

Debugging and Testing Concurrent Programs with Efficient Test Case Generation

効率良いテストケース生成による
並行処理プログラムのデバッグとテスト

SETIADI THEODORUS ERIC

*Department of Information System Fundamentals [FS]
Graduate School of Information Systems*

THE UNIVERSITY OF ELECTRO-COMMUNICATIONS

June 2015

APPROVAL

Name: Setiadi Theodorus Eric
Degree: Doctor of Philosophy
Title of thesis: Debugging and Testing Concurrent Programs
with Efficient Test Case Generation

Examining Committee: Prof. Hiroki Honda
Prof. Tadashi Ohmori
Prof. Akihiko Ohsuga
Asc.Prof. Yasuyuki Tahara
Prof. Emeritus Mamoru Maekawa
Prof. Yoshikatsu Tada

Date Approved: June 2015

Copyright 2015
by

SETIADI THEODORUS ERIC

All rights reserved

効率良いテストケース生成による 並行処理プログラムのデバッグとテスト

概 要

マルチスレッドからなる並行処理プログラムのデバッグは逐次処理プログラムに比較してはるかに難しい。その理由の一つは、エラーの再現が難しいからである。トレースを取るために、並行処理プログラムを再実行したり、コードに何らかの措置を施すと、実行のタイミングが変化したり、異なった実行パスを取ることとなる。即ち、元のエラーが発生した厳密なタイミングは未知である。エラーを再現するためには、たとえ入力変数の値が同一でも、インターリーブを変化させた多くのテストケースの実行が必要となる。しかも、それらを全て実行できるとは限らない。

本論文では、マルチスレッド並行処理プログラムにおいて、異なったスケジュールや割込タイミングの違いに起因するエラー、特に、レース条件の再現のための方法を提案する。実行トレースから得られた限定された情報の範囲で、全ての可能なテストケースを生成し、レース条件を検出するデバッグ/テストシステムを提案した。これまで、*partial order reduction*を用いたテストケース削減に関する多くの研究が存在するが、レース条件の検出という意味ではまだ冗長な部分が存在する。本論文での目的は、効率的にレース条件を検出することにある。そのために、3つの手法を提案する。

一つ目は、レース条件検出能力を維持しながら、冗長なテストケースを削除することである。提案手法の独創性は、分岐に与えるインターリーブの影響に注目し、トレース情報から得られたデータフロー情報を活用し、分岐結果に影響を与えるインターリーブのみを特定することにある。既存手法の多くは、共有変数に影響を与える全てのインターリーブを特定することを行っている。この違いにより、提案手法はテストに必要なインターリーブの数をさらに大きく削減することができる。同一のロック状態と共有変数へのアクセス状態を有する実行パスを“レース等価グループ”として1つのグループにまとめ、そのグループを構成するメンバーの中の1個のみをテストすることにより、レース条件テストに要する労力を軽減する。さらに、提案手法は既存の実行トレースのモデルを拡張し、*lock/unlock*と*wait/notify*依存性により実行不可能なインターリーブを特定できる。こういった実行不可能な

インターリーブをテストケースから省くことにより、テストケース削減に大きく寄与できることを示した。

二つ目は、テストケースを生成するために必要なメモリ容量を削減することである。既存のreachability testing手法は、実行されることがないインターリーブに対してもテストケースを生成する。本提案手法はデータ依存性を解析することにより、ロック状態と共有変数へのアクセス状態に影響を与えるインターリーブのみ生成する。これにより、テストケースを生成するためのグラフの大きさを削減でき、その維持のためのメモリ容量を削減できる。実際、Apache Commons Poolに対する実験結果では、提案手法によってグラフのサイズが990ノードから4ノードに減少した。

三つ目は、繰り返しテストの労力を軽減することである。本提案手法を含むデバッグ/テスト手法においては、プログラムの実行トレースを繰り返し取ることによりプログラム誤りの有無と、その誤りの検出を行う。本提案手法では以前の実行とレースとテストの結果を利用して、レース条件をチェックする手間を軽減する。従来法では、テスト毎に全体の実行トレースを取り、レース条件をチェックする必要がある。本提案手法は、生成されたインターリーブに基づき、ロック状態を変更する可能性がある実行トレースの部分を識別することができる。そのことにより、ロック状態を変更する可能性のある実行トレースの部分のみを再チェックの対象とすることにより、チェックの数を削減できた。

以上の新たな改善策の導入により、本提案手法は与えられた入力値に対して、可能なインターリーブの網羅的なテストを、はるかに少ない労力でもって達成する。インターリーブに起因するプログラムエラーの有無、エラーが発生するインターリーブ（パス）、ロックが正しく行われていない共有変数へのアクセスに関する情報がプログラマに提供される。

ABSTRACT

Debugging multi-threaded concurrent programs is more difficult than sequential programs because errors are not always reproducible. Re-executing or instrumenting a concurrent program for tracing might change the execution timing and might cause the concurrent program to take a different execution path. In other words, the exact timing that caused the error is unknown. In order to reproduce the error, one needs to execute the concurrent program with the same input values many times as test cases by changing interleavings, but it is not always feasible to test them all.

This dissertation proposes a debugging/testing system that generates all possible executions as test cases based on the limited information obtained from an execution trace, and then detects potential race conditions caused by different schedules and interrupt timings on a concurrent multi-threaded program. There are a number of studies about test cases reduction using partial order reduction, but there are still redundancies for the purpose of checking race conditions. The objective is to efficiently reproduce concurrent errors, specifically race conditions, by proposing three methods.

The first is to reduce the numbers of interleavings to be tested. This is achieved by reducing redundant test cases and eliminating infeasible ones. The originality of the proposed method is to exploit the nature of branch coverage and utilize data flows from the trace information to identify only those interleavings that affect branch outcomes, whereas existing methods try to identify all the interleavings which may affect shared variables. Since the execution paths with the same branch outcomes would have equivalent sequences of lock/unlock and read/write operations to shared variables, they can be grouped together in the same “race-equivalent” group. In order to reduce the task for reproducing race conditions, it is sufficient to check only one member of the group. In this way, the proposed method can significantly reduce the number of interleavings for testing while still capable of detecting the same race conditions. Furthermore, the proposed method

extends the existing model of execution trace to identify and avoid generating infeasible interleavings due to dependency caused by lock/unlock and wait/notify mechanisms.

Experimental results suggest that redundant interleavings can be identified and removed which leads to a significant reduction of test cases. We evaluated the proposed method against several concurrent Java programs. The experimental results for an open source program Apache Commons Pool show the number of test cases is reduced from 23, which is based on the existing Thread-Pair-Interleaving method (TPAIR), to only 2 by the proposed method. Moreover, for concurrent programs that contain infinite loops, the proposed method generates only a finite and very few numbers of test cases, while many existing methods generate an infinite number of test cases.

The second is to reduce the memory space required for generating test cases. Redundant test cases were still generated by the existing reachability testing method even though there was no need to execute them. Here, we propose a new method by analyzing data dependency to generate only those test cases that might affect sequences of lock/unlock and read/write operations to shared variables. The experimental results for the Apache Commons Pool show that the size of the graph for creating the test cases is reduced from 990 nodes, as based on the reachability testing method used in our previous work, to only 4 nodes by our new method.

The third improvement is to reduce the effort involved in checking race conditions by utilizing previous test results. Existing work requires checking race conditions in the whole execution trace for every new test case. The proposed method can identify only those parts of the execution trace in which the sequence of lock/unlock and read/write operations to shared variables might be affected by a new test case, thus necessitating that race conditions be rechecked only for those affected parts.

From the new improvements introduced above, the proposed methods accomplish to significantly reduce the efforts for exhaustively checking all possible interleavings. The proposed methods provide programmers the information regarding whether there exist program errors caused by interleavings, the

interleaving (path) when the errors occurred, and accesses to shared variables with inconsistent locking.

Contents

ABSTRACT.....	VI
CONTENTS	IX
LIST OF TABLES.....	XIII
LIST OF FIGURES	XIV
CHAPTER 1. INTRODUCTION	1
1.1 Background.....	1
1.2 Problem and Objective.....	5
1.2.1 Problem	5
1.2.2 Objective.....	6
1.3 Motivation	13
CHAPTER 2. RELATED WORK.....	17
2.1 Error Prevention	17
2.2 Error Detection	18
2.3 Static Error Detection	19
2.4 Dynamic Error Detection	20
2.5 Non-Deterministic Execution.....	23
2.6 Deterministic Execution	23
2.7 Deterministic Replay.....	24
2.8 Deterministic Testing	25
2.9 Partial Approach for Deterministic Testing.....	27
2.9.1 Structural Coverage.....	27
2.9.2 Partial Order.....	30
2.9.3 Partial Components.....	32

CHAPTER 3. BASIC TERMS AND DEFINITIONS	34
3.1 Concurrency Control Using a Lock Mechanism	34
3.2 Race Conditions	35
3.3 Total Replay	37
3.4 Dynamic Access.....	38
3.4.1 Reference Variable	38
3.4.2 Array.....	39
3.5 Conditional Statements/Branches and Loops.....	40
3.6 Model for Concurrent Program Execution Traces	42
3.7 Execution Paths.....	43
3.8 Interleaving and Branching.....	44
3.9 Access-Manner.....	45
3.10 Race-Equivalent	48
3.11 Concurrent-Pair of Access-Manners	51
3.12 No-Race	53
3.13 Use-Define.....	55
CHAPTER 4. SETTING FOR THE PROPOSED METHOD	58
4.1 Requirements.....	58
4.2 Approach.....	59
CHAPTER 5. PROPOSED METHOD	67
5.1 Avoid Testing Redundant Interleavings	67
5.1.1 Creating Different Race-Equivalent Groups.....	68
5.1.2 Creating a Different Race-Equivalent Group by Changing a Control Flow	69
5.2 Avoid Testing Infeasible Interleavings	84
5.3 Reduce Memory Required for Generating Test Cases.....	86
5.3.1 System Overview	87
5.3.2 Concurrent Dependency Graph.....	89
5.3.3 Traversing a Concurrent Dependency Graph.....	94
5.3.4 Generating Test Cases from a Concurrent Dependency Graph	96
5.3.5 Comparison with the Existing Reachability Testing Method.....	99
5.3.6 Generating Test Cases to Check Consistent Locking for Access through Reference Variables .	101
5.3.7 Generating Test Cases: Traversing a Concurrent Dependency Graph of an Access-Manner	103

5.3.8	Generating Test Cases for Checking Consistent Locking of an Access-Manner	105
5.4	Reducing the Effort Involved in Checking Race Conditions.....	106
5.4.1	Executions in the Same Race-Equivalent Group: No Need to Check Race Conditions.....	107
5.4.2	Executions in a Different Race-Equivalent Group: Check Only Some Parts of Execution Traces Affected by A New Test Case	111
CHAPTER 6. IMPLEMENTATION AND EXPERIMENTS		113
6.1	Lock Mechanism in Java.....	113
6.1.1	Lock Objects	113
6.1.2	Synchronized Methods	113
6.1.3	Synchronized Statements	114
6.2	Interrupt as a Thread in Java Program.....	114
6.3	Tracing	115
6.4	Deterministic Testing	116
6.5	Implementation Diagram	117
6.6	Experiment Results: Test Case Reduction	118
6.6.1	Experiment 1: Apache Commons Pool.....	120
6.6.2	Experiment 2: JTelnet.....	123
6.6.3	Experiment 3: jNetMap	124
6.6.4	Experiment 4: JoBo.....	126
6.6.5	Experiment 5: Apache Derby	128
6.7	Experiment Results: Memory Reduction	128
Note:	129
**	Proposed concurrent dependency graph	129
6.7.1	Experiment 1: jNetMap	129
6.7.2	Experiment 2: Apache Commons Pool.....	132
6.7.3	Experiment 3: JoBo.....	135
CHAPTER 7. DISCUSSIONS		138
7.1	Applicability.....	138
7.1.1	Program Characteristics.....	138
7.1.2	Error Types	143
7.1.3	Execution Environment	144
7.2	Limitations.....	145
7.3	Efficiency	146
7.4	Complexity.....	148
7.5	Correctness	152
7.6	Future Work.....	157

