che90]; it seems to be a goal that is feasible in principle, but one that simply has not yet been met. On the other hand, the problem of simultaneously meeting both criteria would subsume the Turing halting problem;[4] it is simply not possible.

Cheng further claims that, "there can never be a correct static detection method in the sense of our correctness criterion." As we have pointed out, this statement is undoubtedly true; it is also meaningless and unconstructive. Cheng later asserts that, "the only possible correct direction ... is dynamic detection."

Contrary to this claim, it is also clear that no dynamic deadlock detection algorithm can be correct in Cheng's sense. For such an algorithm to satisfy criterion (1), it must be able to supply, in finite time, test data (and a schedule compatible with the Ada scheduling specifications) that will force the program into a deadlocked state in finite time, should the program have one. By criterion (2), if no such test set exists, the

[4]Given a sequential program P, embed P into a concurrent program P' that can deadlock, but only after P terminates.

algorithm should also report this in finite time. By a simple argument, any dynamic algorithm that satisfies Cheng's correctness criteria can thus also solve the Turing halting problem.

In more practical terms, dynamic deadlock detection cannot satisfy Cheng's criterion (1), because of the difficulty in generating test sets that force the program into all its possible concurrency states. This is especially true for large programs and programs with conditionally executed rendezvous and guarded entries.

Furthermore, as Cheng himself points out in an article on his own static deadlock detection method [CU91], "As a static analysis method, our method has the same major advantage as that [sic] other proposed static analysis methods have, i.e., the method can guarantee the absence of some tasking ... deadlocks, and the analysis result is independent of run-time environment." Thus, Cheng simultaneously denounces static deadlock detection as worthless, cites important practical advantages for it, and conducts his own research in the area!

In conclusion, Cheng's allegations are inaccurate, misleading, and destructive. They should not be used as a basis for rejecting static deadlock detection as a research goal.

10.4 Static race detection and concurrency analysis.

Much of the original work in static analysis of explicitly concurrent programs was originally directed toward concurrency analysis, for use in static race detection. A *race* occurs when two parts of a program may access a shared resource (e.g., a shared variable) at the same time, and at least one of these parts changes the state of the resource (e.g., writes the variable). If two statements accessing a shared resource cannot be concurrent, they cannot race (though this does not guarantee that their statement instances will occur in a determinate order).

Our *CHT* analysis is a form of concurrency analysis, as is race detection; in principle, some other accurate form of concurrency analysis could be substituted for *CHT*. In this section, we explore some of the work of other researchers on static analysis of concurrency.

Concise overview.

Callahan and Subhlok ([CS88], Section 10.4.1) detect potential data races in polynomial time in programs using post/wait synchronization, by approximating a relation with semantics similar to our *B4*. Their work is translated to rendezvous synchronization by Duesterwald and Soffa ([DS91], Section 10.4.2), who also extend it to interprocedural non-concurrency analysis. While [DS91] has technical flaws, it suggested the method we use for our *B4* analysis.

Mercouroff ([Mer92], Section 10.4.3) proposes the use of a lattice framework of arithmetical congruences for intertask non-concurrency analysis. Emrath and Padua ([EP88], Section 10.4.4) outline a hierarchy of program determinacy criteria, and present a race detection algorithm. Balasundaram and Kennedy ([BK89], Section 10.4.5) analyze post-wait synchronization for non-concurrency.

10.4.1 Non-concurrency: Callahan and Subhlok [CS88, CKS90].

Callahan and Subhlok [CS88] is probably the first publication on polynomial time concurrency analysis. Their goal in doing this research is static race anomaly detection. Working with post-wait synchronization, they formulate a set of data flow equations whose fixed point approximates a relation similar to *B4*, for programs without control loops. The iterative flow analysis equations of [CS88] are the basis of the analysis of [DS91] and our *B4* analysis.

Post-wait synchronization is similar to binary semaphores, but there are important differences. In post-wait, the `wait(EV)` statement does not clear the event variable `EV`; the event variable may be cleared by a `clear` statement, but the authors do not provide any methodology for including the `clear` statement in their analysis. In binary semaphores, the statement $P(x)$ clears the semaphore x non-interruptibly.

The authors represent program synchronization in a *synchronized control flow graph*, which is quite similar to our sync graphs, and which includes directed sync edges. They also include the parallel `case` statement in the constructs they analyze, thus widening their model to include fork-join parallelism. (We believe that it would be fairly easy

to extend our analysis to include the parallel `case` statement; the only change from a non-parallel `case` statement would be that the alternate branches of the `case` statement would not initially be placed in each others' *CHT* sets for the parallel `case` statement.

The authors develop three sets of information. *CPreserved*(n) is the set of synchronizations that happen before all control predecessors of n. *SPreserved*(n) is the set of synchronizations that happen before all *synchronization predecessors* of n, i.e., all nodes that post a signal upon which n waits. The union of these sets, *SCPreserved*(n), is a set of nodes which must have completed at the time n completes. Note that this is in contrast with $B4(n)$, which is a set of nodes that must complete before n *starts.*

Callahan and Subhlok present their theoretical results and algorithm in the form of iterative set equations, but do not formulate the results as a lattice framework. Their set equations can be formulated as a lattice framework for dissimilar edge types, similar to our *B4* framework. This allows us to use past results to address properties such as the convergence time of the problem and the accuracy of the solution.

Callahan and Subhlok prove conservativeness and polynomial execution time for their algorithm, and also prove that determining exact *Preserved*(n) information is co-NP hard. They do not present their analysis as a lattice formulation, but our multiple-edge-type lattice frameworks can easily represent their *SCPreserved* equations.

The authors do not present experimental work. Our own experiments suggest that their *SCPreserved* analysis might yield only a small subset of the *CHT* information that could be found by concurrency state enumeration. However, we should note that finding *CHT* is not the primary goal of the authors.

Later work [CKS90] extends their model to include loops. The extension allows identifying the *SCPreserved* relation between statement instances in loops. The range of the *SCPreserved* relation is changed from nodes m to pairs $(m, \vec{\delta})$, where $\vec{\delta}$ is a *dependence distance vector*, or a vector of integers corresponding to the iteration counters in the program. $(m, \vec{\delta}) \in$ *SCPreserved*(n) means that, any time an instance $(n, \vec{l})$ of n (in their notation) completes with loop counters at $\vec{l}$, then all instances of m with loop counters less than $\vec{l} - \vec{\delta}$ have already completed.

While this extension is useful for race anomaly detection, the extension does not bear directly on the *CHT* computation required for deadlock detection. However, if we know that, for all instances m_j of m, there exists a k such that $m_j \in B4(n_k)$ and $n_k \in B4(m_{j+1})$, then we know that $m \in CHT(n)$. In this way, a relation like the extended *SCPreserved* might be used to refine or extend *CHT* information.

10.4.2 Interprocedural analysis: Duesterwald and Soffa [DS91], [Due91].

Duesterwald and Soffa [DS91, Due91] propose using the technology of [CS88] directly in the analysis of Ada programs, and applying it interprocedurally. Like Callahan and Subhlok ([CS88] et. seq.), their goal is race detection. Our *B4* analysis is closely related to their $Before_{Sync}$ analysis, described later in this section.

Duesterwald and Soffa use a *module interaction graph*, or *MIG*, program representation, somewhat similar to the sync graph. Using the MIG, Duesterwald and Soffa first perform *local control flow analysis*; this derives the relation $Before_{Loc}$. Node $m \in Before_{Loc}(n)$ if there is a control path from m to n, but no control path from n to m; [DS91] derives $Before_{Loc}$ from sets *may_before* and *may_after*. In this way, [DS91] extends [CS88] to programs with loops without using distance vectors. We use the same method to initialize *B4*, but compute the initial value using a lattice framework for control reachability.

The next step is *synchronization analysis*. [DS91] applies Callahan and Subhlok's framework directly to the MIG to derive $Before_{Sync}$. As with [CS88], our multiple edge type lattice frameworks can represent their iterative equations in a more conventional format, allowing a wider range of theoretical analyses.

Direct use of the algorithm of [CS88] in this application has some technical problems, though. Recall that the semantics of "$m \in SCPreserved(n)$" are that all instances of m have completed by the time that any instance of n completes. This still admits the possibility that the last instance(s) of m may co-execute with the first instance of n. Thus, direct application of the framework of [CS88] may result in $Before_{Sync}(n)$ for some node n containing some nodes that are not ordered with respect to n.

For instance, in their Table 1, node 4 is in the $Before_{Sync}$ set for node 10, despite the fact that node 4 *must* be able to happen together with node 10, if the program is to terminate. This situation makes sense only if we take the semantics of the assertion, $4 \in Before_{Sync}(10)$, to be that all instances of node 4 must finish before any instance of node 10 finishes. However, this is apparently not the semantics the authors need to get their *Ordered* relation (analogous to our CHT); rather, they need our sense of the $B4$ relation. We avoid this problem by using completor edges, rather than sync edges, in propagating $B4$ information between tasks. Thus, our $B4$ relation maintains the correct semantics.

In later analysis steps, Duesterwald and Soffa analyze the *activation contexts* of units, e.g., the call points of subroutines. The final phase of the algorithm is to initialize, and solve by recurrence, an "ordered" relation *Ord*, analogous to our CHT. The recurrence is used to propagate *Ord* information from activation sites to the units they activate. The authors claim that this process can be made to work even in the presence of recursion [Due91].

While [DS91] does not discuss iterative refinement of the *Ord* relation, it mentions the possibility of increasing precision by removing spurious sync edges. [Due91] mentions using the *Ord* relation itself to find spurious sync edges.

10.4.3 Congruences: Mercouroff [Mer92].

Mercouroff [Mer92] describes a non-concurrency analysis using a novel lattice framework. The analysis works for channel-based communications (as in CSP). Mercouroff's main contribution is to approximate the number of times that communication statements addressing a particular channel may be executed in each task. This approximation allows him to eliminate spurious sync edges in some cases.

A perfectly accurate representation of this information would be a subset of $\mathcal{N}$, but this is obviously incomputable. The representation Mercouroff uses is the lattice of *arithmetical congruences [Gra91] bound by a finite set*[5]. Elements of this lattice are $\emptyset$,

[5]This term is somewhat of a misnomer, since Mercouroff's approximation is the union of a finite set

$\mathcal{Z}$, or sets $\pi \cup \sigma(c, m)$. The set π is a finite set of integers, whose *width* (maximum minus minimum) is bounded by a constant. Given integers c and m, the set $\sigma(c, m)$ contains all integers of the form $c + km$ for some non-negative integer k. The author claims that his lattice framework converges in polynomial time, presumably in the size of the source program; it should also be noted that there is a factor of $\log(m)$ in the time bound [Gra91].

To find the number of synchronizations that may occur *before* and *after* each program point in a terminating program, both forward and backward analyses are conducted. If the result of the latter analysis for any program point p is the empty set, then there is literally *no* number of synchronizations that can occur after p. Thus, the program cannot terminate if p ever executes.

