Air Force Institute of Technology

AFIT Scholar

Theses and Dissertations

Student Graduate Works

3-2006

Toward the Static Detection of Deadlock in Java Software

Jose E. Fadul

Follow this and additional works at: https://scholar.afit.edu/etd

Part of the Software Engineering Commons

Recommended Citation

Fadul, Jose E., "Toward the Static Detection of Deadlock in Java Software" (2006). *Theses and Dissertations*. 3482.

https://scholar.afit.edu/etd/3482

This Thesis is brought to you for free and open access by the Student Graduate Works at AFIT Scholar. It has been accepted for inclusion in Theses and Dissertations by an authorized administrator of AFIT Scholar. For more information, please contact richard.mansfield@afit.edu.

Toward the Static Detection of Deadlock in Java Software

THESIS

Jose E. Fadul, Captain, USAF

AFIT/GE/ENG/06-19

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

The views expressed in this thesis are those of the author and do not reflect the official policy or position of the United States Air Force, Department of Defense, or the United States Government.

AFIT/GE/ENG/06-19

Toward the Static Detection of Deadlock in Java Software

THESIS

Presented to the Faculty

Department of Electrical and Computer Engineering

Graduate School of Engineering and Management

Air Force Institute of Technology

Air University

Air Education and Training Command

In Partial Fulfillment of the Requirements for the

Degree of Master of Science in Electrical Engineering

Jose E. Fadul, B.S.E.E.

Captain, USAF

March 2006

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

AFIT/GE/ENG/06-19

Toward the Static Detection of Deadlock in Java Software

Jose E. Fadul, B.S.E.E.

Captain, USAF

Approved:

/signed/	3 Mar 2006
Maj Robert P Graham, PhD (Chairman)	date
/signed/	3 Mar 2006
Lt Col Timothy J Halloran (Member)	date
/signed/	3 Mar 2006
Maj Christopher B Mayer (Member)	date

AFIT/GE/ENG/06-19

Abstract

Concurrency is the source of many real-world software reliability and security problems. Concurrency defects are difficult to detect because they defy conventional software testing techniques due to their non-local and non-deterministic nature. We focus on one important aspect of this problem: static detection of the possibility of *deadlock*—a situation in which two or more processes are prevented from continuing while each waits for resources to be freed by the continuation of the other.

This thesis proposes a flow-insensitive interprocedural static analysis that detects the possibility that a program can deadlock at runtime. Our analysis proceeds in two steps. The first extracts the "real" call graph decorated with acquired locks from the target program. The second analyzes this decorated graph to report how a possible deadlock may occur at runtime. We demonstrate our analysis via a prototype implementation that detects deadlock conditions within two small Java programs: Dining Philosophers and Double Lock Equals.

The two principle limitations of our analysis are on the target program: (1) we need its "real" call graph and (2) its overall size is limited. Our prototype tool can only construct the target program's "real" call graph in the presence of perfect aliasing information about its object references. This aliasing information is provided by an external oracle. Combinatorial explosion limits the size of programs our technique is able to analyze to a rough ceiling of 35 kSLOC. Our prototype tool uses a combination of call graph reduction techniques (aided by the insight that our analysis is only concerned with program paths that can acquire one or more locks) and efficient data structure choices to help mitigate this limitation.

To My Family
Thanks for all your Encouragement and Support

Acknowledgements

I would like to express my sincere appreciation to my faculty advisors for their guidance and support throughout the course of this thesis effort. Their insight and experience was certainly appreciated.

Jose E. Fadul

Table of Contents

Page

Abstract iv

Dedication v

Acknowledgements vi

List of Figures x

List of Tables xii

List of Abbreviations xiv

I. Introduction 1
- 1.1 This Thesis 2
- 1.2 A Motivating Example 3
- 1.3 Analysis Overview 6
  - 1.3.1 Two Steps 6
  - 1.3.2 Properties 9
  - 1.3.3 Context Matters 9
- 1.4 Tool Overview 10
- 1.5 Results and Observations 10
- 1.6 Organization 11

II. Definitions and Prior Work 12
- 2.1 What is Deadlock? 12
- 2.2 Prior Approaches to Deadlock Detection 14
  - 2.2.1 Deadlock Detection via Static Analysis 14
  - 2.2.2 Deadlock Detection via Dynamic Analysis . . . 17
  - 2.2.3 Deadlock Detection via Model Checking 18
  - 2.2.4 Java PathFinder 19
  - 2.2.5 Comparison of Detection Approaches 19
- 2.3 Static Call Graphs 21
  - 2.3.1 Definitions 21
  - 2.3.2 Object-Oriented Call Graph Example 26
  - 2.3.3 Context-Sensitive Call Graph Example 26
  - 2.3.4 Call Graph Classification 28

Page

III. CSOOCG Generator 32
3.1 CSOOCG 32
3.2 CSOOCG Generator 36
3.2.1 Starting Point 37
3.2.2 Code Under Test (CUT) Graph 38
3.2.3 Call Completer 38
3.2.4 Reducer Process 42
3.2.5 aliasing Oracle 44
3.2.6 Oracle Interface 44
3.2.7 CSOOCG builder 46
3.3 CSOOCG Soundness 49
3.4 Generator Implementation 50
3.4.1 CSOOCG Data File 51
3.5 A Second Example: Double Lock Equals 52

IV. CSOOCG Analyzer 57
4.1 Overview 57
4.2 Analysis Model 57
4.2.1 Definitions and Functions Table Explained . . . 65
4.3 Analyzer Implementation 66
4.3.1 Trace Generator 66
4.3.2 Trace Eliminator 70
4.3.3 LockMap Builder 70
4.3.4 Deadlock Determinator 70
4.3.5 Presenter 70
4.3.6 Implementation Challenges 72
4.4 A Second Example: Double Lock Equals 73

V. Conclusion 81
5.1 Summary of Contributions 81
5.2 Recommendations for Future Work 82
5.2.1 Integrate an alias analysis 82
5.2.2 Support analysis cut-points and composability of results 83
5.2.3 Find and focus on the nexuses of locking 83
5.2.4 Support util.concurrent-style locks 84
5.2.5 Use a model checker 84
5.3 Concluding Thoughts 84

Page

Appendix A. Use and SetUp 85
A.1 Required Items 85
A.2 Using the Generator 86
A.3 Using the Analyzer 88

Appendix B. Dining Philosophers Results Deadlock Version 89
B.1 Statistics 89

Appendix C. Double Equals Results Deadlock version 93
C.1 Statistics 93

Appendix D. Glossary of Technical Terms 99

Bibliography 102

List of Figures

Figure Page

1.1. The Dining Philosophers . . . 3

1.2. Steps to Detect Deadlock . . . 4

1.3. Dining Philosophers Source Code . . . 7

1.4. Dining Philosophers CSOOCG . . . 8

2.1. Java Locking Semantics Example . . . 22

2.2. Call Graph of Locking Semantics Example . . . 23

2.3. Abbreviated Call Graph of Locking Semantics Example . . . 25

2.4. Class Hierarchy Code Example . . . 26

2.5. UML Class Diagram . . . 27

2.6. Object-Oriented Call Graph of The Hierarchy Code Example . . . 27

2.7. Context-Sensitive Java Code Example . . . 29

2.8. Object-Oriented Call Graph of the Context-Sensitive Java Code Example . . . 30

2.9. Context-Sensitive Call Graph . . . 30

2.10. Regions in a Call Graph Lattice Domain (Grove et al. [12]) . . . 31

3.1. Dining Philosophers Source Code . . . 34

3.2. Dining Philosophers CSOOCG . . . 35

3.3. CUT Graph Semantics . . . 38

3.4. Dining Philosophers Initial CUT Graph . . . 39

3.5. Dining Philosophers Updated CUT Graph . . . 40

3.6. CUT Graph is Object-Oriented . . . 41

3.7. Dining Philosophers Reduced CUT Graph . . . 42

3.8. Dining Philosophers Reduced CSOOCG . . . 43

3.9. Java Source Code Example Demonstrating Aliasing . . . 45

3.10. Alias Oracle Interface . . . 46

Figure Page

3.11. Generator Class Diagram 50

3.12. Abbreviated Data File 53

3.13. Double Lock Equals Java Source Code 55

3.14. Double Lock Equals CSOOCG 56

4.1. Dining Philosophers CSOOCG 58

4.2. Analyzer Class Diagram 67

4.3. Non-Trie Data Structure 68

4.4. Trie Data Structure 69

4.5. Sample HTML Output File Table 71

4.6. Sample Latex Output File Table 72

B.1. Dining Philosophers Source Code 90

C.1. Double Lock Equals Java Source Code 94

List of Tables

Table		Page
1.1.	Dining Philosophers Deadlock Conditions.	6
2.1.	Analysis Classification Table	20
2.2.	Analysis Capabilities Table	20
3.1.	Dining Philosophers Deadlock Conditions.	37
4.1.	Trace Generator Equations	59
4.2.	Definitions and Functions	60
4.3.	Dining Philosophers Results: First Iteration	61
4.4.	Dining Philosophers Results: Second Iteration	62
4.5.	Dining Philosophers Results: Third Iteration	62
4.6.	Dining Philosophers Results: Forth Iteration	63
4.7.	Dining Philosophers Results: Fifth Iteration	63
4.8.	Dining Philosophers Results: Sixth Iteration	64
4.9.	Dining Philosophers Deadlock Conditions	65
4.10.	Double Lock Equals Results: First Iteration	74
4.11.	Double Lock Equals Results: Second Iteration	75
4.12.	Double Lock Equals Results: Third Iteration	76
4.13.	Double Lock Equals Results: Fourth Iteration	77
4.14.	Double Lock Equals Statistics	79
4.15.	Double Lock Equals Deadlock Conditions	80
A.1.	CSOOCG Generator Switch Options	86
A.2.	CSOOCG Analyzer Switch Options	88
B.1.	Dining Philosophers Statistics	89
B.2.	Dining Philosophers Nodes	91
B.3.	Dining Philosophers Edges	91
B.4.	Dining Philosophers Locks	92

Table Page

B.5. Dining Philosophers Lock Map Entries 92

B.6. Dining Philosophers Deadlock Conditions 92

C.1. Double Lock Equals Statistics 93

C.2. Double Lock Equals Nodes 95

C.3. Double Lock Equals Edges 96

C.4. Double Lock Equals Locks 97

C.5. Double Lock Equals Lock Map Entries 97

C.6. Double Lock Equals Deadlock Conditions 98

List of Abbreviations

Abbreviation		Page
CMU	Carnegie Mellon University	2
CSOOCG	Context-Sensitive Object-Oriented Call Graph	6
CUT	Code Under Test	10
IDE	Integrated Development Environment	10
DFS	Depth First Search	14
SDDCE	Scheme for Dynamic Detection of Concurrent Execution	20
UML	Unified Modeling Language	26
AST	Abstract Syntax Tree	37
HTML	Hypertext Markup Language	70

Toward the Static Detection of Deadlock in Java Software

I. Introduction

Multicore processors [9] and programming languages, such as Java [10], have made concurrent programming accessible to a wider audience of practicing programmers. Concurrent programs provide the user a more responsive software experience compared to sequential programs, and make use of the new multicore processor capabilities to improve overall program speedup compared to sequential programs. However, these improvements in responsiveness and speed come at a price, namely increased code complexity.

Concurrent programs are effectively non-deterministic due to the exponential number of possible thread interweavings. This property makes their correctness harder to reason about than that the correctness of a sequential program. In Java, concurrent program threads communicate with each other via fields. These fields are the program's shared state. The shared state between concurrent program threads may become corrupted when an incomplete state change in one thread is interrupted and the shared state is modified by another thread. This undesirable interweaving between threads is called a *race condition*.

The "cure" for race conditions is to add exclusive locking around critical sections of code. An exclusive locking policy insures that shared state is accessed by only one thread at a time. Unfortunately this "cure" introduces a problem called *deadlock*. A deadlock condition occurs when a process waits for an event that will never occur. Cheng [4, 5] and Levine [19], have refined this general definition of deadlock into 18 types of deadlock. The one that is germane to our work is *circular deadlock*. Circular deadlock occurs when two or more threads are waiting on each other to release an exclusive lock. For example, $thread_1$ in a Java program may be stuck waiting for

the availability of an exclusive lock being held by $thread_2$. This prevents $thread_1$ from proceeding to a desired end state, i.e., $thread_1$ is blocked by $thread_2$. At the same time, $thread_2$ may be waiting for the availability of an exclusive lock being held by $thread_1$, i.e., $thread_2$ is blocked by $thread_1$. We say that $thread_1$ and $thread_2$ are stuck in a circular deadlock. In general, this "circle" of deadlock could contain more than two threads. Our interest is to detect, statically, the possibility of circular deadlock occurring within programs written in the Java programming language.

1.1 This Thesis

This thesis describes a flow-insensitive interprocedural static analysis for a subset of Java programs that detects the possibility that a particular program can circularly deadlock at runtime. The subset of Java programs the analysis is designed for includes those Java programs where it is possible to statically determine what Grove, DeFouw, Dean, and Chambers in [11, 12] refer to as the "real" call graph (described in Figure 2.10 on page 31). We demonstrate our analysis via a prototype implementation that detects circular deadlock conditions within two small Java programs: Dining Philosophers and Double Lock Equals. This analysis is implemented using the Fluid analysis infrastructure developed at Carnegie Mellon University (CMU).[1]

Our primary contribution is the development of an analysis that can statically detect the potential for deadlock in object-oriented programs. Prior work has focused on C-like languages [7] that lack dynamic (or runtime) dispatch of function calls. Our work is, to the best of our knowledge, novel in this respect. A secondary contribution is the the formal definition of a Context-Sensitive Object-Oriented Call Graph (CSOOCG), which is a model of the call structure of a program that appears to be potentially useful for other analyses.

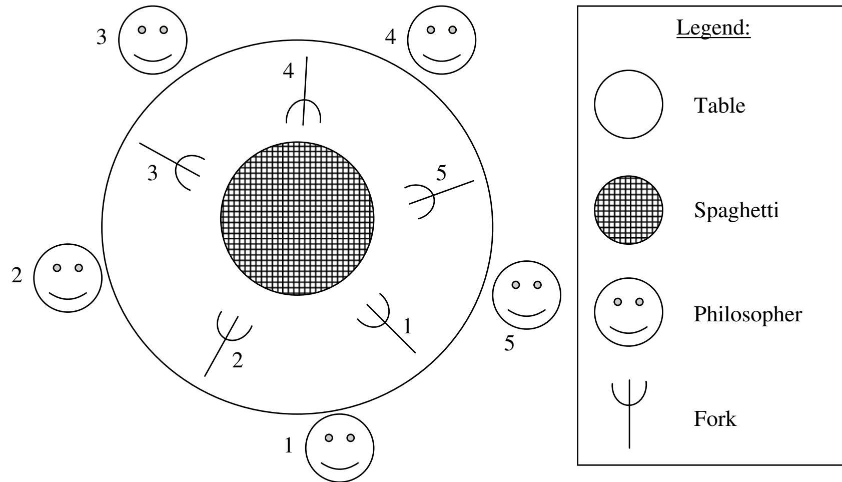


Figure 1.1: **The Dining Philosophers** Five philosophers sitting at the same round table with five forks and one plate of spaghetti. Forks are shared and a philosopher needs two forks to eat.

1.2 A Motivating Example

The Dining Philosophers problem is a classic example of how processes sharing limited resources can become deadlocked. The Dining Philosophers problem, shown in Figure 1.1, has five philosophers sitting at the same round table with five forks and one plate of spaghetti. Each philosopher uses two forks to eat spaghetti. The five philosophers repeatedly eat spaghetti and then think. A philosopher must pick up both forks in order to eat: first the right fork then the left fork. Using this order of picking up forks, circular deadlock occurs when each philosopher picks up his or her respective right forks and is waiting for the availability of the left fork. Once deadlock occurs, the philosophers starve to death. To eliminate this potential for deadlock, the philosophers need to define a consistent fork acquisition order, i.e., a partial order relation between forks used to order fork acquisition by all the philosophers.

To map this problem into a Java program, we can consider the philosophers to be program threads and the forks to be objects used as locks. To detect that deadlock may occur, we need to know when two or more locks are acquired, and the order these locks are acquired in. Finally, for a particular thread of execution (i.e.

[1] The Fluid project website is at `http://www.fluid.cs.cmu.edu`

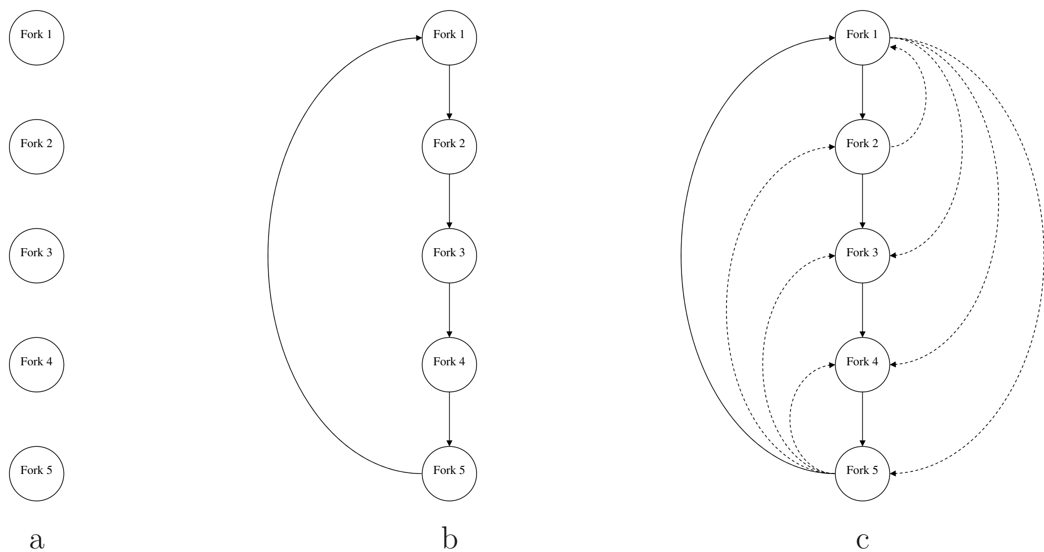


Figure 1.2: **Steps to Detect Deadlock** (a) Identify the shared resources. (b) Determine the order these resources are acquired in. (c) Perform the transitive closure and determine if the order the objects are acquired in is ever reversed. In this figure, circles represent the objects being locked, the solid arrows represent the order the locks are acquired across all threads, and the dashed arrows represent some of the transitive closure orders that need to be added or accounted for. Sub-figure c shows some acquisition orders are reversed, i.e., the graph in sub-figure c is not anti-symmetric.

philosopher) we need to answer the question: Is this order ever reversed? First, as shown in Figure 1.2.a., we note the five forks used as "locks." Next, we need to determine the possible lock acquisition order for each "thread" or philosopher in our example:

Philosopher	Lock Acquisition Order
1	Fork 1, Fork 2
2	Fork 2, Fork 3
3	Fork 3, Fork 4
4	Fork 4, Fork 5
5	Fork 5, Fork 1

From this per-thread lock order, we can determine the lock order across all threads. This is shown in Figure 1.2.b. Next, we compute the transitive closure and determine if the order the forks are acquired in is ever reversed. The transitive closure is shown in Figure 1.2.c. For the purposes of detecting deadlock, we consider the five forks to be elements of a set and the transitive closure of the global acquisition order to be a binary relation on this set. By inspection, we can see that the graph shown in Figure 1.2.c. (of the binary relation representing the transitive closure of the global lock acquisition order) is not anti-symmetric. Thus, cycles exist amongst the fork (lock) acquisition orders and therefore hence deadlock is possible.

We will use a simplified Java implementation of the Dining Philosophers problem that uses only two philosophers and two forks. Our implementation is shown in Figure 1.3. This program can deadlock at runtime. The results from our prototype tool are shown in Table 1.1. The tool uses numbers to model runtime object identifiers (the tool creates a static model of the runtime heap). Each object gets a unique identifier, called its "ID" in the table. In one possible thread of execution we see that the right fork is `Fork` instance 1 and the left is `Fork` instance 2 (which are locked at lines 21 and 22 in the code). For a second possible thread of execution we see that the right fork is `Fork` instance 2 and the left is `Fork` instance 1. Thus, the tool reports the possibility that the program may deadlock at runtime. The tool reports the two

Table 1.1: Dining Philosophers Deadlock Conditions.

CSOOCG Deadlock Conditions					
	Locks Acquired				
ID	First	Second	Method Call	File	Line #
0	heap ID: 1	heap ID: 2			
			Trace 1		
			synchronized (this.right)	Philosopher.java	21
			synchronized (this.left)	Philosopher.java	22
1	heap ID: 2	heap ID: 1			
			Trace 1		
			synchronized (this.right)	Philosopher.java	21
			synchronized (this.left)	Philosopher.java	22

locks involved (including their modeled object identifiers) and the trace in the code where they were acquired.

1.3 Analysis Overview

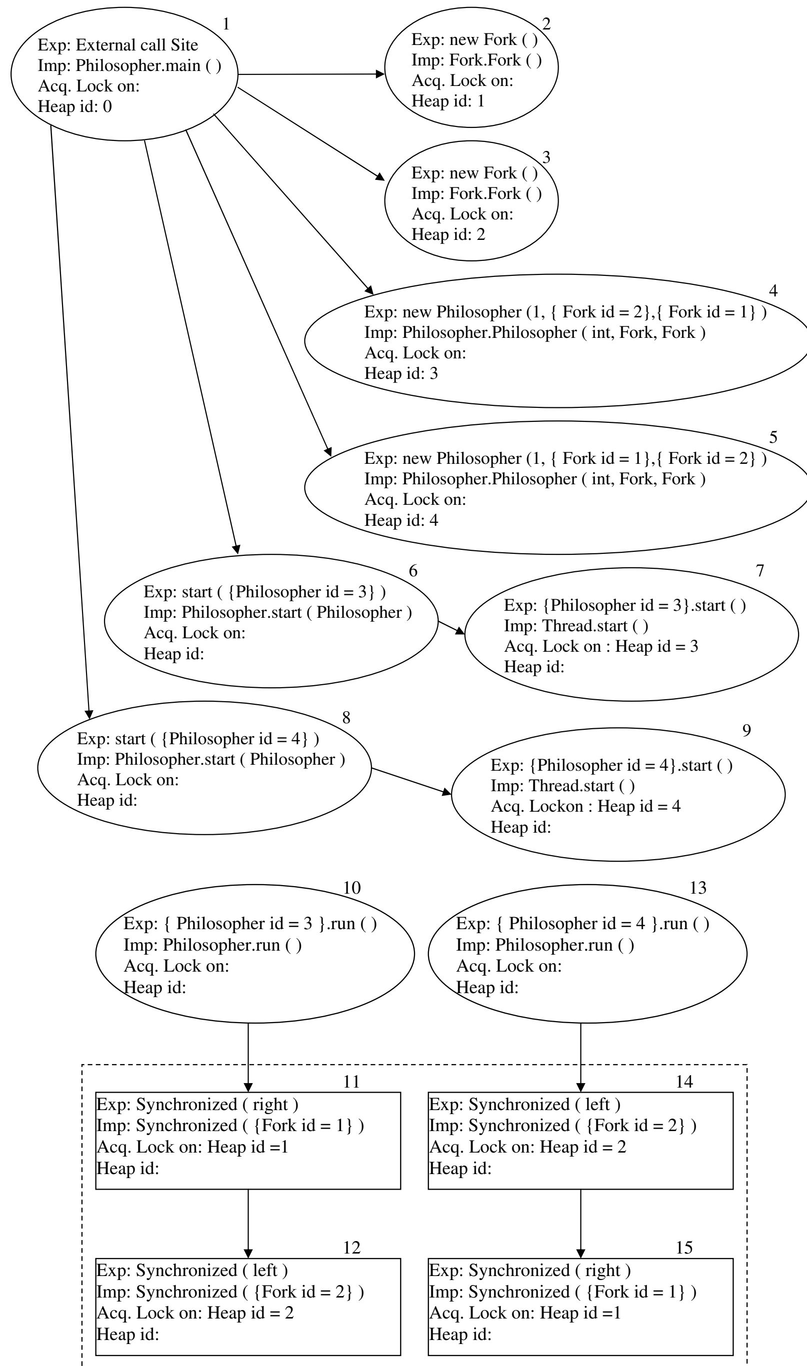
1.3.1 Two Steps. We broke our analysis into two distinct steps: a *CSOOCG generator* and a *CSOOCG analyzer*. The CSOOCG generator runs as part of the Fluid assurance tool. This tool was developed as part of the Fluid project which is dedicated to developing practical software assurance[2] and transformation techniques. This project includes several researchers at Carnegie Mellon University, the Air Force Institute of Technology, and the University of Milwaukee-Wisconsin. For our purposes, this tool provided a well-tested infrastructure for the analysis of Java code. Our CSOOCG analyzer is independent of Fluid.