7.6.1 Correctness Criteria.....	157
7.6.2 Target Program	158
7.6.3 Scope	158
7.6.4 Reduction of the Load of Execution Trace.....	158
7.6.5 Reduction of the Need for Executing Test Cases	159
CHAPTER 8. CONCLUSIONS	162
9. GLOSSARY	165
10. REFERENCES	170
11. APPENDICES.....	177
Appendix A	177
Appendix B	178
Appendix C	178

List of Tables

Table 1. Types of deterministic execution	24
Table 2. An example of finding a set of operations that is affecting branch outcomes using Algorithm 3	77
Table 3. An example of a branch-affect table	80
Table 4. Step-by-step example of Algorithm 5	83
Table 5. Step-by-step example of Algorithm 4 (continued).....	84
Table 6. Comparison between the existing variant graph and the proposed concurrent dependency graph	87
Table 7. Definitions in a concurrent dependency graph	91
Table 8. A set of guidelines from the concurrent dependency graph in Figure 52(a).	95
Table 9. A set of guidelines from the concurrent dependency graph in Figure 52 (b)	95
Table 10. Different values of variables among different execution-variants.....	101
Table 11. A set of guidelines for generating test cases for testing pair2 in Figure 11	104
Table 12. Summary of experiment results	120
Table 13. Grouping of test cases for experiment 1	122
Table 14. Summary of experiment results for JTelnet	123
Table 15. Summary of experiment results for jNetMap	125
Table 16. Branch-affect groups for jNetMap	126
Table 17. Summary of experiment results for JoBo	127
Table 18. Comparison of the experiment results for existing variant graph and the proposed concurrent dependency graph	129
Table 19. A set of guidelines from the concurrent dependency graph in Figure 78	131
Table 20. A set of guidelines from the concurrent dependency graph in Figure 80	134
Table 21. A set of guidelines from the concurrent dependency graph in Figure 85	136

List of Figures

Figure 1. Non-deterministic behavior of a concurrent program.....	2
Figure 2. General method for reproducing concurrent multi-threaded program errors	5
Figure 3. Number of possible interlavings	6
Figure 4. Objective.....	7
Figure 5. Comparison between the existing deterministic replay and the proposed total replay.....	8
Figure 6. Scope for the proposed total replay	8
Figure 7. An example of a bug fix using information from a race detector	10
Figure 8. Applicability of the propose method.....	11
Figure 9. Contributions of the proposed methods	13
Figure 10. Examples of grouping for interleavings.....	14
Figure 11. An example of a control flow for a concurrent program	15
Figure 12. Related work	17
Figure 13. Static method and dynamic method for error detection.....	18
Figure 14. Static error detection using Jlint [Artho01]	20
Figure 15. Recent dynamic methods	22
Figure 16. Deterministic execution for replay and testing.....	24
Figure 17. Examples of execution paths combinations.....	29
Figure 18. An example of a critical section.....	34
Figure 19. An example of two threads $T1$ and $T2$ run concurrently and access a shared variable x	35
Figure 20. Examples of consistent locks	36
Figure 21. Examples of reference variables	39
Figure 22. Sharing of an array element	40
Figure 23. An example of a branch that is affected by interleavings	41
Figure 24. (a) An example of a concurrent program. (b) Flow graph. (c) Flow graph for read and write operations.....	43
Figure 25. An example of a conditional statement.....	43
Figure 26. Examples of different concurrent execution paths for program in Figure 25.....	44
Figure 27. An example of $L(Ti)$ for a usual access-manner with three locks.....	46
Figure 28. An example of a race-equivalent for two executions.....	48

Figure 29. An example of set of access-manners for a loop	50
Figure 30. Examples of some concurrent-pairs of access-manners in an execution trace.....	52
Figure 31. An example of concurrent-pair of access-manners which is no-race.....	55
Figure 32. An example of a concurrent program	57
Figure 33. An example of execution traces and some of its use-defines	57
Figure 34. Comparison between the exhaustive method and reachability testing method	60
Figure 35. An Example of a variant graph from an execution trace	64
Figure 36. Reducing test cases by avoiding redundant interleavings	68
Figure 37. Chain of reactions that can cause a different race-equivalent group.....	69
Figure 38. Interleavings and a branch affecting the occurrence of a race condition	70
Figure 39. Examples of grouping by changing a branch outcome	72
Figure 40. Creating a different race-equivalent group by changing a lock sequence	73
Figure 41. (a) An example of a concurrent program (b) Control flow graph (c) Control flow graph for read and write operations.....	74
Figure 42. Examples of use-defines for the concurrent program in Figure 41	74
Figure 43. An example of a variant graph from an execution trace.....	79
Figure 44. Examples of branch-affect groups for the variant graph in Figure 43 ...	80
Figure 45. Branch-affect group table and branch-condition table for the first test case.....	81
Figure 46. Branch-affect group table and branch-condition table when Algorithm 5 terminates.....	84
Figure 47. An example of the extension of a variant graph.....	86
Figure 48. General idea to reduce memory required for generating test cases.....	87
Figure 49. Overview of the proposed method	88
Figure 50. Components of a concurrent dependency graph	89
Figure 51. Step-by-step illustration for Algorithm 6	92
Figure 52. (a) An example of a concurrent dependency graph. (b) and its optimized version.....	93
Figure 53. An example of test case generation for different cases	98
Figure 54. An example of a test case generation from a guideline	99
Figure 55. Example of a variant graph.	100
Figure 56. Example of lock variables (a) and reference variables (b).....	101

Figure 57. Examples of three executions with different interleavings.....	103
Figure 58. An example of a concurrent dependency graph for the access-manner $M3$ in Figure 30	104
Figure 59. An example of a test case execution for execution-variant $V2$	105
Figure 60. Reducing the effort involved in checking race conditions.....	106
Figure 61. Examples of the same and a different access-manner caused by a branch	108
Figure 62. Different access-manners caused by a loop	109
Figure 63. Different access-manner caused by a branch and a loop	110
Figure 64. Control transfer from thread $T1$ to $T2$	117
Figure 65. Implementation	117
Figure 66. The effectiveness of the proposed method.....	119
Figure 67. An example of a race condition that is difficult to detect	120
Figure 68. A comparison of exhaustive, TPAIR, and the proposed method	121
Figure 69. Comparison of numbers of test cases	123
Figure 70. The source code of the JTelnet and its execution trace.....	124
Figure 71. The source code of the jNetMap	125
Figure 72. Execution trace of jNetMap	126
Figure 73. The source code of JoBo	127
Figure 74. Execution trace of JoBo	127
Figure 75. The source code of Apache Derby	128
Figure 76. Variant graph for the execution of jNetMap	130
Figure 77. Execution trace of the first test execution of jNetMap	131
Figure 78. An example of a concurrent dependency graph for the execution of jNetMap.....	131
Figure 79. The source code of jNetMap.....	132
Figure 80. An example of a concurrent dependency graph for Apache Commons Pool.....	133
Figure 81. Execution trace of the experiment using Apache Commons Pool.....	133
Figure 82. An example of a test program using the Apache Commons Pool library	134
Figure 83. Concurrent dependency graph for the reference variable $_pool$	134
Figure 84. Execution trace of the first test	135
Figure 85. An example of a concurrent dependency graph for JoBo	136
Figure 86. The source code of JoBo.....	136

Figure 87. Interrupt as a thread	141
Figure 88. Deadlock can be avoided by following the interrupt-as-thread principle.	142
Figure 89. Comparison between race detection (a) and deadlock detection (b)....	144
Figure 90. Memory consistency.....	145
Figure 91. Complexity and the actual workload of the proposed method.....	149
Figure 92. Classification of read/write operations.....	150
Figure 93. The percentage of operations affecting branches for several target programs	151
Figure 94. Operations affecting branches for the Apache Commons Pool.....	152
Figure 95. Proof for correctness	153
Figure 96. Example of a concurrent program with an error	157
Figure 97. Example of execution paths combinations	160
Figure 98. Example of possible interleavings for 2 threads and 3 operations	177
Figure 99. Example of possible interleavings for 2 threads and 3 operations	178
Figure 100. Type of interleavings in a concurrent program.....	179

Chapter 1. Introduction

1.1 Background

Multi-core processors are now used in various computer systems ranging from super computers to PCs, and even to small cellular phones. Concurrent programming plays a very important role in fully exploiting the capability of multi-core processors for improving their performance. A concurrent program contains two or more threads/processes that execute concurrently or in parallel and work together to perform a given task. Using multi-threads can increase computational efficiency and resource utilization. For instance, while one thread is waiting for user input or message from network, other threads can perform different computational tasks. From the view point of structuring software systems, modern complex systems are rather naturally structured and perhaps easier to be understood by using multi-threads. They are often real time and require interactive operations. For example, reactive systems, industrial control systems, financial systems, game software, multi agent systems, web servers, etc. can be structured as multi-threaded concurrent programs which create separate threads to service incoming requests from users or devices.

While multi-threaded concurrent programs offer some advantages, debugging and testing of multi-threaded programs are known to be notoriously difficult [Dowell89]. Since they exhibit non-deterministic behavior, they sometimes produce errors or incorrect behaviors that depend on timings. Such errors, for example unintentional race conditions or deadlocks, are very difficult to uncover during testing (see Figure 1). One reason for this difficulty is that the set of possible different interleavings is huge, and it is not feasible to try all of them. The probability of producing a concurrent error is very low because only a few of the interleavings actually produce concurrent errors. Executing the same tests many times under the same test environment might not produce the error because the same interleaving might be created since the scheduler is deterministic. As a result, such errors often remain undetected until even product deployment where different

environmental conditions are waiting. Many errors are not repeatable, and when an error is detected, much effort must be invested in recreating the conditions under which it had occurred.

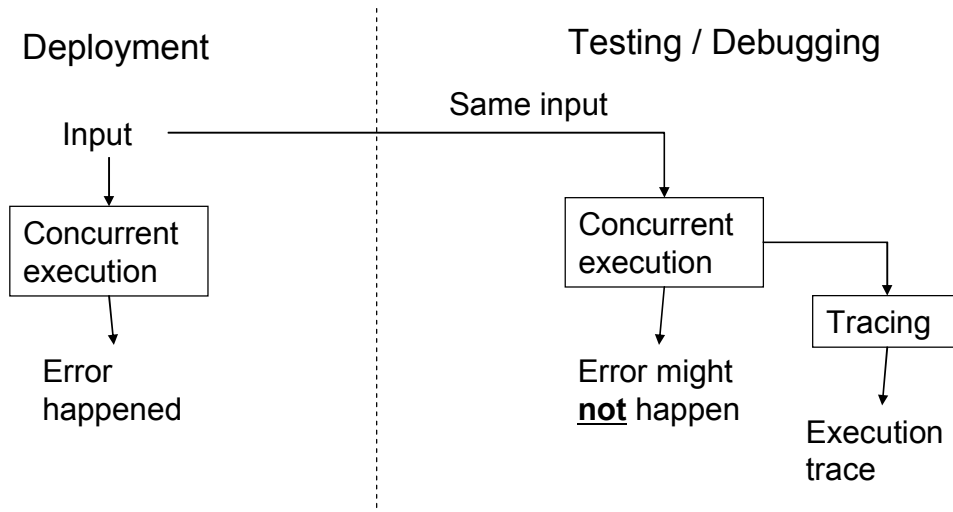


Figure 1. Non-deterministic behavior of a concurrent program

We define the correctness of a concurrent program execution by:

- Data integrity: no race condition.
- Free from concurrency issues: no deadlock, live-locks, and starvation [Rahul08].
- Responsiveness: programs need to response to a user input or an interrupt within a certain time.