When the frameworks reach a fixed point, the congruences of the solution describe how many times each task has accessed each channel at each point in the task. By intersecting the congruences of a read command with the union of congruences for all write commands for the same channel (and vice versa), the congruence information can be refined. If the intersection for a particular synchronization is empty when the summaries for m and n are intersected, then m can never happen together with n. Mercouroff uses this information to refine his representation of the communication structure of the program.

Mercouroff does not propagate *CHT* information between tasks, nor does he refine *CHT* iteratively in any way. This would be an interesting extension to his analysis. It would also be interesting to compare the accuracy of his methods with ours.

It is not clear that Mercouroff's analysis will generalize to synchronization other than channels like those of CSP. Mercouroff's analysis depends heavily on finding expressions for the number of channel accesses in the entire program, based on information available within a single task. It is not clear how this could be extended to communications systems in which more than one task could call the same entry, i.e., Ada.

A demonstration analyzer, based on Mercouroff's methodology, is available for the

---

with a congruence.

Macintosh (tm) computer.

10.4.4 Classification of determinacy: Emrath and Padua [EP88].

In 1988, Emrath and Padua [EP88] discussed a method of race detection in Cedar Fortran [GPHL90] programs, and gave a classification of the types of nondeterminacy which could appear in the program. The authors' aims include both static and dynamic race detection; their work on dynamic races is continued in [EGP89].

The authors' categorization of programs, according to their determinacy for a given input, is:

- *Determinate.* The program produces the same output for a given input.
  - *Internally determinate.* The program has the same trace for a given input.
  - *Externally determinate.* The program produces the same output for a given input, but might not always have the same trace. For instance, additions of integers may be ordered arbitrarily within the trace, but still produce the same output (unless there are overflow errors).
- *Non-determinate.* The program does not always produce the same output for some inputs.
  - *Input-dependently non-determinate.* The program is determinate for some inputs, but not others.
  - *Conclusively non-determinate.* The program is always non-determinate, regardless of its inputs.

  Orthogonal to these categories are:
  - *Associatively non-determinate.* The ordering of operations which are mathematically associative, but which may be influenced by round-off error (such as addition), causes different outputs. This effect may be acceptably small in some cases, so that it is not always considered an error.
  - *Completely non-determinate.* Non-determinacy is caused by some problem other than round-off errors. This is usually regarded as an error.

These categorizations of determinacy may be useful for understanding deadlock, since whether a program deadlocks might (or might not) depend upon its inputs.

Emrath and Padua propose to statically detect both *definite* and *possible* races. In the former, the static tool can infer that there will be a race at run-time; in the latter, there might be a false alarm. If possible races are detected, the authors propose to instrument the program and detect races at run time [EGP89]. Our CHT analysis can, likewise, sometimes infer the possibility of a definite infinite wait or unreachable code anomaly, when some node n is in $CHT(n)$. All other anomalies must be considered possible.

To analyze the execution order of statement instances, Emrath and Padua construct an *ordering graph* representation of the program. Nodes represent synchronization statements; edges represent orderings between them. Control and synchronization edges are not distinguished from each other. Some edges are marked with distance vectors, similar to [CKS90]; an edge with no marking means that the corresponding distance vector is the zero vector.

Conflict arcs in the ordering graph represent potential races. Between each pair of nodes m and n, in which a variable v is accessed by both nodes and is defined in at least one, a conflict arc (m, n) is drawn. A conflict exists between m and n if a write to $v(\vec{X})$ at m may happen together with an access $v(\vec{X} + \vec{\delta})$ at n. The arc (m, n) is thus marked with a distance vector $\vec{\delta}$, corresponding to the conflict distance between m and n.

Given the conflict arc (m, n) with distance vector $\vec{\delta}$, we wish to prove that the conflict arc does not result in a race, i.e. that $m_{\vec{X}}$ is ordered with respect to $n_{\vec{X}+\vec{\delta}}$. This is done by finding a path from m to n, consisting of control and synchronization arcs whose distance vectors sum to $\vec{\delta}$. If such a path is found, then no race exists between m and n. The authors do not detail an efficient methodology for testing whether such a path exists.

In some cases, the race-detection methodology of [EP88] might be applied to estimate CHT; we could establish that $m \in CHT(n)$ by establishing that there is a path of ordering arcs of length δ from m to n, and one of length $\vec{\delta} + (0, ..., 0, 1, 0, ..., 0)$ from

n to m (where the 1 is in the position denoting m's innermost loop). It is not clear that this can be done efficiently.

10.4.5 Balasundaram and Kennedy [BK89].

Balasundaram and Kennedy [BK89] work on detection of race conditions in programs using post-wait synchronization, as do Callahan and Subhlok [CS88]. Starting with a simple model of unconditional task execution, they progressively expand it to allow branching and loops with event arrays. In the process of doing this, they outline a non-concurrency computation that may be viewed as a somewhat primitive variant on [CS88], but which includes at least some loops. Deadlock detection is discussed briefly.

The *program execution graph* is a condensed flow graph of a program; `wait` statements begin the basic blocks in the program execution graph, and `post` statements end them. (Either the `post` or `wait` statement may be omitted, but each basic block must contain one of the two.) Directed synchronization edges are also drawn from every `post(EV)` statement to every `wait(EV)` statement for the same event variable `EV`.

Given the program execution graph, the authors define the *co-graph* as its "negation;" an undirected edge (m, n) exists in the co-graph iff there is no path between m and n in the program execution graph. In a program without branching or loops, edge (m, n) is in the co-graph iff m may happen together with n; the co-graph thus represents the negation of CHT.

Later, the model is augmented by adding conditional branching, loops, and multiple `post(EV)` statements; the modifications to the model discussed for each of these additions is safe, but may lose information. For instance, if the program contains multiple `post(EV)` statements, then all synchronization edges on `EV` are ignored when constructing the co-graph. However, if the `post(EV)` statements are all in one task and postdominate n in control flow, we could add the corresponding edge from n to the `wait(EV)` statements, and construct the co-graph accordingly.

The authors discuss the problem of deadlock detection, and mention that their co-graph building algorithm may be modified to detect deadlock cycles in programs

without control loops. However, their detection algorithm is somewhat naive, and would probably raise many false alarms.

Balasundaram and Kennedy handle non-nested loops without branching in the next section. They peel one iteration, to represent "scalar" `post` and `wait` statements (those not referencing event arrays). The scalar `post` and `wait` statements are then removed from the loop body. The analysis requires that all event array references be made through induction variables.

The authors form a *safe distance matrix*: if entry (m, n) of the matrix is the integer k, then there have been at least k more instances of m than n at any point in program execution. The authors give an example, but not a general algorithm, in which the safe distance matrix can be used to detect the possibility of deadlock.

10.5 Formal proof methods.

In this section, we examine some proposed methods for formally proving that programs are free of deadlock. The great advantage of these proofs is their accuracy: they have the potential for deadlock detection without false alarms. No static analysis method can have such precision, since any method which is guaranteed to terminate must discard some information that might be used to prove deadlock freedom.

Unfortunately, formal proof methods also have many disadvantages. Any general algorithm to prove program termination must solve the Turing halting problem; thus, formal proof methods cannot be fully automated by any algorithm that is guaranteed to terminate. Programmers are called upon to supply additional information, such as loop invariants, for use in correctness proofs. Even though software assistance for the proofs is available, the proofs themselves are tedious, and require highly trained experts to be done at all. If the program is changed, then the proof must be re-done to reflect the changes.

The worst problem is that, in the current state of the art, proof of freedom from deadlock depends upon state enumeration. This means that a set of deadlock states must be enumerated, and then the programmer must prove that these states are not

reached. Generally, a separate proof must be made for each state. Since there may be exponentially many states, there may also be exponentially many proofs. For programs with complex synchronization, this makes formal proof of deadlock freedom totally impractical at the present time.

Concise overview.

Avrunin et. al. ([ADWR86], Section 10.5.1) outline a proof technique which represents statement execution counts as regular expressions with additional constraints. Dillon ([Dil90], Section 10.5.2) presents a proof method which first analyzes local program units, then combines them for a global proof of correctness. The work of Apt ([Apt83], Section 10.5.3) is related to both of these; he also enumerates potentially blocked states to prove they cannot be entered. Section 10.5.4 gives a concise summary of model-checking approaches.

10.5.1 Constrained regular expressions: Avrunin, Dillon, Wileden, and Riddle [ADWR86].

Avrunin, Dillon, Wileden, and Riddle [ADWR86] propose the use of *constrained expressions* to express concurrency. A constrained expression consists of a regular expression, which generates concurrency histories, and a set of constraints to filter out impossible histories. The language generated by a constrained expression represents a superset of the set of possible concurrency histories of the program.

Constraints are written as regular expressions, with an additional *shuffle operator*, denoted †, representing the arbitrary interleaving of zero or more copies of its string argument. (Unfortunately, the authors do not show how † is used to constrain concurrency.) Associated with each constraint is a subset of the regular expression's alphabet, called the *constraint alphabet*. Given a string in the regular language, the string is in the constrained language iff, after deleting all letters not in the constraint alphabet, the string is in the language of the constraint.

Tasks are translated into *task expressions*, representing their possible sequences of

instructions. The interleavings of the possible sequences of events for the tasks is a superset of the possible sequences for the program. Constraints are added to this interleaving, to insure that all rendezvous observe the correct semantics.

If the resulting language contains a string which indicates, for instance, deadlock, then the program may be able to deadlock. Proving deadlock freedom involves proving that the constrained expression cannot generate any such string; such proofs cannot currently be automated, and would have to be re-done each time the program is modified to correct an error. The authors propose a software tool set to aid in making the proofs.

10.5.2 Isolation-based proofs: Dillon [Dil90].

Dillon [Dil90] proposes a methodology for formally proving safety properties, including freedom from deadlock, for Ada programs. The method is based on using symbolic execution of the isolated tasks to generate *verification conditions*, which are then combined for the whole program and proven to assure that the program meets its safety specifications. The resulting technique is very accurate, but cannot be fully automated.

Dillon uses an *isolation-based* approach to proof, in which the program is divided into units that are analyzed separately and then combined. The process starts with the source program, which is annotated by the programmer with local assertions, a global invariant condition, auxiliary variables used in the proof, and brackets to delimit places where the global invariant may be temporarily violated.

Each program unit (procedure, task, or accept statement) is symbolically executed separately, to produce a *symbolic execution forest* for the unit. The unit entry, and the entry to each loop in the unit, define a tree in this forest. The nodes represent task states; each task state includes the symbolic store (*evaluation map*), *path condition*, and location counter at a specific point in the unit. Edges represent transitions from one program point to the next.

Assertions are evaluated symbolically, with respect to the evaluation map, and are usually proved by showing that the path condition implies the assertion. The leaf nodes of the symbolic execution trees are associated with exit or loop assertions; a

proof of these assertions constitutes the *local proof* of correctness. The final step in the partial correctness proof is proving that the units cooperate to maintain the global invariant. The global invariant and local path conditions can then be used to prove *safety assertions* about the entire program.