The communication vehicle between the two steps of our analysis is the *Context-Sensitive Object-Oriented Call Graph* (CSOOCG). The CSOOCG is a model of calls that may be made by a particular Java program at runtime, with what locks are acquired and released during those calls. The CSOOCG for our Dining Philosophers program is shown in Figure 1.4 (this figure will be explained further in Chapter III).

[2]In this thesis the word "assurance" is synonymous with verification—proof that an implementation is consistent with a precise behavioral specification or model

```
1  public class Philosopher extends Thread {
2
3    public static final class Fork {  }
4
5    final Fork right;
6
7    final Fork left;
8
9    final int identity;
10
11   Philosopher(int identity, Fork right, Fork left) {
12     this.identity = identity;
13     this.right = right;
14     this.left = left;
15   }
16
17   @Override
18   public void run() {
19     while (true) {
20       // Thinking
21       synchronized (right) {
22         synchronized (left) {
23           // Eating
24         }
25       }
26     }
27   }
28
29   public static void main(String[] args) {
30     final Fork f1 = new Fork();
31     final Fork f2 = new Fork();
32     final Philosopher p1 = new Philosopher(1, f1, f2);
33     final Philosopher p2 = new Philosopher(2, f2, f1);
34     start(p1);
35     start(p2);
36   }
37
38   private static void start(final Philosopher p) {
39     p.setName(‘‘philosopher-’’ + p.identity);
40     p.start();
41   }
42 }
```

Figure 1.3: **Dining Philosophers Source Code**.



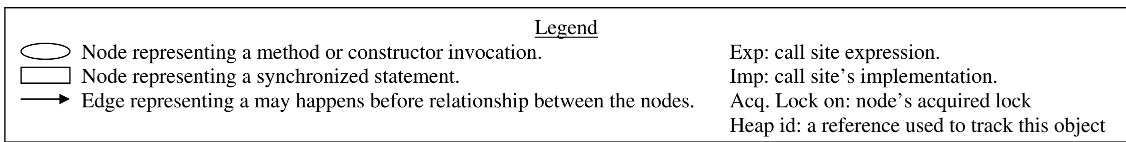


Figure 1.4: **Dining Philosophers CSOOCG** This is the CSOOCG for the Dining Philosophers program example in Figure 3.1.

1.3.2 Properties. Our CSOOCG analysis is a flow-insensitive interprocedural analysis. Our analysis only handles locks acquired by synchronized syntactic blocks, e.g., synchronized methods and statements in Java. It does not support the new `util.concurrent.locks` lock objects, which would require a flow-sensitive analysis to track individual lock acquisition and release statements. If the acquisition and release of a lock is defined by a syntactic block, then determining the set of calls which may be invoked within such a block is *flow-insensitive*. Our analysis is *interprocedural* because to deadlock a thread must try to acquire two or more locks and the only way to acquire two or more locks is to nest synchronized blocks by having one "call" the other. Thus, we must consider the *whole program* or most of the program to produce useful results.

1.3.3 Context Matters. Deadlock detection requires us to know each call's context because, in Java, locks are associated with object references (not object reference variables). Thus to understand which objects (on the heap) are being locked requires us to understand which object is being referenced by a particular object reference expression. The context sensitivity of the CSOOCG uses an external alias oracle to resolve reference variables within the program. Our intent is to cleanly separate the alias analysis from our own analysis. We have left to future work the integration of a state-of-the-art alias analysis tool into our analysis tool.

We need to make a important distinction about use of the alias oracle by our analysis. The alias oracle *must* produce correct information about any reference used as a lock within the program for our results to be correct (i.e., sound). It is not required that this be the case for creating the context-sensitive call graph. In this case our results will be correct, in the sense that they are conservative, but we will report cases of possible deadlock which really can't occur at runtime (i.e., we will have false positive results).

Our prototype tool uses the aliasing oracle to build the CSOOCG. It uses it to determine what objects are locked and the context of all calls. Future work could relax the latter if an increase in false positive results is deemed acceptable.

1.4 Tool Overview

Our deadlock detection prototype tool requires five major components

1. Java program Code Under Test (CUT),
2. Eclipse IDE,
3. Fluid assurance tool,
4. CSOOCG generator, and
5. CSOOCG analyzer.

The Eclipse Java Integrated Development Environment (IDE)[3] is the platform used by the Fluid assurance tool. Our CSOOCG generator uses the Fluid analysis infrastructure to extract the CSOOCG for the Java CUT. The generated CSOOCG is saved to an XML data file to be subsequently processed by the CSOOCG analyzer. The CSOOCG analyzer determines if the CSOOCG contains traces that can deadlock at runtime.

1.5 Results and Observations

The CSOOCG analysis algorithm is able to detect deadlock conditions in both of our test programs using an external alias oracle. Without an aliasing oracle, our results are incorrect (due to the issue of tracking lock references noted earlier in this chapter).

The combinatoric explosion of possible paths in our CSOOCG has caused insufficient memory errors to occur and remains the most significant challenge to this

[3]Eclipse is a Java IDE created and maintained by the Eclipse Foundation at `http://www.eclipse.org`.

work. The *Reducer* and *Trace Eliminator* processes (described in Chapter III and Chapter IV) were designed to reduce memory requirements and thus help us to scale to larger programs. On a typical desktop, our approach can roughly scale up to 35kSLOC of Java.

We observed that the actual number of traces that acquire two or more locks in most programs is small. For our Dining Philosophers test case the number of traces before the *Trace Eliminator* is 15 and the number after is 2. For our Double Lock Equals test case the number of traces before the *Trace Eliminator* is 19 and the number after is 4. A similar observation was also made by Engler et al. [7, page 240], Rugina and Rinard [21, page 71] and Holzmann [16].

1.6 Organization

The rest of this thesis is organized as follows:

- Chapter II, "Definitions and Prior Work," defines deadlock and various static call graphs. This chapter also presents relevant prior work in deadlock detection.
- Chapter III, "CSOOCG Generator," provides a deeper understanding of the CSOOCG and the CSOOCG generator process with a closer look at how the CSOOCG for the Dining Philosophers is generated.
- Chapter IV, "CSOOCG Analyzer," provides a deeper understanding of the CSOOCG Analyzer process with a closer look at how the CSOOCG for the Dining Philosophers is analyzed to determine if possible deadlock condition exist.
- Chapter V, "Conclusion," summarizes our results and covers possible future directions that this research topic may follow.

Several appendices contain instructions on how to set up and use our prototype tool as well as the data from our case studies of the Dining Philosophers and Double Lock Equals Java programs.

II. Definitions and Prior Work

In this chapter we present definitions and relevant related work on deadlock detection. We first define deadlock and then compare and contrast our work to prior deadlock detection approaches and tools. We then define several different forms of a static call graph to situate our CSOOCG, the "heart" of our proposed approach, in prior static analysis work.

2.1 What is Deadlock?

Deadlock occurs when two or more programs are waiting to acquire exclusive access to common resources held by one of the waiting programs. For example, consider two programs, program 1 and program 2:

pseudo code for program 1.

```
acquire(lock_a); acquire(lock_b);
//do some work
release(lock_b); release(lock_a);
```

pseudo code for program 2.

```
acquire(lock_b); acquire(lock_a);
//do some work
release(lock_a); release(lock_b);
```

These programs are attempting to acquire exclusive access to common resources A and B. Resource A is protected by `lock_a` and resource B is protected by `lock_b`. A program holding `lock_a` has exclusive access to resource A. Similarly, A program holding `lock_b` has exclusive access to resource B. Let us consider the following execution steps for program 1 and program 2:

Step 1. Program 1 acquires `lock_a`.

Step 2. A context switch occurs, causing program 1 to be paused while program 2 starts execution.

Step 3. Program 2 acquires `lock_b` and is unable to acquire `lock_a`. This causes program 2 to wait until `lock_a` is available, i.e., wait until program 1 releases `lock_a`.

Step 4. Another context switch occurs, causing program 2 to be paused while program 1 resumes execution.

Step 5. Program 1 is unable to acquire `lock_b`. This causes program 1 to wait until `lock_b` is available, i.e., wait until program 2 releases `lock_b`.

This circular waiting between program 1 and program 2 is defined as deadlock. Program 1 can not continue execution and hence is unable to release `lock_a`. Similarly, program 2 is unable to release `lock_b`. This example can be fixed by having all programs acquire multiple locks in a consistent order. This solution would require one of the two programs to change their order of lock acquisition. For example, program 1 could acquire `lock_b` before acquiring `lock_a` as shown below.

pseudo code for program 1 modified.

```
acquire(lock_b); acquire(lock_a);
//do some work
release(lock_a); release(lock_b);
```

This solution could be applied easily to small programs where multiple locks are acquired and released in single methods. It is more difficult to detect and correct this problem in larger programs where methods may be holding locks when they call other methods that acquire other locks. The numerous program execution interweaving possibilities create many possible lock acquisition orders. These possibilities increase the complexity of deadlock detection. Furthermore, unlike the domain of relational database management systems, we cannot simply detect deadlock at runtime and "rollback" the errant program. This is because (1) dynamic deadlock detection is costly (in terms of runtime overhead) and (2) general programs have side-effects other than on (tables of) data.

2.2 ***Prior Approaches to Deadlock Detection***

Prior work in the area of deadlock detection in concurrent programs have used the following techniques: dynamic (including postmortem) analysis, model checking, and static analysis. We will compare our approach to prior results based upon each of these techniques.

We consider an approach to be *sound* if it does not miss any possible deadlock conditions within a program, i.e., no false negatives. We consider an approach to be *complete* if every deadlock condition reported can actually occur at runtime, i.e., no false positives.

2.2.1 Deadlock Detection via Static Analysis. Static approaches, such as [1, 7, 18, 20, 25], attempt to detect possible deadlock conditions without executing the program. Our approach falls into this category. Static analysis approaches to deadlock typically create an abstract representation of the program or may try to find a given source code pattern within the code. This abstract representation may take the form of graphs, where the nodes represent program states of interest and edges represent program transition statements. Static approaches are also typically very memory-intensive and suffer from large numbers of false positives. Few tools in this category have been created, the most similar tool, which in fact inspired our work, is RacerX.

2.2.1.1 RacerX. RacerX [7] is a static data race and deadlock detection tool for concurrent programs written in C. According to the authors, Engler and Ashcraft, RacerX's data race and deadlock detection method is a top-down, flow- and context-sensitive, interprocedural lockset analysis [7, page 240]. They accomplish this analysis via a depth-first search (DFS) of the program's control flow graph (CFG). The search originates from the roots of each call graph (i.e., where threads start their execution) and the lockset is adjusted at every node visited in the CFG. The lockset adjustments entail adding and deleting locks from the lockset based on the operating

system library procedures for acquiring and releasing locks. Additional source code stubs are used to suppress false positives and improve RacerX's accuracy. the source code stubs are programmer's annotation used to inject the programmer's intent.

RacerX's [7] analysis proceeds as follows. The tool first perform a DFS traversal of the CFG that (1) adds and/or removes locks from the lockset and (2) calls the race and deadlock checker on every node in the CFG. The locksets are cached at both the statement and function level of the program. The cached locksets are used to improve RacerX's execution speed. RacerX's analysis is deterministic in the sense that two execution paths from the same node with the same lockset will produce the same result. So the cached locksets may be considered a set of locksets. This means that if an execution path reaches a previously traversed node in the CFG with a copy of the lockset already in its cache, then the DFS traversal along this path can be terminated. This means the DFS can continue on to the next branch of the analysis as if it had reached a leaf node. This reduces the execution time of the analysis by eliminating unnecessary duplicate processing. The terminology used by Engler for the cached locksets are "statement cache" and "summary cache." The statement cache identifies the cached lockset associated with each statement. The summary cache identifies the cached lockset associated with each function. The statement and function level of analysis act the same way, namely that if the lockset is contained in the cache, then there is no need to reprocess the statement or function.

The locksets being cached at the function and statement levels are commonly referred to as the "entry lockset". This identifies the state of the lockset just before the function or statement is executed. The resulting lockset is referred to as the "exit lockset". The exit lockset is the lockset resulting from the function or statement execution with a given entry set.

The exit locksets produced per function are used to determine the overall state of the concurrent program. An exit lockset can result from each possible CFG path. This could lead to an exponentially large number of locksets based on the number

of threads, the number of functions and the number of function interactions. Engler reports that the actual number of exit locksets created is much less than the theoretical maximum. Most functions either do not acquire any locks or release all their acquired locks before completion (i.e., most functions produce an empty exit lockset).

Locksets are initialized at the roots of the program's call graph. These roots are functions that have no callers. Locksets are transfered around the global CFG and are cached at both the statement and function levels. The transfer function entails adding and removing locks from the entry lockset. Statements and functions capable of adding and removing locks are based on the operating system functions used to perform lock acquisition and release.

RacerX's deadlock detection method uses a two-step process: "(1) constraint extraction, which extracts all locking constraints and (2) constraint resolving, which does a transitive closure [to identify and flag cycles]." The constraint extraction step is applied at every lock acquisition node in the global CFG. This step determines the constraints between the newly acquired lock and the locks in the current lockset. For example, if the current lockset contains locks x and y, then the acquisition of lock z creates two constrains $x \rightarrow z$ and $y \rightarrow z$. This first step also collects trace information concerning how the code was traversed between acquiring lock x and lock z as well as between acquiring lock y and lock z.

Constraint resolution creates the transitive closure of all constraints and detection cycles. This step uses a user–defined value n, which represents the maximum number of threads to consider in this deadlock analysis, i.e., RacerX detects deadlock conditions involving 2,. . ., n threads. This step also records the shortest error path, ranks the errors and displays the results. For example, given the following three constraints $x \rightarrow y$, $y \rightarrow z$ and $z \rightarrow x$, RacerX computes the transitive closure which identifies the following additional constraints $x \rightarrow z$, $y \rightarrow x$, $z \rightarrow y$, $x \rightarrow x$, $y \rightarrow y$, and $z \rightarrow z$. In this case the cyclic deadlock condition between the three threads is flagged. This deadlock condition is created when thread 1 holds lock x and is waiting

on lock y, thread 2 holds lock y and is waiting on lock z, and thread 3 holds lock z and is waiting on lock x.

RacerX is similar to our prototype tool except that it does not support dynamic method dispatch which is nearly ubiquitous in object oriented languages like Java. In fact, our analysis started as an implementation of the RacerX algorithm for Java. Another difference is that our approach is sound for the subset of the Java programs we handle, while RacerX is not sound for the C language. The drawback to our decision is that, currently, RacerX scales much better than our approach.

2.2.2 Deadlock Detection via Dynamic Analysis. Dynamic analysis approaches to deadlock detection, such as [3, 23], monitor the execution of one or more runs of a concurrent program to determine if deadlock may occur in another execution of the program. Unlike dynamic analysis approaches, our approach has the advantage that we don't have to run the program. In addition, dynamic analysis techniques are never sound because they only see one (or a few) of the possible executions of the program. We now describe the Eraser tool which supports dynamic deadlock detection in Java programs.

2.2.2.1 Eraser. Eraser [23] is a dynamic analysis program originally designed to detect possible data race conditions in multithreaded programs and later modified to detect possible deadlock conditions. Eraser uses dynamic analysis techniques to monitor the correctness of multithreaded concurrent programs. The monitoring effect is achieved by adding and removing locks to locksets associated with specific variables. This process maintains a map-like data structure. The memory location of the shared variable is used as the key to the map structure and the lockset is the associated value. The locksets initially contain all the locks identified in the multithreaded concurrent program. The set of locks held by a thread that accesses these shared variables is intersected with the appropriate lockset in the map. If the result of the intersection is the empty set, then the variable is not consistently protected during all accesses and may indicate a possible race condition. Eraser has been modi-

fied to detect deadlock conditions using the same lockset information gathered during the data race detection analysis. Manual annotations are added to the source code to help minimize the false positives and false negatives reported by Eraser. Eraser is considered an automated process–after the annotations have been added. Eraser is not a sound or complete algorithm. Eraser is able to handle aliasing due to its dynamic nature, i.e., it is easy to determine an object's memory address during program execution. Eraser is able to find, but not assure the absence of, data races and deadlock conditions.

Eraser and our CSOOCG analyzer have very little in common. Both analyses build, maintain and analyze the contents of a lock data structure. In Eraser this data structure is the lock sets in our CSOOCG analyzer it is a lock map. This is where the commonality between these two approaches ends. Eraser does not attempt to cover all possible executions paths to determine where data races and deadlocks may occur. Our analysis traverses every possible execution path in an attempt to locate all possible deadlock conditions. Eraser–being a dynamic analysis–does not face the same aliasing problem that our static analysis does. As we can see, their are more differences between our CSOOCG analyzer and Eraser than there are similarities.

2.2.3 Deadlock Detection via Model Checking. Model checking approaches to deadlock detection, such as [6, 14–17], use an abstract representation of a concurrent program (typically based upon a model logic) and, using the model checking tool, verify specific properties about that abstraction of the program. The abstraction of the program is called a *model* of the program. Model checking approaches to deadlock detection typically use a static analysis to create the model of the program. Our approach is similar to deadlock detection via model checking. We suffer the same scalability problems typical to model checking approaches (due to combinatoric explosion). We have proposed, in Chapter V, that future work consider replacing our CSOOCG analyzer component (or part of it) with a model checker.

2.2.4 Java PathFinder. Java PathFinder (JPF) [14] is an attempt to automate the creation of a Promela model from Java source code for simple Promela interpreter (SPIN) analysis. JPF is part of a larger effort by NASA to make formal method applicable within NASA's areas of space, aviation, and robotics. The main strength of JPF is the maturity of SPIN. The main weakness is scalability, the tool reports only being able to support 2KSLOC. The size of the Promela model to be analysis must be small and finite, where finite means that only a specified small number of threads and variables may be created in the model. A secondary weakness is the need for annotations to add programmer's intent to the Java program. These annotations are assertions to be included in the Promela model for SPIN to verify.

2.2.5 Comparison of Detection Approaches. In this section we summarize the classifications and capabilities of all the analyses covered and additional analyses that were not discusses but follow a similar approach. Table 2.1 covers their classifications and Table 2.2 covers their capabilities. We also discuss the similarity and difference between the analyses.

Our approach does not use annotations to suppress false positive and/or false negatives, which Eraser and RacerX do. This means that the user does not have to add additional information to his source code to use our tool (however, we admit that our current manual aliasing oracle perhaps more than makes up for this advantage). Our approach is implemented in Java to analyze Java source code. This increases the difficulty of our analysis compared to RacerX, which analyzes C code, and does not have to consider dynamic dispatch of methods.

Table 2.1: Analysis Classification Table

Analysis Method	Approach Classification	Programming Language	Limitations
Eraser	Dynamic	C/C++	not exhaustive
SDDCE[a]	Dynamic	Java	not exhaustive
Ada TIG	Model Checker	Ada	combinatoric explosion problem
RacerX	Static	C	combinatoric explosion problem
Ownership Type	Static	Java	combinatoric explosion problem
CSOOCG[b] Analysis	Static	Java	combinatoric explosion problem

[a] SDDCE is "A Scheme for Dynamic Detection of Concurrent Execution of Object Oriented Software".
[b] CSOOCG is a Context Sensitive Object Oriented Call Graph.

Table 2.2: Analysis Capabilities Table

Analysis Method	Sound	Complete	Ability to Handle Aliasing	Deadlock Conditions	Data Race Conditions
Eraser	No	No	Yes	Find	Find
SDDCE[a]	No	No	Yes	Find	Find
Ada TIG	No	No	No	Find	Find
RacerX	No	No	No	Find	Find
Ownership Type	Yes	No	No	Find	Find
CSOOCG[b] Analysis	No	No	Yes	Find	N/A

[a] SDDCE is "A Scheme for Dynamic Detection of Concurrent Execution of Object Oriented Software".
[b] CSOOCG is a Context Sensitive Object Oriented Call Graph.

2.3 Static Call Graphs

Here we define the terms call graph (CG), object-oriented call graph (OOCG) and context-sensitive call graph (CSCG). We then proceed to discuss the CSCG classification technique proposed by Grove et al. in [12]. We consider a CG to be *sound* if and only if it accurately represents all possible runtime executions. This section is intended to to situate our CSOOCG, the "heart" of our proposed approach, in prior static analysis work.

2.3.1 Definitions. Grove et al. in [12] define a call graph as "a directed graph that represents the calling relationships between the programs procedures," where procedures represent procedures, functions, and methods. This definition of a CG is typical of the definitions used by Ryder [22], Callahan [2], and Mary W. Hall and Ken Kennedy [13]. We need to account for a subset of the Java language's locking semantics, namely synchronized statements and synchronized methods. We do not account for the new `util.concurrent`-style locks added to Java 5.

Figure 2.1 contains a short example of the Java language's locking semantics accounted for in our call graph representation. This Java example contains four methods (`main`, `addOne_v1`, `addOne_v2` and `addOne_v3`) and two instance variables (`lock` and `count`). The `main` method creates a new instance of `LockSemantics` and assigns it to a local variable named `m_LS`. This `main` method proceeds to call `addOne_v1`, `addOne_v2` and `addOne_v3` before terminating. Method `addOne_v1` is a synchronized method. Methods `addOne_v2` and `addOne_v3` are not synchronized methods, but contain a synchronized statement. In Java, a synchronized method, e.g., `addOne_v1`, is shorthand for enclosing all the method's statements and call sites within a synchronized statement that acquires a lock on the `this` pointer, e.g., `addOne_v2`. Hence, call sites implemented by `addOne_v1()` and `addOne_v2()` will acquire a lock on the object reference pointed to by the receiver variable, i.e. `this` pointer. Call sites implemented by `addOne_v3()` will acquire a lock on the object reference pointed to by the `lock` variable. Our call graph definition must account for both the implicit acquisition of

```
1        public class LockSemantics {
2            private final Object lock = new Object();
3            private int count = 0;
4            public static void main(String[] args) {
5                LockSemantics m_LS = new LockSemantics();
6                m_LS.addOne_v1();
7                m_LS.addOne_v2();
8                m_LS.addOne_v3();
9            }
10           public synchronized void addOne_v1() {
11               count++;
12           }
13           public void addOne_v2() {
14               synchronized (this) {
15                   count++;
16               }
17           }
18           public void addOne_v3() {
19               synchronized (lock) {
20                   count++;
21               }
22           }
23       }
```

Figure 2.1: **Java Locking Semantics Example** This is a short contrived Java code example to illustrate the two different locking semantics accounted for in our call graph representation.

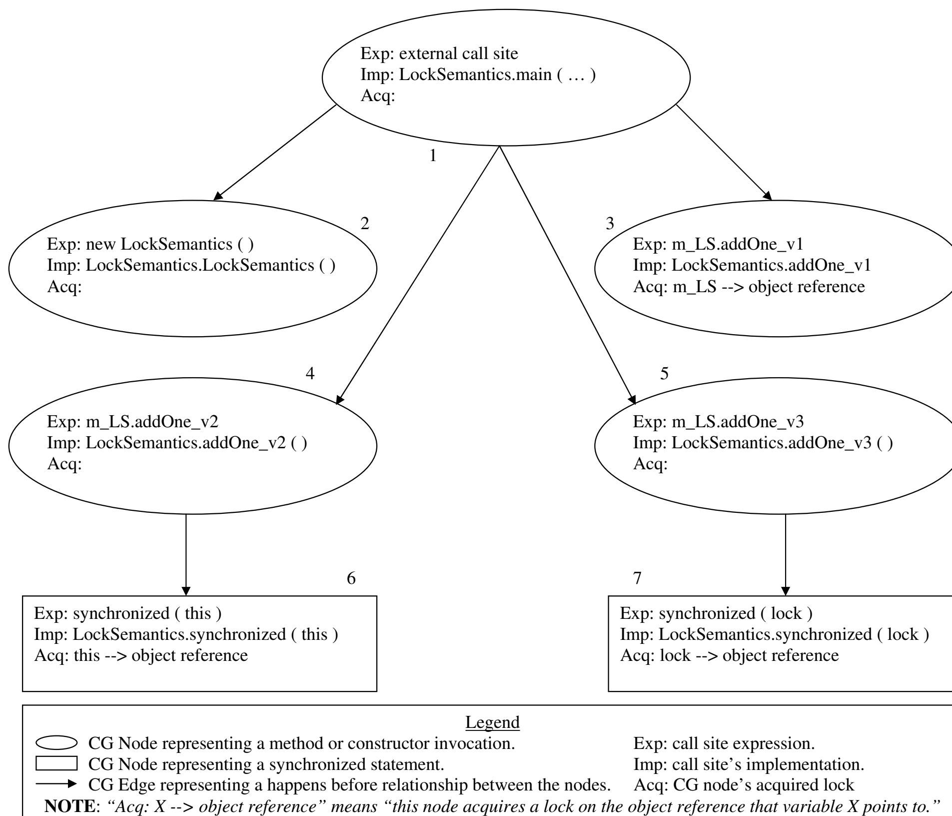


Figure 2.2: **Call Graph of Locking Semantics Example** This is a call graph representation of the Java Locking Semantics Example code in Figure 2.1

the object reference pointed to by the receiver variable and the explicit acquisition of the object reference pointed to by an object reference variable.