In this research, we concentrate on checking race conditions in concurrent programs whose concurrency control is based on locking mechanism. Race conditions occur mostly because shared variables are accessed by threads using inconsistent locking or even no locks [Savage97] [Edelstein03] [Lee96]. Programmers often fail to apply appropriate locks due to difficulties in predicting the execution path or interrupt timing because of the complexity of concurrent programs, especially when branches are affected by access to shared variables and interleavings. To detect race conditions, a programmer can execute the concurrent program and check the execution trace using a dynamic race detector. Unfortunately, concurrent errors might not be easy to detect because a re-executed concurrent program might execute with a different interleaving. Adding additional

commands or instrumentation of the source code to record intermediate results for testing concurrent programs might change the interleaving, so that errors may not show up. Furthermore, dynamic race detectors can detect potential errors only if they show up in a re-execution.

The execution of a concurrent program depends on both input values and interleavings. Race conditions cannot always be detected during testing because their occurrences depend on interleavings. In a concurrent program, a branch can take a different execution path due not only to a different input value, but also to a different interleaving. This situation happens when the program's conditional statement depends on shared variables and the shared variables are affected by interleavings. A change of branch outcomes can affect the sequence of lock/unlock and read/write operations to shared variables, thus affecting the occurrence of race conditions. Hence, an execution trace might contain race conditions that depend on the branches and interleavings.

A typical debugging scenario proposed in this dissertation is as follows:

1. An error that is thought to be caused by timing is discovered, but the exact thread interleaving and interrupt timings are not known.
2. Trace the program using the same set of input values. From the result of execution trace, examine race conditions, deadlocks, and responsiveness (late response). If at least one cause of the error is found (which is lucky), fix it and then continue debugging. Here, an execution trace is a time-stamped trace of all the threads comprising the program.
3. Usually in most cases, the error is not reproduced. Tracing affects the execution of lock/unlock and/or read/write operations to shared variables and interrupt timings. Therefore there is no guarantee that the execution order and timing of the program with tracing is the same as the one in which the error was detected.

For program debugging, one of the common and powerful methods is that, for investigating the cause of incorrect behaviors, additional commands are added or instrumented into the code to display intermediate results, and the program is executed again using the same input. This re-execution of a program is called a "replay". However, this program replay debugging method causes the change of timing and the error might not be reproduced. Gathering trace information while executing a program

using even the same input values can cause schedules and timings to be different from those in the execution in which the actual error had occurred. Therefore, the error cannot always be reproduced because it might execute a different execution path. Assuming almost all logical errors have been removed, then the errors are most likely caused by a different timing. It is well known that debugging such remaining or hidden or infrequent errors is difficult. The exact timing when error had occurred is unknown, so it is difficult to find the true cause of the error. Since the exact interleaving is unknown, we need to try all possible interleavings as test cases to find the execution path where the error had occurred. The problem is that there can be many possibilities of interleavings and interrupt timings.

A program replay is broadly divided into two classes:

- Deterministic replay : a program is re-executed exactly the same interleaving and interrupt timing as previous execution.
- Non-deterministic replay : a program is re-executed, but might not exactly the same interleaving and interrupt timing as previous execution.

If the previous execution is the one that contains an error, then we can reproduce the error using deterministic replay. For a sequential program, it is expected that this deterministic replay is always possible. For a concurrent program, a deterministic replay is difficult. To do a deterministic replay of a concurrent program, one controls the scheduling of threads in the system to obtain the same execution path. If the complete information is obtained concerning the execution in which an incorrect behavior is found, then a deterministic replay is preferable for debugging. A number of techniques for deterministic replay have been devised and it becomes popular because it provides the same degree of debugging easiness as that for sequential programs. When the complete information is not obtained, then a non-deterministic replay is applied.

In a non-deterministic replay, a single execution of a concurrent program with a particular value of an input variable x is insufficient to determine the correctness of the concurrent program when the actual execution timing of the error is unknown. In order to reproduce the same error for debugging multi-threaded concurrent program, it is necessary to change/alter the interleavings (timings) as test cases and test all the possible executions produced from the same input values. Figure

2 shows a general method for reproducing concurrent multi-threaded program errors. Unfortunately there are many possible thread interleavings and interrupt timings, which means it requires a large number of test cases and it is not always feasible to test all of them. The number of different interrupt timings, in particular, is almost unlimited because their granularity is very small. Randomly choosing which execution to be replayed with some heuristics can help to increase the probability of manifesting concurrent error, but often comes with many redundant test cases.

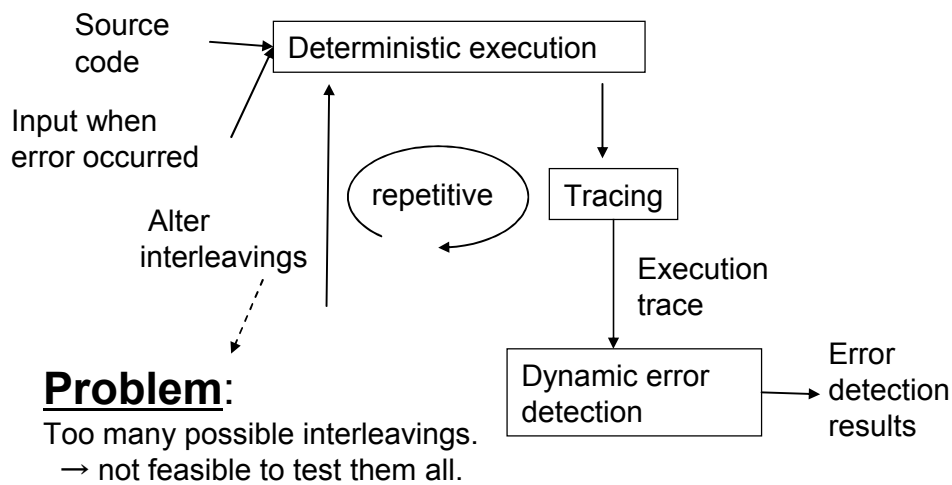


Figure 2. General method for reproducing concurrent multi-threaded program errors

1.2 Problem and Objective

1.2.1 Problem

It is difficult to detect race conditions in concurrent programs if the exact interleaving that causing the error is unknown. In the case of debugging sequential programs, the output results depend only on the input values. Even though instrumentation is added to display/output intermediate results or tracing is applied, it does not affect the process and the result of program execution. Therefore, in investigating the cause of the error, it is possible to repeat the execution and then narrowing down the cause of the error. Unfortunately, this is not the case in debugging concurrent programs because the execution depends not only on the input value but also on interleavings. As such, we must consider all possible interleavings for testing.

Unfortunately, blindly executing all possible interleavings is not usually feasible because of their huge number. Figure 3 shows an example that the number of possible interleavings grows in factorial order as the number of threads and operations increase.

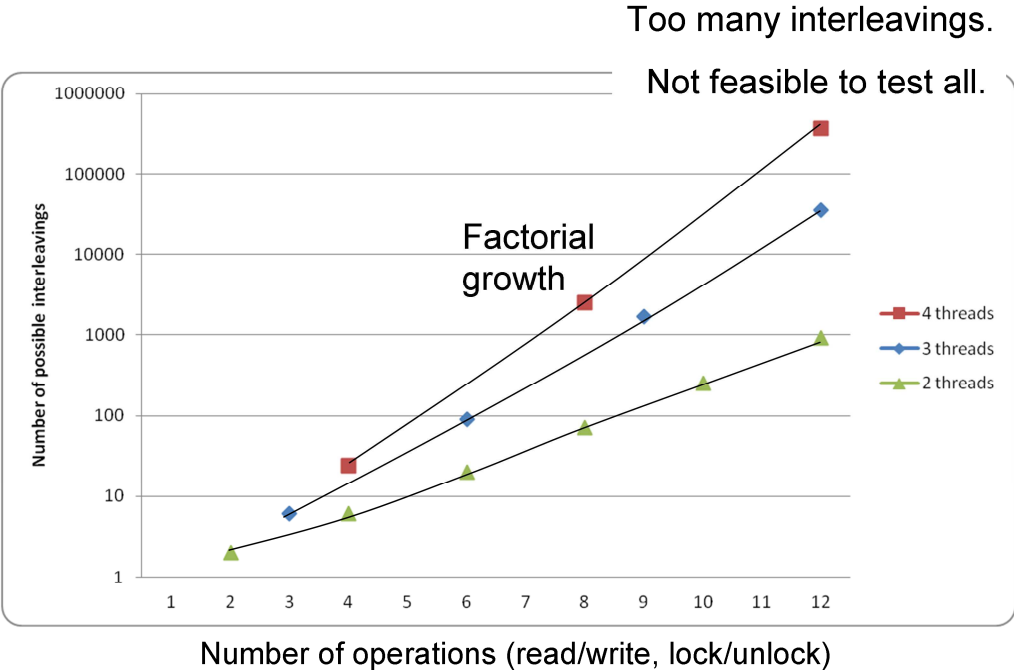


Figure 3. Number of possible interlavings

The main problem is how to reduce this number of testing and the efforts for detecting concurrent errors. Two major issues in testing concurrent programs are efficiency and precision. It is beneficial if, for concurrent programs, the same execution is reproducible during testing and debugging just like sequential programs.

1.2.2 Objective

The objective of this research is to realize debugging capabilities/situations for concurrent programs similar to those for sequential programs even though the exact interleaving that causing the error is unknown. Note that our definition of concurrent programs includes interrupts. Our goal for testing and debugging concurrent systems is the ability to repeat an execution as close as the actual execution in which an incorrect behavior will be manifested even when a trace is

taken. This is achieved by realizing a deterministic replay for concurrent programs which we call it “total replay” (see Figure 4). Total replay for concurrent programs aims at reproducing all possible executions effectively based on limited trace information under the following assumptions:

- Input values are known, but
- Interleaving is unknown (see Figure 6)

Namely, we guarantee that all possible execution paths or all different interleavings are produced and we reduce redundant executions or tests as much as possible while still capable of detecting the same error. Even though the input values are fixed, the range of execution reproduction is still very large (see Figure 3) due to a wide range of different interleaving caused by scheduling and interrupt timings.

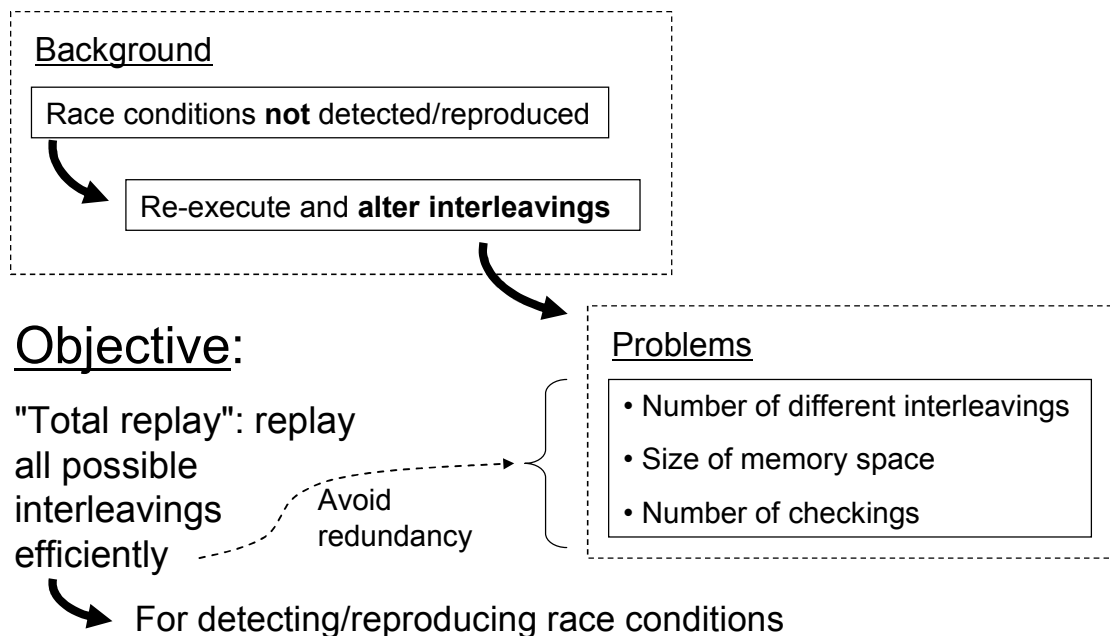


Figure 4. Objective

Figure 5 shows the difference between the existing deterministic replay and the proposed total replay.