Dillon introduces the *multitask state*, which is a vector of task states, in the proof of safety assertions. A *control set* is a set of multitask states (or, alternatively, a predicate which is true for some set of multitask states). A safety assertion is valid for a control set if the verification conditions for each multitask state in the control set is a theorem.

For the example of deadlock, Dillon defines a *blocking assertion*, a predicate which is true if all tasks in the program are blocked or terminated and at least one is blocked. (Thus, Dillon's notion of deadlock encompasses all infinite wait anomalies.) If the intersection of the path conditions for each task state in each multitask state disproves the blocking assertion, then the program cannot deadlock. Dillon also sketches example proofs of distributed termination, mutual exclusion, and absence of communication failure.

While Dillon's methods are precise, they cannot be automated at the present time, and must be applied by experts trained in formal methods. These proofs are likely to be time-consuming and tedious, even with computer assistance. Thus, they are unlikely to come into general use in the near future. Also, if the control set (or a useful superset) for the program cannot be described by a simple predicate, the number of proof steps required may explode combinatorially.

Later published work on constrained expressions [ADW89, ABC+91] describes the implementation of a concurrency analysis tool based on constrained expressions, and gives empirical results for small test programs.

10.5.3 Optimizing the proof process: Apt [Apt83].

In [Apt83], Apt proposes a formal proof technique for freedom from infinite waits (which he calls deadlocks). Apt's technique is related to both [Dil90] and [ADWR86]; proof of freedom from infinite wait depends upon enumerating the *blocked configurations* (infinite

wait states) of the program. For each of these blocked configurations, the programmer is required to prove that the conjunction of assertions at each program point in each task in the configuration is logically inconsistent.

As in [ADWR86], Apt uses regular expressions to prune unreachable configurations. Apt also proposes to optimize the number of blocked configurations by dividing the program into *communication-closed layers*, which is equivalent to the parceling optimization made in [Tay83b]. This reduces the number of blocked configurations, thus easing the proof task. However, the value of this technique has not been demonstrated in practice; it is not known whether the majority of programs can be decomposed into communication-closed layers. The disadvantages of Dillon's technique also apply to Apt's: lack of full automation, tedium, the need for trained experts, the need to re-prove if the program changes, and the possibility of combinatorially complex proofs.

10.5.4 Model checking.

The *model checking* approach [CES86] has been applied to verify properties of concurrent systems, especially protocol validation. Model checking is closely related to formal proof methods; unlike the other formal methods given here, it can be automated (but it may suffer from combinatorial explosion problems).

Model checking might be applied to deadlock detection as follows. Suppose we can construct a formula f_0 which holds only in non-deadlocked concurrency states. A model checker would test that the formula holds globally by generating the state space of the program, then constructing all logic formulas of length $\leq |f_0|$ that hold in each state. Although model checking can be automated, it may be combinatorially explosive; much of the work in the field is aimed at minimizing the size of the state spaces that must be generated [BCM+90, CGL92].

10.6 Summary of static analysis methods.

Table 10.1 is a summary of the static and formal methods for analyzing synchronization. Time bounds are "Exp" for algorithms exponential in lines of source code, "Poly" for

polynomial algorithms, "N/A" for algorithms that cannot be fully automated. "Contribution or influence" refers to the contribution or influence each method has had on our work.

Of these synchronization analysis methods, only Saxena's work [Sax77] centers on safe static deadlock detection in polynomial time. However, due to the coarse "grain size" of his graph representation, we suspect that his algorithm may have accuracy problems in practice. Bristow et. al. [BDER79] proposed a method of infinite wait anomaly detection as a side issue, but it is also likely to have accuracy problems.

The remainder of the infinite wait anomaly detection algorithms require exponential (or worse) time. Taylor's state enumeration [Tay83b] provided a simple means to validate the deadlock analyzer implementation, and check both the accuracy of deadlock detection and *CHT* estimation. Many of the other state enumeration methods could also be used for this purpose.

Of the methods for non-concurrency analysis, probably that of Duesterwald and Soffa [DS91] is most closely related. Based on experimental evidence and theoretical review, we believe that their work has problems with both accuracy and safety. Nonetheless, their research had a large influence on the evolution of our *B4* analysis.

Bristow et. al. [BDER79] provided perhaps the genesis of the idea of iterative refinement of *CHT*. The work of Mercouroff [Mer92] is interesting because it is perhaps *least* closely related to the other non-concurrency analyses; an empirical comparison to our *CHT* analysis, or a combination of approaches, might provide a good line of future work.

10.7 Static intertask data flow analysis.

In this section, we examine work in static data flow analysis of rendezvousing programs, where data may be passed between tasks as arguments to the rendezvous. This area is related to our work in two ways. First, non-concurrency analysis may be used to eliminate some rendezvous between nodes that can never happen together. Thus, the corresponding parameter passing is eliminated from the data flow problem, improving

Method	Section	Time bounds	Information returned	Contribution or influence
[Tay83b]	10.3.1	Exp	Infinite waits; CHT_{perf}	Validation and accuracy benchmark for CHT and deadlock computation software
[YT88]	10.3.1	Exp	Infinite waits; CHT_{perf}	Partitioning in Concurrent C
[LC89]	10.3.1	Exp	Infinite waits; CHT_{perf}	
[McD89]	10.3.1	Exp	Infinite waits; CHT_{perf}	
[Sax77]	10.3.1	Poly	Deadlocks	Deadlock cycle pruning; nested critical sections
[BDER79]	10.3.1	Poly	Infinite waits; races	Iterative refinement of CHT
[ZYN86]	10.3.1	Exp	Deadlocks	
[MSS89]	10.3.2	Exp	Infinite waits	
[CU91]	10.3.2	Exp	Deadlocks; livelocks	
[KB90]	10.3.3	?	Livelock-induced starvation	
[CS88]	10.4.1	Poly	CHT	$B4$ problem formulation
[DS91]	10.4.2	Poly	CHT	$B4$ problem formulation
[Mer92]	10.4.3	Poly	CHT	
[EP88]	10.4.4	Exp	Races	
[BK89]	10.4.5	?	Races	
[ADWR86]	10.5.1	N/A	Proofs	
[Dil90]	10.5.2	N/A	Proofs	
[Apt83]	10.5.3	N/A	Proofs	
[CES86]	10.5.4	Exp	Model checking	

Table 10.1: Summary of related synchronization and non-concurrency analysis techniques.

the accuracy of the solution.

Second, we have developed here a set of lattice frameworks, capable of representing data flow problems where the solutions meet at nodes from edges of more than one type (i.e., control and sync edges). These lattice frameworks can often also be used to represent the data flow problems, enabling ready comparison of convergence properties with other problems so represented. The authors typically represent the solution process as an algorithm, which does not aid such comparison.

Concise overview.

Long and Clarke ([LC91], Section 10.7.1) describe an iterative data flow framework for intertask analysis of classical data flow problems (e.g., reaching definitions.) Reif and Smolka ([RS90], Section 10.7.2) give a more general framework for intertask data flow analysis, based on abstract interpretation. Moser ([Mos90], Section 10.7.3) gives techniques for finding data dependences statically within small regions of the program. Cheng ([Che92], Section 10.7.4) describes an extended version of the program dependence graph for use in concurrent programs.

10.7.1 Classic iterative analysis: Long and Clarke [LC91].

Long and Clarke [LC91] build upon the work of Taylor and Osterweil [TO80], in that they divide programs into components which are each analyzed only once. The components that Long and Clarke analyze are called *fragments*; a fragment represents a task, a loop body, or the code contained inside an `accept` statement. A *rendezvous graph*, representing invocation ordering between fragments, is used to determine the proper order of analysis for the fragments.

A control flow graph corresponding to each fragment is created during the analysis. Since accept statements may be nested, so may fragments. The invocation of a nested fragment is represented by a single node in the control flow graph for the containing fragment.

The nodes in the rendezvous graph represent fragments or procedures; edges represent invocation. Long and Clarke restrict themselves to acyclic rendezvous graphs, though they mention that techniques such as node splitting can sometimes be used to accommodate cyclic rendezvous graphs.

When a fragment is reached in the containing code, Long and Clarke treat this as an *implicit procedure call* of the fragment by the containing code. In contrast, the fragment is *explicitly* called by the entry call statements corresponding to its accept. Fragments are thus said to have two types of parameters: *explicit parameters*, which are passed between the fragment and its entry calls, and *implicit parameters*, which reach the fragment from its containing fragment.

Summary information ("gen" and "kill" sets) are kept for the explicit and implicit parameters of each fragment. Once the summary information for the fragment has been determined, it can be instantiated in its explicit and implicit callers.

Fragments are analyzed in reverse topsort order of the rendezvous graph; thus, the analysis for calling fragments may make use of that for called fragments. The summary information for the explicit and implicit parameters of a nested fragment are used to represent the effect that it has upon the solution for the containing fragment or entry calls.

Long and Clarke use two classic data flow problems, available expressions and live variables, as example analyses. From these analyses, they can also detect several data flow anomalies, including uninitialized variables, dead definitions, and definitions that do not influence the output of the program.

If CHT information were available, the accuracy of Long and Clarke's solution could be improved for analysis at entry calls. Assuming that several entries had the same signal type as a call c, we need find the meet of the intertask solution only over those entries that are not in $CHT(c)$.

Implicit in their summary process, Long and Clarke assume that data flow information cannot propagate from external to internal calls, or vice versa. Because they make this assumption, they need not use multiple-edge lattice frameworks. However, this

assumption may be violated in special cases, lowering the accuracy of their solutions. For example, variables in calling tasks may be live, through parameter passing in entry calls, to a use in the called task following the accept statement. This case will be missed by their analysis.

10.7.2 Abstract interpretation: Reif and Smolka [RS90].

Reif and Smolka [RS90] develop a broad framework, based on abstract interpretation, for data flow analysis in a system of communicating processes. The processes communicate via non-blocking TRANSMIT statements; synchronization behavior is thus similar to that for the general semaphore model. The authors develop a *flow graph model* of program representation and execution, similar to our sync graphs and execution models. From the flow graph of a program, an *event spanning graph* is produced and used to guide the data flow analysis. The authors discuss data flow analysis in the presence of both *static* and *dynamic communication*; the difference is that channel references corresponding to a single expression may change during execution in the latter, but not in the former. This corresponds roughly to the difference between Ada and Concurrent C.

The flow graph of [RS90] is essentially a control flow graph in a limited language. Nodes represent program statements of four kinds: assignment, TRANSMIT, RECEIVE, and no-op. Directed edges represent control transitions between the statements. Reif and Smolka also develop a formal semantics of their programming model.