An example of our CG representation of our lock semantics is provided in Figure 2.2. Our CG nodes represent particular method and constructor invocations which could occur at runtime. Our CG node representation enables us to account for lock acquisitions. The explicit lock acquisitions are accounted for by considering synchronized statements to be method calls that are always implemented by its associated block statement. Of course, we consider the object reference acquired by the CG node to be the same object reference acquired by the synchronized statement. Our CG edges represent a may call relation and a happens before ordering between the nodes. Our definition of a CG captures the information we will need to determine all the object references being acquired for a given program and the order in which these object references are acquired.

There are times when the full node description does not need to be displayed in the call graph's diagram (e.g., when there is only one possible implementation or when no locks are acquired). In this call graph section (Section 2.3), we use the full node description in the call graph's diagrams to insure the reader's full understanding of our meanings. In the remainder of this thesis we use an abbreviated node description in the call graph diagrams to improve readability. The abbreviated call graph diagram for our lock semantics example code is shown in Figure 2.3.

Not only is there more than one way to define a call graph, but there is more than one way to create it. How a call graph is created determines the classification of the call graph. The call graph in Figure 2.2 was created base on the structure of the Java lock semantics example, Figure 2.1. If the call graph creation process uses object-oriented information to determine the nodes or the edges then the resulting call graph is an *OOCG*. Similarly, if the call graph creation process uses context-sensitive information to determine the nodes or the edges then the resulting call

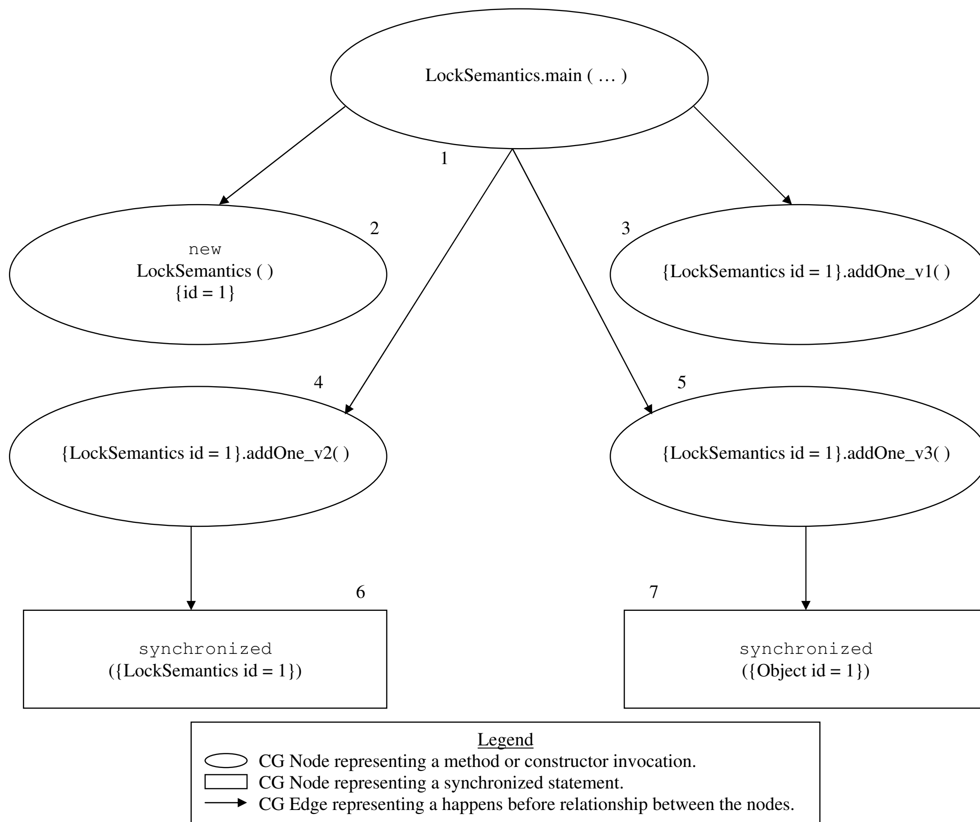


Figure 2.3: **Abbreviated Call Graph of Locking Semantics Example** This is an abbreviated call graph representation of the Java Locking Semantics Example code in Figure 2.1

```
1 public class foo {
2     public static void main(String[] args) {
3         DD d = new FFF();
4         d.m2();
5     }
6 }
```

Figure 2.4: **Class Hierarchy Code Example** This short contrived Java code example is part of the class hierarchy example used to explain OOCG concepts.

graph is a *CSCG*. Our *CSOOCG* generator process uses both object-oriented and context-sensitive information, making our CSOOCG both an OOCG and a CSCG.

2.3.2 Object-Oriented Call Graph Example. Statically creating an OOCG requires object-oriented information concerning the target program. This information includes knowledge of class and interface hierarchies. This information is need to resolve sub-typing relationships between classes and interfaces. This information is also need to resolve method invocations–with respect to determining overridden methods. For example, consider method call `d.m2()` in `foo.main( ...)`, Figure 2.4 and the object-oriented information available in the Unified Modeling Language (UML) class diagram in Figure 2.5. In Java, we can determine the static type of the receiver variable `d`, the subtypes of the static type, and whether or not the subtypes have overridden method `m2()`. The OOCG would need to create four nodes to represent the four possible runtime implementations of method call `d.m2()`, Figure 2.6.

2.3.3 Context-Sensitive Call Graph Example. Statically creating a CSCG requires context-sensitive information concerning the target program. This information includes knowledge of a method call's possible runtime receiver object reference and knowledge of the method call's arguments. Given an OOCG, this information could be used to select possible runtime implementation from the possible object-oriented implementations identified in the construction of the OOCG. For example, consider the the Java code in Figure 2.7, along with the OOCG in Figure 2.8 and the

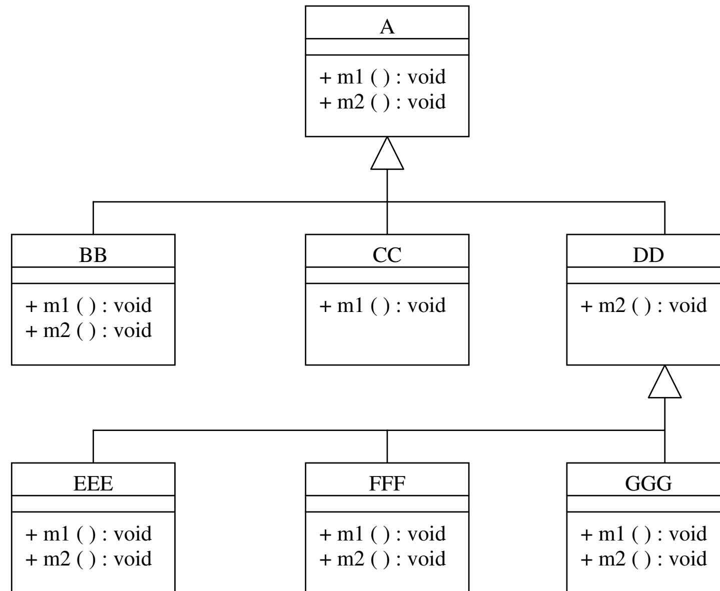


Figure 2.5: **UML Class Diagram** This is a UML class diagram used to illustrate a class hierarchy. This is part of the class hierarchy example used to explain OOCG concepts.

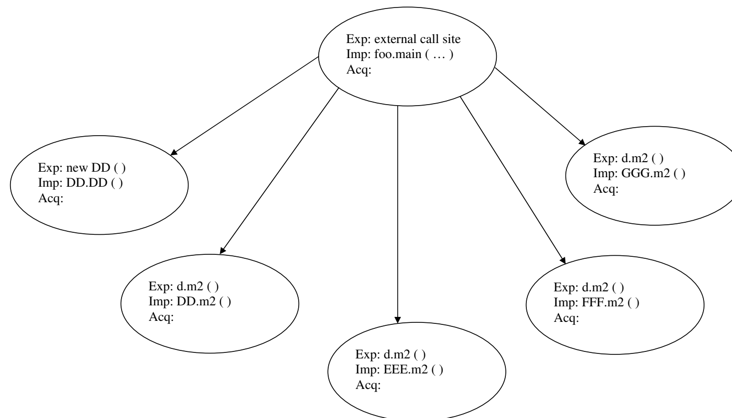


Figure 2.6: **Object-Oriented Call Graph of The Hierarchy Code Example** This is an OOCG representation of the class hierarchy code example in Figure 2.4 based on the object-oriented information provided by the UML diagram in Figure 2.5

CSCG in Figure 2.9. Note: the OOCG is also the CSCG, because no context-sensitive information was used to create the OOCG. We start by describing the Java code.

The Java code in Figure 2.7 contains four classes. An abstract Vehicle class and three concrete classes, namely AirVehicle, GroundVehicle and VehicleExample. The abstract Vehicle class is extended by the AirVehicle and GroundVehicle classes. The VehicleExample class contains the program's entry point, i.e., "public static void main(String[ ]...)." This `main` method serves as the root of our call graphs, Figure 2.8 and Figure 2.9. We start by considering how context-sensitive information improves our OOCG representation of the Java code in Figure 2.7.

The context-sensitive call graph requires the ability to determined statically the context of every expression involved in a call. In this example the information would be used to identifying possible runtime implementation of method calls `v.isAirVehicle()` and `v.isGroundVehicle()`. The context of these two calls determines which of the two possible implementations will be invoked at runtime. This is an improvement to the accuracy of the OOCG representation in Figure 2.8.

This example helps explain how context sensitive can be use to improve the accuracy of an OOCG. This same concept is applied in our analysis. The CSOOCG generator utilizes context-sensitive information to improve its program representation accuracy.

2.3.4 Call Graph Classification. Grove et al. in [12] provides technique to classify program call graphs. This technique system considers the CG's soundness and precision. Grove considers a CSCG to be *sound* if it accurately represents all possible runtime executions and considers a CG to be *precise* if it only represents possible runtime executions. This provides us with a technique to classify our CSOOCG.

Grove has constructed a graphical lattice, shown in Figure 2.10, representing varying degrees of soundness, precision and optimism of CGs. An optimistic CG does not contain all possible runtime executions, but optimistically creates nodes and edges to reflect the most likely runtime executions. The points on the lattice represent CGs,

```
1  public abstract class Vehicle {
2    public abstract boolean isAirVehicle();
3    public abstract boolean isGroundVehicle();
4  }
5
6  public class AirVehicle extends Vehicle {
7    @Override
8    public boolean isAirVehicle() {
9      return true;
10   }
11   @Override
12   public boolean isGroundVehicle() {
13     return false;
14   }
15 }
16
17 public class GroundVehicle extends Vehicle {
18   @Override
19   public boolean isAirVehicle() {
20     return false;
21   }
22   @Override
23   public boolean isGroundVehicle() {
24     return true;
25   }
26 }
27
28 public class VehicleExample {
29   public static void main(String[] args) {
30     Vehicle transport = new AirVehicle();
31     printStatus(transport);
32   }
33   private static void printStatus(Vehicle v){
34     String s1 = ‘‘Transport is an Air Vehicle: ’’;
35     String s2 = ‘‘Transport is a Ground Vehicle: ’’;
36     System.out.println(s1 + v.isAirVehicle());
37     System.out.println();
38     System.out.println(s2 + v.isGroundVehicle());
39   }
40 }
```

Figure 2.7: **Context-Sensitive Java Code Example** This is a short contrived Java code example to illustrate the improvement in accuracy compare to an OOCG.

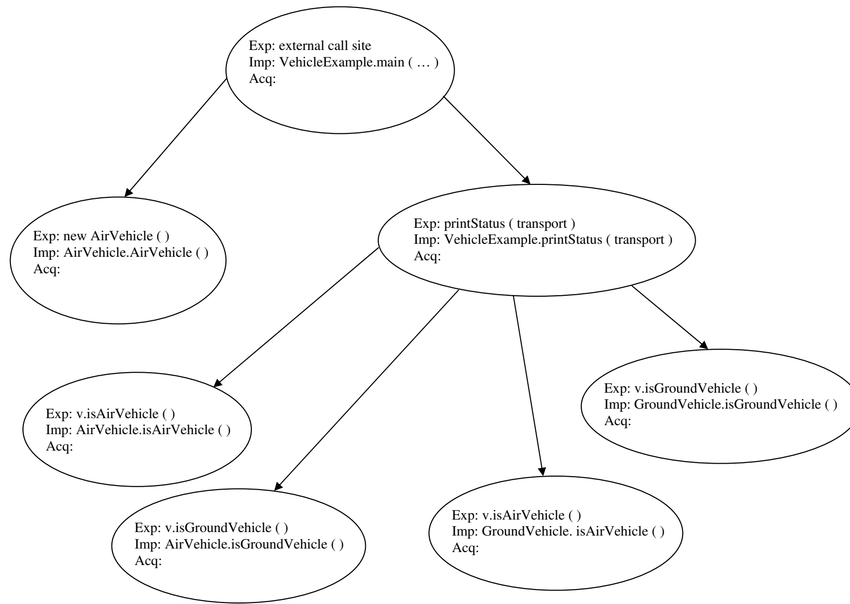


Figure 2.8: **Object-Oriented Call Graph of the Context-Sensitive Java Code Example** This is a context-insensitive call graph representation of the Java code in Figure 2.7

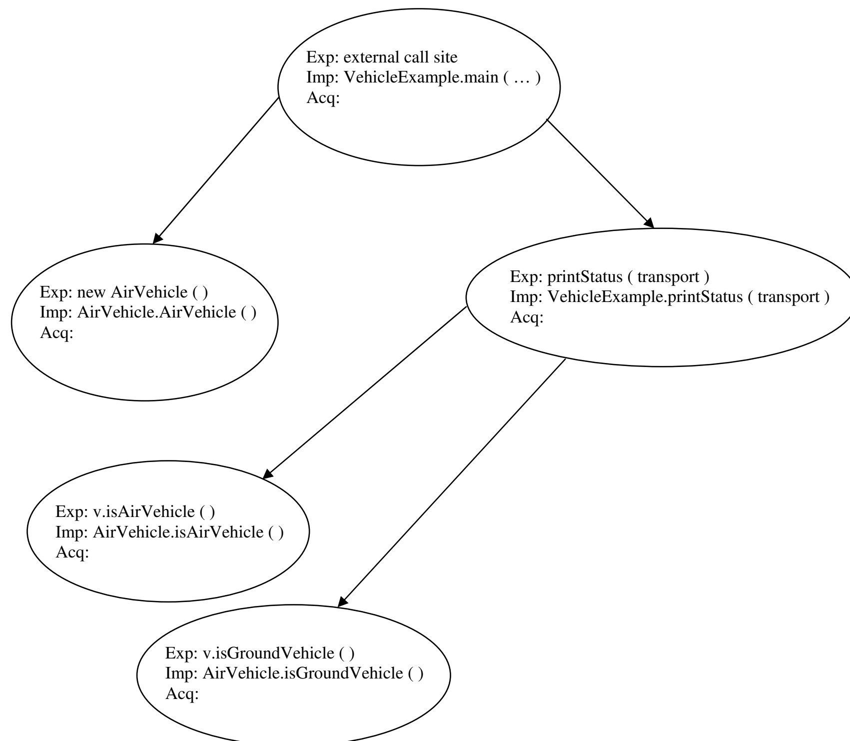


Figure 2.9: **Context-Sensitive Call Graph** This is a context-sensitive call graph representation of the Java code in Figure 2.7

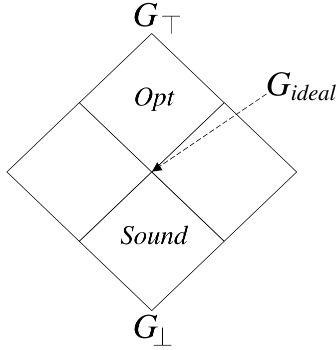


Figure 2.10: **Regions in a Call Graph Lattice Domain (Grove et al. [12])** ***NOTE:*** *This figure, title and description is taken in its entirety from Grove et al. [12].* This diagram depicts a lattice whose elements are call graphs. We order one call graph below another (and depict it in the cone below the other) if it is more conservative (less precise) than the other. The top and bottom elements, corresponding to the empty call graph and the complete call graph, respectively, are denoted by $G_\top$ and $G_\perp$. The point G_{ideal} identifies the "real" but usually uncomputable call graph, which can be described precisely as the greatest lower bound over all call graphs corresponding to actual program executions. Any particular program execution induces a call graph in the cone above G_{ideal} labeled Opt (for "optimistic"). Any call graph produced by a correct call graph construction algorithm must be in the cone below G_{ideal} labeled Sound.

where $G_\top$ represents an empty CG and is considered to be the most optimistic and unsound CG. The remaining two points G_{ideal} and $G_\perp$ are considered sound CG with varying degrees of precision. $G_\perp$ is the most vague (least precise) CG, which consider a call to be implemented by all methods in the program. G_{ideal} is the most precise CG, which is considered to be statically unattainable.

Our CSOOCG is at the G_{ideal} point of of the lattice shown in Figure 2.10. This limits the Java programs our technique can consider to those where it is possible to construct a G_{ideal} call graph. What is required to construct a CSOOCG is nearly perfect static aliasing information about each Java reference variable. We accomplish this via an aliasing oracle as we describe in the next chapter.

III. CSOOCG Generator

This chapter explains what our context-sensitive object-oriented call graph (CSOOCG) is and how it is created. This chapter also defines the aliasing oracle interface needed by our CSOOCG generator. The Dining Philosophers case study is used to help solidify the CSOOCG generator process.

3.1 CSOOCG

A CSOOCG is a Java program's call graph decorated with the locks that may be acquired at runtime. This call graph consist of two types of nodes. One type of node represents method and constructor invocations, where an invocation is defined to be a call site and the implementation of this call site. The other type of node represents synchronized statements. Both node types are needed to capture the two locking mechanisms we are interested in, namely the lock acquired by synchronized statements and the locks acquired by method calls implemented by synchronized method definitions–method definitions with the "synchronized" modifier.

We represent each synchronized statement in the program as if it were a private method invoked only at the point where the synchronized statement occurs. This allows us to account for the locks acquired by these synchronized statements in our CSOOCG. For example, for the purpose of CSOOCG construction, we pretend that

```
public void safePrint(Object o) {
  // code before the synchronized statement
  synchronized(o) {
    System.out.println(o);
  }
  // code after the synchronized statement
}
```

looks like

```
public void safePrint(Object o) {
  // code before the synchronized statement
  syncBlock(o);
  // code after the synchronized statement
}
```

```
private synchronized syncBlock(Object o) {
  System.out.println(o);
}
```

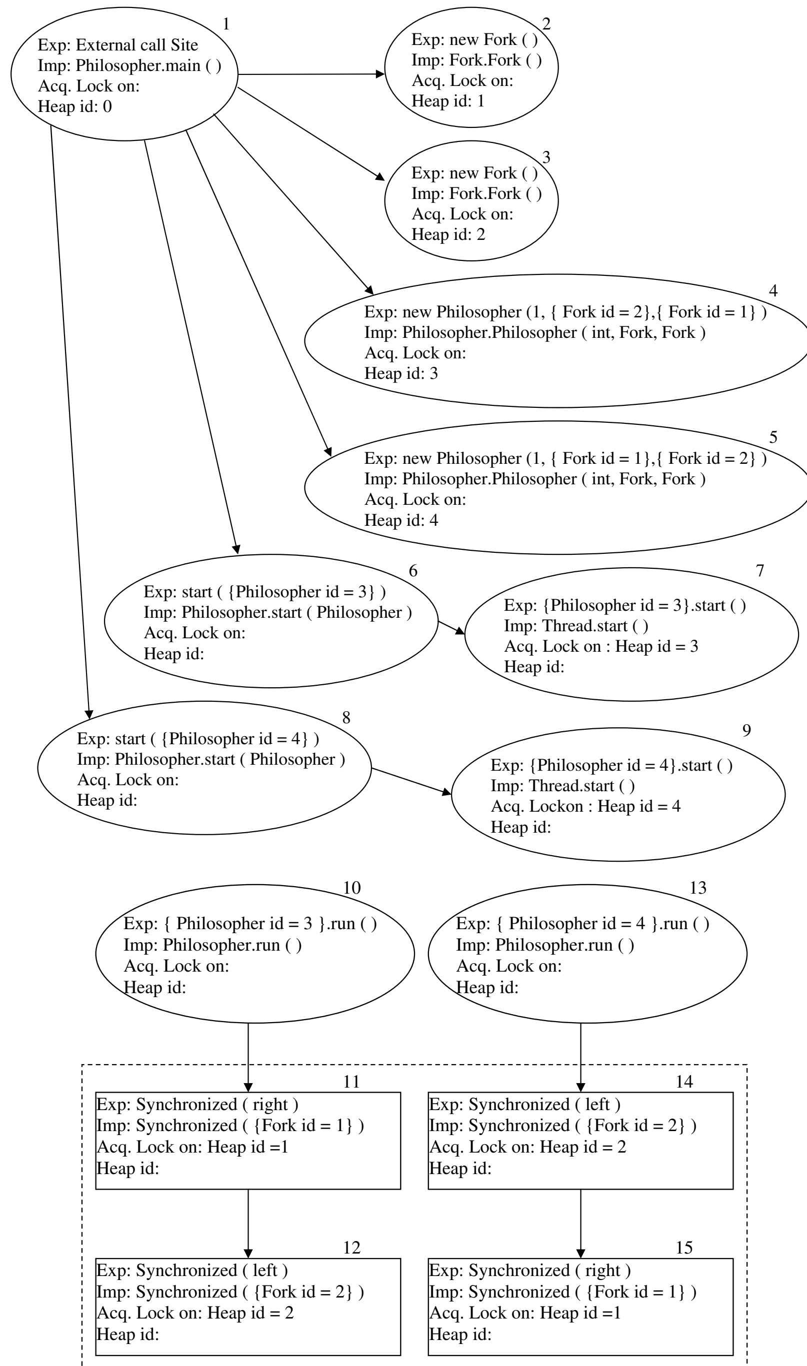
where the introduced method name, in this case `syncBlock`, would be unique within the class. This example is only intended to build intuition, however, and this refactoring of the code is not concretely carried out. The actual CSOOCG node (as we shall see below) replaces its reference to the call site with a reference to the synchronized statement and its reference to the called method or constructor implementation with a reference to the block of statements within the synchronized statement.

The Java source code and CSOOCG for the Dining Philosophers are shown in Figure 3.1 and Figure 3.2, respectively. The CSOOCG for the Dining Philosophers code consists of 3 disconnected graph components. One graph component represents the `main` method call and all the nodes this `main` method call can reach. The remaining two graph components represent the `run` method calls that are *started* in the `main` method and all the nodes this `run` method call can reach. Each connected graph component has one and only one *root node*. Root nodes are the entry points to graph components. Each root node is either a call to `run` or a call to `main`.

The directed edges of the CSOOCG represent a may happen before relationship between the CSOOCG nodes. This relationship between the nodes is used to generate possible runtime executions in the CSOOCG Analyzer. When a runtime execution acquire two or more locks then the edges along this execution are used to establish lock acquisition orders. The node at the tail of a directed edge occur before the node at the head of the same directed edge. In Figure 3.2, for example, the root node, node 1, represents an external invocation of `Philosopher.main (String[] args)`. This invocation creates an object on the heap that we refer to as heap id = 0. This root node would be invoked before all the other nodes in its component, namely nodes 2 through 9. Node 6 represents the `start(p1)` call site located within `Philosopher.main(String[] args)` on line 34 of Figure 3.1 and node 7 represents the `p.start()` call site located within `Philosopher.start(final`

```
public class Philosopher extends Thread {

  public static final class Fork {  }

  final Fork right;

  final Fork left;

  final int identity;

  Philosopher(int identity, Fork right, Fork left) {
    this.identity = identity;
    this.right = right;
    this.left = left;
  }

  @Override
  public void run() {
    while (true) {
      // Thinking
      synchronized (right) {
        synchronized (left) {
          // Eating
        }
      }
    }
  }

  public static void main(String[] args) {
    final Fork f1 = new Fork();
    final Fork f2 = new Fork();
    final Philosopher p1 = new Philosopher(1, f1, f2);
    final Philosopher p2 = new Philosopher(2, f2, f1);
    start(p1);
    start(p2);
  }

  private static void start(final Philosopher p) {
    p.setName(``philosopher-'' + p.identity);
    p.start();
  }
}

```

Figure 3.1: **Dining Philosophers Source Code** This is the Java source code for the Dining Philosopher program.



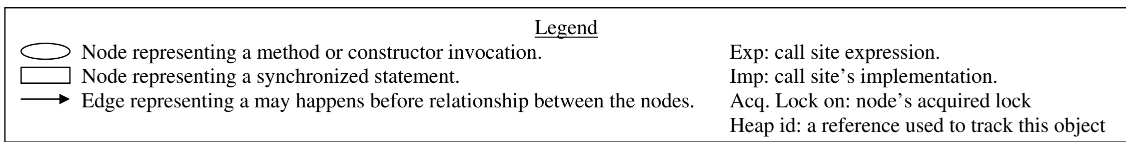


Figure 3.2: **Dining Philosophers CSOOCG** This is the CSOOCG for the Dining Philosophers program example in Figure 3.1.