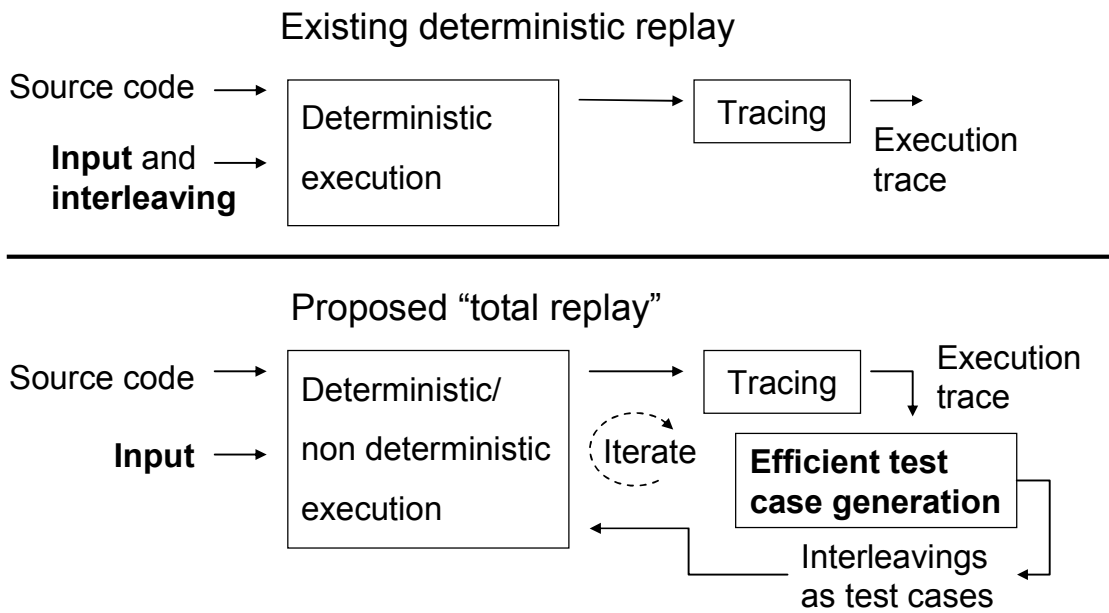


Figure 5. Comparison between the existing deterministic replay and the proposed total replay

Figure 6 shows a debugging process using the proposed method.

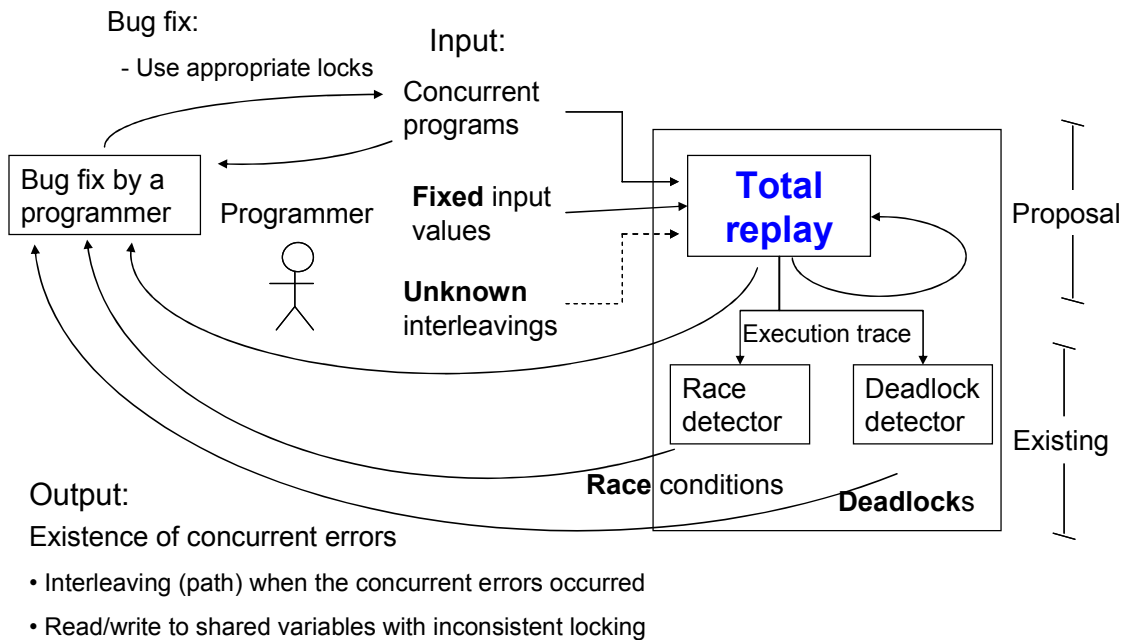


Figure 6. Scope for the proposed total replay

Debugging process:

1. Detection of race conditions by checking the consistent locks in the execution trace.
 - 1.1. Suppose:
 - There are N times accesses to a shared variable x .
 - $ActiveLocks(x,i)$ is the set of locks acquired by the thread when accessing the shared variable x for the i -th access.
 - Consistent locks for accessing the shared variable x is $ActiveLocks(x,0) \cap ActiveLocks(x,1) \cap ActiveLocks(x,2) \cap \dots \cap ActiveLocks(x,N)$.
 - 1.2. Race conditions exist for an access to a shared variable x if the consistent locks in step 1.1. is empty.
2. If concurrent errors are found by a race detector.
 - 2.1. A race detector will give information about the name of lock variables, shared variable names, and line of code where access to the shared variable is not protected by consistent locks. The execution path can be obtained from the execution trace.
 - 2.2. Fix the error by adding appropriate locks in the source code, i.e. deciding the consistent locks for accessing the shared variable.
3. If concurrent errors are **not** found, the error might be caused by other bugs in the program. Such causes are not within the scope of the proposed method.

Figure 7 shows an example of a bug fix using information from a race detector.

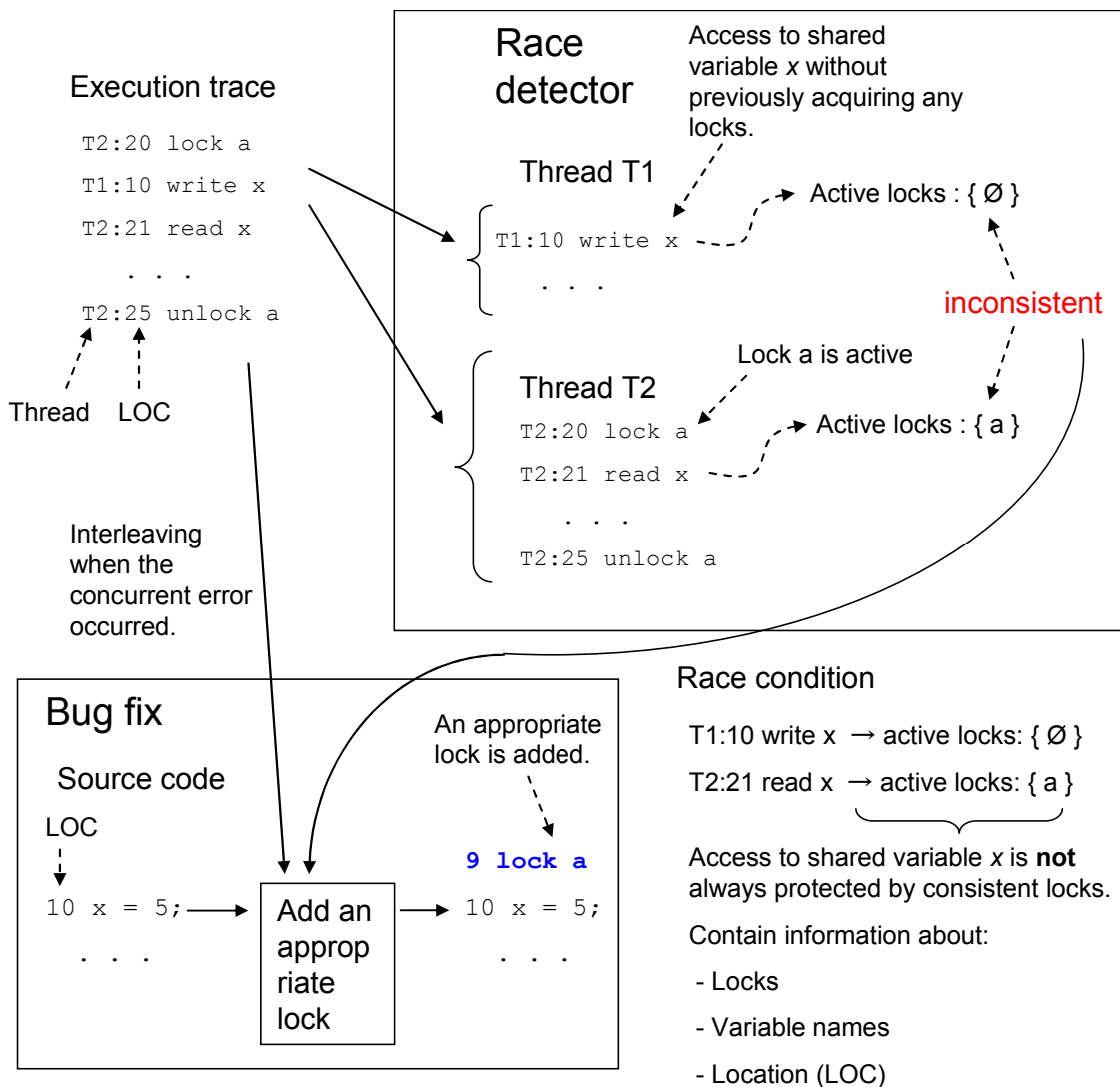


Figure 7. An example of a bug fix using information from a race detector

The applicability of the proposed method:

- At the end of software development phase after all logical errors and conceptual errors are removed, or
- After deployment when an error is found by users

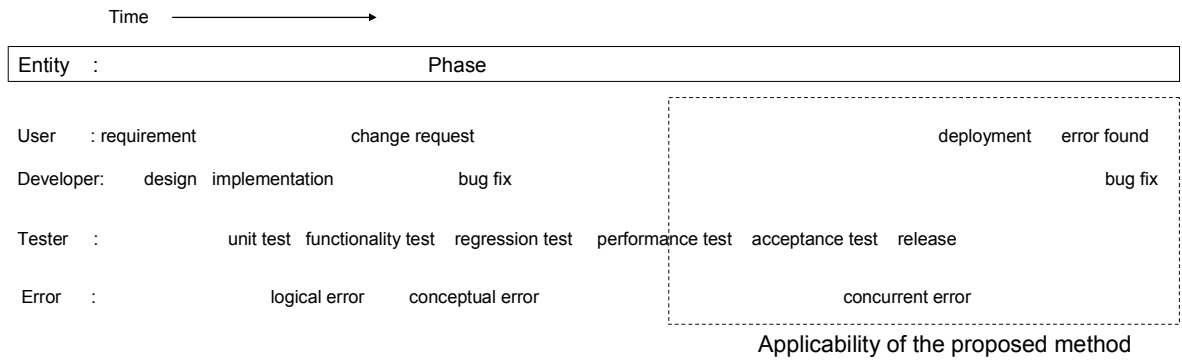


Figure 8. Applicability of the propose method

In this dissertation, we propose a new efficient dynamic method to minimize the number of test cases for detecting concurrent errors. Our method is particularly applied for finding concurrent errors where the detection or the reproduction rate is very low. It iteratively uses previous execution traces as guidelines for generating new test cases. The focus is a debugging of a concurrent program whose behavior has been found anomalous.