The event spanning graph of a program in the [RS90] model represents "enable to proceed" relations between statements, and resembles a DAG which spans the sync graph of a binary semaphore program. Directed *message link* edges are included from TRANSMIT to RECEIVE nodes for the same channel (i.e., signal) reference. Reif and Smolka generate the event spanning graph by an algorithm which simulates the execution of the program until all nodes have been visited once.[6] The topological order

[6]The algorithm apparently does not function correctly for some special cases involving TRANSMIT statements in loops. In these cases, the TRANSMIT must be executed more than once to enable a single RECEIVE statement to be reached in control flow, and to then complete execution.

of the event spanning graph forms a node list for data flow analysis.

The data flow analysis resembles abstract interpretation. Predicates for the specific problem being solved are propagated through the flow graphs, in the topological order of the event spanning graph. The predicates represent strongest post-conditions about the program variables. Section 4 addresses data flow analysis in the presence of static communication between processes; section 5 extends this to dynamic communication.

Non-concurrency information could not be directly used to refine the solution, because the TRANSMIT statement in the model of [RS90] does not block. Instead, we would have to solve a "message kill" problem, similar to *SemKill*, for that purpose.

Reif and Smolka use a ρ_n function for each RECEIVE node n, to achieve the same effect as our multiple-edge-type data flow frameworks. Our frameworks could directly subsume their ρ functions, encoding their analysis as a more standard data flow framework.

10.7.3 Data dependency graph construction: Moser [Mos90].

Moser, in [Mos90], outlines a technique for constructing the data dependency graphs of Ada programs, based on composition of subgraphs during parsing. The expected operational advantage of doing this is that the graph may be obtained for a small portion of the program at any time. Thus, the large space and time complexity needed for dependency analysis of the whole program may thus be avoided in localized, intraprocedural optimizations.

It is possible that a local analysis of non-concurrency properties might also be of use in dependence analysis and optimization. For example, locally available *CHT* information might eliminate some spurious data dependence edges from the graphs of [Mos90]. However, none of the four *CHT* refinements listed in Chapter 5 are safe for local analyses; more work is needed in this area to identify locally safe means of propagating *CHT* information between tasks.

10.7.4 Task dependence nets: Cheng [Che92].

In [Che92], Cheng augments the standard program dependence graph (PDG) model [FOW87] to represent task interactions. He calls the resulting model a *task dependence net (TDN)*, and proposes to use it "to develop a unified representation tool useful in knowledge-based debugging of concurrent Ada programs."

In both the TDN and PDG, nodes represent program statements; in the TDN, they also represent compound statements such as task and procedure bodies and calls and loop bodies. Edges represent possible dependences of one statement on another.

The TDN has edges of two types not used in the PDG. The first added edge type represents *communication dependence*, or data dependence between statements in in different tasks. The second added edge type represents possible *synchronization dependence*; edge (m, n) exists if m is in a different task from n, and the termination of m may directly influence whether n starts or terminates. In a sense, synchronization dependences should thus be akin to our completor edges, since both represent relations in which the start of a node in one task is enabled by the completion of a node in another. However, some of the synchronization dependence edges of their example indicate a direct synchronization relationship (e.g., (f_8, p_9)); others have no obvious explanation (e.g., (t_5, p_2)).

Like the PDG, the TDN can have either a static version, representing all possible dependences, or a dynamic version, representing the dependences that occur in an actual run of the program. Cheng proposes to use the TDN in conjunction with Weiser slicing [Wei84] of either the static or dynamic TDN as appropriate, to aid in program debugging.

10.8 Dynamic deadlock detection.

Dynamic deadlock detection has the same major goal as static deadlock detection: to identify the problem of deadlocks so they can be remediated. Dynamic detection is more precise than static, since it will report deadlock only when one actually occurs. However, dynamic methods are perforce unsafe; they will not identify deadlocks until

they are actually executed, and thus some programs with potential deadlocks may slip through the testing process.

Besides this overall goal, there are two main relations between dynamic deadlock detection and our methods. Most of the dynamic methods surveyed use some form of deadlock cycle detection; many also make use of program representations similar to the sync graph.

Concise overview.

Knapp ([Kna87], Section 10.8.1) surveys the field of dynamic deadlock detection. Although his area of interest is in distributed databases, many of the methods he surveys can be adapted to concurrent programming applications. Helmbold and Luckham ([HL85], Section 10.8.2) describe an interactive debugger with facilities to present and analyze "deadness errors" in Ada programs. Shih and Stankovic ([SS90], Section 10.8.3) list a hierarchy of models for deadlock in Ada programs, and give algorithms for deadlock detection in two of these models. Last, Chen et. al. ([CKU91], Section 10.8.4 employ an algorithm designed for database use to detect deadlocks in Ada programs.

10.8.1 Survey of dynamic deadlock detection: Knapp [Kna87].

Edgar Knapp surveys the area of dynamic deadlock detection in [Kna87]. Although the surveyed work focuses upon applications in distributed databases, much of it is applicable to the problem of detecting deadlock in any concurrent system, including programs or operating systems. Here we overview the survey, and describe similarities between our static deadlock detection method and one of Knapp's classes of dynamic methods.

Knapp presents a general model of dynamic databases, which is intended to apply to all the methods he reviews. In Knapp's model, a set of *transactions* request, hold, and release *resources* during their operating lifetime. Transactions may be partitioned into separate, concurrent *transaction agents* and distributed across processors. Thus, deadlock can occur when some agents are mutually waiting on each other to release

resources.

The relations of transaction agents waiting on each other are described by their *waits-for graph (WFG)*. The WFG is somewhat akin to the general resource graph structure described by Holt [Hol72]. Nodes in the WFG represent agents; a directed edge (m, n) exists if m is blocked and is waiting on a resource held by n. If the system deadlocks, there is a cycle in the WFG; this is a necessary condition for deadlock, but may not be a sufficient one, depending on the complexity of the transaction model.

Knapp classifies deadlock detection algorithms according to the complexity of the waiting queries to which they can be applied. In order of increasing complexity, they are:

- *One resource*: Only one resource is shared.
- *AND*: Multiple resources are shared, and an agent may be simultaneously waiting for all of several resources to be released.
- *OR*: An agent waits on any of several resources. The difference in complexity between the AND and OR models is that, while a cycle is a sufficient condition for deadlock in the AND model, it is not sufficient in the OR model. Instead, we must detect a *knot* (a cycle without exiting edges).
- *AND-OR*: Agents individually either wait on any or all of several resources to be released. The AND-OR model has been used as a framework to represent the Ada rendezvous [SS90, CKU91].
- $\binom{n}{k}$: Agents may specify that they wish to obtain any k resources out of a pool of n resources, where n is fixed but k may vary. This generalizes the AND-OR model.
- *Unrestricted:* Agents wait on arbitrary combinations of resources.

The algorithms to detect deadlocks can likewise be either *centralized* or *distributed*.

Knapp further classifies the deadlock detection algorithms in broad classes determined by their method of operation:

- *Path-pushing*. These algorithms build a local version of the WFG at each site, and pass it to another site for further elaboration. This process continues until some

site has enough information to announce or disprove the existence of a deadlock. As it is constructed, the WFG is effectively "pushed" along the deadlock path.

- *Edge-chasing.* These algorithms propagate *probe messages* along the edges of the WFG. When a site receives a version of a probe message that it has sent, it can announce a deadlock. Our static deadlock algorithm may be viewed as similar to an edge-chasing algorithm; *SACnt* summaries would correspond to the probe messages.
- *Diffusing computations.* If a deadlock is suspected, a computation is initiated at the root of the WFG. Although the WFG is never built explicitly, the structure of the computation reflects the structure of the WFG; it grows by sending *query* messages and shrinks by receiving *replies.* If the computation shrinks back to just the root node, a deadlock is reported.
- *Global state detection.* The idea is to determine a *consistent global state* without freezing the computation, and detect deadlock in that state. The consistent global state is derived using a "happened before" relation, similar to that of Lamport [Lam78].

In Section 5, Knapp notes that many of the published deadlock detection algorithms in use have errors, and that, "only rigorous proofs, using as little operational augmentation as possible, suffice to show the correctness of these algorithms." Presenting a short comparison of the complexity of the algorithms, he concludes that, "the differences in message complexity of the algorithms have turned out to be less significant than expected," and advises, "it is ... advisable — in order to avoid unnecessary complexity — to choose the least general technique that is still general enough to solve the problem at hand." Our experiences have borne out both of these assertions.

10.8.2 Deadlock and interactive debugging: Helmbold and Luckham [HL85].

Helmbold and Luckham [HL85] describe a debugger with facilities for dealing with "deadness errors" in Ada programs. A program to be monitored is instrumented with

calls to a monitor task; the monitor task keeps track of the state of each task, and may be queried by a user. The monitor offers features such as single-stepping through task interactions, taking "snapshots" of the global state of the program, maintaining histories, and dynamic deadlock detection. An extended example with their debugger shows how such a tool might be used in practice.

The authors note that, "dead state descriptions can become unmanageable and contain much that is irrelevant to describing the real error." This observation is important for static deadlock detection, since information on potential deadlocks must eventually be presented to the programmer for debugging. Helmbold and Luckham propose collecting and displaying additional task state information; of their suggestions, the one most relevant to static deadlock detection is that "it becomes important to know at which call or accept a statement is waiting." Our algorithm can present the potential head nodes of a deadlock cycle without modification, and therefore addresses this concern.

10.8.3 Real-time systems, resource allocation, and dynamic deadlock: Shih and Stankovic [SS90].

Shih and Stankovic [SS90] discuss the problems of dynamic deadlock detection in programs of several levels of complexity, in the terms of [Kna87]. They are concerned with resource acquisition, as well as the rendezvous mechanism, in deadlock detection, and also characterize the types of deadlocks occurring in real time programs. Finally, they give edge-chasing algorithms for deadlock detection for problems in the two lowest levels of complexity.

The authors enumerate a hierarchy of Ada deadlock models, along the lines of the problem complexity levels of [Kna87]. With each of these models is associated a set of restrictions on the Ada programs that can be handled. For instance, in the single-resource model, resources must be non-sharable, only one outstanding request is allowed, and at most two tasks may be involved in either a rendezvous or task termination.

In real-time systems, the authors divide deadlocks into *stable* and *temporal* deadlocks. Stable deadlocks last forever; temporal deadlocks will eventually time out because of a `delay` alternative in the deadlock. It is the authors' goal to avoid reporting temporal deadlocks, as long as they do not cause any task to miss a deadline. A third class of potential problems is *non-deadlocked blocking*, which includes livelocks, starvation ("orphan tasks," in the authors' terminology), and normal blocking conditions.

Following [Hol72], Shih and Stankovic represent dynamic Ada task activity using a *general resource graph (GRG)*. The GRG is a generalization of the task waits-for graph (TWFG). Vertices represent tasks and resources; each edge represents a *request* for a resource, an *assignment* of a resource to its holder, or a *production* of a consumable resource. For each of the complexity levels discussed, the authors characterize the sufficient conditions on the GRG to report deadlock. For instance, a cycle is sufficient to report deadlock in the AND model, but not in the OR model, where knots must be detected.