`Philosopher p)` on line 40 of Figure 3.1. The edge between node 1 and node 6 indicate that node 1 may be invoked before node 6. The edge between node 6 and node 7 indicate that node 6 may be invoked before node 7. Through transitivity we conclude that node 1 may be invoked before node 7.

This Dining Philosophers example has a deadlock condition, as is explained in Chapter IV, that is highlighted by a dashed rectangular box in Figure 3.2. The four CSOOCG nodes in this box become part of two disjoint traces that acquire two object locks in reversed order. The CSOOCG contains information needed to identify object locks and the execution traces that acquire them. For example, information concerning conditional statements, loops and assignment statements is not captured or represented in the CSOOCG. The CSOOCG for the Dining Philosophers does not contain nodes for the `p.setName(‘‘philosopher-’’ + p.identity)` call (which sets the receiver thread's textual name). This is because the method invoked by `p.setName(...)` is implemented in the `Thread` class which is outside the source code of our example. Our prototype tool doesn't analyze Java libraries (i.e., Jar files).

The only exception to this rule is calls that invoke `Thread.start()`. Calls that invoke Thread.start() create a new graph component rooted at Run(). The CSOOCG in Figure 3.2, contains two calls, nodes 7 and 9, that call `thread.start()`. These two calls cause two threads to be created and started. These two new threads of execution are represented in our CSOOCG by the root nodes 10 and 13.

The actual report of the possible deadlock condition within the Dining Philosophers example is shown in Table 3.1. This report informs the programmer of the two possible runtime traces which might deadlock the program (or at least those two threads). Chapter IV explains how the CSOOCG is analyzed to produce this output.

3.2 CSOOCG Generator

The CSOOCG Generator process extracts from the Java source code under test (CUT) all the method and constructor invocations as well as its synchronized

Table 3.1: Dining Philosophers Deadlock Conditions.

CSOOCG Deadlock Conditions					
	Locks Acquired				
ID	First	Second	Method Call	File	Line #
0	heap ID: 1	heap ID: 2			
			Trace 1		
			synchronized (this.right)	Philosopher.java	21
			synchronized (this.left)	Philosopher.java	22
1	heap ID: 2	heap ID: 1			
			Trace 1		
			synchronized (this.right)	Philosopher.java	21
			synchronized (this.left)	Philosopher.java	22

statements. This information is store in a CUT graph. This CUT graph captures the may happen before execution relationship between the three CSOOCG nodes elements; method invocations, constructor invocations and synchronized statements. An aliasing oracle is used to statically determine the possible runtime object reference of reference variables being lock and reference variables acting as method call receivers. The aliasing oracle is used to determine the identity of reference variables being locked and to improve the precision of the CSOOCG. Here improved precision refers to eliminating execution paths through the CSOOCG that may not occur at runtime. The CSOOCG generation process creates a CUT graph form the Java source files and uses an aliasing oracle to determine the identity of objects being acquired along CSOOCG paths and the improve the precision of the CSOOCG.

3.2.1 Starting Point. The creation of a CSOOCG starts with a forest of abstract syntax trees (ASTs) consisting of an AST for each Java class. This forest of ASTs is traversed with a double visitor to capture the program's structure. We call the structure we create from this double visitor traversal of the AST forest a CUT graph.

A CUT graph is a bipartite graph. One set of nodes (rectangles) represents blocks of code and the other (ellipses) represents call sites. An edge from a code

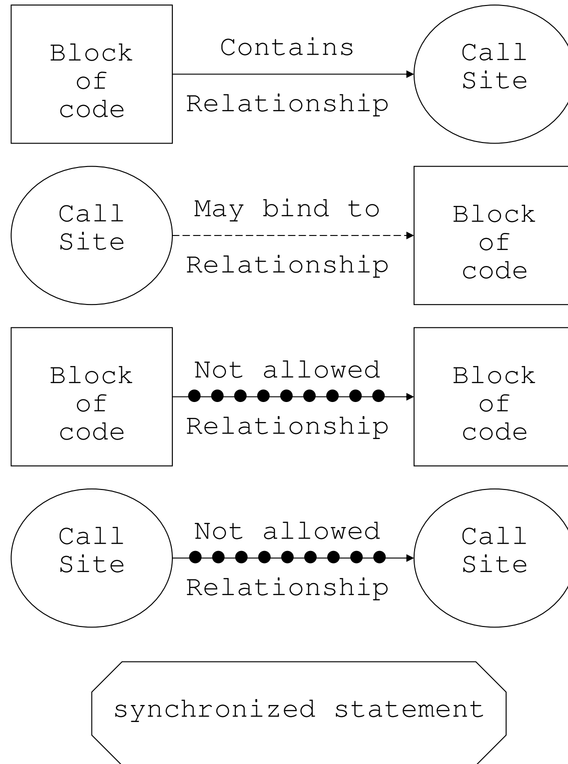


Figure 3.3: **CUT Graph Semantics**

block node to a call site node means that the code contains the call site. An edge from a call site to a code block means that the site may bind to the code block (possibly not until runtime). Synchronized statements are represented by an octagon. Synchronized statements represent both a block of code and a "call" site in our CUT graph (saving us a node and an edge).

3.2.2 Code Under Test (CUT) Graph. The CUT graph is built in several stages. The initial CUT graph consist of nodes for each code block and call site in the AST forest, but no "may bind to" edges. Figure 3.4 shows the initial CUT graph for the Dining Philosophers.

3.2.3 Call Completer. This step completes the CUT graph (later Figure 3.5 shows the complete CUT graph for the Dining Philosophers) by connecting calls sites to their possible implementations. It uses the call site receivers' and arguments' typing

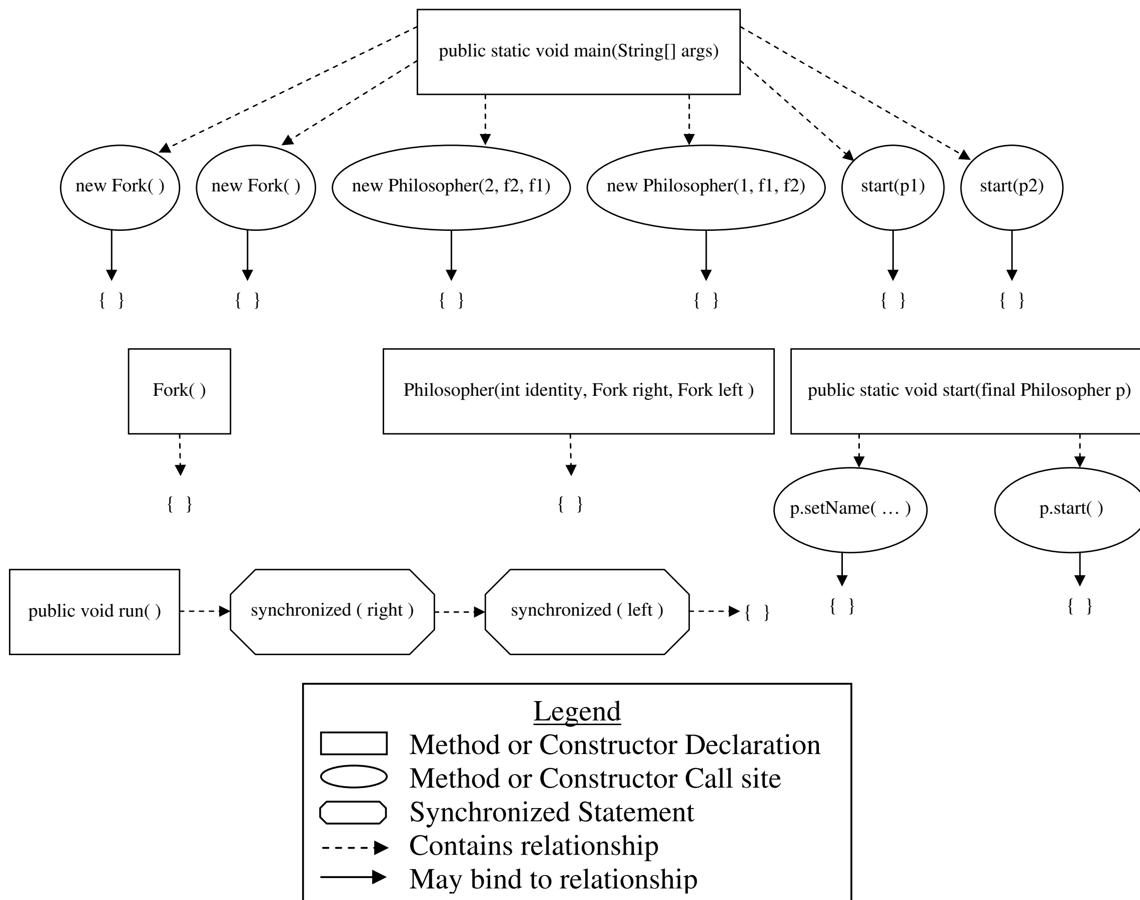


Figure 3.4: **Dining Philosophers Initial CUT Graph** This is the initial CUT graph for the Dining Philosophers program example in Figure 3.1.

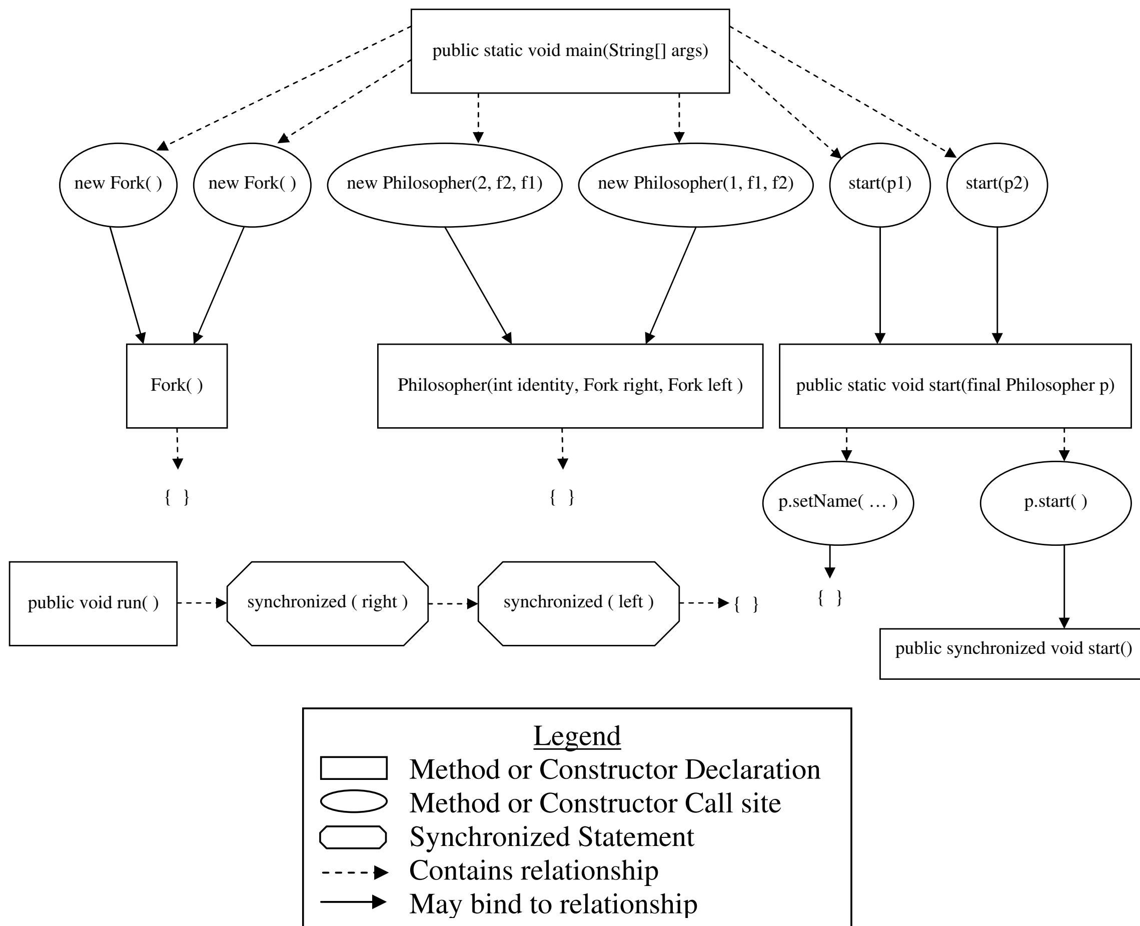


Figure 3.5: **Dining Philosophers Updated CUT Graph** This is the updated CUT graph for the Dining Philosophers program example in Figure 3.1.

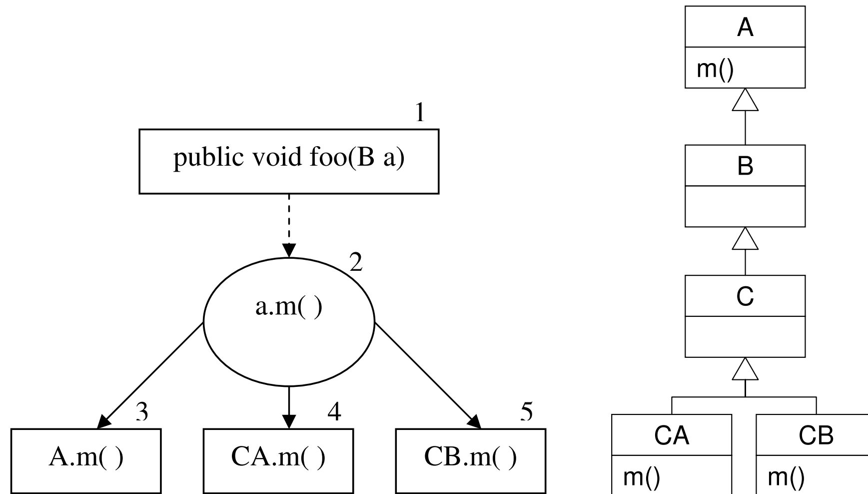


Figure 3.6: **CUT Graph is Object-Oriented**

information to determine possible runtime binding to an implementation. The receiver is the only runtime type considered because Java only performs dynamic dispatch on the receiver object. The static type of the call site's receiver and arguments do limit the possible implementations to those located within the declared class type or within classes that extend the declared class type (and in some cases, like the situation in Figure 3.5, this is enough to uniquely identify the runtime binding). At this point any call sites with no "may bind to" edges means the possible call binding is outside the program's source code (it is most likely contained within a Jar file acting as a library for the program).

The CUT graph is object-oriented but not context-sensitive. Consider the class hierarchy shown in Figure 3.6 and the following Java code segment:

```
public void foo(B a) {
  a.m( )
  }
```

The corresponding CUT graph for this segment of Java code is shown in Figure 3.6. The possible object-oriented implementations of `a.m()` is illustrated by the three edges from Node 2 to Nodes 3, 4 and 5. Our Dining Philosophers example does not have multiple implementations possibilities for the call sites.

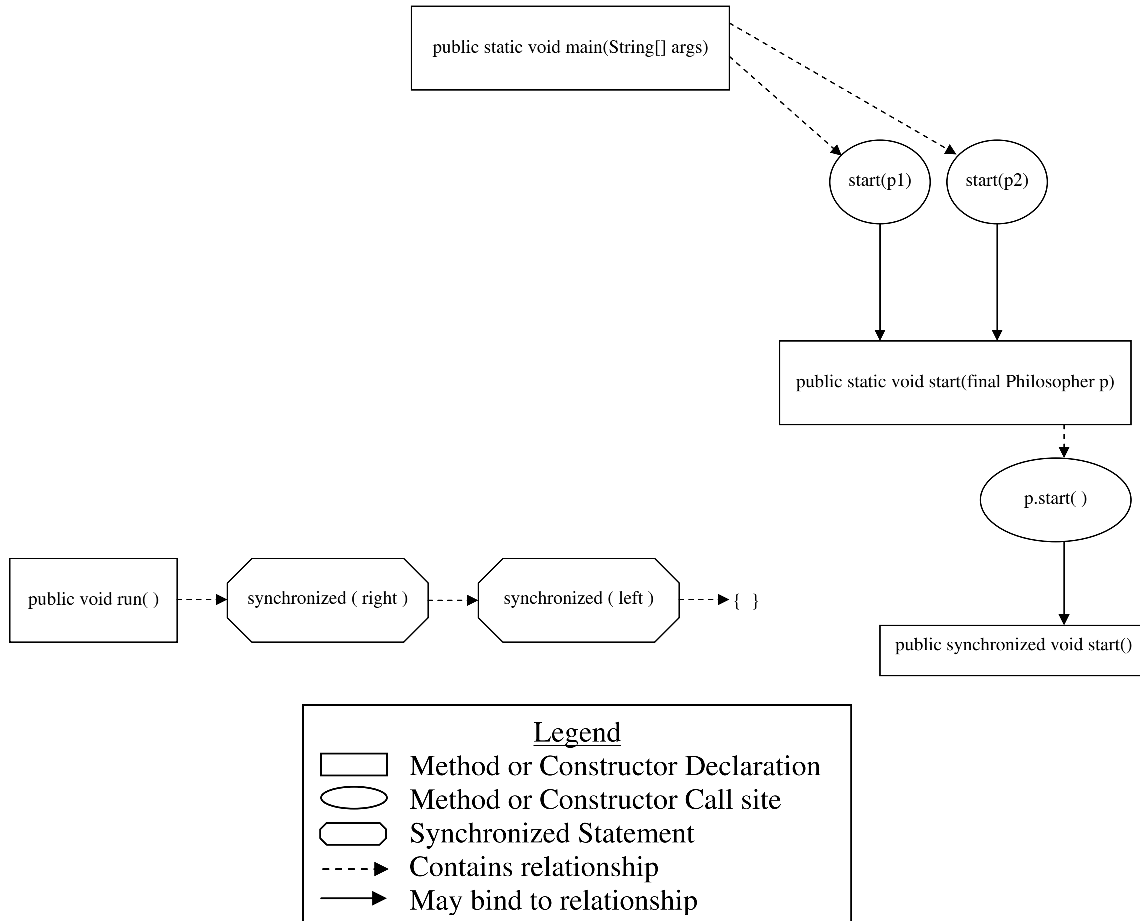


Figure 3.7: **Dining Philosophers Reduced CUT Graph** This is the reduced CUT graph for the Dining Philosophers program example in Figure 3.1.

3.2.4 Reducer Process. It is possible to reduce the CUT graph because we only care about the traces that acquire locks or start threads. Reducing the size of the updated CUT graph reduces the size of the resulting CSOOCG. Nodes in CUT graph paths that do not acquire at least one lock or start a thread can be deleted. A single depth first search traversal of the CUT graph's entry points is sufficient to determine unneeded nodes. The unneeded nodes can then be deleted from the graph along with their edges. We call this new version of the updated CUT graph the reduced CUT graph, Figure 3.7. The corresponding reduced CSOOCG created from this reduced CUT graph is shown in Figure 3.8.

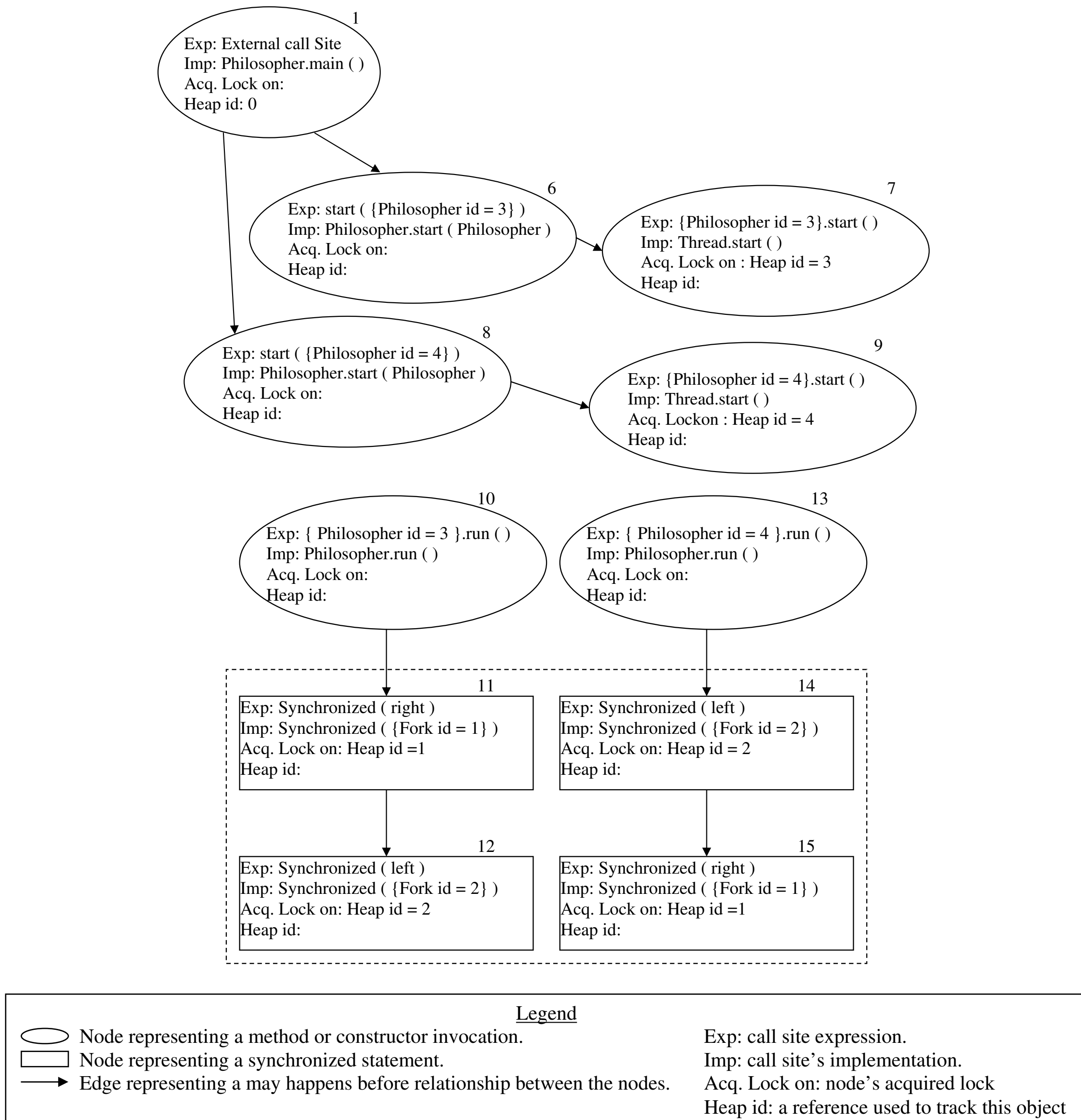


Figure 3.8: **Dining Philosophers Reduced CSOOCG** This is the reduced CSOOCG for the Dining Philosophers program example in Figure 3.1.

3.2.5 aliasing Oracle. It is not possible to create a perfect CSOOCG for most Java programs. This is primarily due to aliasing and late binding (dynamic binding). The identity of every locked object must be known. This means that the references (aliasing) to locked objects must be statically resolvable. The runtime binding of every method call must be able to be statically determined in order to create a precise CSOOCG. The precision minimizes false possitive, but the main reason for an aliasing oracle is the need to statically resolve the identity of references variables being locked.

The aliasing oracle does not directly resolve runtime method bindings but, it does statically models the program's runtime heap to allow us to understand what objects each reference variable may point to at runtime. This allows our analysis to understand aliasing of references in the program for the purpose of determining runtime method bindings and lock object aliasing.

Consider Figure 3.9, we have a Java method with a synchronized statement that acquires a lock on the object reference pointed to by `o1`. What does `o1` reference at runtime? We can't be sure statically, it might be what `p1` referenced or what `p2` referenced. To be conservative we consider both. More precisely, if we define a function, *hm*, which takes a reference as its only argument and returns the set of possible runtime object references, then the set returned would be the union of possible runtime references of `p1` and `p2`, i.e.,

$$hm(\texttt{o1}) = hm(\texttt{p1}) \cup hm(\texttt{p2}).$$

If an aliasing oracle is given `o1` as an expression, then it would return the union of the object references pointed to by expressions `p1` and `p2`. We did not write this alias analysis as part of our work, but did design an interface to it as described below.

3.2.6 Oracle Interface. Figure 3.10 is a Unified Modeling Language (UML) class diagram of the aliasing oracle used by the CSOOCG generator to resolve ref-

```
1          void m(Object p1, Object p2) {
2            Object o1;
3            if (Math.random() > 0.5)
4              o1 = p1;
5            else
6              o1 = p2;
7            synchronized (o1) {
8              // do stuff
9            }
10         }
```

Figure 3.9: **Java Source Code Example Demonstrating Aliasing**

erence variables into possible runtime object references. The aliasing oracle provides the `getHeapReference` method for this function. The `getHeapReference` method takes three parameters, the expression's IRNode (a Fluid internal representation (IR) node), the context the call is being made under, and the Fluid IBinder (which is used to determine static bindings in the Fluid analysis infrastructure). The expression is the reference variable we wish to resolve into the possible runtime object references. The Fluid IBinder provides static binding, such as binding for local variables. The context the call is being made under encompasses three things:

1. the receiver of the call site,
2. the mapping between the actual arguments and the formal parameters and
3. the method definition the call site is located in.