The contributions of this dissertation are as follows (refer to Figure 9):

1. Reducing the number of test cases for detecting concurrent errors:

- ✧ Eliminating redundant test cases: The proposed method reduces the number of interleavings to be tested by exploiting the branch coverage information from the execution trace. This is an improvement over the existing reachability testing methods [Hwang95]. The existing reachability testing methods try to identify all interleavings which may affect shared variables, although they may not necessarily affect the sequence of lock/unlock and read/write operations to shared variables; thus redundant interleavings are included. These redundant interleavings are, however, reduced in our method, resulting in a significant reduction in the number of interleavings for checking race conditions. Our method is different from previous methods because it can distinguish those interleavings that can affect branch outcomes and the sequence of lock/unlock and read/write operations to shared variables from those that cannot. Then the proposed method reduces the number of interleavings necessary to be tested by the following:
 - Grouping the interleavings which have the same sequence of lock/unlock

and read/write operations to shared variables.

- Testing only one member from each group.

To the best of our knowledge, this idea has not been exploited so far.

- ✧ Eliminating infeasible test cases: The existing reachability testing methods do not consider the synchronization event dependency of the execution path, e.x. lock-unlock and wait-notify mechanisms. There exist infeasible interleavings due to this dependency. The proposed method extends the existing model of variant graphs (will be defined in section **4.2 Approach**) to identify infeasible interleavings due to this dependency, thereby further contributing to reducing the number of test cases.

2. Reducing memory space required for generating test cases.

Our method exploits data dependency to generate only those test cases that might affect sequences of lock/unlock and read/write operations to shared variables. Our new proposed method requires smaller size graphs for generating test cases compared to the existing reachability testing methods. This means the required memory space is reduced.

3. Reducing the effort involved in checking race conditions.

Our method identifies only the parts of the execution trace whose sequences of lock/unlock and read/write operations to shared variables might be affected by a new test case. Race conditions are then checked again only for those affected parts. For other (unaffected) parts, we can reuse the results from previous executions, thereby reducing the effort involved in checking race conditions.

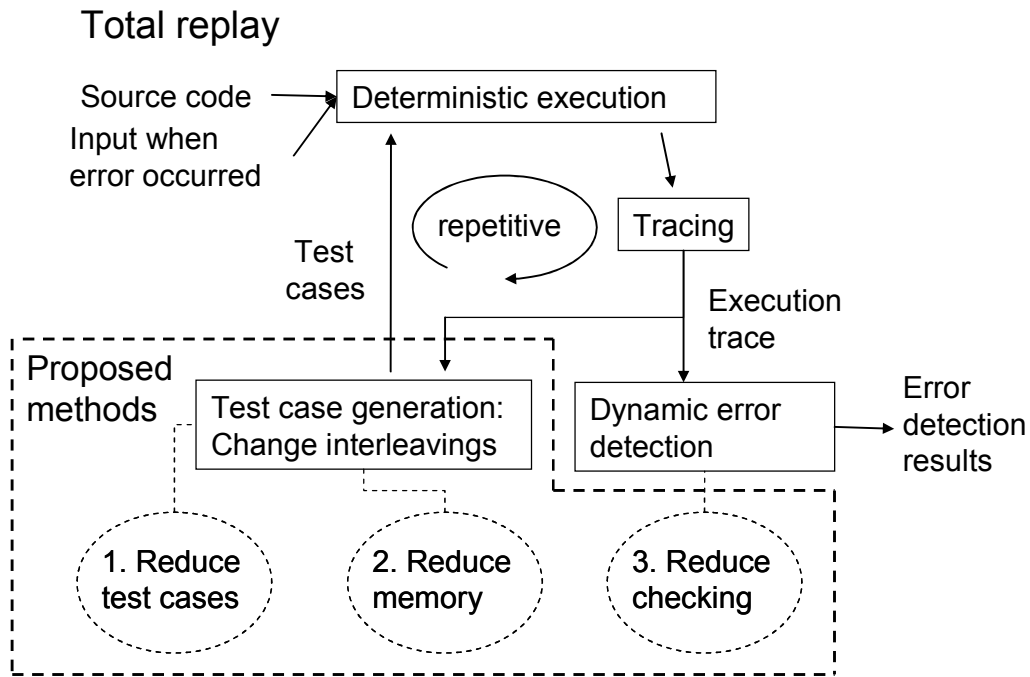


Figure 9. Contributions of the proposed methods

1.3 Motivation

Several methods have been proposed to reduce the number of interleavings for testing. Partial order reduction is a general method which considers only those interleavings that may affect an execution of a program based on certain criteria. One example of the partial order reduction method reduces the number of interleavings by considering only those that may affect the values of shared variables [Godefroid96] [Clarke00] [Godefroid97] and by ignoring the order of “independent” operations. Two operations are said to be independent if any different order of the operations does not affect the values of shared variables. Examples of independent operations are two read operations from different threads accessing the same shared variable. Such interleaving is left unordered because its order is irrelevant to the resulting values of any shared variables.

Unfortunately, such partial order reduction still leaves some redundancy when exploring different execution paths in threads for detecting potential race conditions. Consider the example in Figure 10. In the case that the loop in the thread $T2$ is executed only once, there are six possible different interleavings. The first and the second

interleavings are different only in the order of independent operations, so they will have the same values for shared variables. A similar situation happens for the fifth and sixth interleavings. By ignoring the order of independent operations, there will be only four groups of interleavings with different combinations of values for the shared variables x and y . For the members of the same group, the same read or write operation is guaranteed to use the same value of the shared variable. If the branch depends only on the shared variable x , there are actually only two groups that matter for changing the execution path of thread $T1$. These groups are determined by whether $CS_1 x$ is executed before $CS_A x$ (group 1) or vice versa (group 2). When the loop in the thread $T2$ is executed several times or possibly becomes an infinite loop, there are more possible interleavings that affect the value of the shared variable y , but still there are only two groups of interleavings with respect to different values of the shared variable x . We will use this idea for exploring different execution paths efficiently.

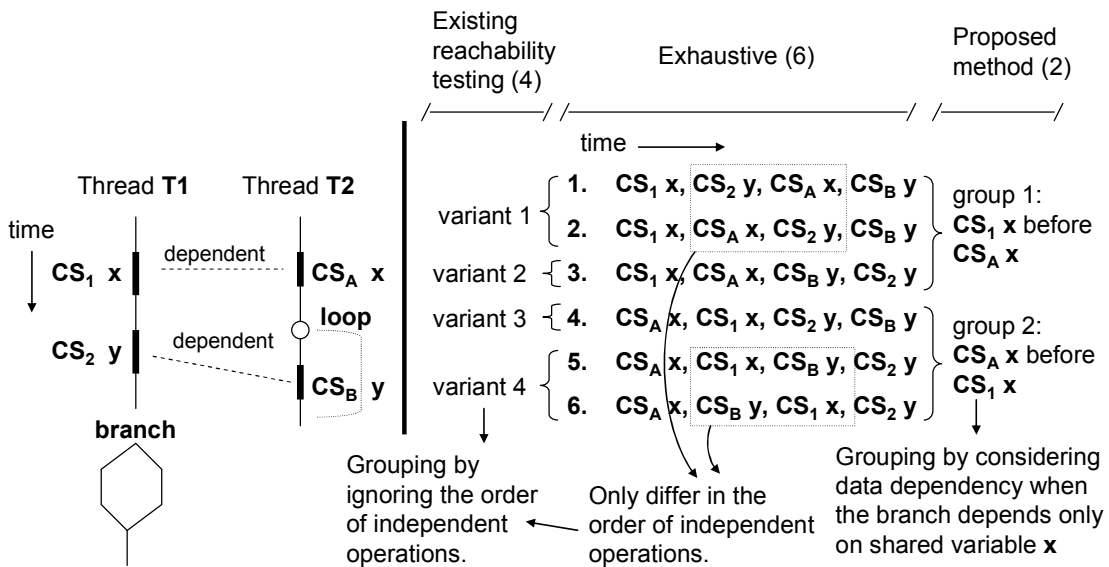


Figure 10. Examples of grouping for interleavings

Figure 11 shows a control flow graph for a concurrent program and its possible execution paths. A thread can take a different execution path when its control flow changes. In a concurrent program, its control flow depends on input data, interleavings, and branches. A different execution path might have a different lock sequence or different read/write operations to shared variables. To detect concurrent errors, we need to find all different interleavings that can change the execution path.

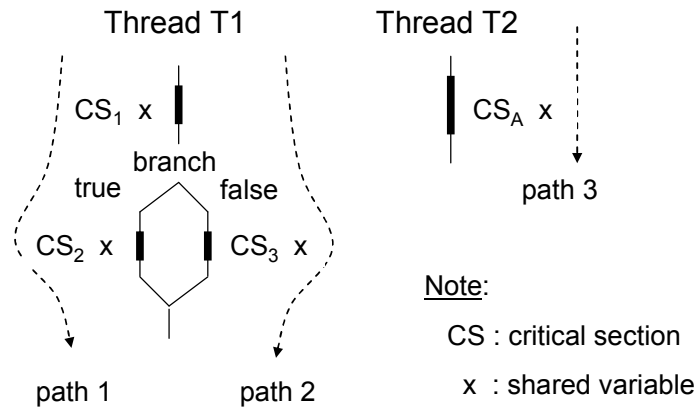


Figure 11. An example of a control flow for a concurrent program

Suppose that path 1 is executed concurrently with path 3 (path 1 || path 3) when the program is first tested. In this case, there are three other possible different interleavings with the following results:

No.	Possible interleavings	Branch outcome	Concurrent paths	Note
1.	$CS_A x, CS_I x, CS_2 x$	<i>true</i>	path1 path3	
2.	$CS_I x, CS_A x, CS_2 x$	<i>false</i>	path2 path3	Infeasible interleaving. It becomes $CS_I x, CS_A x, CS_3 x$
3.	$CS_I x, CS_2 x, CS_A x$	<i>true</i>	path1 path3	

Referring to Figure 11, let us assume that the first interleaving is taken when the program is first tested. The other two interleavings are other possible test cases. Assuming that the branch is conditioned by the shared variable x , the conditional statement of the branch is affected only by the order of $CS_A x$ and $CS_I x$:

- $CS_A x$ is executed before $CS_I x \rightarrow$ branch outcome is *true*
- $CS_I x$ is executed before $CS_A x \rightarrow$ branch outcome is *false*

If the branch condition is *true*, then the execution of path 1 will be concurrent with path 3 (path 1 || path 3). On the other hand, if the branch condition is *false*, then we will have the combination of the execution of path 2 concurrent with path 3 (path 2 || path 3). In this example, $CS_A x$ is executed before $CS_I x$ in the first and the third interleavings, so the branch outcome will be *true* and result in the same execution path 1 for thread $T1$. Since thread $T1$ follows the same execution path in the first and third interleavings, there will be no change in the sequence of lock/unlock and read/write operations to shared variables. For exploring different execution paths

in thread $T1$ caused by the branch, it is sufficient to test only either the first or the third interleaving.