For the single-resource and AND models, the authors present dynamic edge-pushing algorithms to update the GRG and detect deadlock. The probe messages maintain information on the soonest timing constraint of any task in the path; this is the *probe deadline*. The probe deadline is maintained to eliminate false deadlock cycles in real-time systems, where tasks may time out and enable others.

10.8.4 Cheng, Kasahara and Ushijima [CKU91].

Cheng, Kasahara and Ushijima [CKU91] describe a dynamic deadlock detector for Ada, based upon the task waits-for graph (TWFG) representation of program tasking. See Section 10.2.3 for a description of the authors' version of the TWFG. The paper's contribution is to adapt technology developed for dynamic deadlock detection in databases and operating systems to the authors' own model of Ada tasking.

The authors give an algorithm to detect deadlock. It operates by updating the TWFG dynamically as the program executes, and calling another algorithm upon appropriate updates to actually detect the deadlock cycle. The authors do not present a

deadlock cycle detection algorithm; rather, Hermann and Chandy's algorithm [HC83] is suggested for this purpose. It is detailed briefly in [Kna87], from which much of the terminology of the paper is borrowed. Note that, while [HC83] is a distributed algorithm, the implementation of [CKU91] uses a centralized monitor task.

Section 3 of [CKU91] describes the implementation, which is called EDEN. The authors instrument the object program to collect tasking events and pass them to the EDEN run-time monitor. This instrumentation changes the semantics of the program slightly, in order to allow the EDEN monitor to refer to tasks during their activation (Section 3.2). Information on what tasks may call what other tasks' entries must also be maintained by EDEN, so that the TWFG can be updated. The authors show a script for a single run of EDEN, but do not provide performance data.

10.9 Conclusions.

Most of the work in static deadlock detection that we have seen relies upon concurrency state enumeration and/or formal proof methods. The former may be combinatorially explosive; the latter cannot be fully automated and requires tedious proofs "by hand." We avoid both of these problems. The small fraction of previous work dealing with static polynomial-time deadlock detection methods has had problems with safety or accuracy, which our algorithm framework alleviates.

We have shown how *CHT* and multiple-edge-type lattice frameworks can be applied to intertask data flow analysis. Last, we have presented references to the dynamic methods, which, like our own static method, detect cycles of tasks waiting for each other.

Chapter 11

Future work.

11.1 Future work in static deadlock detection.

This chapter describes some possible avenues for future investigation, continuing the research presented here. An (incomplete) list of these research directions includes:

- **The Ada model.**
  - *Improvements to the analyzer front end.* The parser and sync graph generator of the ggen program is a research tool; the user is required to inline, by hand, any subroutines, functions, packages, and generics that incorporate tasking, and to merge source files. In a more complete analyzer, this could be done automatically.
  - *Efficiency improvements to the analyzer.* For instance, the representation for *SACnt* records is known to be larger than necessary.
  - *More complete modeling of Ada tasking.* In particular, we would like to directly model subprograms with included tasking, dynamic task creation, and the `abort` statement.
  - *Modeling Ada 9X synchronization.* The Ada 9X draft mapping document [Ada92] specifies further additions to the tasking model. These additions include non-blocking signaling, asynchronous multicast, support for large numbers of tasks, and statement-level parallelism. The impact of these additions upon static deadlock detection should be investigated theoretically and experimentally.
- **Binary semaphores.**
  - *Acquire test programs; implement and test algorithm.*
  - *Expand semaphore kill information* by weakening the preconditions necessary for semaphores to be killed.
  - *Extend model to full semaphores.* This may help in the task of acquiring test data, since many operating systems provide semaphores as synchronization primitives.
- **Concurrent C.**

  - *Develop algorithms for required data flow analysis.* An actual implementation of deadlock detection for Concurrent C will require an algorithm to perform approximate reaching definitions and def-use chains (Section 9.1.1).
  - *Acquire test programs; implement and test algorithm.*

- **Other synchronization paradigms.** If there are synchronization paradigms for which we can obtain a substantial amount of test data, these should be investigated for deadlock detection. The communication and synchronization primitives used in distributed operating systems might be a reasonable choice.

- **Improving the accuracy of deadlock detection.**

  - *Modeling memory state.* The performance of the deadlock analyzer might be improved by using additional semantic information. For example, it might be possible to deduce that conditionals are linked within, or between tasks [SMHY92], thereby improving the *CHT* relation.
  - *Graph reduction techniques.* Tu, Schatz, and Murata [TSM90] proposed a set of graph reduction techniques to reduce the complexity of state enumeration in Petri nets. We might perform similar reductions to reduce the complexity of polynomial deadlock detection in sync graphs.
  - *Directed state space enumeration.* When our algorithm reports a potential deadlock, it can also be made to report a set of deadlock head nodes. It might be possible to use this information to direct a state space enumeration algorithm, generating only those parts of the state space which may contain deadlocks. This may improve the execution time of state space enumeration.

- **Non-concurrency analysis.**

  - *Generalizing critical sections,* as outlined in Section 5.1.5.
  - *Propagating pinning information* along control and sync edges. An example is given in Section 5.1.2.
  - *Congruences as a CHT refinement.* The work of Mercouroff [Mer92] uses congruences to detect non-concurrent synchronization statements in the CSP

language. Future work may investigate the generalization of Mercouroff's method to other types of synchronization, and the use of *CHT* information to improve the congruence analysis.

- *Other uses for non-concurrency information.* Non-concurrency analysis has potential uses in areas other than deadlock detection, including optimization, data flow analysis, and dead code elimination. We have described some of the possible uses of *CHT* information in concurrent programming, in Section 11.2. An empirical investigation of its effectiveness would be worthwhile.

- **Detecting other synchronization anomalies.** Under the assumption that control paths in any task are taken independently of paths in any other task, any program containing synchronizations inside a loop can starve. In practice, any starvation detector making this assumption would always indicate a possible starvation in programs with such structures. Most often, these would be false alarms.

  With additional modeling of memory states, though, it may become possible to link conditionals and iteration counters between tasks. If this can be done, it may then be possible to reliably detect starvation with reasonable accuracy. Livelock detection, though theoretically undecidable, might also yield to memory state modeling for many common programs.

11.2 Additional applications for non-concurrency analysis.

Information about guaranteed non-concurrency has many possible uses in compilation and debugging, apart from static deadlock detection. Specifically, it is useful for data flow analysis [RS90, LC91], rendezvous optimization, dead code elimination, and race anomaly detection. In this section, we briefly outline some of these expected benefits. This area will likely be an important one for future research.

Debuggers and program understanding tools.

In debugging and hand-optimizing concurrent programs, it may be valuable to know that some pair of code segments can never happen together. For example, if the code segments that access a shared resource are non-concurrent, the programmer can safely delete any locks that are protecting the resource. The sophisticated programmer will doubtless find many other uses for such a feature. Thus, non-concurrency analysis may become a valuable part of program understanding tools.

We could envision, for example, a debugger that includes a structured source code editor. On the user's command, the editor would highlight all code that might be concurrent with the code at the current cursor position. Alternately, the debugger could answer queries about whether two statements may be concurrent.

Data flow analysis.

We take the example of def-use associations. If tasks pass data to each other during rendezvous, non-concurrency information can eliminate some infeasible def-use pairs. Statements which are non-concurrent cannot rendezvous with each other; no definitions can reach uses via rendezvous which can never occur. Similar examples exist for other classic data flow problems.

Optimizing rendezvous.

Rendezvous can be optimized by choosing the proper implementation for a particular statement instance. In a simple form of rendezvous synchronization, signaling tasks call an accepting task; both wait until the call is completed. Each signaling task may call the accepting task from more than one call point; the accepting task may also accept the calls at any of several entry points.

Entries might be implemented in two alternate ways: by polled flags and by call queues. Polling will be more efficient when few call points can execute simultaneously with the entry; beyond some threshold, the time to poll will be greater than the overhead for queue management. Non-concurrency analysis might determine that this threshold

has not been reached for a particular entry, even though the total number of call points is greater than the threshold. If so, we can gain speed by using polling.

Dead code elimination.

Dead code can also be detected using non-concurrency information. If some statement in a task can never happen together with *any* other statement in the program (including the beginning and end of the program), then it can never happen at all. The code may thus be eliminated from the program. If the code is dead because some `accept` alternative of an entry can never rendezvous, then the entry may also be optimized by removing the alternative.

In some cases, the code may be dead because some predecessor in the same task *must* starve or deadlock; here, the dead code is only a symptom of a more serious anomaly.

Race detection.

A *race* occurs whenever two tasks may simultaneously access the same shared object (variable or resource), and if at least one of those accesses changes the state of the resource (e.g., advances a stream or writes a variable).[1] If we can prove that two accesses are non-concurrent, then there can be no races between them. Callahan and Subhlok [CS88] apply this methodology to post/wait synchronization in programs without loops.

[1] To guarantee that the program produces determinate results, we must usually also guarantee that conflicting statement instances are *totally ordered*, rather than just non-concurrent.

Appendix A

Boolean matrix notation.

In this thesis, many formulae are given in the form of operations on Boolean matrices, somewhat after Cooper and Kennedy [CK84]. We find that this is both a succinct and precise way to explain many of the operations we perform on relations from objects to sets of objects; it is far less prone to misinterpretation than verbal explanations. This appendix provides intuition on how the Boolean matrix representation works, and explains some common idioms used later.

We do not contend that the algorithms described in this thesis should always be implemented as Boolean matrix operations; efficiency would often dictate otherwise in a practical implementation. We describe the Boolean matrix operations for notational purposes only.

A.1 Subtraction, negation, union, and intersection.

An m by n Boolean matrix typically represents a function from a universe of m elements, to sets from a universe of n elements total. For instance, we might wish to construct a function from each task to the nodes in the task, for the program of Figure A.1. The corresponding matrix (call it T) is of size $|Tasks| \times |N|$ (ignoring the begin and end nodes), and is as follows:

$$T = \begin{bmatrix} 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 \end{bmatrix}$$

The bit at the i'th row and j'th column of T is denoted $T[i,j]$.

We may subtract matrices of like dimensions. For instance, suppose we have a matrix F, which maps from each task to the set of nodes which may be the *first* to execute in the task. Then, row i of the matrix $(T - F)$ is the set of nodes which are in task i, and which *cannot* be the first to execute in task i:

$$T - F = \begin{bmatrix} 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 \end{bmatrix} - \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$

Note that, in Boolean subtraction, $0 - 1 = 0$.