This `getHeapReference` method returns a set of static object references, that statically model runtime objects (and their associated types).

We now provide an example of how a runtime method binding is resolved using our aliasing oracle. This aliasing oracle is external to our CSOOCG analysis. Consider the CSOOCG for the Dining Philosophers in Figure 3.2 (on page 35). Node 11, a synchronized block, was created with this aliasing oracle. The call site expression for this node was `synchronized(right)`. Our CSOOCG generator process determines that this expression is a synchronized block and that a lock is acquired on the run-

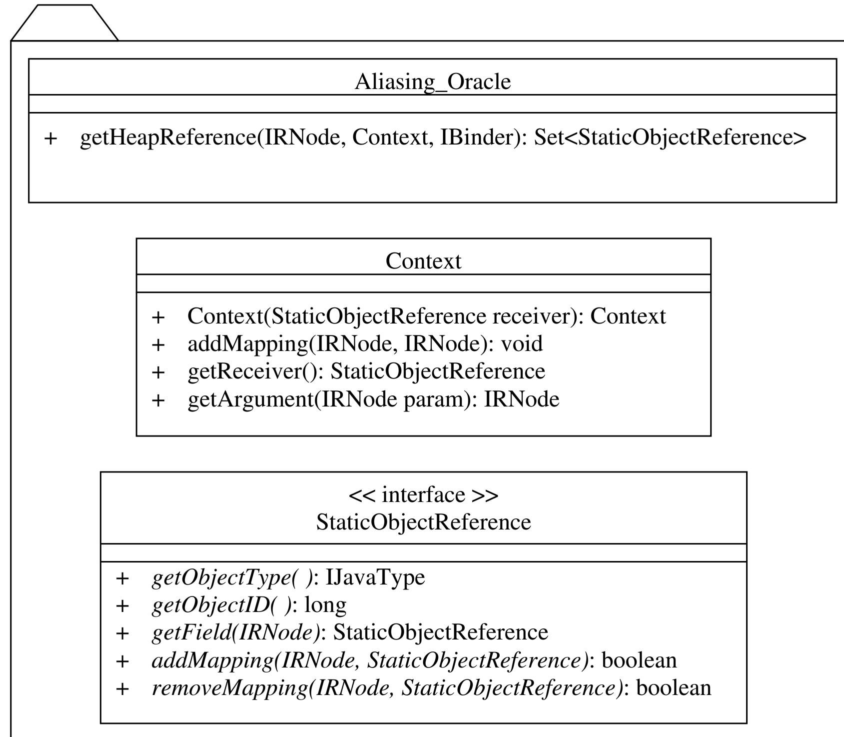


Figure 3.10: **Alias Oracle Interface**

time objects references pointed to by `right`. We must determine the context of this call, i.e., determine the possible runtime objects that `this` may be pointing to. The context of a synchronized block is determined by the receiver of the enclosing method invocation. In this case the receiver is `Philosopher id = 3`. This is the receiver of the call to `Thread.start()` in CSOOCG node 7. This call created and started CSOOCG node 10, {`Philosopher id = 3`}`.run()`. We provide the reference variable, `right`, and the context of this call site, `Philosopher id = 3`, along with the current binder to the aliasing oracle `getHeapReference` method. The return value is set of object references, namely {`FORK id = 1`}. We annotate in node 11 that it acquires a lock on heap id = 1.

3.2.7 CSOOCG builder. The CSOOCG builder process uses the aliasing oracle and the reduced CUT graph to produce a the CSOOCG. The CSOOCG is created via continued traversals of the reduced CUT graph until all call site nodes,

have been visited. We consider a CUT graph call site node to be visited if the call site is reached under the same calling context. Here we define the context to be the receiver's identity and the identity of all the passed in reference variable arguments. The CUT graph traversal starts with the `main` method definitions. These `main` method definitions are entry points to CUT graph components and become entry points in the CSOOCG components. There is a one-to-one correspondence between the `main` entry points in the reduced CUT graph and the `main` entry points in the CSOOCG. For example, the main entry point for our Dining Philosophers is `Philosophers.main`. The receiver for this `main` entry point is the enclosing class object, i.e., the Philosopher static class object. The remaining call site's receivers are determined by querying the aliasing oracle with the call site's receiver reference variable and the context under which this call would be made at runtime.

For the `main` method definition from our Dining Philosophers example code the remaining component call sites are:

- new Fork( )
- new Fork( )
- new Philosopher(1, f1, f2)
- new Philosopher(2, f2, f1)
- start(p1)
- start(p2)
- p1.start()
- p2.start()

The constructor call sites do not need an aliasing oracle to determine their runtime implementations. For the remaining 4 method calls the CSOOCG builder ask the aliasing oracle for the set of possible object references these method call site receivers may be at runtime. The response from the aliasing oracle is used to

select the possible runtime implementation of the method call sites. The resulting implementations of the above calls are listed below:

- Fork.Fork( )
- Fork.Fork( )
- Philosopher.Philosopher(1, f1, f2)
- Philosopher.Philosopher(2, f2, f1)
- Philosopher.start(p1)
- Philosopher.start(p2)
- Thread.start()
- Thread.start()

Remaining CUT graph call site and implementation node pairs are added to the CSOOCG as a single node. Each time a CUT graph call site and implementation node pair are visited, may results in a new unique CSOOCG node.

The `p.start()` call sites are implemented by `Thread.start()` and requires additional processing. We first create a unique CSOOCG node for this special case and then create a new CSOOCG component with the corresponding `run` method definition as its entry point. The receiver for this `run` entry point is the same receiver identified in the `start()` call site. The `run` CUT graph components are traversed in the same way as the `main` CUT graph components (i.e., they are staring points in the CUT graph). In our Dining Philosophers example we create two CSOOCG nodes, one for the `p1.start()` invocation in `Philosopher.main` and one for `Philosopher.run()`. In Figure 3.8, these two nodes are 7 and 10 for the first implementation of `p.start()` nodes 9 and 13 for the second invocation of `p.start()`.

Synchronized statements in the CUT graph are another special case. We consider synchronized statements to be both a call site and an implementation and hence one synchronized statement satisfies the "call site and implementation node pair"

requirement for a single CSOOCG node. In our Dining Philosophers example, the CUT graph node for `synchronized(right)` becomes node 11 and 14 in CSOOCG Figure 3.8–one node for each thread's invocation of the synchronized statement. Determining the set of object reference locks that may be acquired by a CUT graph synchronized statement node is done by querying the aliasing oracle. In our example this means the reference variable `right` is sent to the aliasing oracle, along with the context (heap id = 3 for the philosopher object corresponding to `p1`), and the current binder. The response from the aliasing oracle is {Fork id = 1}. We annotate in CSOOCG node 11, that it acquires a lock on heap id=1. CUT graph call sites pointing to an empty set are ignored, because their implementation's source code is not available.

CSOOCG creation is complete when all the updated CUT graph `main` entry points and identified `run` entry points have been traversed.

3.3 CSOOCG Soundness

In Chapter II the term soundness and precision with respect to a call graph were highlighted, where Grove et al. [12] considers a call graph to be *sound* if it accurately represents all possible runtime executions and *precise* if it only represents possible runtime executions. The CSOOCG is sound due to the representation of all possible object-oriented implementations of a call site in the CUT graph. During the generation of the CSOOCG all the CUT graph call site nodes are visited. When an aliasing oracle is not available the CSOOCG nodes corresponding to each possible implementation is be generated. This accounts for all the possible implementations of each call site in the original Java source code. This generation process reproduces in the CSOOCG all the possible runtime implementations of each call site in the original Java source code. This is why we state that the CSOOCG is sound.

The aliasing oracle use to determine the possible runtime identity of object reference variables being acquired is also used to improve the precision of the CSOOCG. We can provide the aliasing oracle with the receiver variable of a method call site

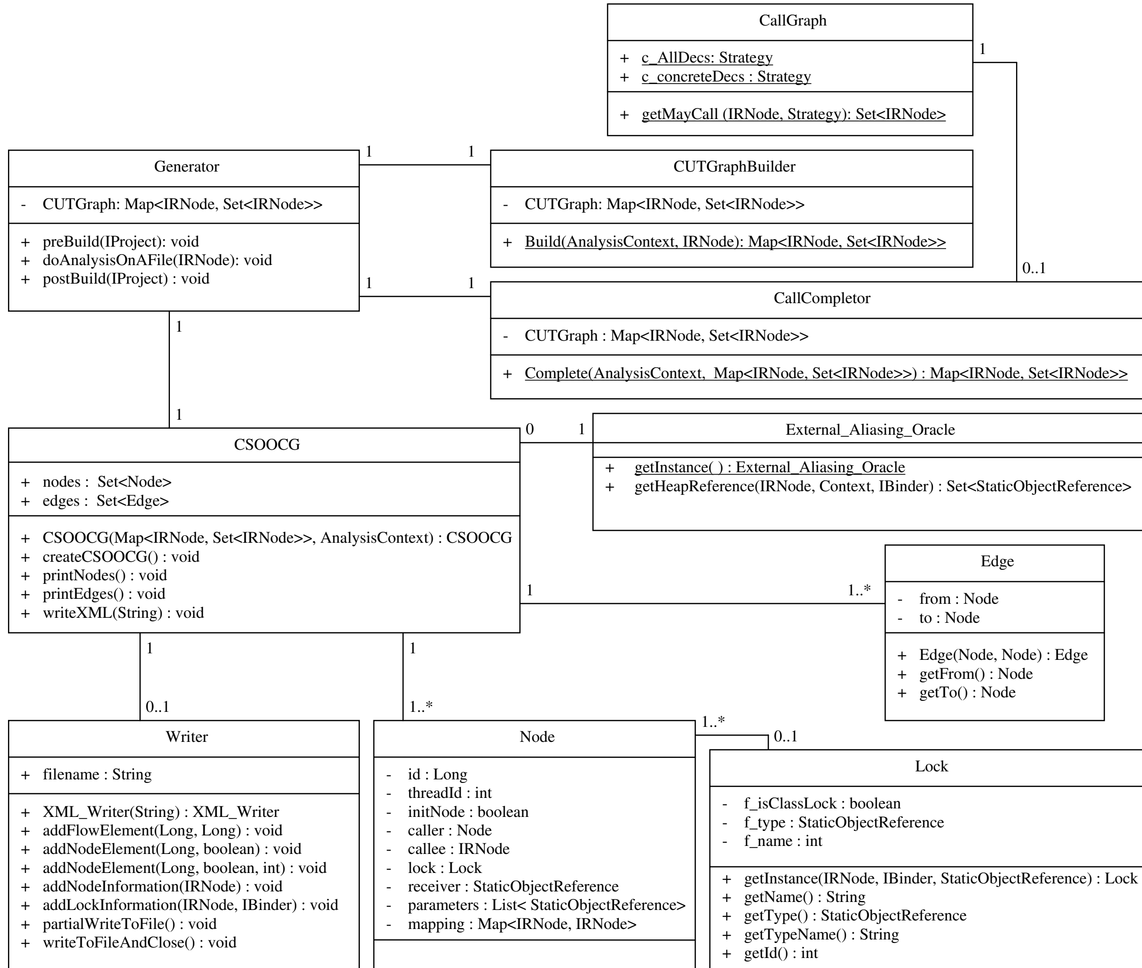


Figure 3.11: **Generator Class Diagram** This is a UML class diagram representing the classes used to create and store the CSOOCG. This diagram also shows the links between these classes.

and receive as a response the set of object reference the receiver variable may be at runtime. The types of this set of object reference can be used to determine the valid runtime implementations. The information helps improve the precision of the CSOOCG. When the aliasing information is prefect, the CSOOCG is considered sound and precise, that is Grove's G_{ideal}.

3.4 Generator Implementation

The CSOOCG generation process is made up of many sub-processes. Figure 3.11 is a UML class diagram displaying the most important of these classes. We discuss

these classes and their relationships to each other. The primary class in this class diagram is the *Generator*. The *Generator* class is called by the Eclipse IDE. First, Eclipse invokes `preBuild` before the CUT files are loaded into Eclipse. Then, Eclipse invokes `doAnalysisOnAFile` for each CUT file as it is loaded into Eclipse. Once all the CUT files are loaded into Eclipse the `postBuild` method is invoked. Each method invocation is designed to perform a special task. The `preBuild` method is designed to initialize all the CSOOCG data structures, namely the CUT graph map and the CSOOCG. The `doAnalysisOnAFile` method is designed to build the CUT graph incrementally as the CUT files are loaded into Eclipse. The `postBuild` method implementation calls the *Call Completor* process to add all possible method call implementations to the CUT graph. The *Call Completor* uses the Fluid call graph utility class to determine method call implementations. The `postBuild` then calls on the *Builder* class to create the CSOOCG and save it to disk via the *Writer* class.

3.4.1 CSOOCG Data File. Now we need to store our CSOOCG (either the CSOOCG in Figure 3.2 or the reduced CSOOCG in Figure 3.8) in a file that the CSOOCG analyzer can read and process. Why are we not analyzing the CSOOCG now? Our design breaks up the analysis into two parts in order to reduce the amount of memory required to perform our analysis. The generation of the CSOOCG relies upon the Fluid IR to build a forest of ASTs. Once we generate the CSOOCG, we no longer need this forest of ASTs. It is not possible to delete this forest of ASTs and still maintain the CSOOCG for our analysis. This is why we create a data file representation of the CSOOCG to pass on to the CSOOCG analyzer.

The XML file format was selected to maximized the data file's portability. We could not create the whole XML document in memory, so we create and save the XML document in stages, where each successive stage is appended to the previous stage's data file.

An abbreviated CSOOCG data file for our Dining Philosopher example is shown in Figure 3.12. This data file contains four distinct sections. The first section is the CSOOCG nodes, Figure 3.12 lines 4–5. This first section contains three pieces of information; node Id, root node boolean value, and lock acquired heap id–if any. The second section contains the CSOOCG directed edges, Figure 3.12 lines 6–7. The third section contains additional CSOOCG node information, e.g., the name of the file containing the code this CSOOCG node represents, Figure 3.12 lines 8–27. The fourth section contains lock information, e.g., lock ID 4 is acquired by a synchronized statement on object `this.left`, which is heap object ID 2. The first two sections are used by the CSOOCG Analyzer (described in the next chapter) to determine if deadlock may occur in the Java program. The additional two sections are used to help the the CSOOCG Analyzer produce results that the programmer can understand.

3.5 A Second Example: Double Lock Equals

The Double Lock Equals example, Figure 3.13, illustrates how deadlock conditions may exist in the simplest of code segments. The main idea behind our Double Lock Equals example is the need to lock both objects before comparing them for equality. The first object locked is the *this* object. The second object locked is the passed in *object*. The problem is not the locking order of the reference variables, but the actual locking order of the objects referenced by these reference variables. The overridden equals method is called twice with its argument and receiver object swapped[1]. This means the locking order is also swapped between the two calls. This inconsistency in locking order is detectable by our CSOOCG analysis.

This program has nineteen nodes of interest. The `DoubleLockEqualsMain.main` method is the entry point for this program. This `main` method instantiates two `DoubleLockEqualsMain` objects by calling the default constructor for this class. These two constructor calls are illustrated in Figure 3.14 as nodes 2 and 4. These two nodes

[1]This swapping between items to determine equality is similar to the definition of equality between sets, i.e., $A = B \Leftrightarrow A \subset B \wedge B \subset A$.

```
<?xml version=‘‘1.0’’ encoding=‘‘UTF-8’’?>
<!--These are the node and flow values used in the CSOOCG-->
 <CSOOCGData>
  <Node Id=‘‘1’’ Root=‘‘true’’ Lock=‘‘’’/>
  <Node Id=‘‘2’’ Root=‘‘false’’ Lock=‘‘’’/>
  <Flow From=‘‘1’’ To=‘‘2’’/>
  <Flow From=‘‘1’’ To=‘‘3’’/>
  <NodeInformation Id=‘‘1’’>
    <Kind>Method call</Kind>
    <Package/>
    <Filename>Philosopher.java</Filename>
    <Line>29</Line>
    <Type>public class Philosopher # extends # # #</Type>
    <MethodCall>Philosopher.main(String [])</MethodCall>
    <NewExpression/>
    <SynchronizedBlock/>
  </NodeInformation>
  <NodeInformation Id=‘‘2’’>
    <Kind>New Expression</Kind>
    <Package/>
    <Filename>Philosopher.java</Filename>
    <Line>33</Line>
    <Type>public class Philosopher # extends # # #</Type>
    <MethodCall/>
    <NewExpression>new Philosopher (2, f2, f1)</NewExpression>
    <SynchronizedBlock/>
  </NodeInformation>
  <LockInformation Id=‘‘3’’>
    <AcquiredBy>Synchronized Statement: node id = 12</AcquiredBy>
    <Static>false</Static>
    <Final>false</Final>
    <MethodCall/>
    <SynchronizedBlock>synchronized (this.right)</SynchronizedBlock>
    <Object>heap ID: 1</Object>
  </LockInformation>
  <LockInformation Id=‘‘4’’>
    <AcquiredBy>Synchronized Statement: node id = 13</AcquiredBy>
    <Static>false</Static>
    <Final>false</Final>
    <MethodCall/>
    <SynchronizedBlock>synchronized (this.left)</SynchronizedBlock>
    <Object>heap ID: 2</Object>
  </LockInformation>
</CSOOCGData>
```

Figure 3.12: **Abbreviated Data File** This is an abbreviated data file used as a small representation of the overall contents stored in the actual XML file.

create and start the new thread objects via nodes 3 and 5. Starting these two threads equates to calling their corresponding `run` methods. These two run method are nodes 6 and 13 in Figure 3.14. These two `run` methods contain calls to the overridden equals method. These calls correspond to nodes 7 and 10 for node 6, and nodes 14 and 17 for node 13. The Equals method contains two synchronization blocks. These nested blocks correspond to nodes 8, 9, 11, 12, 15, 16, 18, and 19. These eight synchronization nodes are the only nodes that acquire locks in our figure.

Now that we have identified all the nodes in the *CSOOCG*, we will identify their directed edges, i.e., their "may happen before" ordering. The program's `main` entry point make two calls to the DoublLockEqualsMain constructors. These two calls are represented by the two arrows leaving the `main` method call, node 1, and entering the constructor calls, nodes 2 and 4. The directed edges illustrate a happens before relationship between the CSOOCG nodes. The remaining directed edges are easily read off Figure 3.14.

This chapter has covered the CSOOCG generation process in detail with a step by step example to solidify the concepts and needs behind each step. The CSOOCG data structure contains all the possible runtime execution paths and identifies the actual object references being acquired along these paths. This information is passed on to the CSOOCG analyzer process to determine if deadlock conditions exist in the original CUT files used to create the CSOOCG.

```java
public final class DoubleLockEqualsMain extends Thread {

  static final DoubleLockEquals f1 = new DoubleLockEquals();

  static final DoubleLockEquals f2 = new DoubleLockEquals();

  public static void main(String[] args) {
    (new DoubleLockEqualsMain()).start();
    (new DoubleLockEqualsMain()).start();
  }

  @Override
  public void run() {
    while (true) {
      boolean result = f1.equals(f2) == f2.equals(f1);
    }
  }
}

public final class DoubleLockEquals {

  private long f_readCount = 0;

  @Override
  public boolean equals(Object obj) {
    if (obj instanceof DoubleLockEquals) {
      synchronized (this) {
        synchronized (obj) {
          this.f_readCount++;
          ((DoubleLockEquals) obj).f_readCount++;
          return super.equals(obj);
        }
      }
    }
    return false;
  }

  // OTHER IMPLEMENTATION CODE

}
```

Figure 3.13: **Double Lock Equals Java Source Code** This is the Java source code for the Double Lock Equals program.

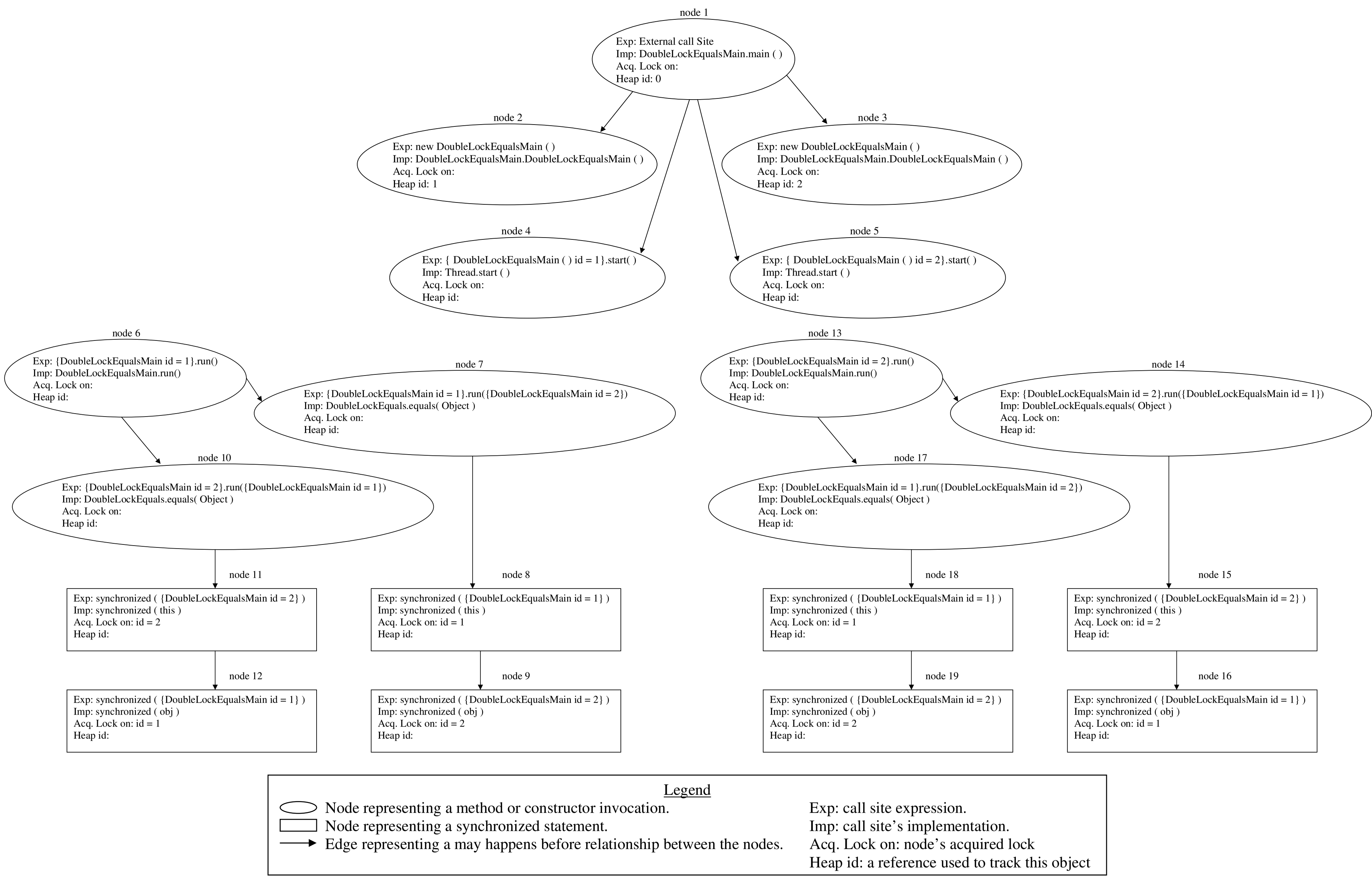


Figure 3.14: **Double Lock Equals CSOOCG** This is the reduced CSOOCG for the Double Lock Equals program example in Figure 3.13.

IV. CSOOCG Analyzer

This chapter describes the context-sensitive object-oriented call graph (CSOOCG) analyzer. This component takes the CSOOCG created by the CSOOCG generator, described in Chapter III, and produces results for the programmer. In this chapter we continue to use the Dining Philosophers program as a running example (and present results for the Double Lock Equals lock program at the end of the chapter).

4.1 Overview

The CSOOCG analyzer proceeds as follows. It first traverses the CSOOCG and identifies the possible runtime traces. These traces are used to determine the acquisition order relation between the program's lock object references. It then computes the transitive closure of these lock object references, with respect to this relation, and determine if any cycles exist. If a cycle does exist, then we report the lock object references involved in creating the cycle and the traces that acquire them as a possible deadlock condition to the programmer. If no cycles exist, then we report to the programmer that no deadlock conditions were found. For programs with a CSOOCG based upon perfect aliasing information (and for which we have all the program's source code) this is a strong assurance that the program is deadlock-free.