By considering the dependency between the conditional statement in the branch and the shared variables, we can avoid testing interleavings that do not change the execution path of a thread. For the example shown in Figure 11, if we know from the previous executions that the branch is not affected by the shared variable x , then there is no need to test the second or the third interleaving. Of course, the final result for the value of the shared variable x can be different in those interleavings because it might also depend on the order of the critical sections. If the execution path in thread $T1$ changes to path 2, we compare the sequence of lock/unlock and read/write operations to shared variables between CS_2 and CS_3 before checking the race conditions for the concurrent execution of path 2 and path 3 (path 2 || path 3). If the sequence of lock/unlock and read/write operations to shared variables in CS_2 and CS_3 is the same or “equivalent” (will be defined in section **3.10 Race-Equivalent**), then the race conditions are the same as in the first test case (path 1 || path 3) in the previous execution, thus reducing the effort for checking race conditions.

This dissertation consists of eight chapters. Chapter 2 presents the related work and gives a survey on related existing systems for debugging concurrent programs. Chapter 3 presents the basic terms and definitions that will be used in this dissertation. Race conditions are introduced as one of major anomalies in concurrent executions of multiple threads, and concurrency control mechanisms are shown as means for solving race problems. In Chapter 4, we set the conditions and requirements for the method that this dissertation seeks to provide. Among many possible different situations and objectives for debugging and testing concurrent programs, this section makes the conditions and the objectives specific. Chapter 5 proposes a new method to reduce the number of different interleavings for test cases. The method utilizes data flows from the trace information to identify only those interleavings that affect branch outcomes. The number of necessary test cases may be significantly reduced. Chapter 6 discusses implementation methods in Java and presents some of experimental results in comparison to some existing methods. Chapter 7 does some discussions and indicates possible future work. Finally, Chapter 8 gives the conclusions.

Chapter 2. Related Work

Figure 12 shows the area of this research among the related work

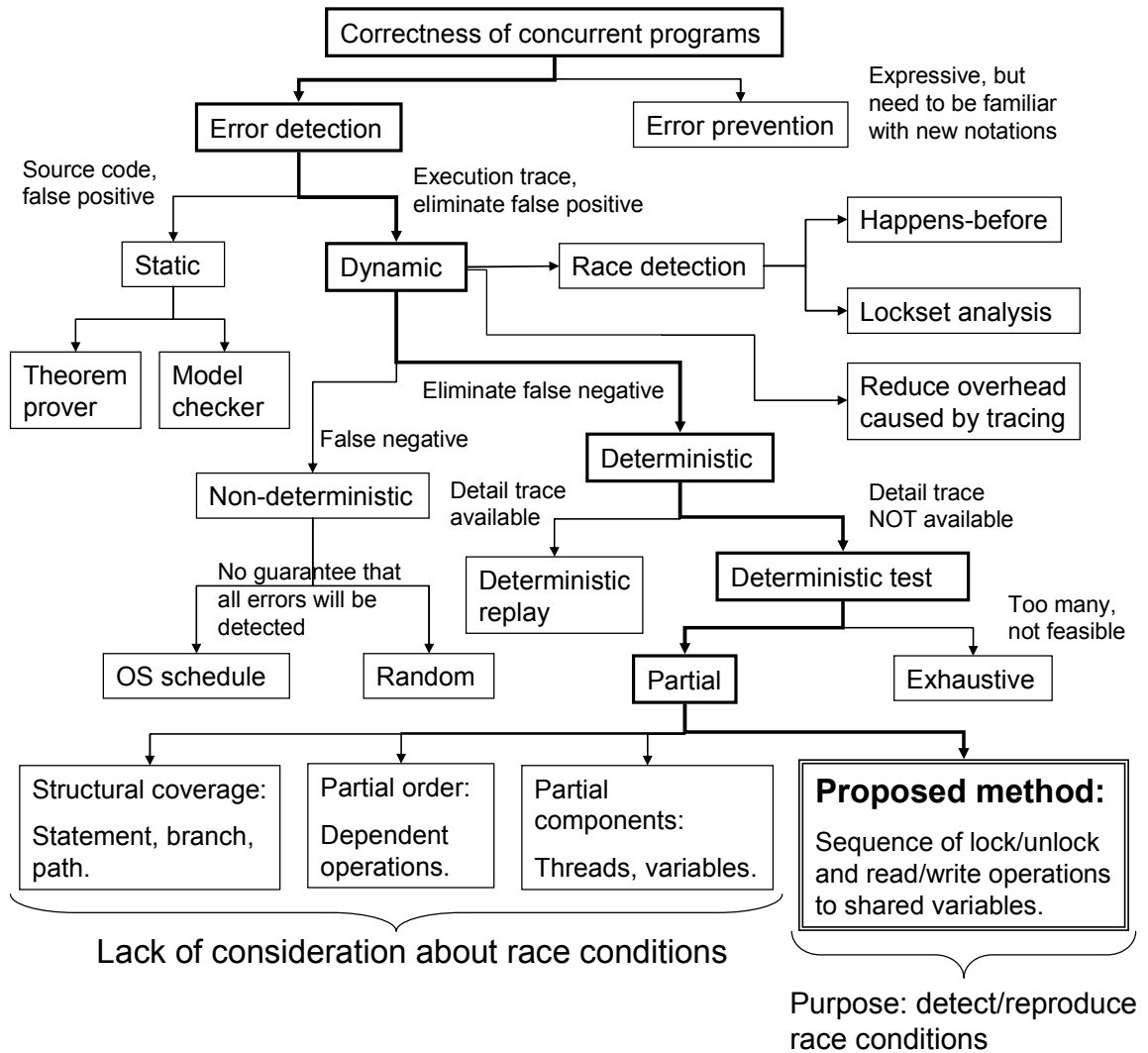


Figure 12. Related work

2.1 Error Prevention

The type-based system [Abdelqawy12] [Beckman06] introduces annotations directly in source code to prevent race conditions. This is an improvement in terms of

expressiveness, for example:

- Different objects of the same class can be protected by different locks.
- Different fields of the same object can be protected by different locks.

When we specify a field to be guarded by a lock, the type system can verify that all accesses of that field are protected by that corresponding lock. However, it has some drawbacks:

- Difficult to use: programmers need to be familiar with the notations of type-based system.
- Concurrent errors might still escape: programmers cannot predict the flow of execution because of the complexity of concurrent programs.

When concurrent errors escape, error detection systems can be useful to detect them.

2.2 Error Detection

Error detection can be classified into two classes; static and dynamic methods (see Figure 13):

- Static methods : employ only source code analysis without executing the program [Boyapati01] [Engler03] [Flanagan00] [Henzinger04] [Sterling93].
- Dynamic methods : actually execute a program and detect errors from the execution trace [Nishiyama04] [Praun01] [Chris01] in addition to the source code information.

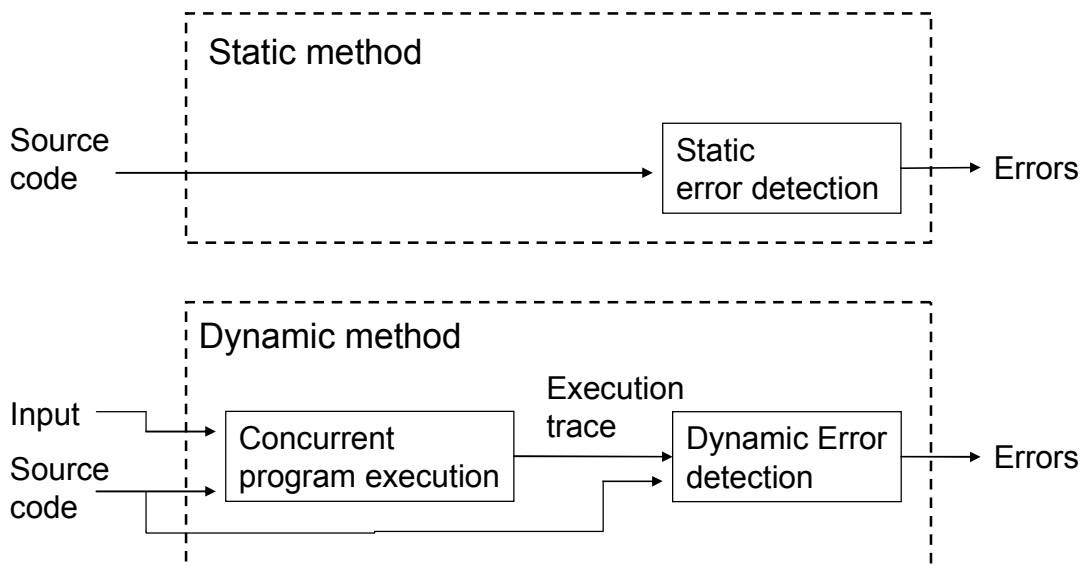


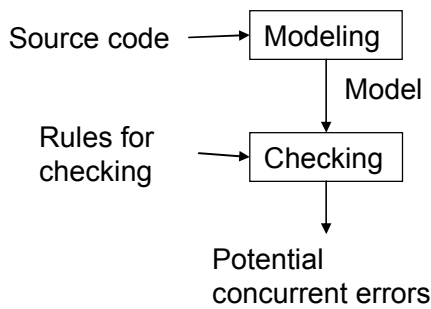
Figure 13. Static method and dynamic method for error detection

2.3 Static Error Detection

Since static methods do not know the precise execution of a program that causes the error, they need to use a conservative (safe) approach by considering all possible executions in order not to overlook potential errors [Yuan05]. Static methods are suitable for ensuring that a program is free from errors because they check all possible program behaviors. However, they often suffer from the detection of false positives when debugging; that is, potential race conditions that do not actually exist in the execution are detected, because it cannot determine the precise set of possible interleavings that cause the errors. Furthermore, dynamic read/write operations to shared variables through reference variables cannot be determined until the execution.

Some types of static methods:

- Inspect all possible different execution sequences for a concurrent program by generating paths from a source code or a model based on control flow analysis [Yang98] [Bertolino94].
- State space search method based on model checking [Godefroid97] [Havelund00] [Holzmann91] [Cleaveland94] [Mutilin06]. This approach can systematically exercise all the possible different sequences of synchronization events in a concurrent program, but suffers from state space explosion problems. Techniques such as partial order reduction [Godefroid96] [Clarke00] can suppress state explosion, but it is necessary to record all execution history to avoid exercising the same execution sequence and unfortunately static model is often too large to build for many applications [Ramalingam02]. A recent work by Jasaitis R., et al [Jasaitis13] extends an existing model checking tool [Havelund00] to verify a distributed system in which the program runs in different machines.
- Check the properties of race conditions and deadlocks from a model based on source code [Artho01], see Figure 14.