Similarly, we may perform unary negation (or one's complement) on matrices; this

```
task body t1 is
begin
  t2.sig1;        -- (1)
  accept sig2;    -- (2)
  t2.sig1;        -- (3)
end t1;

task body t2 is
begin
  if Cond then
    accept sig1; -- (4)
    t1.sig2;     -- (5)
  end if;
  accept sig1;    -- (6)
end t2;
```

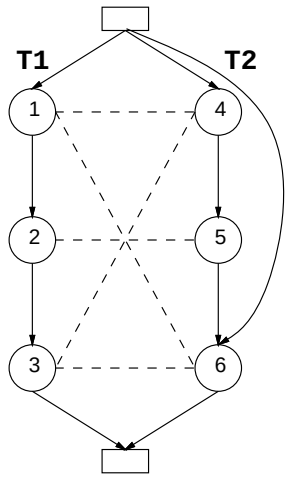

Figure A.1: An Ada program fragment and its sync graph.

is identical to subtracting the matrix from a matrix of all ones. The negated matrix has the opposite meaning from the positive matrix. For example, row i in matrix $-F$ is then the set of nodes which are either not in task i, or which are in task i but cannot be the first nodes to execute in i:

$$-F = \begin{bmatrix} 0 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 1 & 0 \end{bmatrix}$$

Note the difference in meaning between $-F$ and $T - F$.

Set union and intersection may be performed between matrices of corresponding dimensions; they correspond to element-wise "or" and "and" (Boolean $+$ and $*$) operations respectively. Continuing, we let L be, for each task, the set of rendezvous nodes that may be the *last* ones executed in the task. Then row i of $(F \cup L)$ is the set of nodes that may either begin or end task i, or may do both:

$$F \cup L = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 \end{bmatrix} \cup \begin{bmatrix} 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 \end{bmatrix}$$

Similarly, row i of $(F \cap L)$ is the set of nodes that may be both first and last in task i; the only bit set in $(F \cap L)$ is $[2, 6]$, denoting that node 6 is the only node that can both begin and end a task (and that that task is task 2).

A.2 Transpose.

The transpose of a matrix is denoted by a subscripted capital letter T. Transposing a matrix is the same as interchanging the domain and range of the function to which the matrix corresponds. For example, row j of T_T would be the set of tasks to which node j belongs:

$$T_T = \begin{bmatrix} 1 & 0 \\ 1 & 0 \\ 1 & 0 \\ 0 & 1 \\ 0 & 1 \\ 0 & 1 \end{bmatrix}$$

(T_T happens to be a singleton set for each node, though this is not a general case.)

A.3 Boolean matrix multiplication.

Our definition of Boolean matrix multiplication is the same as that for integer matrix multiplication, with Boolean operators substituted for integer ones. Thus, the bit $[i, j]$ of matrix $(M \times N)$ is 1 iff any of the 1 bits in row i of matrix M correspond to any of the 1 bits in column j of N.

Matrix multiplication corresponds to a form of function composition. In the more familiar notation of functions from objects to sets, suppose M has the signature $A \rightarrow 2^B$ and N has the signature $B \rightarrow 2^C$ for some sets A, B, and C. Then, using functional notation, for each $i \in A$,

$$(M \times N_T)(i) = \{k \in C \overset{\text{such that}}{} \exists j \in B \overset{\text{such that}}{} j \in M(i) \wedge k \in N(j)\}.$$

Continuing our example, suppose R represents the nodes with which each node can rendezvous. Then row i of the matrix $F \times R_T$ represents the set of nodes that can

rendezvous with some first node of task i:

$$F \times R_T = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 & 0 & 0 \end{bmatrix}$$

Note that since R is symmetric, we could have multiplied by R rather than R_T, saving a transposition. We often use this simplification in dealing with symmetric matrices.

A.4 Multiplication by a matrix of all ones.

Suppose we have an $m \times n$ matrix A. Multiplying A by an $n \times n$ matrix of all ones ONE yields a matrix such that, if any row i of A had any 1 bits, row i of $A \times ONE$ has all ones. This has many uses in practice, particularly in constructing masks to eliminate unwanted rows.

Suppose that we wished to find, for each task, the set of "first" nodes as before, but only for those tasks where no single node could be either first or last. If some node could be both first and last, we wish the set to be empty. The 2×6 matrix $(F \cap L) \times ONE$ has all zeroes in row 1 and all ones in row 2, since only task 2 has a node that can be either first or last (node 6). We can use the array $(F \cap L) \times ONE$ to mask F, giving us the array we want:

$$\begin{aligned} F - ((F \cap L) \times ONE) &= \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 \end{bmatrix} - \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix} \\ &= \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \end{aligned}$$

References

[ABC+91] G. S. Avrunin, U. A. Buy, J. C. Corbett, L. K. Dillon, and J. C. Wileden. Experiments with an improved constrained expression toolset. In *Proceedings of the ACM SIGSOFT '91 Fourth Symposium on Testing, Analysis, and Verification (TAV4)*, pages 178–187, Victoria, BC, October 1991.

[Ada92] Ada 9X Mapping/Revision Team. Ada 9X mapping volume I: mapping rationale version 4.1. Technical Report IR-MA-1249-2, Intermetrics, Inc., Cambridge, MA, March 1992.

[ADW89] G. S. Avrunin, L. K. Dillon, and J. C. Wileden. Experiments with automated constrained expression analysis of concurrent software systems. In R. A. Kemmerer, editor, *Proceedings of the ACM SIGSOFT '89 Third Symposium on Testing, Analysis, and Verification (TAV3)*, pages 124–130, December 1989.

[ADWR86] G. S. Avrunin, L. K. Dillon, J. C. Wileden, and W. E. Riddle. Constrained expressions: adding analysis capabilities to design methods for concurrent software systems. *IEEE Transactions on Software Engineering*, pages 278–291, February 1986.

[Ame83] American National Standards Institute. *ANSI/MIL-STD 1815A (1983) reference manual for the Ada programming language*. United States Government Printing Office, 1983.

[Apt83] K. R Apt. A static analysis of CSP programs. In *Lecture Notes in Computer Science #164 - Logics of Programs*, pages 1–17. Springer-Verlag, 1983.

[ASU86] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[Ban78] J. P. Banning. *A Method for Determining the Side Effects of Procedure Calls*. PhD thesis, Department of Electrical Engineering, Stanford University, August 1978.

[BCM+90] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. In *Proceedings of the Fifth IEEE Symposium on Logic in Computer Science*, pages 428–439, Philadelphia, PA, 1990.

[BCW88] R. A. Becker, J. M. Chambers, , and A. R. Wilks. *The New S Language*. Wadsworth and Brooks/Cole, Pacific Grove, CA, 1988. ISBN 0-534-09192-X.

[BDER79] G. Bristow, C. Drey, B. Edwards, and W. Riddle. Anomaly detection in concurrent programs. In *Proceedings of the 4th International Conference on Software Engineering*, pages 265–273, 1979.

[BK89] V. Balasundaram and K. Kennedy. Compile-time detection of race conditions in a parallel program. In *Proceedings of 1989 International Conference on Supercomputing*, pages 175–185, June 1989.

[BK90] V. Balasundaram and K. Kennedy. A mechanism for keeping useful information in parallel programming tools: the data access descriptor. *Journal of Parallel and Distributed Computing*, 9:154–170, 1990.

[Bur70] J. Burgess. Sex, honor, and the Italian driver. *Reader's Digest*, pages 110–113, September 1970.

[Cal88] D. Callahan. The program summary graph and flow-sensitive interprocedural data flow analysis. In *Proceedings of the SIGPLAN '88 Conference on Programming Languages Design and Implementation*, pages 47–56, June 1988.

[CAU88] J. Cheng, K. Araki, and K. Ushijima. Tasking communication deadlocks in concurrent Ada programs. *Ada Letters*, pages 61–70, Septenber/October 1988.

[CC77] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the 4th Annual ACM Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, January 1977.

[CD73] E. G. Coffman and P. J. Denning. *Operating systems theory.* Prentice-Hall, 1973. ISBN 0-13637-868-4.

[CES86] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, April 1986.

[CGL92] E. M. Clarke, O. Grumberg, and D. E. Long. Model checking and abstraction. In *Conference Record of the 19th Annual ACM Symposium on Principles of Programming Languages*, pages 343–354, January 1992.

[Che90] J. Cheng. A classification of tasking deadlocks. *Ada Letters*, pages 110–127, May/June 1990.

[Che91] J. Cheng. A survey of tasking deadlock detection methods. *Ada Letters*, pages 82–91, January/February 1991.

[Che92] J. Cheng. The task dependence net in Ada software development. *Ada Letters*, XII(4):24–35, July/August 1992.

[CK84] K. D. Cooper and K. Kennedy. Efficient computation of flow insensitive interprocedural summary information. In *Proceedings of the ACM SIGPLAN '84 Symposium on Compiler Construction*, June 1984.

[CK88] D. Callahan and K. Kennedy. Analysis of interprocedural side effects in a parallel programming environment. *Journal of Parallel and Distributed Computing*, 5:517–550, October 1988.

[CKS90] D. Callahan, K. Kennedy, , and J. Subhlok. Analysis of event synchronization in a parallel programming tool. In *Proceedings of the 2nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 21–30, 1990.

[CKU91] J. Cheng, Y. Kasahara, and K. Ushijima. A tasking deadlock detector for Ada programs. In *COMPSAC91*, Tokyo, September 1991.

[CS88] D. Callahan and J. Subhlok. Static analysis of low-level synchronization. In *Proceedings of the ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging*, pages 100–111, 1988.

[CU91] J. Cheng and K. Ushijima. Analyzing Ada tasking deadlocks and livelocks using extended Petri nets. In *Lecture Notes in Computer Science #499*, pages 125–146. Springer-Verlag, 1991.

[CWZ90] D. R. Chase, M. Wegman, and F. K. Zadeck. Analysis of pointers and structures. In *Proceedings of the SIGPLAN '90 Conference on Programming Languages Design and Implementation*, pages 296–310, June 1990. SIGPLAN Notices, Vol 25, No 6.

[Dij68] E. W. Dijkstra. Co-operating sequential processes. In F. Genuys, editor, *Programming Languages; NATO Advanced Study Institute*, pages 43–112. Academic Press, London, 1968.

[Dil90] L. K Dillon. Verifying general safety properties of Ada tasking programs. *IEEE Transactions on Software Engineering*, pages 51–63, January 1990.

[DS91] E. Duesterwald and M. L. Soffa. Concurrency analysis in the presence of procedures using a data flow framework. In *Proceedings of the ACM SIGSOFT '91 Fourth Symposium on Testing, Analysis, and Verification (TAV4)*, pages 36–48, October 1991.

[Due91] E. Duesterwald. Static concurrency analysis in the presence of procedures. Technical Report 91-6, Department of Computer Science, University of Pittsburgh, March 1991.

[EGP89] P. A. Emrath, S. Ghosh, and D. A. Padua. Event synchronization analysis for debugging parallel programs. In *Proceedings of Supercomputing '89*, November 1989. ISBN 0-89791-341-8.

[EP88] P. A. Emrath and D. A. Padua. Automatic detection of nondeterminacy in parallel programs. In *Proceedings of the ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging*, pages 89–141, 1988.

[FOW87] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, 1987.

[FW74] G. M. Furnival and Jr. Wilson, R. W. Regressions by leaps and bounds. *Technometrics*, 16(4):499–511, November 1974.