4.2 Analysis Model

In this section we present a formal model of the analysis done on the CSOOCG to produce results for the programmer. Consider Figure 4.1, the CSOOCG for the Dining Philososphers. This CSOOCG contains 15 nodes and 12 edges in three connected graph components. We consider an edge to be an ordered pair where the first element in the order pair represents the tail of an edge and the second element represents the head of an edge. In our example, the ordered pair $< 1,\ 2 >$ represents the edge leaves CSOOCG node 1 and arrives at CSOOCG node 2. We say that node 1 *may be invoked before* node 2.

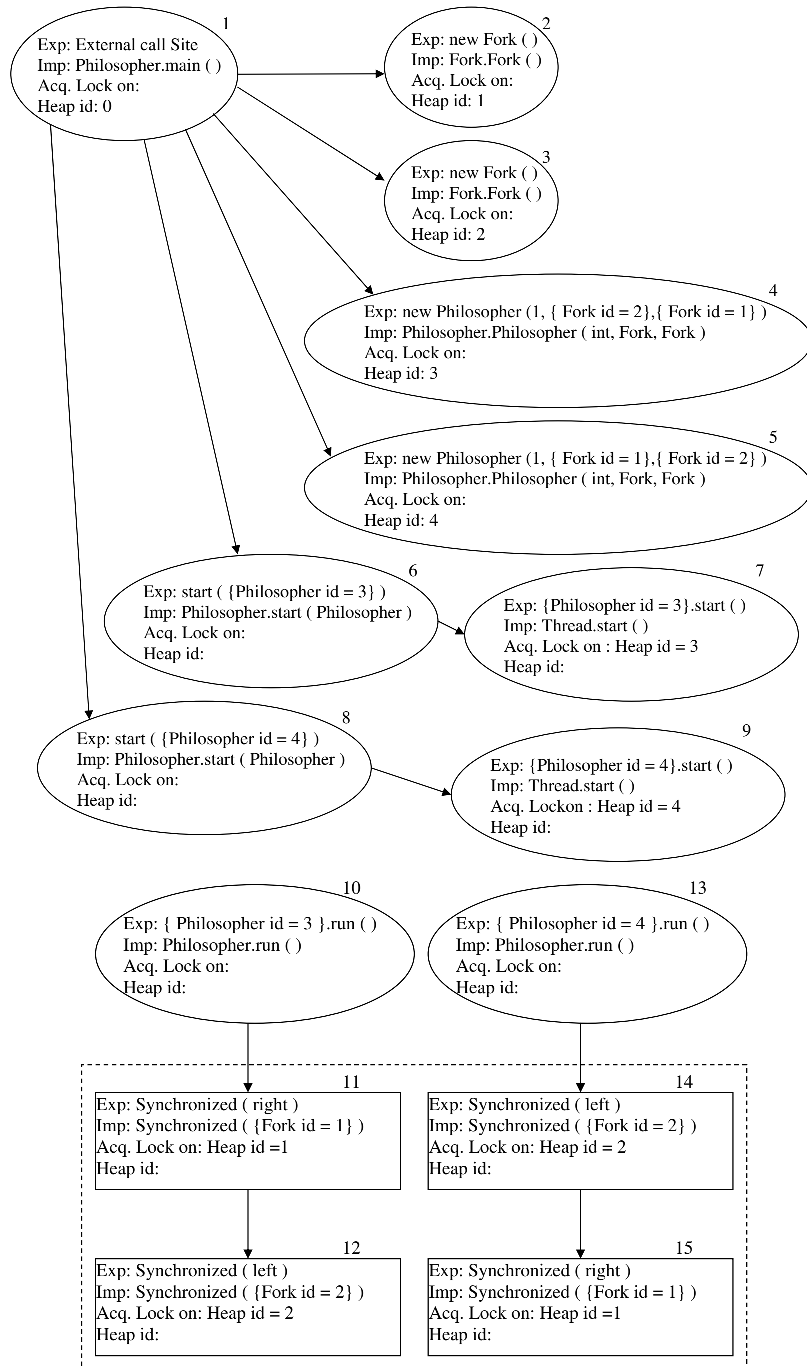


Figure 4.1: **Dining Philosophers CSOOCG** This is the CSOOCG for the Dining Philosophers program example in Figure 3.1.

Table 4.1: Trace Generator Equations

$$C_{Entry}(n) = \{C_{Exit}(n') | (n', n) \in flow(CSOOCG)\}$$

$$C_{Exit}(n) = \begin{cases} \{t \frown < n >| \; t \in C_{Entry}(n) \wedge \neg \; addsRedundantCycle(t, n)\} \\ \cup \\ \{< n >| \; C_{Entry}(n) = \varnothing \wedge isRootNode(n)\} \end{cases}$$

Note: At the start of the first iteration of this specification all the $C_{Entry}(n)$ and $C_{Exit}(n)$ sets are empty.

We need to generate all the possible traces through the CSOOCG, where a trace is a possible runtime execution represented by a sequence of nodes that traverse the CSOOCG via a path. This is accomplished by running an analysis, similar to a *forward-may* flow-sensitive analysis, on the CSOOCG until it reaches a fixed point. This analysis uses the trace generation (transfer) equations define in Table 4.1 (with supporting defintions and functions defined in Table 4.2. The C_{Entry} set of a node is defined to be the set of all traces entering this node. The C_{Exit} set of a node is defined to be the set of all traces leaving this node. The C_{Entry} and C_{Exit} are initially empty, i.e., $C_{Entry} = C_{Exit} = \varnothing$. There are two cases in which a node is added to a trace:

Case 1: The C_{Entry} set is empty and the node is a root node of a graph component.

Case 2: Node, n, being appended to a trace, t, does not cause a redundant cycle, where a redundant cycle is any cycle that does not add new information to the trace. For example, consider the trace, $t =< 1, 2, 3, 4, 2, 3 >$. Appending node 4 to trace t will result in a redundant cycle. This means that no new nodes have been visited from the last occurrence of node 4, i.e., $t(5..6) \subset t(1..3)$ is true. The $addsRedundantCycle(t, n)$ function is defined in Table 4.2. This function determines whether adding a given node to a given trace will result in a redundant cycle.

The results of the first application of this equation to our Dining Philosophers example is shown in Table 4.3. Only node 1 is added to a trace and placed in the exit

Table 4.2: Definitions and Functions

1. $CSOOCG$ is the Context Sensitive Object Oriented Call Graph
2. $nodes(CSOOCG)$ is the set of Java call site invocations in the CSOOCG
3. $n \in nodes(CSOOCG)$
4. $flow(CSOOCG)$ is the set of all directed edges in the CSOOCG of the form $\mathcal{P}(nodes(CSOOCG) \times nodes(CSOOCG))$
5. $isRootNode(n)$ returns true if n is a root node for a $CSOOCG$ graph component, false otherwise.
6. $hasLock(n)$ returns $true$ if n acquires a lock, $false$ otherwise.
7. $getLock(n)$ returns the lock object refernce acquired by n. $getLock(n)$ is undefined if $hasLock(n)$ returns $false$.
8. $trace$ is an ordered list of k nodes, i.e., $< n_1, n_2, ...n_k >$, representing a trace through the CSOOCG.
9. l is an ordered list of lock object references, i.e., $< lock_1, lock_2, ... >$, representing the locks acquired along a $trace$.
10. $getLocklist(trace)$ returns l, a list of locks, acquired along a given trace. **note:** this list maybe empty.
11. C is a set of traces.
12. $C_{Entry}(n)$ is a set of traces entering a given node.
13. $C_{Exit}(n)$ is a set of traces exiting a given node.
14. # is an operator used to get the cardinality of a list, e.g., $\# < 2, 4, 6, 8 >= 4$.
15. $\frown$ is an operator used to concatenate trace elements and sequences together, e.g., $< 1, 3, 4, 2 > \frown < 5, 8, 6 > \frown < 7 >=< 1, 3, 4, 2, 5, 8, 6, 7 >$.
16. S is a set of integers representing the indexes of a list, i.e., $S = \{t \in C \wedge i \in \mathbb{N} | 1 \leq i \leq \#t\}$.
17. $t\ (i)$ is the ith element of sequence t.
18. $t\ (i..j)$ is the sequence of inclusive elements of sequence t from $t\ (i)$ to $t\ (j)$, i.e., $< t(i), t(i+1), t(i+2), ..., t(j-1), t(j) >$.
19. $occurs(t, n)$ returns true if $\exists_{i \in S}(t(i) = n)$, false otherwise.
20. $lastIndexOf(t, n)$ returns the index value of the last occurrence of n in sequence t or -1 if n is not in sequence t.
21. $subSequence(t, start, end)$ returns a sequence $t\ (start..end)$, if $start, end \in S \wedge 1 \leq start \leq end \leq \#t$ then $t(start..end)$.
22. $addsRedundantCycle(t, n)$ returns $true$ if $occurs(t, n) \wedge \exists_{x \in S}(x = lastIndexOf(t, n) \wedge t(x + 1..\#t) \subset t(1..x - 1))$ is $true$, $false$ otherwise.

Table 4.3: Dining Philosophers Results: First Iteration

Node	C_{Entry}	C_{Exit}
1	$\varnothing$	$\{<1>\}$
2	$\varnothing$	$\varnothing$
3	$\varnothing$	$\varnothing$
4	$\varnothing$	$\varnothing$
5	$\varnothing$	$\varnothing$
6	$\varnothing$	$\varnothing$
7	$\varnothing$	$\varnothing$
8	$\varnothing$	$\varnothing$
9	$\varnothing$	$\varnothing$
10	$\varnothing$	$\varnothing$
11	$\varnothing$	$\varnothing$
12	$\varnothing$	$\varnothing$
13	$\varnothing$	$\varnothing$
14	$\varnothing$	$\varnothing$
15	$\varnothing$	$\varnothing$

set of node 1. This is the main root node of a component being added in accordance with case 1 above. An additional application of this equation causes the results seen in Table 4.4. This second iteration increases the lengths to two by insuring that appending a given node to a trace does not cause a redundant cycle in the resulting trace. This continues until Table 4.6. This table captures the run Philosophers threads being starting by nodes 7 and 9. The only other interesting point, is when the iterating process reaches a fixed point, shown in Table 4.8. For this analysis, the fixed point represents the point where the trace generation process has created all possible runtime execution paths. The trace generation algorithm is completed when it reaches a fixed point.

Once we complete the trace generation process, we collect all the C_{Exit} sets into a single set. This set, L, which represents all possible runtime executions of the program is defined formally as

$$L = \bigcup_{n \in nodes(CSOOCG)} Cexit(n)$$

Table 4.4: Dining Philosophers Results: Second Iteration

Node	C_{Entry}	C_{Exit}
1	$\varnothing$	$\{<1>\}$
2	$\{<1>\}$	$\{<1,\ 2>\}$
3	$\{<1>\}$	$\{<1,\ 3>\}$
4	$\{<1>\}$	$\{<1,\ 4>\}$
5	$\{<1>\}$	$\{<1,\ 5>\}$
6	$\{<1>\}$	$\{<1,\ 6>\}$
7	$\varnothing$	$\varnothing$
8	$\{<1>\}$	$\{<1,\ 8>\}$
9	$\varnothing$	$\varnothing$
10	$\varnothing$	$\varnothing$
11	$\varnothing$	$\varnothing$
12	$\varnothing$	$\varnothing$
13	$\varnothing$	$\varnothing$
14	$\varnothing$	$\varnothing$
15	$\varnothing$	$\varnothing$

Table 4.5: Dining Philosophers Results: Third Iteration

Node	C_{Entry}	C_{Exit}
1	$\varnothing$	$\{<1>\}$
2	$\{<1>\}$	$\{<1,\ 2>\}$
3	$\{<1>\}$	$\{<1,\ 3>\}$
4	$\{<1>\}$	$\{<1,\ 4>\}$
5	$\{<1>\}$	$\{<1,\ 5>\}$
6	$\{<1>\}$	$\{<1,\ 6>\}$
7	$\{<1,\ 6>\}$	$\{<1,\ 6,\ 7>\}$
8	$\{<1>\}$	$\{<1,\ 8>\}$
9	$\{<1,\ 8>\}$	$\{<1,\ 8,\ 9>\}$
10	$\varnothing$	$\varnothing$
11	$\varnothing$	$\varnothing$
12	$\varnothing$	$\varnothing$
13	$\varnothing$	$\varnothing$
14	$\varnothing$	$\varnothing$
15	$\varnothing$	$\varnothing$

Table 4.6: Dining Philosophers Results: Forth Iteration

Node	C_{Entry}	C_{Exit}
1	$\varnothing$	$\{<1>\}$
2	$\{<1>\}$	$\{<1,\ 2>\}$
3	$\{<1>\}$	$\{<1,\ 3>\}$
4	$\{<1>\}$	$\{<1,\ 4>\}$
5	$\{<1>\}$	$\{<1,\ 5>\}$
6	$\{<1>\}$	$\{<1,\ 6>\}$
7	$\{<1,\ 6>\}$	$\{<1,\ 6,\ 7>\}$
8	$\{<1>\}$	$\{<1,\ 8>\}$
9	$\{<1,\ 8>\}$	$\{<1,\ 8,\ 9>\}$
10	$\varnothing$	$\{<10>\}$
11	$\varnothing$	$\varnothing$
12	$\varnothing$	$\varnothing$
13	$\varnothing$	$\{<13>\}$
14	$\varnothing$	$\varnothing$
15	$\varnothing$	$\varnothing$

Table 4.7: Dining Philosophers Results: Fifth Iteration

Node	C_{Entry}	C_{Exit}
1	$\varnothing$	$\{<1>\}$
2	$\{<1>\}$	$\{<1,\ 2>\}$
3	$\{<1>\}$	$\{<1,\ 3>\}$
4	$\{<1>\}$	$\{<1,\ 4>\}$
5	$\{<1>\}$	$\{<1,\ 5>\}$
6	$\{<1>\}$	$\{<1,\ 6>\}$
7	$\{<1,\ 6>\}$	$\{<1,\ 6,\ 7>\}$
8	$\{<1>\}$	$\{<1,\ 8>\}$
9	$\{<1,\ 8>\}$	$\{<1,\ 8,\ 9>\}$
10	$\varnothing$	$\{<10>\}$
11	$\{<10>\}$	$\{<10,\ 11>\}$
12	$\varnothing$	$\varnothing$
13	$\varnothing$	$\{<13>\}$
14	$\{<13>\}$	$\{<13,\ 14>\}$
15	$\varnothing$	$\varnothing$

Table 4.8: Dining Philosophers Results: Sixth Iteration

Node	C_{Entry}	C_{Exit}
1	$\varnothing$	$\{<1>\}$
2	$\{<1>\}$	$\{<1,\ 2>\}$
3	$\{<1>\}$	$\{<1,\ 3>\}$
4	$\{<1>\}$	$\{<1,\ 4>\}$
5	$\{<1>\}$	$\{<1,\ 5>\}$
6	$\{<1>\}$	$\{<1,\ 6>\}$
7	$\{<1,\ 6>\}$	$\{<1,\ 6,\ 7>\}$
8	$\{<1>\}$	$\{<1,\ 8>\}$
9	$\{<1,\ 8>\}$	$\{<1,\ 8,\ 9>\}$
10	$\varnothing$	$\{<10>\}$
11	$\{<10>\}$	$\{<10,\ 11>\}$
12	$\{<10,\ 11>\}$	$\{<10,\ 11,\ 12>\}$
13	$\varnothing$	$\{<13>\}$
14	$\{<13>\}$	$\{<13,\ 14>\}$
15	$\{<13,\ 14>\}$	$\{<13,\ 14,\ 15>\}$

We remove from L all traces that do not acquire two or more locks. Therefore, in our example,

$$L = \{<10,\ 11,\ 12>, <13,\ 14,\ 15>\}$$

which is the set of traces that acquire two or more locks. We can now use these traces to determine the lock acquisition order.

The first trace, $\{<10,\ 11,\ 12>\}$, acquires locks on id=1 and id=2, where id=1 and id=2 represent possible runtime object references. The order these objects are acquired in is $<id = 1,\ id = 2>$. We add this to a set of ordered pairs and consider the next trace. The second trace, $\{<13,\ 14,\ 15>\}$, acquires locks on id=1 and id=2. The order these objects are acquired in is $<id = 2,\ id = 1>$. We add this second ordered pair to our list, such that our set of ordered pairs is: $\{<id = 1,\ id = 2>,\ <id = 2,\ id = 1>\}$.

Now we perform a transitive closure on this set of ordered pairs. No transitive ordered pairs are added. The final step is to determine is this set of ordered pairs

Table 4.9: Dining Philosophers Deadlock Conditions

CSOOCG Deadlock Conditions					
	Locks Acquired				
ID	First	Second	Method Call	File	Line #
0	heap ID: 2	heap ID: 3			
			Trace 1		
			synchronized (this.right)	Philosopher.java	21
			synchronized (this.left)	Philosopher.java	22
1	heap ID: 3	heap ID: 2			
			Trace 1		
			synchronized (this.right)	Philosopher.java	21
			synchronized (this.left)	Philosopher.java	22

is antisymmetric? The set is not antisymmetric because it contains a cycle, namely $< id = 1,\ id = 2 >$ and $< id = 2,\ id = 1 >$ are acquired in opposite ordered forming a cycle. This indicates that a possible runtime deadlock condition exits in the Java source code used to generate the CSOOCG.

4.2.1 Definitions and Functions Table Explained. The first four items in Table 4.2 define the CSOOCG. The CSOOCG is defined by a set of nodes, item 2, and a set of edges, item 4. These first four item provide a basis for traversing the CSOOCG.

The trace starting point is determined by the value returned from item 5, $isRootNode(n)$. If the value returned by $isRootNode(n)$ is true, then the node n is a starting point, or root, of a CSOOCG graph component. The acquired lock information is accessed by items 6 and 7 in Table 4.2. Item 6, $hasLock(n)$, is used to determine if the CSOOCG node, n, acquires a lock and item 7, $getLock(n)$, is used to access the lock object reference acquired by node, n.

Five sets are created and used by the trace generation algorithm. These data structures are traces, lock list, entry sets, exit sets and the index set. A *trace* is a

list of nodes. The location of these nodes in the list is consistent with their happens before ordering. The entry and exit sets are a set of traces. The last set is the index set, S. The index set is a set of integers representing the indexes of a trace.

four operators are defined for the sequences. The first operator, #, is used to extract the cardinally of a given list, e.g., $\# < 2, 4, 8 >= 3$. The second operator, $\frown$, is used to concatenate sequences together. The third operator, $t(i)$, is used to extract the ith element in sequence t. The fourth operator, $t(i..j)$, is used to extract a subsequence from sequence t.

The next four functions are used to access sequence elements. The first function, $occurs(t, n)$, returns a boolean value. This boolean value is true if the given node, n, is an element in sequence t, or false if node, n, is not an element of sequence t. The second function, $LastIndexOf(t, n)$, returns the index of the last occurrence of n in sequence t. If n does not occur in sequence t, then the value returned by $LastIndexOf(t, n)$ is -1. The third function, $subSequence(t, start, end)$, is used to extract a subsequence from sequence t. The fourth function, $addsRedundantCycle(t, n)$, is used to detect recursion in the trace generation algorithm. This last function returns true if node, n, occurs in sequence, t, and all the elements from the last occurrence of node, n, to the end of sequence t have previous occurred in sequence t, false otherwise.

4.3 Analyzer Implementation

The Analyzer is made up of 13 classes and 1 interface as shown in Figure 4.2. The primary class in Figure 4.2 is the Analyzer. The Analyzer uses two pieces of information to complete an analysis: the name of the data file to analyze and the strategy to use for the analysis. The TrieAnalysis is the default strategy used by the Analyzer.

4.3.1 Trace Generator. The first step in our analyzer is the generation of all possible traces. This is accomplished by a DFS traversal of the CSOOCG with a special redundant cycle detection algorithm used to terminate recursive loops.

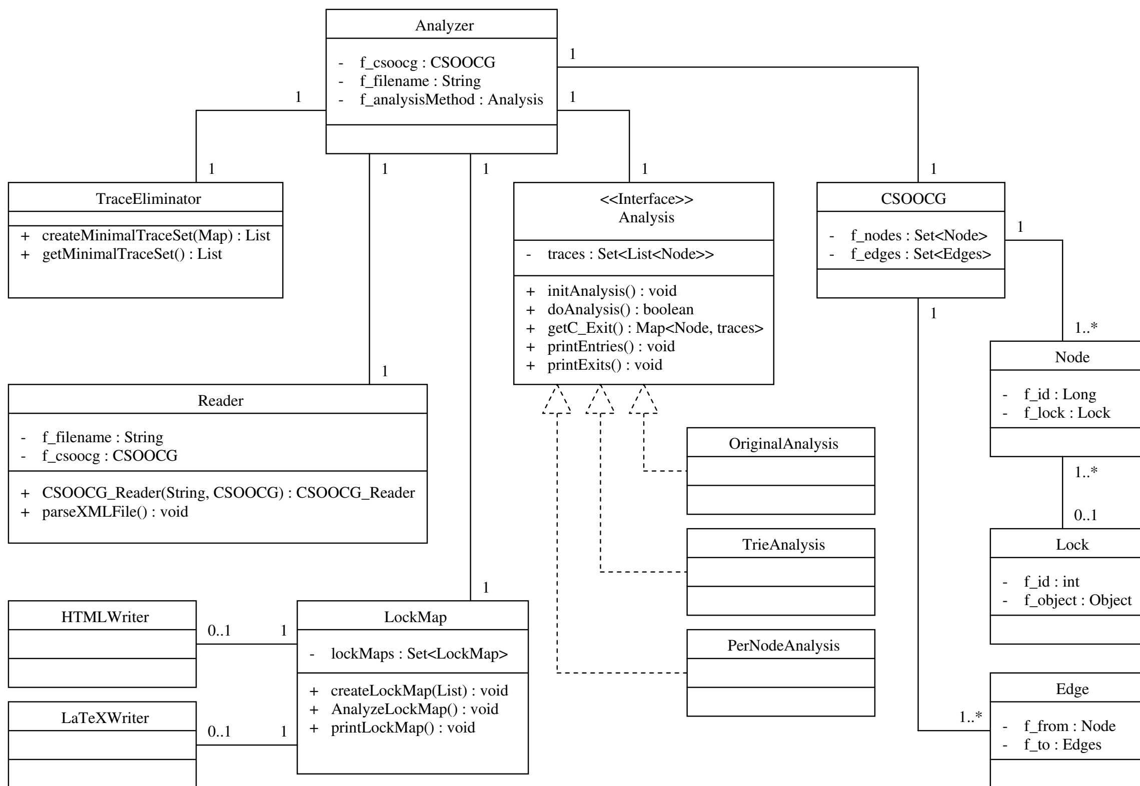


Figure 4.2: **Analyzer Class Diagram** This is a UML class diagram representing the classes used to analyze the *Data File*. This diagram also shows the links between these classes.

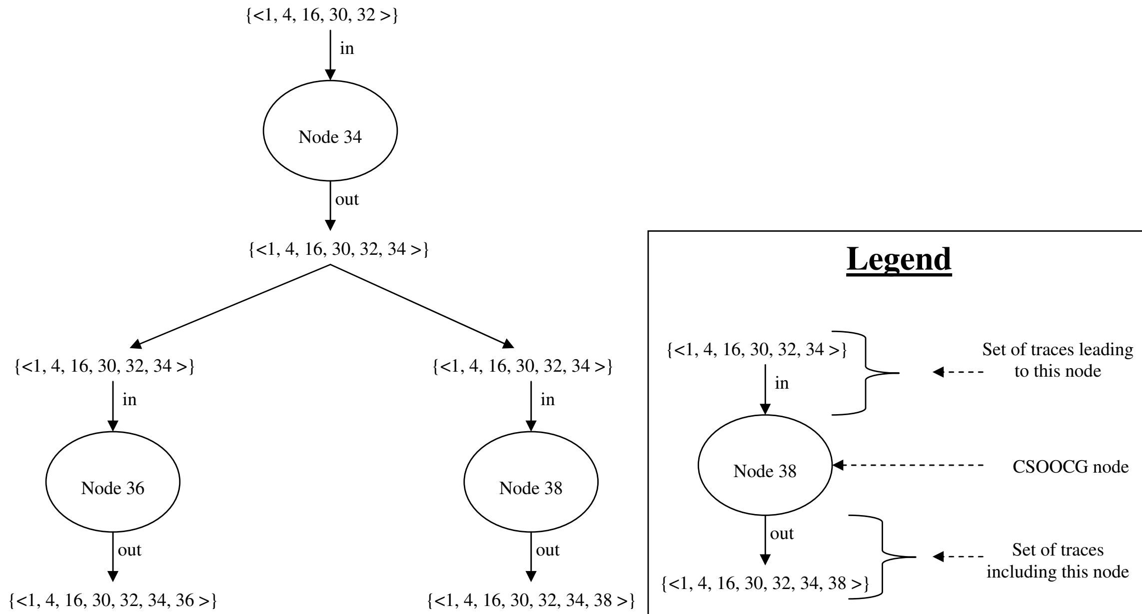


Figure 4.3: **Non-Trie Data Structure** This figure illustrates how execution traces are tracked and stored when a trie data structure is not used.

The results of this *Trace Generator* are sets of execution traces for each node in the CSOOCG. These sets of traces may contain redundant information that we can eliminate in the following process, the *Trace Eliminator*.