Drawback: false positives

- Infeasible execution paths due to branching and data dependency
- Infeasible interleavings due to synchronization, ex. wait-notify

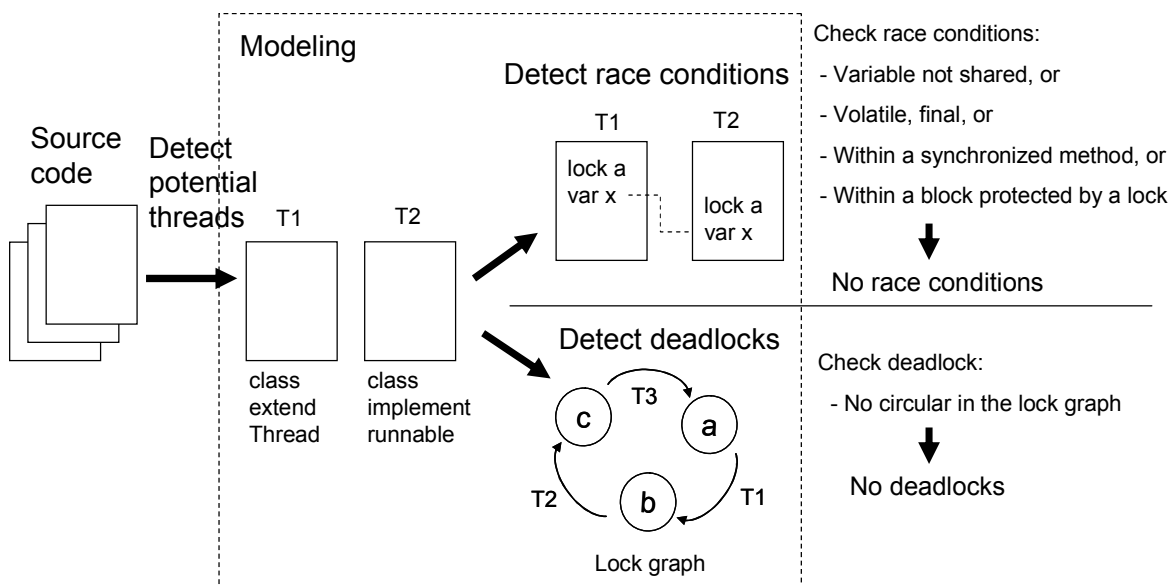


Figure 14. Static error detection using Jlint [Artho01]

2.4 Dynamic Error Detection

In order to increase the precision, dynamic approaches [Nishiyama04] [Praun01] [Chris01] are needed which actually execute a program, and the history of program execution is recorded and analyzed. Since it analyzes the actual execution, it results fewer false positive detection compared to static methods. For debugging purpose, we should also be able to fully utilize information from execution traces. The information obtained in a trace is concerned with read/write

operations to shared variables, lock commands, and synchronized commands related to concurrency control. Some mechanisms for obtaining traces are JVM, Aspect, and manual insertion. Since it detects only potential races based on a particular execution, it does not guarantee the tested program to be bug free.

The dynamic approaches for helping debugging concurrent programs can be applied either for showing/detecting potential errors from a particular execution [Setiadi05] [Setiadi04] [Setiadi04_2] or for localizing/pinpointing the cause of error. Delta debugging [Jong02] localizes the cause of error from the difference/delta between correct and incorrect execution based on a binary search method which, if a binary search is luckily applicable, makes it feasible even for a large program. Others use specifications to find any errors/violations by deterministically execute concurrent program [Chung01][Bochmann94][Carver98]. Very limited conditions need be satisfied for these tools to be applicable in localizing the error. Due to the limitations of such tools, they can be utilized only if the error can be reproduced, that is having the same complete execution trace with timing information from the execution in which the error had occurred, or knowing the correctness/specification of the program, for example a particular incorrect value of a variable, or an occurrence of an exception. Unfortunately, often programs do not stop or produce an exception at the time when error occurs, and it is not always easy for programmers to write a specification.

Existing trace analysis techniques for dynamically detecting potential races are based on:

- Happens-before analysis : Happens-before analysis [Lamport78] based tools [Adve91] [Chris01] [Dinning90] [Crummey91] establish temporal ordering on program statements.
- Lockset analysis : Lockset analysis based tools [Savage97] [Nishiyama04] [Praun01] verify that a program execution satisfies a locking discipline. Eraser [Savage97], for example, is a lockset analysis that identifies potential race from a particular execution by checking lock consistency for access to shared variables.f

Most of research directions in this field are to reduce:

- False positives [Yuan05] [Nishiyama04] [Netzer91].

- Overhead caused by tracing [Choi91] [Huang11].

J. Huang, J. Zhou, and C. Zhang [Huang11] identified one of the causes of redundancy to be that an execution trace often contains a large number of events that are mapped to the same lexical statements in the source code (see Figure 15). However, removing them without careful analysis might cause false negatives because they might affect the reproduction of race conditions. This situation happens when a number of events from the same lexical statement in the source code affect a conditional statement in a branch whose “then” and “else” statements have a different sequence of lock/unlock and read/write operations to shared variables.

By checking happens-before relation or locking discipline for lockset, potential races can be detected, but only if the execution trace contains potential errors i.e. only identification of race conditions that actually occurred in the current run. Unfortunately, the chance that a race condition will occur is low, and an actual race detection tool does nothing to improve it. In a concurrent program, a branch can take a different path not only caused by different input values, but also caused by different scheduling and interrupt timings. Therefore it is not always possible to get the same trace of the execution in which the error had occurred.

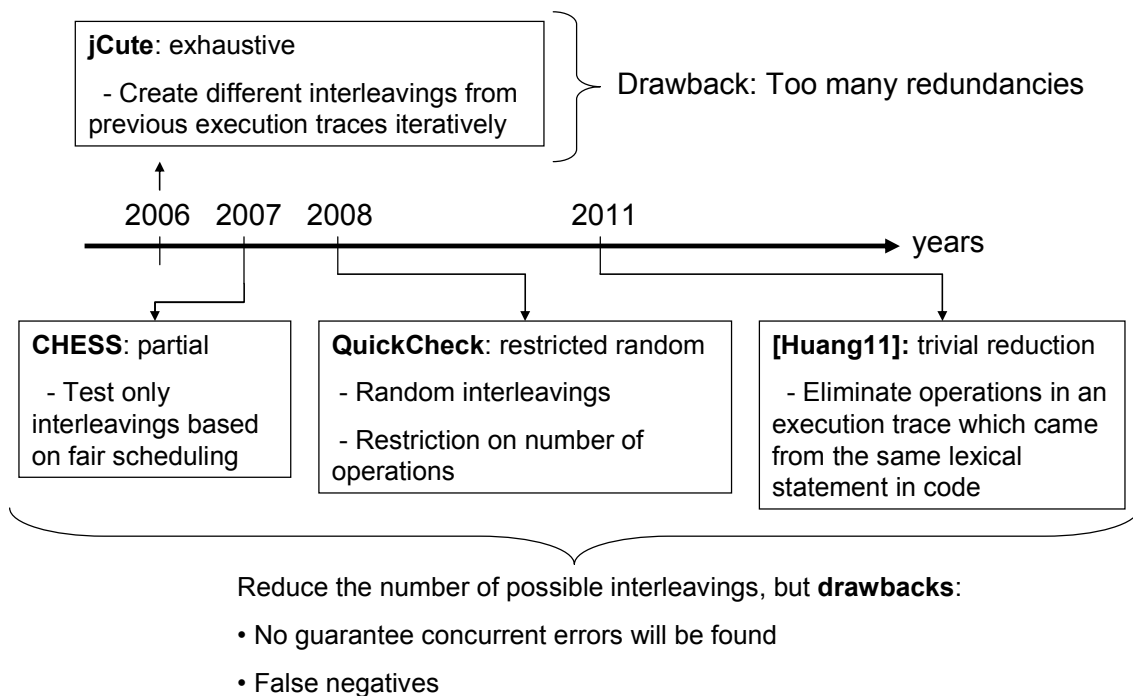


Figure 15. Recent dynamic methods

2.5 Non-Deterministic Execution

Non-deterministic testing executes concurrent programs without precise control of the interleavings. It does not guarantee that errors will be detected, i.e. causing false negatives, because it might only execute some of the possible interleavings. The interleavings can be decided either by:

- Operating system schedule.
- Random.

An operating system schedule determines interleavings based on some policies. Therefore, the same interleaving might be executed even though a concurrent program is executed several times. This causes concurrent errors not to be detected. In order to increase the possibility of occurrence of concurrent errors, “CHESS” [Musuvathi07] generates all interleavings of a given scenario written by a tester based on a “fair scheduling”. In a fair scheduling, “all threads get opportunities to make progress” [Musuvathi07].

A random approach determines interleavings arbitrarily. Since it is random, it might not have a good coverage for detecting errors. An improved random approach that uses a heuristic has been developed for the following purposes:

- Exploring execution paths which have high probability for causing the error [Ben06] [Eytani07] [Stoller02] [Ben03]
- Reducing the search space [Edelstein03]
- Localizing the cause of errors [Ben03] [Edelstein03]

2.6 Deterministic Execution

Deterministic execution controls the interleaving of a concurrent execution. There can be two type of deterministic execution based on the origin of the interleaving as shown in Table 1.

Table 1. Types of deterministic execution

Deterministic execution	Origin of the interleaving	Description
Deterministic replay	Previous execution	Replay exactly the same interleaving as previous execution.
Deterministic testing	Generated as a test case	Execute an interleaving as specified by a test case.

If we have the interleaving from previous execution when a concurrent error occurred, then we can easily reproduce the error using a deterministic replay. Otherwise, we need to use a deterministic testing to find the interleaving in which the error occurred. Figure 16 illustrates the two types of deterministic execution.

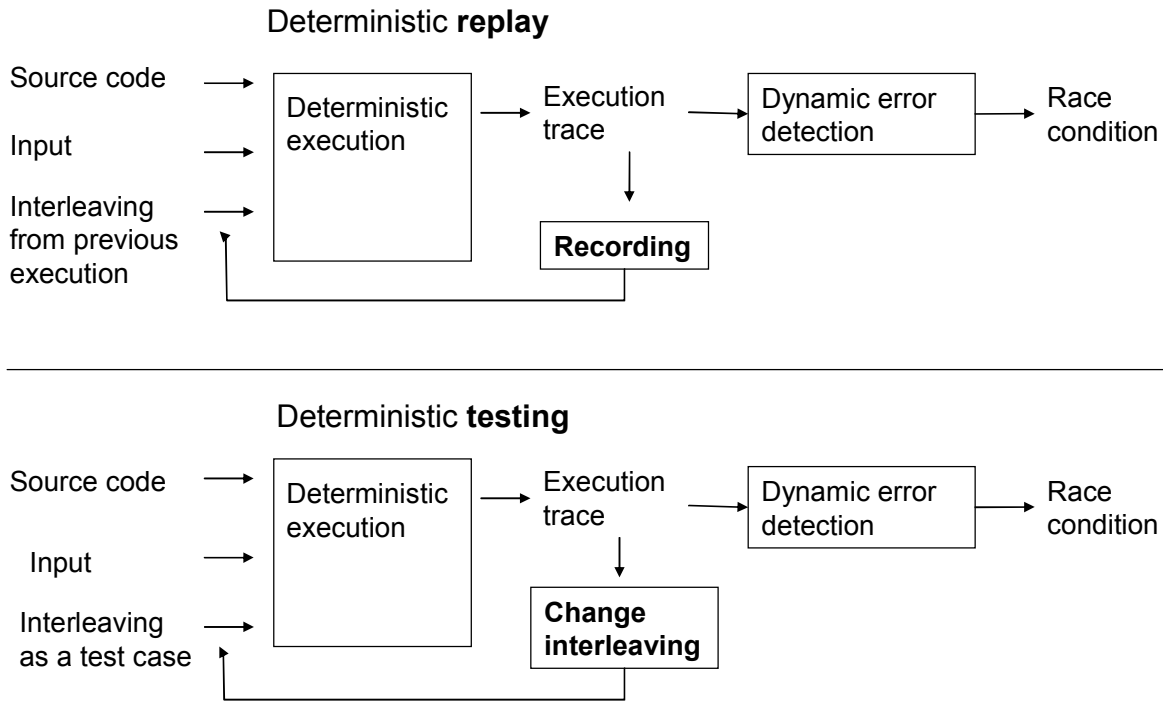


Figure 16. Deterministic execution for replay and testing

2.7 Deterministic Replay

A deterministic replay can reproduce concurrent errors by replaying the program. Some tools enforce a particular schedule to replay based on the information from an execution trace. DejaVu [Jong98] is a deterministic replay system for a modified Sun Microsystems' Java Virtual Machine. Jreplay [Baur03]

instruments Java byte code to replay specific thread schedule. Deterministic replay techniques are available for replaying a concurrent program with the same interleaving. Such techniques record the concurrent execution trace in a recording mode. The recorded execution can be replayed later in a replaying mode for dynamic analysis.