[GNV88] E. R. Gansner, S. C. North, , and K. P. Vo. DAG — a program that draws directed graphs. *Software — Practice and Experience*, 18(11):1047–1062, November 1988.

[GPHL90] M. D. Guzzi, D. A. Padua, J. Hoeflinger, and D. H. Lawrie. Cedar Fortran and other vector and parallel Fortran dialects. *Journal of Supercomputing*, 4(1):37–62, 1990.

[GR86] N. H. Gehani and W. D. Roome. Concurrent C. *Software - Practice and Experience*, 16(9):821–844, September 1986.

[GR88] N. H. Gehani and W. D. Roome. Rendezvous facilities: Concurrent C and the Ada programming language. *IEEE Transactions on Software Engineering*, 14(11):1546–1553, November 1988.

[Gra91] P. Granger. Static analysis of linear congruence equalities among variables of a program. In *Lecture Notes in Computer Science #493*, pages 169–192. Springer-Verlag, New York, 1991.

[Hab69] A. N. Habermann. Prevention of system deadlock. *Communications of the ACM*, 12(7):373–377, July 1969.

[HC83] T. Hermann and K. M. Chandy. A distributed procedure to detect AND-OR deadlock. Technical Report TR LCS-8301, Department of Computer Science, University of Texas, Austin, Texas, 1983.

[Hec77] M. S. Hecht. *Flow analysis of computer programs.* Elsevier North-Holland, New York, 1977. ISBN 0-444-00210-3.

[HHN92] L. J. Hendren, J. Hummel, , and A. Nicolau. Abstractions for recursive pointer structures: improving the analysis and transformation of imperative programs. In *Proceedings of the SIGPLAN '92 Conference on Programming Languages Design and Implementation*, pages 249–260, San Francisco, June 1992. ISBN 0-89791-475-9.

[HL85] D. Helmbold and D. Luckham. Debugging Ada tasking programs. *IEEE Software*, pages 47–57, March 1985.

[HN90] L. J. Hendren and A. Nicolau. Parallelizing programs with recursive data structures. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):105–133, 1990.

[Hol72] R. C. Holt. Some deadlock properties on computer systems. *ACM Computing Surveys*, 4(3):179–196, September 1972.

[HPR89] S. Horwitz, P. Pfeiffer, and T. Reps. Dependence analysis for pointer variables. In *Proceedings of the SIGPLAN '89 Conference on Programming Languages Design and Implementation*, pages 28–40, June 1989.

[HY82] K. Hao and Y. T. Yeh. Detection of inherent deadlocks in distributed programs. In *Proceedings of IEEE International Conference on Distributed Computing Systems*, pages 518–523, Florida, October 1982.

[Job91] J. D. Jobson. *Applied multivariate data analysis*, volume 1: regression and experimental design. Springer-Verlag, New York, 1991. ISBN 0-387-97660-4.

[KB90] G. M. Karam and R. J. A. Buhr. Starvation and critical race analyzers for Ada. *IEEE Transactions on Software Engineering*, pages 829–843, August 1990.

[Kna87] E. Knapp. Deadlock detection in distributed databases. *ACM Computing Surveys*, 19(4):303–328, December 1987.

[Knu71] D. E. Knuth. An empirical study of FORTRAN programs. *Software - Practice and Experience*, 1:105–133, 1971.

[KR78] B. W. Kernighan and D. M. Ritchie. *The C Programming Language*. Prentice-Hall, Englewood Cliffs, NJ, 1978. ISBN 0-13-110163-3.

[KU76] J. B. Kam and J. D. Ullman. Global flow analysis and iterative algorithms. *Journal of the ACM*, 23(1):158–171, 1976.

[Lam78] L. Lamport. Time, clocks, and the ordering of events in distributed systems. *Communications of the ACM*, 21(7):558–565, July 1978.

[LC89] D. L. Long and L. A. Clarke. Task interaction graphs for concurrency analysis. In *Proceedings of the 11th International Conference on Software Engineering*, pages 44—52, April 1989.

[LC91] D. L. Long and L. A. Clarke. Data flow analysis of concurrent systems that use the rendezvous model of synchronization. In *Proceedings of the ACM SIGSOFT '91 Fourth Symposium on Testing, Analysis, and Verification (TAV4)*, pages 21–35, Vancouver, BC, October 1991.

[Lon86] D. Longley. *Dictionary of information technology*. Macmillan Press, New York, 1986. ISBN 0-19-520519-7.

[LR92] W. Landi and B. G. Ryder. A safe approximation algorithm for interprocedural pointer aliasing. In *Proceedings of the SIGPLAN '92 Conference on Programming Languages Design and Implementation*, pages 235–248, June 1992.

[LT79] T. Lengauer and R. E. Tarjan. A fast algorithm for finding dominators in a flowgraph. *ACM Transactions on Programming Languages and Systems*, 1(1):121–141, July 1979.

[McD89] C. E. McDowell. A practical algorithm for static analysis of parallel programs. *Journal of Parallel and Distributed Computing*, 6(3):515–536, June 1989.

[Mer92] N. Mercouroff. An algorithm for analyzing concurrent processes. In *Lecture Notes in Computer Science #598*. Springer-Verlag, 1992. ISBN 0-38755-511-0.

[MM92] T. J. Marlowe and S. P. Masticola. Safe optimization for hard real-time programming. In *Second International Conference on Systems Integration*, pages 438–446, June 1992.

[Mos90] L. E. Moser. Data dependency graphs for Ada programs. *IEEE Transactions on Software Engineering*, 16(5):498–509, May 1990.

[MR90a] S. P. Masticola and B. G. Ryder. Static infinite wait anomaly detection in polynomial time. In *Proceedings of the 1990 International Conference on Parallel Processing*, pages II78–II87, August 1990.

[MR90b] S. P. Masticola and B. G. Ryder. Static infinite wait anomaly detection in polynomial time. Technical Report LCSR-TR-141, Laboratory for Computer Science Research, Rutgers University, 1990. This is an expanded version of the ICPP '90 paper.

[MR91a] T. J. Marlowe and B. G. Ryder. Properties of data flow frameworks: a unified model. *Acta Informatica*, 28(2):121–164, 1991.

[MR91b] S. P. Masticola and B. G. Ryder. A model of Ada programs for static deadlock detection in polynomial time. In *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging, Santa Cruz, CA*, pages 91–102, May 1991. Also published as ACM SIGPLAN Notices, vol. 26, no. 12, December 1991.

[MR93] S. P. Masticola and B. G. Ryder. Non-concurrency analysis. In *Proceedings of the 4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, May 1993. Accepted for publication.

[MS82] J. Martinez and M. Silva. A simple and fast algorithm to obtain all invariants of a generalized Petri net. In *Application and Theory of Petri Nets (Informatik-Fachberichte 52)*, pages 301–310. Springer-Verlag, New York, 1982.

[MSS89] T. Murata, B. Shenker, and S. M. Shatz. Detection of Ada static deadlocks using Petri net invariants. *IEEE Transactions on Software Engineering*, pages 314–326, March 1989.

[MSW+93] T. J. Marlowe, A. D. Stoyenko, L. R. Welch, P. LaPlante, and S. P. Masticola. Incremental analysis for reuse and change in hard-real-time systems. In *Tenth IEEE Workshop on Real-Time Operating Systems and Software*, May 1993. To appear.

[Pet81] J. L. Peterson. *Petri Net Theory and the Modeling of Systems.* Prentice-Hall, Englewood Cliffs, N.J., 1981. ISBN 0-13661-983-5.

[RMP88] B. G. Ryder, T. J. Marlowe, and M. C. Paull. Conditions for incremental iteration: examples and counterexamples. *Science of Computer Programming*, 11:1–15, 1988.

[RS90] J. H. Reif and S. A. Smolka. Data flow analysis of distributed communicating processes. *International Journal of Parallel Programming*, 19(1):1–30, 1990.

[Sax77] A. Saxena. Static detection of deadlocks. Technical Report CU-CS-122-77, Department of Computer Science, University of Colorado at Boulder, 1977.

[Set89] R. Sethi. *Programming languages: concepts and constructs.* Addison-Wesley, Reading, Massachussetts, 1989. ISBN 0-201-10365-6.

[SMBT90] S. M. Shatz, K. Mai, C. Black, and S. Tu. Design and implementation of a Petri net based toolkit for Ada tasking analysis. *IEEE Transactions on Parallel and Distributed Systems*, 1(4):424–441, October 1990.

[SMHY92] A. D. Stoyenko, T. J. Marlowe, W. A. Halang, and M. Younis. Enabling efficient schedulability analysis through conditional linking and program transformations. Technical Report CIS-92-20, Department of Computer and Information Science, New Jersey Institute of Technology, Newark, NJ, 1992.

[SS90] C-S. Shih and J. A. Stankovic. Distributed deadlock detection in Ada runtime environments. In *TRI-Ada '90*, Baltimore, MD, December 1990.

[Tar81a] R. E. Tarjan. Fast algorithms for solving path problems. *Journal of the ACM*, 28(3):594–614, July 1981.

[Tar81b] R. E. Tarjan. A unified approach to path problems. *Journal of the ACM*, 28(3):577–593, July 1981.

[Tay83a] R. N. Taylor. Complexity of analyzing the synchronization structure of concurrent programs. *Acta Informatica*, 19:57–84, 1983.

[Tay83b] R. N. Taylor. A general-purpose algorithm for analyzing concurrent programs. *Communications of the ACM*, 26(5):362–376, May 1983.

[TLK92] R. N. Taylor, D. L. Levine, , and C. D. Kelly. Structural testing of concurrent programs. *IEEE Transactions on Software Engineering*, 18(3):206–215, March 1992.

[TO80] R. N. Taylor and L. J. Osterweil. Anomaly detection in concurrent software by static data flow analysis. *IEEE Transactions on Software Engineering*, SE6(3):265–277, May 1980.

[TSM90] S. Tu, S. M. Shatz, and T. Murata. Applying Petri net reduction to support Ada-tasking deadlock detection. In *Proceedings of IEEE International Conference on Distributed Computing Systems*, pages 96–103, Paris, May 1990.

[Wam85] G. K. Wampler. Static concurrency analysis of Ada programs. Master's thesis, University of California, Irvine, CA, 1985.

[Wei84] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, SE-10(4):352–357, July 1984.

[YT88] M. Young and R. N. Taylor. Combining static concurrency analysis with symbolic execution. *IEEE Transactions on Software Engineering*, 14(10):1499–1511, October 1988.

[Zad84] F. K. Zadeck. Incremental data flow analysis in a structured program editor. In *Proceedings of the ACM SIGPLAN '84 Symposium on Compiler Construction*, pages 132–143, June 1984.

[ZYN86] B. Zhou, R. T. Yeh, and P. A. Ng. Principle of deadlock detection in Ada programs. In *Proceedings of IEEE International Conference on Distributed Computing Systems*, pages 572–579. IEEE, 1986.