An Example on how the trie data structure helped reduce the amount of memory our CSOOCG analyzer requires. Figure 4.3 illustrates how we originally stored CSOOCG execution trace information. This figure shows three nodes that are a part of a larger CSOOCG. The arrow entering a node contain the set of execution call traces that lead to this node. The arrow leaving a node contains the set of execution call traces that include the node. This data structure uses 43 memory units to store these six traces. The trie data structure, shown in Figure 4.4 right side, maintains the trace information for a CSOOCG starting from the CSOOCG graph component's root node. Now the arrows entering and leaving a CSOOCG node needs only one trie pointer value to represent a trace, Figure 4.4 left side. Using a trie data structure lowers the number of memory units we needed from 43 down to 22. We save 50 percent of memory.

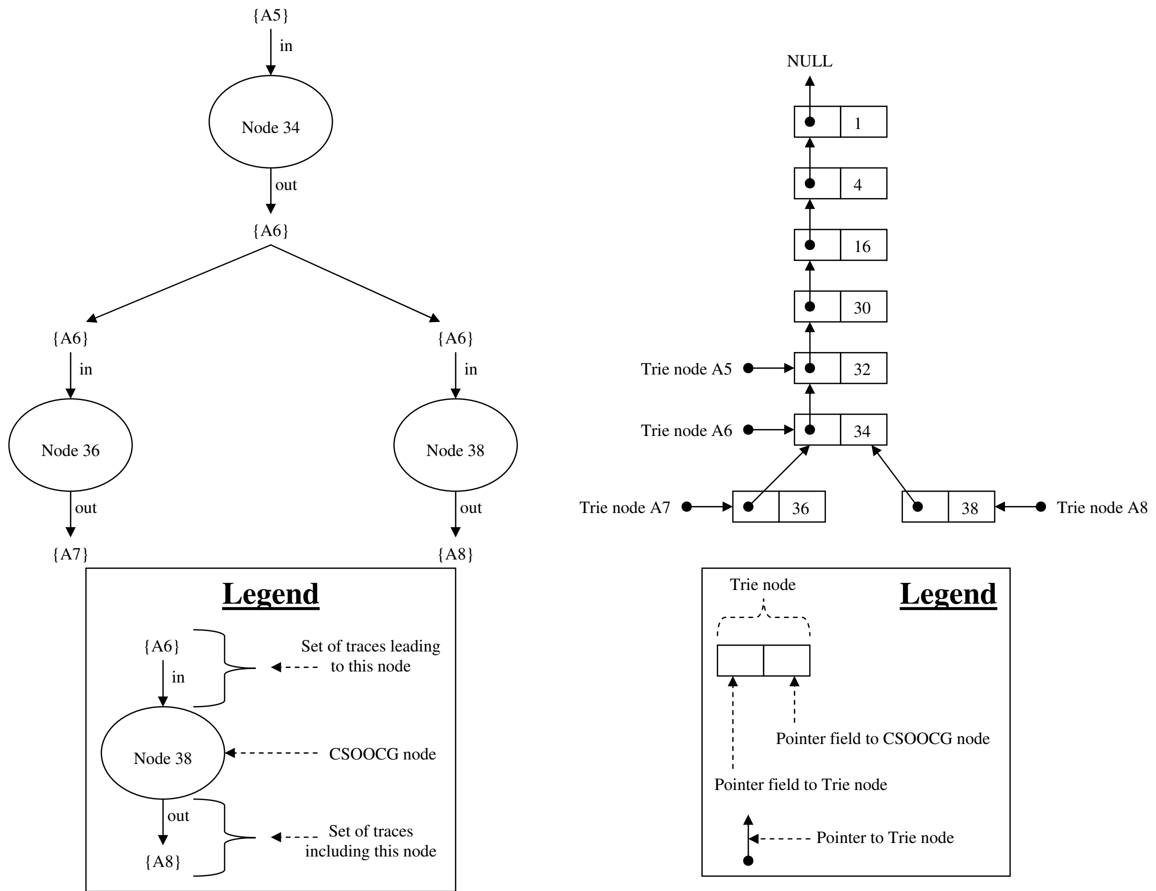


Figure 4.4: **Trie Data Structure** This figure illustrates how execution traces are tracked and stored when a trie data structure is used.

4.3.2 Trace Eliminator. The *Trace Eliminator* removes all the traces that acquire less than two locks. A trace must acquire at least two lock to be involved in a deadlock condition. The remaining traces are farther reduced by removing all the traces that are fully contained in larger traces. This allows us to create a minimal set of traces that represent the CUT execution paths of interest. This minimal traces set is analyzed to determine all possible lock acquisition orders, i.e., our *LockMap Builder* perform a *transitive closure* on the locks acquired by each trace.

4.3.3 LockMap Builder. Our *LockMap Builder* process performs the transitive closure on the minimal set of traces and uses the results to create our lock map. The lock map is a mapping from a pair of acquired locks–the map key–to a set of subtraces that acquire these locks–the associated map value. The subtraces consist of the call trace from the first acquired lock to the second acquired lock. One of the invariants of the lock map is that The subtraces contains two or more calls, i.e., the length of the subtraces is greater than or equal to 2.

4.3.4 Deadlock Determinator. The *Deadlock Determinator* process searches the lock map's keys to determine if all locks are acquired in a consistent order. If two acquired lock pairs are acquired in opposing order, then a possible deadlock condition exist. These two acquired lock pairs are identified and passed on to the *Presenter* process.

4.3.5 Presenter. The *Presenter* process receives from the *Deadlock Determinator* all possible deadlock conditions and formats the information for the user. Three different format options are available; plain text on the console screen, HTML formatted statistics and LaTeX2e formatted statistics. Appendices B and C contains Presenter output information for our two Java test cases: Dining Philosophers and Double Lock Equals. The call traces identify the calls from the CUT not the CSOOCG nodes–this way the user can locate and correct the problem in the CUT.

Java Project Code Under Test: Dining_Philosophers	
Method Declarations:	3
Constructor Declarations:	2
Method Calls:	6
Constructor Calls:	4
Synchronized Satements:	2
Synchronized Method Declarations:	0
CSOOCG Nodes:	15
CSOOCG Edges:	12
CSOOCG Locks:	4
CSOOCG Traces (before eliminator):	15
CSOOCG Traces (after eliminator):	2
Deadlock Conditions found:	1

Figure 4.5: **Sample HTML Output File Table**

Table B.1: Dining Philosophers Statistics

Java Project Code Under Test: **Dining Philosophers**	
Method Declarations:	3
Constructor Declarations:	2
Method Calls:	6
Constructor Calls:	4
Synchronized Satements:	2
Synchronized Method Declarations:	0
CSOOCG Nodes:	15
CSOOCG Edges:	12
CSOOCG Locks:	4
CSOOCG Traces (before eliminator):	15
CSOOCG Traces (after eliminator):	2
Deadlock Conditions found:	1

Figure 4.6: **Sample Latex Output File Table**

4.3.6 Implementation Challenges. We encounter three challenging problem during our CSOOCG analyzer implementation. These problems were detecting redundant cycles, inefficient memory storage of traces and providing meaning full results. Detecting redundant cycles is required to achieve a fixed point in our trace generation process. At first our inability to detect a redundant cycle was hard to identify as a problem–our analyzer would enter an endless loop and appear to hang. Creating a prototype tool to evaluate our redundant cycle detecting function enable us to solve this problem. Inefficient memory storage of traces cause problems concerning the size of programs we could analyze. We decided to use a trie data structure to improve the efficiency of how we stored traces in memory. This require a re-implementation of our redundant cycle detecting algorithm which made use of the trie's properties, namely the property that each trie node has one and only one parent.

In order to provide meaningful results to the user concerning deadlock conditions found during our analysis, we must maintain a large amount of information relating each CSOOCG node back to its original CUT file call site. Such information includes the CUT file's name, CUT file's package and the call site's line number in the CUT file.

This information is currently maintained in memory but not needed by the CSOOCG analyzer until the results are reported to the user.

4.4 A Second Example: Double Lock Equals

For Double Lock Equals, our *CSOOCG Analyzer's Trace Generator* starts by setting all the C_{Entry} and the C_{Exit} sets to the emptyset. Then the *Analyzer* constructs the C_{Exit} sets for all nineteen nodes. This requires the considerations of all entries sets into the nodes and the $isRootNode(n)$ return value of these nodes. Lucky each node only has one arrow entering it, hence only one C_{Entry} set must be considered per C_{Exit} set for a given node.

The first iteration through the *Trace Generator* causes the system to initialize itself. The C_{Entry} values are all empty during the *Trace Generator's* first pass. This means that the C_{Exit} sets will only contain one element in the trace sequence, i.e., the initial node for whom the C_{Exit} set is being determined. Hence, the results of the first pass of our *Trace Generator* is shown in Table 4.10.

The C_{Entry} and C_{Exit} set for each node is determined by the *Trace Generator* equations in Table 4.1. In this example, we notice that nodes 1, 6, and 13 does not have any arrows coming into them. We consider these nodes to be the root nodes for graph components. This means that the C_{Entry} sets and the C_{Exit} sets for these nodes will not change from one pass to another of the *CSOOCG Analysis*. This is to say that the C_{Entry} sets of these nodes will always be the empty set. The remaining nodes will change at least once because they have at lease one arrow coming into them.

The second iteration of the *Trace Generator* process leads to the results of Table 4.11. Here we notice that nodes 3 and 5 have been added to the exit trace sequence of node 1. This results in two new traces being created and cashed in the exit sets of nodes 3 and 5. Similarly, nodes 7 and 9 have been added to the exit trace of node 6. Nodes 14 and 17 have also been added to the exit trace of node 13. The

Table 4.10: Double Lock Equals Results: First Iteration

Node	C_{entry}	C_{exit}
1	$\varnothing$	$\{< 1 >\}$
2	$\varnothing$	$\varnothing$
3	$\varnothing$	$\varnothing$
4	$\varnothing$	$\varnothing$
5	$\varnothing$	$\varnothing$
6	$\varnothing$	$\{< 6 >\}$
7	$\varnothing$	$\varnothing$
8	$\varnothing$	$\varnothing$
9	$\varnothing$	$\varnothing$
10	$\varnothing$	$\varnothing$
11	$\varnothing$	$\varnothing$
12	$\varnothing$	$\varnothing$
13	$\varnothing$	$\{< 11 >\}$
14	$\varnothing$	$\varnothing$
15	$\varnothing$	$\varnothing$
16	$\varnothing$	$\varnothing$
17	$\varnothing$	$\varnothing$
18	$\varnothing$	$\varnothing$
19	$\varnothing$	$\varnothing$

Table 4.11: Double Lock Equals Results: Second Iteration

Node	C_{entry}	C_{exit}
1	$\varnothing$	$\{< 1 >\}$
2	$\{< 1 >\}$	$\{< 1,\ 2 >\}$
3	$\{< 1 >\}$	$\{< 1,\ 3 >\}$
4	$\{< 1 >\}$	$\{< 1,\ 4 >\}$
5	$\{< 1 >\}$	$\{< 1,\ 5 >\}$
6	$\varnothing$	$\{< 6 >\}$
7	$\{< 6 >\}$	$\{< 6,\ 7 >\}$
8	$\varnothing$	$\varnothing$
9	$\varnothing$	$\varnothing$
10	$\{< 6 >\}$	$\{< 6,\ 10 >\}$
11	$\varnothing$	$\varnothing$
12	$\varnothing$	$\varnothing$
13	$\varnothing$	$\{< 13 >\}$
14	$\{< 13 >\}$	$\{< 13,\ 14 >\}$
15	$\varnothing$	$\varnothing$
16	$\varnothing$	$\varnothing$
17	$\{< 13 >\}$	$\{< 13,\ 17 >\}$
18	$\varnothing$	$\varnothing$
19	$\varnothing$	$\varnothing$

Table 4.12: Double Lock Equals Results: Third Iteration

Node	C_{entry}	C_{exit}
1	$\varnothing$	$\{< 1 >\}$
2	$\{< 1 >\}$	$\{< 1,\ 2 >\}$
3	$\{< 1 >\}$	$\{< 1,\ 3 >\}$
4	$\{< 1 >\}$	$\{< 1,\ 4 >\}$
5	$\{< 1 >\}$	$\{< 1,\ 5 >\}$
6	$\varnothing$	$\{< 6 >\}$
7	$\{< 6 >\}$	$\{< 6,\ 7 >\}$
8	$\{< 6,\ 7 >\}$	$\{< 6,\ 7,\ 8 >\}$
9	$\varnothing$	$\varnothing$
10	$\{< 6 >\}$	$\{< 6,\ 10 >\}$
11	$\{< 6,\ 10 >\}$	$\{< 6,\ 10,\ 11 >\}$
12	$\varnothing$	$\varnothing$
13	$\varnothing$	$\{< 13 >\}$
14	$\{< 13 >\}$	$\{< 13,\ 14 >\}$
15	$\{< 13,\ 14 >\}$	$\{< 13,\ 14,\ 15 >\}$
16	$\varnothing$	$\varnothing$
17	$\{< 13 >\}$	$\{< 13,\ 17 >\}$
18	$\{< 13,\ 17 >\}$	$\{< 13,\ 17,\ 18 >\}$
19	$\varnothing$	$\varnothing$

Table 4.13: Double Lock Equals Results: Fourth Iteration

Node	C_{entry}	C_{exit}
1	$\varnothing$	$\{< 1 >\}$
2	$\{< 1 >\}$	$\{< 1,\ 2 >\}$
3	$\{< 1 >\}$	$\{< 1,\ 3 >\}$
4	$\{< 1 >\}$	$\{< 1,\ 4 >\}$
5	$\{< 1 >\}$	$\{< 1,\ 5 >\}$
6	$\varnothing$	$\{< 6 >\}$
7	$\{< 6 >\}$	$\{< 6,\ 7 >\}$
8	$\{< 6,\ 7 >\}$	$\{< 6,\ 7,\ 8 >\}$
9	$\{< 6,\ 7,\ 8 >\}$	$\{< 6,\ 7,\ 8,\ 9 >\}$
10	$\{< 6 >\}$	$\{< 6,\ 10 >\}$
11	$\{< 6,\ 10 >\}$	$\{< 6,\ 10,\ 11 >\}$
12	$\{< 6,\ 10,\ 11 >\}$	$\{< 6,\ 10,\ 11,\ 12 >\}$
13	$\varnothing$	$\{< 13 >\}$
14	$\{< 13 >\}$	$\{< 13,\ 14 >\}$
15	$\{< 13,\ 14 >\}$	$\{< 13,\ 14,\ 15 >\}$
16	$\{< 13,\ 14,\ 15 >\}$	$\{< 13,\ 14,\ 15,\ 16 >\}$
17	$\{< 13 >\}$	$\{< 13,\ 17 >\}$
18	$\{< 13,\ 17 >\}$	$\{< 13,\ 17,\ 18 >\}$
19	$\{< 13,\ 17,\ 18 >\}$	$\{< 13,\ 17,\ 18,\ 19 >\}$

traces continue to grow by one element per iteration. This is what is expected from the specifications listed in Table 4.1.

The results of the third and fourth iterations of the *Trace Generator* process are listed in Tables 4.12 and 4.13. The completion of the fourth iterations brings the trace generator process to its final state. This final state of this process is called the fixed point. This means that the results of the *Trace Generator* will not change due to additional iterations. This fixed point signifies the completion of the *Trace Generator* process and we proceed to the *Trace Eliminator* process.

The *Trace Eliminator* examines the traces in the C_{Exit} sets and selects the traces that acquire two or more locks. The results of this process are:

1. $\{< 6,\ 7,\ 8,\ 9 >\}$
2. $\{< 6,\ 10,\ 11,\ 12 >\}$
3. $\{< 13,\ 14,\ 15,\ 16 >\}$
4. $\{< 13,\ 17,\ 18,\ 19 >\}$

These four traces are passed to the Lockmap builder process. The resulting lock map is:

1. $< DoubleLockEquals\ id = 2,\ DoubleLockEquals\ id = 1 > \rightarrow \{< 8,\ 9 >,\ \ < 15,\ 16 >\}$
2. $< DoubleLockEquals\ id = 1,\ DoubleLockEquals\ id = 2 > \rightarrow \{< 11,\ 12 >,\ \ < 18,\ 19 >\}$

These two lock map entries are then processed by the *Deadlock Determinator*. The *Deadlock Determinator* process is capable of detecting the reversed lock acquisition order in the two acquired locks pairs. This deadlock condition is passed on to the *Presenter* process. The *Presenter* process consolidates all the *CSOOCG Analyzer's* results for the user. The *Presenter* process' results for our Double Lock Equals examples is located in appendix C and provided in Table 4.14 and Table 4.15.

The *Presenter* formats the *CSOOCG Analyzer's* results into HTML and LaTeX2e files. Both files are populated with information similar to the tables located in appendices B and C.

Table 4.14: **Double Lock Equals Statistics** This table contains statistical information concerning the number and types of declarations, calls, CSOOCG elements, CSOOCG traces and deadlock conditions found in our Double Lock Equals example. These numbers help quantify our work.

Java Project Code Under Test: **Double Lock Equals**	
Method Declarations:	3
Constructor Declarations:	2
Method Calls:	6
Constructor Calls:	4
Synchronized Statements:	2
Synchronized Method Declarations:	0
CSOOCG Nodes:	19
CSOOCG Edges:	16
CSOOCG Locks:	4
CSOOCG Traces (before eliminator):	19
CSOOCG Traces (after eliminator):	4
Deadlock Conditions found:	1

Table 4.15: **Double Lock Equals Deadlock Conditions** This table list the deadlock conditions found in our Double Lock Equals example.

CSOOCG Deadlock Conditions					
	Locks Acquired				
ID	First	Second	Method Call	File	Line #
0	heap ID: 2	heap ID: 1			
			Trace 1		
			synchronized (this)	DoubleLockEquals.java	27
			synchronized (obj)	DoubleLockEquals.java	28
			Trace 2		
			synchronized (this)	DoubleLockEquals.java	27
			synchronized (obj)	DoubleLockEquals.java	28
1	heap ID: 1	heap ID: 2			
			Trace 1		
			synchronized (this)	DoubleLockEquals.java	27
			synchronized (obj)	DoubleLockEquals.java	28
			Trace 2		
			synchronized (this)	DoubleLockEquals.java	27
			synchronized (obj)	DoubleLockEquals.java	28

V. Conclusion

The increased use of concurrency in mission critical software applications increases the potential for concurrency-related security and reliability problems within these applications. Traditional testing is ineffective because concurrent code is essentially nondeterministic. Traditional inspection techniques are ineffective because these problems are typically non-local in the code, i.e., examining a small portions of the program's code will not find them.

This thesis has described a flow-insensitive interprocedural static analysis for a subset of Java programs that detects if a program may deadlock at runtime. Our analysis, as we have described in Chapter III and Chapter IV, proceeds in two steps. The first extracts the "real" call graph decorated with acquired locks from the target program, which we call the CSOOCG. The second analyzes our CSOOCG to report how a possible deadlock may occur at runtime.

The two principle limitations of our analysis are on the target program: (1) we need its "real" call graph and (2) its overall size is limited. Our prototype tool can only construct the target program's "real" call graph in the presence of perfect aliasing information about its object references. This aliasing information is provided by an external oracle, which was described in Chapter III. Combinatorial explosion limits the size of programs our technique is able to analyze to a rough ceiling of 35 kSLOC. To enable us to reach this ceiling, our prototype tool uses a combination of call graph reduction techniques, aided by the insight that our analysis is only concerned with program paths that can acquire one or more locks, and efficient data structure choices to help mitigate this limitation.

5.1 Summary of Contributions

Our primary contribution is the development of a static analysis technique that can detect the potential for deadlock in object-oriented programs. Prior work has focused on C-like languages [7] that lack runtime dispatch of function calls. Our approach takes into account the runtime dispatch of Java methods based upon the

dynamic type of the receiver object. Given correct aliasing information about a program, our analysis is not susceptible to false negative results, i.e., it will not miss potential deadlock cases. It can, however, result in false positive reports, i.e., reports of deadlock which cannot occur at runtime. This is primarily due to our lack of understanding of what the program does, i.e., our lack of design intent about the program, and that we make no attempt to model constraints within each procedure, e.g., to determine if only one of two synchronized blocks will ever be executed.

A secondary contribution is the the formal definition of a Context-Sensitive Object-Oriented Call Graph, which is a model of the call structure of a program that appears to be potentially useful for other analyses.

An engineering contribution of our work is the modular design of our two-step analysis. Our design is composed of a series of distinct steps with clear interfaces. In particular, our use of the CSOOCG structure as the primary interface between the CSOOCG generator and the CSOOCG analyzer allows improvements to be made to one component without impact upon the other. In addition, our definition of a clear interface to the aliasing oracle allows our design to try and perhaps "swap-out" several alias analyses.

5.2 Recommendations for Future Work

In this section we speculate on several improvements which could be made to our result via future research.

5.2.1 Integrate an alias analysis. Our approach defines an interface to, but does not implement, an alias analysis. Due to two reasons an analysis analysis was not attempted in our work. First, understanding the body of existing work on alias analyses (even for just object-oriented languages) is a large task. Second, without a clear definition of what the requirements for an alias analysis, selecting an analysis is not possible. Thus we decided to develop a precise specification of what

was needed (which we presented in Chapter III) rather than co-develop both our deadlock detection analysis and an alias analysis.

As we have indicated, the lack of an automatic alias analysis is one of the major limitations of our approach which we would like to address in future work. Our approach requires an analysis analysis which is conservative in the model of the runtime heap it presents to our CSOOCG generator. This requirement exists because we do not want to have the alias analysis introduce the potential for false negative results.

5.2.2 Support analysis cut-points and composability of results. Our current approach requires the whole program to produce useful results. This is a highly undesirable requirement in practice. Today's software systems are built from components, therefore our analysis should be able to analyze components separately and compose its results when components are brought together into a whole.

One observation is that a component or library which does not "callback" its client can be assured separately and not considered by our analysis. The discriminator here is that no call into the library ever invokes a method within the client. Assuring this property is non-trivial. For example, the `java.util` collection library appears to have this property, however in specific cases calls to `hashCode` and `compareTo` are made on objects placed within the collections (an example of the first is the `HashSet` class and an example of the second is the `TreeMap` class). These constitute a "callback" to the client code. Clearly, more research on defining strong analysis cut-points is needed. We believe this will require some degree of programmer expressed design intent about the program, i.e., it will not be fully automatic.

5.2.3 Find and focus on the nexuses of locking. In prior work by Engler et al. [7, page 240], Rugina and Rinard [21, page 34], and Holzmann [17] has noted that only a small amount of the overall code in a typical program locks–on the order of a third or a fourth. Our current approach does not optimize CSOOCG creation or

analysis based upon this empirical finding. We believe future work should consider optimized our approach based upon this empirical finding which could help to reduce the size of the CSOOCG and improve the memory efficiency of our analysis.

5.2.4 Support util.concurrent-style locks. The Java programming language version 5 added locks which are not restricted to syntactic blocks. As we have indicated, Our approach does not support these locks. This is a possible area of future work as it is expected that more and more production Java software will being to use the `util.concurrent` library. Our analysis would have to become flow-sensitive to determine where lock acquisitions and releases may be made on particular `util.concurrent` lock objects. In addition, locks with special semantics (e.g., a reader-writer lock) will require special handling by our analysis.

5.2.5 Use a model checker. It is possible that the CSOOCG analyzer could largely be replaced with an model checker. This would allow our approach to take advantage of the significant engineering effort put into the model checker while at the same time possibly increasing the scalability of our approach. The disadvantage of using a model checker is the loss of context it can cause, e.g., our approach must be able to take results from the model checker (typically in the form of counterexample traces) and communicate them to the programmer in a understandable manner.

5.3 Concluding Thoughts

As the concurrency in mission critical software systems increases our ability to detect, in a principled manner correct, concurrency related faults within these systems must also increase. This research has focused on static deadlock detection for Java and, while not without limitations, shows some promise of becoming a practical tool for software quality assurance.

Appendix A. Use and SetUp

A.1 Required Items

The following items are required for our CSOOCG analysis:

1. The Java program source code to be analyzed,
2. a compatible version of the Eclipse IDE,
3. a compatible version of the Fluid plug-in for Eclipse,
4. a compatible version of the CSOOCG generator plug-in for Eclipse, and
5. a copy of the CSOOCG Analyzer.

Our CSOOCG analysis is a two part process. Part one is the creation of the CSOOCG data file and part two is the analysis of this data file. The first four items are used to create and store the CSOOCG data file. The last item, i.e., the CSOOCG analyzer, is used to analyze the data file and determine if deadlock conditions exist in the Java program's source code.

The first step is to get a copy of the Eclipse IDE. A copy of eclipse can be downloaded from http://www.eclipse.org. The next step is to get a copy of Fluid plug-in for Eclipse from CMU. The final step is to get a copy of the CSOOCG Generator plug-in for eclipse and a copy of the CSOOCG Analyzer from AFIT.