Index

Accept head node function, 176

Accept node, 281

Accept?, xxix, 76

AInSameType, xxix

Ancestor, 11

Anti-ordering task, 62

Artifactual deadlock, 21, 147

B4, xxix, 43

B4 analysis, 30, 94, 247, 322

B4 summary, 95

Begin node, 56

Binary semaphore, 225

Bottom function, 165

Bracket node, 123

Bracket node set, 237

Call body, 121, 123, 241, 317

Call node, 121, 241, 317

CallB, xxix, 123, 241, 317

$CallB_{set}$, xxix, 241, 317

CallSameInst, xxix, 320

Can't happen together, 24

Candidate critical section structure, 131, 321

Candidate strict interval, 117

CC cycle location graph, 331

CC execution graph, 285

CC sync graph, 280

CD sync graph, 37

CDADA sync hypergraph, 75

CDCC sync graph, 329

CDSEM sync graph, 229, 235, 268

CHT, xxix, 24, 234

CHT_{perf}, xxix, 25, 200, 208, 347

CHTSelf, xxix, 304

$CHTSelf_{Perf}$, xxix, 304

Clause task group, 64

Clear semaphores, 225

CLG, 23, 38

Closed guard, 81

CompleteRend, xxix, 102

Completor, 30, 96, 248, 323

Completor edge, 34, 97, 248, 324

Completors, xxix, 96, 248, 323

Concurrency state, 5, 13

Conditional entry call, 89

Conditional identity function, 165, 333

Conditional increment function, 165

Conditional inflationary function, 333

Constant function, 176, 333

Constraint 1, 21, 59, 78, 90, 232, 293

Constraint 2, 22, 59, 79, 91, 232, 293

Constraint 3, 22, 59, 80, 91, 232, 294
Constraint 4, 22, 59, 80, 91, 232, 294
Constraints on deadlock cycles, 11, 231
Control edge, 14, 229
CPred, xxix, 307
CPreds, xxix, 99, 309
CReach, xxx, 97
Create node, 276, 281
Create set, 288
Critical section, 118
Critical section body, 121, 123, 241, 317
Critical section call graph, 145
Critical section structure, 121, 240, 317
Crits, xxx, 241, 317
CRITSTRUCTS, 320
CSB, xxx, 123, 242, 318
CSB_{set}, xxx, 241, 317
CSStruct, xxx, 241, 317
Cycle detection form Ada sync hypergraph, 75
Cycle detection form CC sync graph, 329
Cycle detection form SEM sync graph, 229, 235, 268
Cycle detection sync graph, 37
Cycle location graph, 23, 38, 60, 75, 235
"Cycle shape" constraint, 21, 59, 78, 90, 232, 293

D_{CC}, 332
Deadlock, 2, 20, 72
Deadlock cycle, 20
DeadPath, xxx, 39, 43, 164, 173, 174, 235, 332
Def, xxx, 300
Descendant, 11
DOM, xxx, 116
Dominator strong components, 47
D_{SACnt}, 174
D_{Sem}, 235, 268
D_{SigCnt}, 164
Dynamic nest level, 147
Dynamic testing, 4

E_C, xxx, 14
$E_{Completors}$, xxx, 248, 324
Edge function, 97, 165, 176, 254, 263, 269, 332
E_{MayHT}, xxx, 262
End node, 56
Entry (of an execution wave), 16
Entry node, 121, 236, 241, 317
E_{PCM}, xxx, 300
E_S, xxx, 15
Execution wave, 15, 57, 230, 290
Exit node, 121, 236, 241, 317, 319

Feasible execution wave, 58, 231
Firing function, 18
Forbidden nodes, 62
Formal proof methods, 6
Frontier node, 86

Guard condition, 81

HCLG, 75

Head node, 21

"Head nodes can happen together" constraint, 22, 59, 80, 91, 232, 294

"Head nodes cannot proceed" constraint, 22, 59, 79, 91, 232, 293

Heterogeneous cross-product frameworks, 36

Homogeneous cross-product frameworks, 34

Hyperedge, 15

Hyperedge cycle location graph, 75

I, xxx, 29

Identity array, 29

InClause, xxx, 111

Inflationary function, 25, 35, 45, 50

Initially-clear semaphores, 226

Initially-set semaphores, 226

K-bounded, 48

L, xxx, 152

Lattice framework, 43

LE, xxx, 152

Loop connectedness, 49

Loop unrolling, 39

Loops, xxx, 302

LS, xxx, 152

MayCall, xxx, 282

Model checking, 374

Multi-source problems, 8, 33

MustCall, xxx, 282

MustClear, xxx, 266

MustSet, xxx, 232, 233, 235, 260

Mutex body, 237

MutexB, xxx

N, xxxi, 13

N_{Accept}, xxxi, 282

$N_{Bracket}$, xxxi, 237, 335

N_{Create}, xxxi, 337

Nested critical section call, 146

NextStates, xxxi, 17

N_{Head}, xxxi, 173

$N_{NonHead}$, xxxi, 312

Node, 229, 281

NodeInsts, xxxi, 284

"Nodes outside deadlock don't break infinite wait" constraint, 22, 59, 80, 91, 232, 294

NodeType, xxxi, 131

Non-concurrency analysis, 8

Normalized execution wave, 290

N_{PCM}, xxxi, 300

N_{Signal}, xxxi, 293

ONE, xxxi, 28

ONE_{Vector}, xxxi, 312

Open guard, 81

Order-accepting nodes, 65

Order-sending node, 64
Ordering chain, 151
Ordering task, 65

PADA sync hypergraph, 70
Parse form Ada sync hypergraph, 70
Parse form Concurrent C sync graph, 276
Parse form SEM sync graph, 229, 234
PartInst, xxxi, 308
Partition begin node, 281
Partition end node, 281
Partition on the CC sync graph, 299
Partner, 103, 309
Partner instance, 308
Partners, xxxi, 103, 109, 309
$Partners_{other}$, xxxi, 310
$Partners_{Pinned}$, xxxi, 112, 312
Path summary, 32
PCC sync graph, 276
Pinning, 105, 310
Pins, 112
Polarity, 56
POSTDOM, xxxi, 116
Predecessor, 11
Process, 274
Process, xxxi, 292
Process begin node, 276, 281
Process creation multigraph, 299
Process end node, 281
Process variables, 275
ProcInst, xxxi, 292
ProcInsts, xxxi, 284
ProcRef, xxxi, 292
Procs, xxxii, 302
PSEM sync graph, 229, 234
Pseudotransitive closure function, 48
Pseudotransitivity, 27

Rapid lattice framework, 35, 43, 49, 98
Rapid lattice frameworks, 49
Reference function, 283, 284, 291
Refinement, 25, 50, 94, 210, 214, 236, 303, 353, 365, 390
Rendezvous, 3, 55
Return node, 121, 241, 317, 319

SACnt, xxxii, 174, 190, 200, 216
Safe approximation, 23
Scope, xxxii, 76
Scope depth, 76
$Scope_{set}$, xxxii, 76
Selective wait, 81
SEM sync graph, 229
Semaphore kill, 235
SemKill, 235
Set semaphores, 225
SigCnt, xxxii, 164
Signal type, 55, 282
Signaling head node function, 177
Signaling node, 281
Signaling node count summary, 164

Signaling node group, 64
Signaling node/accept scope depth count summary, 174
$SigN_{set}$, xxxii, 76
Sigs, xxxii, 78
SigType, xxxii, 131, 334
Simple critical section, 146
Sisters, 81
Sisters, xxxii, 111
Starter, 111, 309
Starter instance, 308
Starters, xxxii, 111, 309
$Starters_{other}$, xxxii, 310
StartRendSig, xxxii, 111
Starvation, 20, 73
State enumeration, 5
Strict interval, 114
StrictInt, xxxii, 115, 237, 317
Strong critical section structure, 317
STRONGSTRUCTS, 320
Successor, 11
Sync edge, 15, 56, 229, 281
Sync environment, 284
Sync graph, 11, 13, 56, 229, 276
Sync hyperedge, 69
Sync hypergraph, 69
Sync subgraph, 70
Sync variable, 283
Sync variable instance, 283
Synchronization, 2
Synchronization model, 3
Synchronization statement, 3
SyncVarInsts, xxxii, 283
SyncVars, xxxii, 302
Synergy of *CHT* refinements, 187, 214

Tail node, 21, 298, 327, 328
TakeOff, xxxii, 111
Tarjan's algorithm, 47
Task, 2
Task, xxxii, 109
Task end node, 56
Task instance, 2
Task name, 55
Task-wise pseudotransitivity, 27
TaskFrontNodes, xxxii, 111
Tasking model, 2
Tasking statement, 56
Tasks, xxxii, 220
Test function, 333
Timed entry call, 89
Top function, 165, 176, 333
Trans, xxxiii, 309
Transaction, 274
Transaction node, 281
Transaction statement, 281
Transaction variable, 275
Transactor, 309
Transactor instance, 307
TransInst, xxxiii, 307
Transition function, 17

Transitive connection, 19

Transitively nested critical section call,
146

UngSis?, xxxiii, 85

UnionB4, xxxiii, 99

Unrolling loops, 39

Use, xxxiii, 300

Wait mode, 73

Waits, xxxiii, 74

Weak critical section structure, 315, 317

WEAKSTRUCTS, 320

Widen, xxxiii, 28, 113, 134, 158, 314

Widening, 26

Vita

Stephen P. Masticola

1956 Born, Wilmington, Delaware, December 15.

1974 Graduated from Christiana High School, Newark, Delaware.

1974-78 Attended University of Pennsylvania, Philadelphia, Pennsylvania.

1978 B.S.E. *summa cum laude* in Computer Science and Engineering, University of Pennsylvania.

1978-79 Programmer, Sperry-Univac, Blue Bell, Pennsylvania.

1979-80 Software and Hardware Engineer, Barclay Analytical, Wynnewood, Pennsylvania; position was transferred to Crown International, Elkhart, Indiana.

1980-82 Senior Software Engineer, Kulicke and Soffa Industries, Horsham, Pennsylvania.

1982-87 Senior Member of Technical Staff, RCA, Somerville, New Jersey.

1987-93 Graduate work in Computer Science, Rutgers University, New Brunswick, New Jersey.

1987-89 Teaching Assistant, Rutgers University.

1987-88 Member of Technical Staff, Bell Communications Research, Piscataway, New Jersey. Summer positions, both years.

1989 M.S. in Computer Science, Rutgers University.

1989-93 Graduate Research Assistant, Rutgers University.

1990 "Static infinite wait anomaly detection in polynomial time." [MR90b]

1991 "A model of Ada programs for static deadlock detection in polynomial time." [MR91b]

1992 "Safe optimization for hard real-time programming." [MM92]

1993 "Non-concurrency analysis." [MR93]

1993 "Incremental analysis for reuse and change in hard-real-time systems." [MSW+93]

1993 Ph.D. in Computer Science, Rutgers University.