Once all the software has been obtained and loaded onto the computer system, the analysis process can begin. We open Eclipse and insure our two eclipse plug-ins, Fuild and CSOOCG Generator, are loaded and successfully compile. Then start a "meta-Eclipse" session within Eclipse. A meta-Eclipse session is an Eclipse application running within the Eclipse's IDE. This meta-Eclipse session is were the CUT files are loaded and manipulated. The results of this meta-Eclipse session is the CSOOCG data file. This data file is stored under the same name as the Java project containing the CUT files with the date appended to the end.

The CSOOCG Analyzer processes the data file and determines if deadlock conditions exist within the source CUT files. The results of the CSOOCG analysis are

Table A.1: CSOOCG Generator Switch Options

Program Switches	Feature	Default
DEADLOCK_DEBUG	Turn the CSOOCG generator on	off
ALIAS_READY	Turn the Alias Oracle methods on	off
DINING_BROKEN	Turn the Dining Philosophers deadlock alias information on	off
DLE_BROKEN	Turn the Double Lock Equals deadlock alias information on	off

stored in an HTML file with the same name as the data file. The information contained in this HTML file is similar to the information contained in the following three appendices.

The CSOOCG Generator and CSOOCG Analyzer both utilize command line switches to determine internal variable values. These command line switches are covered in the next two sections.

A.2 Using the Generator

A number of program command line switches have been added to the CSOOCG generator to enable certain features. These switches and the features they control are listed in Table A.1.

There are five useful switch configuration options the reader should know. The first switch configuration is:

- -Xms64m -Xmx1024m -Xss8192k -DDEADLOCK_DEBUG

This is the base configuration used to run the CSOOCG Generator. The first two options, Xms64m and Xmx1024m, are used to set the minimum and maximum amount of memory accessible by the meta-Eclipse application. The third option, Xss8192k, is used to set the amount of stack memory to allocate to the meta-Eclipse application. The fourth and last option in the base switch configuration, DDEADLOCK_DEBUG, is used to create a system variable called "DEADLOCK_DEBUG." The CSOOCG Generator searches the system variables for "DEADLOCK_DEBUG," if this variable

is found then the CSOOCG Generator process is run else the process is not run. This base configuration is used to locate all the traces that acquire two or more locks. This configuration does not create a CSOOCG–strictly speaking. It creates an object oriented call graph, i.e., the context sensitivity part of the CSOOCG is missing. The context sensitivity used to create the CSOOCG is provided by an external process, e.g., the alias oracle. This base switch configuration is useful in determine if any execution traces acquire two or more locks.

The second switch configuration is:

- -Xms64m -Xmx1024m -Xss8192k -DDEADLOCK_DEBUG -DALIAS_READY

This second switch configuration adds a fifth option, DALIAS_READY, to enable the external alias oracle. This external alias oracle provides the context sensitivity information used to create the CSOOCG. This configuration requires an external alias oracle process. To date, this external process is not fully operational within the Fluid plug-in for Eclipse. This leads us to the next three test switch configurations.

The next two switch configurations are used to test out the Dining Philosophers and Double Lock Equals Java programs. These two options are:

- -Xms64m -Xmx1024m -Xss8192k -DDEADLOCK_DEBUG -DALIAS_READY -DDINING_BROKEN
- -Xms64m -Xmx1024m -Xss8192k -DDEADLOCK_DEBUG -DALIAS_READY -DDLE_BROKEN

The two separate switch configurations are used to activate hard coded alias oracles. The sixth options, DDINING_BROKEN and DDLE_BROKEN, correspond to hard coded alias oracle for Dining Philosophers deadlock version and Double Lock Equals deadlock version, respectively.

Table A.2: CSOOCG Analyzer Switch Options

Program Switches	Feature	Default
USE_TRIE	Use Trie analyzer	Enabled
USE_Original	Use Original analyzer	Disabled
USE_SyncNodesOnly	Use the Per Node analyzer on the CSOOCG nodes that acquire locks	Disabled
USE_PerNode	Use the Per Node analyzer	Disabled
Latex_OFF	Turns off the LaTeXgeneration process.	Enabled
HTML_OFF	Turns off the HTML generation process.	Enabled

A.3 Using the Analyzer

A number of program command line switches have been added to the CSOOCG Analyzer to enable and disable certain features. These switches and the features they control are listed in Table A.1.

The six optional command line switches listed in Table A.1 may be used in any combination. The user should be aware of the hierarchal relationship between the first four switches. This relationship determines which analysis is chosen when two or more conflicting switches are entered. These first four switches are listed in Table A.1 in order of precedence. This means that the DUSE_TRIE option overrides the DUSE_Original, DSyncNodesOnly and DUSE_PerNode options, while the DUSE_Original option overrides the DSyncNodesOnly and DUSE_PerNode options. Of course, the DSyncNodesOnly option overrides the DUSE_PerNode option. The features controlled by these options are listed in Table A.1 along with their default setting. The default options within the CSOOCG Analyzer are sufficient to process most data files.

Appendix B. Dining Philosophers Results Deadlock Version

The tables in this appendix contain the results of the CSOOCG analysis for the Dining Philosophers example, Figure B.1. These results include statistical information, list of nodes, list of edges, list of locks, list of traces that acquire two or more locks and list of deadlock conditions.

B.1 Statistics

Table B.1: **Dining Philosophers Statistics** This table contains statistical information concerning the number and types of declarations, calls, CSOOCG elements, CSOOCG traces and deadlock conditions found in our Double Lock Equals example. These numbers help quantify our work.

Java Project Code Under Test: **Dining Philosophers**	
Method Declarations:	3
Constructor Declarations:	2
Method Calls:	6
Constructor Calls:	4
Synchronized Statements:	2
Synchronized Method Declarations:	0
CSOOCG Nodes:	15
CSOOCG Edges:	12
CSOOCG Locks:	4
CSOOCG Traces (before eliminator):	15
CSOOCG Traces (after eliminator):	2
Deadlock Conditions found:	1

```
public class Philosopher extends Thread {

  public static final class Fork {  }

  final Fork right;

  final Fork left;

  final int identity;

  Philosopher(int identity, Fork right, Fork left) {
    this.identity = identity;
    this.right = right;
    this.left = left;
  }

  @Override
  public void run() {
    while (true) {
      // Thinking
      synchronized (right) {
        synchronized (left) {
          // Eating
        }
      }
    }
  }

  public static void main(String[] args) {
    final Fork f1 = new Fork();
    final Fork f2 = new Fork();
    final Philosopher p1 = new Philosopher(1, f1, f2);
    final Philosopher p2 = new Philosopher(2, f2, f1);
    start(p1);
    start(p2);
  }

  private static void start(final Philosopher p) {
    p.setName(‘‘philosopher-’’ + p.identity);
    p.start();
  }
}

```

Figure B.1: **Dining Philosophers Source Code** This is the Java source code for the Dining Philosopher program.

Table B.2: **Dining Philosophers Nodes** This table list the CSOOCG nodes for our Dining Philosophers Example.

CSOOCG Nodes		
ID	Name	Lock
4	p.start()	heap ID: 3
11	new Fork ()	
6	Philosopher.start(p2)	
8	Philosopher.run()	
7	p.start()	heap ID: 4
3	Philosopher.start(p1)	
14	synchronized (this.right)	heap ID: 1
15	synchronized (this.left)	heap ID: 2
9	new Philosopher (2, f2, f1)	
1	Philosopher.main(String [])	
2	new Fork ()	
13	synchronized (this.left)	heap ID: 2
5	Philosopher.run()	
12	synchronized (this.right)	heap ID: 1
10	new Philosopher (1, f1, f2)	

Table B.3: **Dining Philosophers Edges** This table list the edges in our Dining Philosophers example.

CSOOCG Edges		
Count	Tail	Head
0	1	6
1	3	4
2	5	14
3	14	15
4	1	3
5	1	11
6	1	2
7	1	10
8	8	12
9	1	9
10	12	13
11	6	7

Table B.4: **Dining Philosophers Locks** This table list the lock object references that may be acquired when the Dining Philosophers example program is executed.

CSOOCG Locks		
ID	Object	Type
3	heap ID: 3	Philosopher
2	heap ID: 2	Fork
4	heap ID: 4	Philosopher
1	heap ID: 1	Fork

Table B.5: **Dining Philosophers Lock Map Entries** This table list the lock map created for our Dining Philosophers example.

CSOOCG Lock Maps					
	Locks Acquired				
ID	First	Second	Method Call	File	Line #
0	heap ID: 1	heap ID: 2			
			Trace 1		
			synchronized (this.right)	Philosopher.java	21
			synchronized (this.left)	Philosopher.java	22
1	heap ID: 2	heap ID: 1			
			Trace 1		
			synchronized (this.right)	Philosopher.java	21
			synchronized (this.left)	Philosopher.java	22

Table B.6: **Dining Philosophers Deadlock Conditions** This table list the deadlock conditions found in our Dining Philosophers example.

CSOOCG Deadlock Conditions					
	Locks Acquired				
ID	First	Second	Method Call	File	Line #
0	heap ID: 1	heap ID: 2			
			Trace 1		
			synchronized (this.right)	Philosopher.java	21
			synchronized (this.left)	Philosopher.java	22
1	heap ID: 2	heap ID: 1			
			Trace 1		
			synchronized (this.right)	Philosopher.java	21
			synchronized (this.left)	Philosopher.java	22

The tables in this appendix contain the results of the CSOOCG analysis for the Double Lock Equals example, Figure C.1. These results include statistical information, list of nodes, list of edges, list of locks, list of traces that acquire two or more locks and list of deadlock conditions.

C.1 Statistics

Table C.1: **Double Lock Equals Statistics** This table contains statistical information concerning the number and types of declarations, calls, CSOOCG elements, CSOOCG traces and deadlock conditions found in our Double Lock Equals example. These numbers help quantify our work.

Java Project Code Under Test: **Double Lock Equals**	
Method Declarations:	3
Constructor Declarations:	2
Method Calls:	6
Constructor Calls:	4
Synchronized Statements:	2
Synchronized Method Declarations:	0
CSOOCG Nodes:	19
CSOOCG Edges:	16
CSOOCG Locks:	4
CSOOCG Traces (before eliminator):	19
CSOOCG Traces (after eliminator):	4
Deadlock Conditions found:	1

```java
public final class DoubleLockEqualsMain extends Thread {

  static final DoubleLockEquals f1 = new DoubleLockEquals();

  static final DoubleLockEquals f2 = new DoubleLockEquals();

  public static void main(String[] args) {
    (new DoubleLockEqualsMain()).start();
    (new DoubleLockEqualsMain()).start();
  }

  @Override
  public void run() {
    while (true) {
      boolean result = f1.equals(f2) == f2.equals(f1);
    }
  }
}

public final class DoubleLockEquals {

  private long f_readCount = 0;

  @Override
  public boolean equals(Object obj) {
    if (obj instanceof DoubleLockEquals) {
      synchronized (this) {
        synchronized (obj) {
          this.f_readCount++;
          ((DoubleLockEquals) obj).f_readCount++;
          return super.equals(obj);
        }
      }
    }
    return false;
  }

  // OTHER IMPLEMENTATION CODE

}
```

Figure C.1: **Double Lock Equals Java Source Code** This is the Java source code for the Double Lock Equals program.

Table C.2: **Double Lock Equals Nodes** This table list the CSOOCG nodes for our Double Lock Equals Example.

CSOOCG Nodes		
ID	Name	Lock
3	DoubleLockEqualsMain.run()	
19	synchronized (obj)	heap ID: 1
12	synchronized (this)	heap ID: 2
6	new DoubleLockEqualsMain ()	
16	synchronized (obj)	heap ID: 2
5	DoubleLockEqualsMain.run()	
8	#.f1.equals(#.f2)	
7	new DoubleLockEqualsMain ()	
14	#.f1.equals(#.f2)	
15	synchronized (this)	heap ID: 1
18	synchronized (this)	heap ID: 2
10	synchronized (obj)	heap ID: 2
1	DoubleLockEqualsMain.main(String [])	
17	#.f2.equals(#.f1)	
2	(new # #).start()	heap ID: 6
13	synchronized (obj)	heap ID: 1
4	(new # #).start()	heap ID: 7
11	#.f2.equals(#.f1)	
9	synchronized (this)	heap ID: 1

Table C.3: **Double Lock Equals Edges** This table list the edges in our Double Lock Equals example.

CSOOCG Edges		
Count	Tail	Head
0	12	13
1	11	12
2	5	17
3	15	16
4	9	10
5	5	14
6	3	11
7	1	7
8	17	18
9	3	8
10	1	6
11	8	9
12	1	4
13	1	2
14	18	19
15	14	15

Table C.4: **Double Lock Equals Locks** This table list the lock object references that may be acquired when the Double lock Equals example program is executed.

CSOOCG Locks		
ID	Object	Type
1	heap ID: 7	DoubleLockEqualsMain
2	heap ID: 1	DoubleLockEquals
3	heap ID: 2	DoubleLockEquals
0	heap ID: 6	DoubleLockEqualsMain

Table C.5: **Double Lock Equals Lock Map Entries** This table list the lock map created for our Double Lock Equals example.

CSOOCG Lock Maps					
	Locks Acquired				
ID	First	Second	Method Call	File	Line #
0	heap ID: 2	heap ID: 1			
			Trace 1		
			synchronized (this)	DoubleLockEquals.java	27
			synchronized (obj)	DoubleLockEquals.java	28
			Trace 2		
			synchronized (this)	DoubleLockEquals.java	27
			synchronized (obj)	DoubleLockEquals.java	28
1	heap ID: 1	heap ID: 2			
			Trace 1		
			synchronized (this)	DoubleLockEquals.java	27
			synchronized (obj)	DoubleLockEquals.java	28
			Trace 2		
			synchronized (this)	DoubleLockEquals.java	27
			synchronized (obj)	DoubleLockEquals.java	28

Table C.6: **Double Lock Equals Deadlock Conditions** This table list the deadlock conditions found in our Double Lock Equals example.

CSOOCG Deadlock Conditions					
	Locks Acquired				
ID	First	Second	Method Call	File	Line #
0	heap ID: 2	heap ID: 1			
			Trace 1		
			synchronized (this)	DoubleLockEquals.java	27
			synchronized (obj)	DoubleLockEquals.java	28
			Trace 2		
			synchronized (this)	DoubleLockEquals.java	27
			synchronized (obj)	DoubleLockEquals.java	28
1	heap ID: 1	heap ID: 2			
			Trace 1		
			synchronized (this)	DoubleLockEquals.java	27
			synchronized (obj)	DoubleLockEquals.java	28
			Trace 2		
			synchronized (this)	DoubleLockEquals.java	27
			synchronized (obj)	DoubleLockEquals.java	28

This appendix consolidates and defines noteworthy terms used within this dissertation.

Acquired Locks is an entity within the Analyzer representing an order pair of locks acquired along a trace.

Analyzer is a collection of processes used to determine if a given Data File contains deadlock conditions.

AST is an abbreviation for Abstract Syntax Tree.

Builder is a process within the Generator used to build the CSOOCG from a given CUT Graph.

Call Completor is a process within the Generator used to determine all possible method call site and new expression (constructor call site) implementations for a given CUT Graph.

CSOOCG is an abbreviation for Context Sensitive Object Oriented Call Graph. This is an entity created in the Generator, stored in the Data File and used in the Analyzer. This entity represents the Java program being analyzed.

CUT is an abbreviation for program Code Under Test. This is the Java program code being analyzed for possible deadlock conditions.

CUT Graph is an entity within the CSOOCG generator. The CUT Graph is a map data structure containing AST elements. These elements include method declarations (MD), constructor declarations (CD), synchronized statements (SS), method calls (MC), and new expressions (NE).

The mapping is as follows:

1. $MD \rightarrow \{SS|MC|NE\}$ where:
2. $CD \rightarrow \{SS|MC|NE\}$ | represents the logical inclusive "or" operation
3. $SS \rightarrow \{SS|MC|NE\}$ and the { } symbols are used to represent a set
4. $MC \rightarrow \{MD\}$ and → represents the actual mapping from
5. $NE \rightarrow \{CD\}$ map key to map value set.

Items 1, 2 and 3 capture AST call sites within method declarations, constructor declarations and synchronized statements. Items 4 and 5 capture all possible call site implementations, e.g., method calls maybe implemented by one or more method declarations.

CUT Graph Builder is a process within the Generator used to build a CUT Graph for a given Java file.

Data File is an XML formatted ASCII file containing all the needed information to construct a CSOOCG. The information in this file defines CSOOCG nodes, edges and lock objects.

Deadlock Determinator is a process within the Analyzer use to determine if deadlock conditions exist in a given lock map.

fAST is a Fluid Java abstract syntax tree.

Generator is an Eclipse plug-in used to generate the Data File.

Lock Map is an entity within the Analyzer representing the mapping from acquired locks to the set of traces that acquire them.

Lock Map Builder is a process within the Analyzer used to build the lock map from a set of traces.

Output Files are two optional analyzer result files. The first is an HTML formatted ASCII file and the second is a LaTeX formatted ASCII file. Both files contain the same analyzer statistical information, e.g., the number of methods calls in the CUT, the number of deadlock conditions found . . . etc.

Presenter is a process within the Analyzer used to translate the Analyzer's results into the Output Files.

Reader is a process within the Analyzer use to read in the Data File from disk.

Reducer is an optional process within the Generator used to reduce the size of a given CUT Graph.

Trace is an entity within the Analyzer representing an execution path through the CSOOCG. A trace is store as a list of CSOOCG nodes.

Trace Eliminator is a process within the Analyzer used to minimize the size of a list of traces. This is done utilizing the following two rules:

- Eliminate traces that are fully contained in larger traces and share the same first CSOOCG node.
- Eliminate traces that do not acquire at least two locks.

Trace Generator is a process within the Analyzer used to generate a list of all possible traces.

Trie is an abbreviation for retrieval. This is a tree like data structure invented by Edward Fredkin [8, 24] and used within the CSOOCG analyzer process to maintain trace information in a minimal amount of memory.

Writer is a process within the Generator used to write out the Data File to disk.

Bibliography

1. Boyapati, Chandrasekhar, Robert Lee, and Martin Rinard. "Ownership types for safe programming: preventing data races and deadlocks". *OOPSLA '02: Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, 211–230. ACM Press, New York, NY, USA, 2002.

2. Callahan, David, Alan Carle, Mary Wolcott Hall, and Ken Kennedy. "Constructing the Procedure Call Multigraph". *IEEE Trans. Softw. Eng.*, 16(4):483–487, 1990.

3. Chen, Huo Yan, Yu Xia Sun, and T.H. Tse. "A scheme for dynamic detection of concurrent execution of object-oriented software". *IEEE International Conference on Systems, Man and Cybernetics.*, volume 5, 4828–4833. IEEE, October 2003.

4. Cheng, Jingde. "A classification of tasking deadlocks". *Ada Lett.*, X(5):110–127, 1990.

5. Cheng, Jingde. "A survey of tasking deadlock detection methods". *Ada Lett.*, XI(1):82–91, 1991.

6. Demartini, C. and R. Sisto. "Static Analysis of Java Multithreaded and Distributed Applications". *PDSE '98: Proceedings of the International Symposium on Software Engineering for Parallel and Distributed Systems*, 215. IEEE Computer Society, Washington, DC, USA, 1998.

7. Engler, Dawson and Ken Ashcraft. "RacerX: effective, static detection of race conditions and deadlocks". *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, 237–252. ACM Press, New York, NY, USA, 2003.

8. Fredkin, Edward. "Trie memory". *Commun. ACM*, 3(9):490–499, 1960.

9. Geer, David. "Chip Makers Turn to Multicore Processors". *Computer*, 38(5):11–13, May 2005.

10. Gosling, James, Bill Joy, Guy L. Steele, and Gilad Bracha. *Java Language Specification.* Addison-Wesley, 3nd edition, 2005.

11. Grove, David and Craig Chambers. "A framework for call graph construction algorithms". *ACM Trans. Program. Lang. Syst.*, 23(6):685–746, 2001.

12. Grove, David, Greg Defouw, Jeffery Dean, and Craig Chambers. "Call Graph Construction in Object-Oriented Languages". *Proceedings of the 12th ACM conference on Object-oriented programming, systems, languages, and applications (OOPSLA 1997)*, 108–124. ACM Press, 1997.

13. Hall, Mary W. and Ken Kennedy. “Efficient call graph analysis”. *ACM Lett. Program. Lang. Syst.*, 1(3):227–242, 1992.

14. Havelund, Klaus and Jens U. Skakkebæk. “Applying Model Checking in Java Verification”. *Proceedings of the 5th and 6th International SPIN Workshops on Theoretical and Practical Aspects of SPIN Model Checking*, 216–231. Springer-Verlag, London, UK, 1999.

15. Havelund, Klaus and Thomas Pressburger. “Model Checking Java Programs using Java PathFinder.” *STTT*, 2(4):366–381, 2000.

16. Holzmann, Gerard J. “The Model Checker SPIN”. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997.

17. Holzmann, Gerard J. *The SPIN Model Checker, Primer and Reference Manual*. Addison-Wesley, Reading, Massachusetts, first edition, 2003.

18. Hovemeyer, David and William Pugh. “Finding bugs is easy”. *SIGPLAN Not.*, 39(12):92–106, 2004.

19. Levine, Gertrude Neuman. “Defining deadlock”. *SIGOPS Oper. Syst. Rev.*, 37(1):54–64, 2003.

20. Long, Douglas L. and Lori A. Clarke. “Task interaction graphs for concurrency analysis”. *ICSE ’89: Proceedings of the 11th international conference on Software engineering*, 44–52. ACM Press, New York, NY, USA, 1989.

21. Rugina, Radu and Martin C. Rinard. “Pointer analysis for structured parallel programs”. *ACM Trans. Program. Lang. Syst.*, 25(1):70–116, 2003.

22. Ryder, Barbara G. “Constructing the Call Graph of a Program.” *IEEE Trans. Software Eng.*, 5(3):216–226, 1979.

23. Savage, Stefan, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. “Eraser: a dynamic data race detector for multithreaded programs”. *ACM Trans. Comput. Syst.*, 15(4):391–411, 1997.

24. Wikipedia. “Edward Fredkin — Wikipedia, The Free Encyclopedia”, 2005. URL `http://en.wikipedia.org/w/index.php?title=Edward_Fredkin\&oldid=28511622`. [Online; accessed 29-January-2006].

25. Zhou, B., R. T. Yeh, and P.A. Ng. “Principle of Deadlock Detection in Ada Programs”. *Proc. of the 6th IEEE ICDCS*, 572–579. IEEE, 1986.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704–0188

The public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden to Department of Defense, Washington Headquarters Services, Directorate for Information Operations and Reports (0704–0188), 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202–4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to any penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number. **PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ADDRESS.**

1. REPORT DATE *(DD–MM–YYYY)*
23–03–2006

2. REPORT TYPE
Master's Thesis

3. DATES COVERED *(From — To)*
Sept 2004 — Mar 2006

4. TITLE AND SUBTITLE

Toward the Static Detection of Deadlock
in Java Software

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)

Fadul, Jose, E., Capt, USAF

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Air Force Institute of Technology (AFIT/EN)
Graduate School of Engineering and Management
2950 Hobson Way
WPAFB OH 45433-7765

8. PERFORMING ORGANIZATION REPORT NUMBER

AFIT/GE/ENG/06-19

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

HCSS/R21
Mr. William B. Martin
Chief, High Confidence Software and Systems Division Information Assurance Research Group National Security Agency
9800 Savage Road
Fort George G. Meade, MD 20755-6511
e-mail: wbmarti@alpha.ncsc.mil
comm: 1-(301)-688-1057

10. SPONSOR/MONITOR'S ACRONYM(S)

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION / AVAILABILITY STATEMENT

Approval for public release; distribution is unlimited.

13. SUPPLEMENTARY NOTES

14. ABSTRACT Concurrency is the source of many real-world software reliability and security problems. Concurrency defects are difficult to detect because they defy conventional software testing techniques due to their non-local and non-deterministic nature. We focus on one important aspect of this problem: static detection of the possibility of deadlock-a situation in which two or more processes are prevented from continuing while each waits for resources to be freed by the continuation of the other. This thesis proposes a flow-insensitive interprocedural static analysis that detects the possibility that a program can deadlock at runtime. Our analysis proceeds in two steps. The first extracts the "real" call graph decorated with acquired locks from the target program. The second analysis this decorated graph to report how a possible deadlock may occur at runtime. We demonstrate our analysis via a prototype implementation that detects deadlock conditions within two small Java programs. The two principle limitations of our analysis are on the target program: (1) we need its "real" call graph and (2) its overall size is limited. Construction of the "real" call graph requires perfect aliasing information. The program's size our technique is able to analyze is roughly 35 kSLOC.

15. SUBJECT TERMS

software engineering, software tools, software metrics, computer program verification, verification, Deadlock detection, static analysis

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			Robert P. Graham, Maj, USAF (AFIT/ENG)
U	U	U	UU	119	19b. TELEPHONE NUMBER *(include area code)* 1-937-255–3636, ext. 7256 robert.graham@afit.edu