A commercial tool for deterministic replay [Total10] is capable of reproducing the original execution order of threads, thus the same interleaving can be replayed. When a concurrent error is detected during a recording mode, a deterministic replay requires only one execution to replay the error and obtain the execution trace containing the error. This is useful for debugging concurrent programs. However, this is only effective if programmers can identify the errors when a concurrent program is running in recording mode during software development or a testing cycle. Unfortunately, due to the huge number of all possible interleavings, not all of them can be tested during software development or the testing cycle because of time and cost restrictions. Sometimes only regression tests are performed after fixing bugs and the software is quickly deployed in real situations, leaving the possibility that other errors remain. In recording mode, all the information necessary for replaying can be traced using instrumentation [Baur03] or a specialized virtual machine [Jong98]. Hence, recording mode will be different from the normal execution which is known as probe effect:

- Timing: timings change and programs run more slowly because it is taking all the information necessary for replaying.
- Memory: require more memories to store information concerning interleavings and program states.

Therefore, executions cannot always be traced during the deployment of systems that require high performance or where resources are limited, such as in embedded systems. To reduce the probe effect, a special hardware device can be used to communicate with the performance monitor through JTAG (refer to IEEE 1149) for tracing, but many hardware constructions cannot run at full speed when JTAG is used [Sebek02]. The advantage of this approach is that an execution can be traced with minimum interference, but the drawback is expensive hardware costs.

2.8 Deterministic Testing

In cases when an error has happened in the absence of a complete execution trace for replaying, programmers need to test the concurrent program while taking trace information to see if the same error can be detected. Unfortunately, the error might not be easy to detect because a concurrent program can have a different interleaving during re-execution. In this situation, programmers need to control the interleaving and use deterministic testing. Deterministic testing can enforce a particular interleaving specified in test cases. However, the number of possible different interleavings can be huge. The method proposed in this research helps in the efficient generation of test cases to reproduce the same or equivalent execution.

Some tools for deterministic replay can also be used for deterministic testing. For example, in Jreplay [Baur03] programmers can control the interleaving by enforcing thread switching using some additional locks, and can write them in the locations where a thread switch should occur. Enforcing a thread switch is realized by unblocking the next thread in the schedule followed by blocking all other threads, including the current thread. An additional lock object is assigned to each thread. The wait and notifyAll methods are used to implement the block and unblock operations that suspend and resume an execution of a thread. A binary semaphore is used to prevent deadlocks in the control transfer method due to interceptions by the JVM scheduler. Another method devised by Pugh and Ayewah [Pugh07] uses a clock to synchronize the order of executions in multiple threads. Programmers can delay operations within a thread until the clock has reached a desired tick.

Using the trace information, determining which interleavings to be inspected among all the possible execution is important because it has direct impact on the replay efficiency. Basically, there are two approaches:

- (1) Partial : inspect only some of all possible interleavings based on certain criteria.
- (2) Exhaustive : inspect all possible interleavings [Lei06] [Lei04].

In principle, finding all errors requires an exhaustive approach. Unfortunately, exhaustive approaches often suffer from an explosion of the number of possible execution paths to be inspected. To overcome this problem, the concept of partial approach is introduced.

2.9 Partial Approach for Deterministic Testing

The idea behind a partial approach is to identify a group of executions with the same coverage based on some criteria. For each particular group, it is sufficient to test only one interleaving. It is useful for improving efficiency in testing because it reduces the number of tests. In the field of concurrent programs, there exist some criteria to determine which interleavings should be tested based on:

- Program structural (will be defined in subsection **2.9.1 Structural Coverage**)
- Order of operations (will be defined in subsection **2.9.2 Partial Order**)
- Program components (will be defined in subsection **2.9.3 Partial Components**)

2.9.1 Structural Coverage

Structural coverage is based on control flow, which originally was defined for sequential programs. In program testing, we can identify several levels of criteria based on program structure [Prather87] [Taylor92]. These are statement coverage, node coverage, branch coverage and path coverage.

Statement Coverage and Node Coverage

Statement coverage executes all statements in the graph at least once. Node coverage encounters all decision node entry points in the flow graph. Statement coverage and node coverage are rather weak criteria, representing necessary but by no means sufficient conditions for conducting a reasonable test.

Branch Coverage

Branch coverage is a stronger criterion. It encounters all exits branches of each decision node in the flow graph. Some existing researchers worked on testing branch coverage for sequential programs [Prather87] [Gupta00]. In the case of sequential programs, the execution path only depends on the input. Generating data for branch coverage can be obtained by solving linear constraints in the conditional statements of the branches. In the case of concurrent programs, where the input is already known, we need to find different interleavings that cause the change in branch

outcome.

It is necessary to apply branch coverage for checking race conditions since different branch outcome might execute different sequences of lock/unlock and read/write operations to shared variables in a thread. However, it is not sufficient because of the following reasons:

- Branch coverage might not cover all possible concurrent combination of execution paths.

Different interleavings might create different combinations of branch outcomes containing race conditions. Take examples in Figure 17, assumed some test executions execute the combination 3 and combination 4, and found no race conditions. Branch coverage is fully covered because both branches have been executed as *true* and *false*. However, race conditions might exist in different interleavings for the combination 1 and 2.

- Race conditions might still occur even with the same branch outcomes.

Even with the same branch outcomes, different interleavings can change the sequence of lock/unlock and read/write to shared variables causing race conditions. This situation might happen when there are assignments to lock variables or reference variables in different threads (will be explained in **Section 5.3.6 Generating Test Cases to Check Consistent Locking for Access through Reference Variables**).

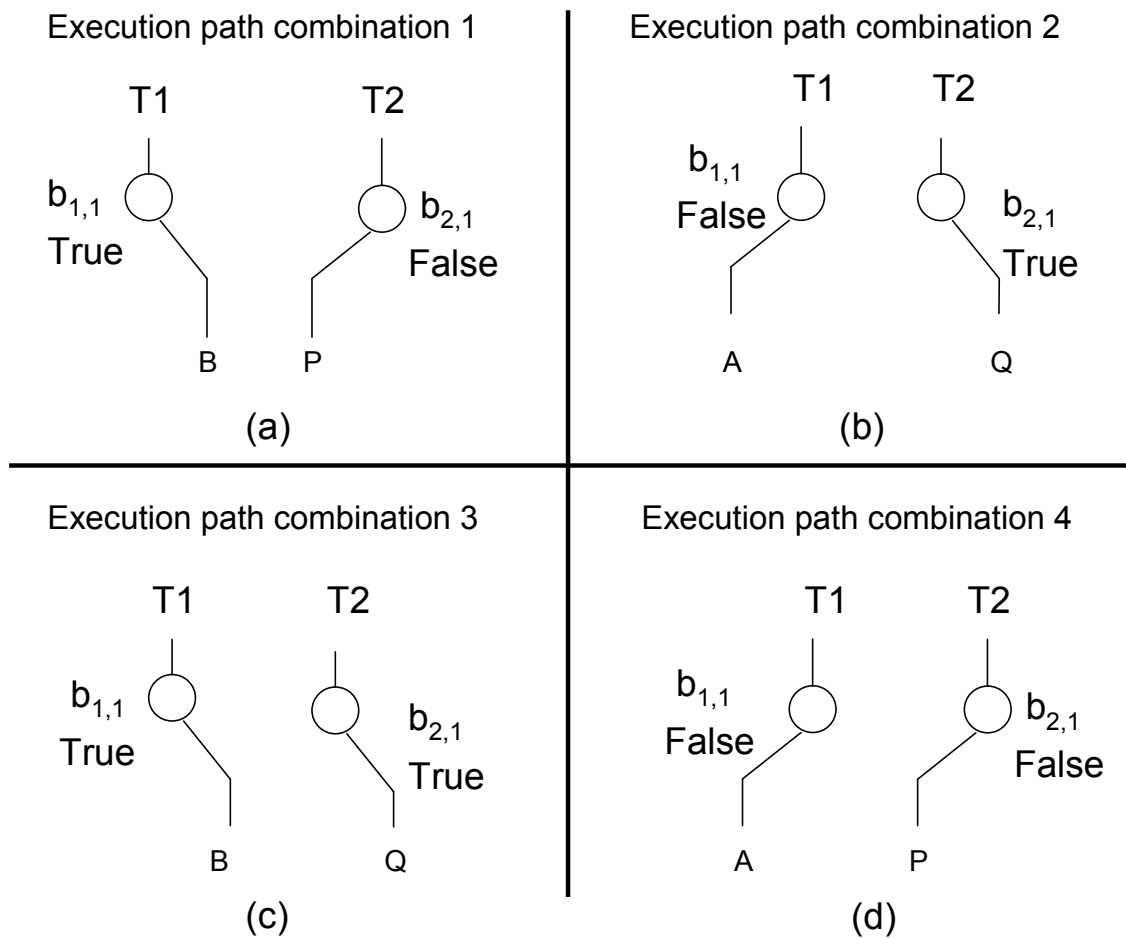


Figure 17. Examples of execution paths combinations

In order to reduce the necessity to execute test cases, the proposed method can identify and create a new execution path by combining the execution paths found in the previous trace, but there is no guarantee that the new execution path will be executed in the actual situation. Therefore there might be overhead for checking unnecessary paths in this approach.

Path Coverage

Path coverage is the most thorough of all. Usually it is required to ensure the correctness of a concurrent program. However, it is normally difficult to achieve because the number of possible execution paths might be huge. Especially in concurrent programs with complex loops and paths.

In particular, interrupts may create an unlimited number of different paths because interrupts may occur at any point of time in the program execution. This problem can be partially mitigated by converting an interrupt processing to a thread. Modern programming languages such as Java may also help mitigating this problem by their capability of encapsulating concurrent operations. In this dissertation, we assume that an interrupt processing is converted to a thread. Nevertheless, since different execution paths might exercise different sequences of lock/unlock and read/write operations to shared variables that can affect consistent locking, it is mandatory to check path coverage to ensure that all concurrent read/write operations to shared variables are consistent.

Koushik Sen and Gul Agha [Sen06] [Sen06_b] explored different execution paths by generating new interleavings as well as new input. Their tool, known as “jCute”, generates all possible interleavings based on previous executions by changing the order of thread executions, starting from the smallest indexed thread. In jCute, some redundancies remain in detecting race conditions, because not all of the generated interleavings will change the sequences of lock/unlock and read/write operations to shared variables.

All of the coverage criteria discussed above are mainly based on program structure, but incomplete for exploring all sequences of lock/unlock and read/write operations to shared variables in order to detect race conditions.

2.9.2 Partial Order

This type of coverage considers only some order of operations by exploiting particular characteristics of concurrent programs.

Order of Dependent Operations

The partial order reduction, which was originally developed for static methods, can be applied to reduce the possible executions. It reduces the execution by defining “equivalency” between execution paths. It reduces possible interleavings by considering only the order of “dependent” operations that affect the value of variables. When the operations are “independent”, meaning that their execution does not

interfere with each other, changing their order of execution will not modify their combined effect. Since changing the order of independent operations will not affect the value of shared variables, partial order reduction method ignores the order of independent operations. Examples of independent operations are two read operations to the same shared variable, and two write operations to different shared variables. This method covers all possible different values of shared variables caused by interleavings. This is necessary if we want to test all possible different values for shared variables affected by different interleavings.

The reachability testing method proposed by [Hwang95] is an example of a dynamic approach that utilizes the partial order reduction. Some of the focuses of those researches in this field are to reduce the cost of tracing [Huang11] and to reduce the search space [Flanagan05]. The reachability testing method is based on prefix-based testing. The advantage of prefix-based testing is to be able to start non-deterministic testing from a specific program state other than the initial state. This kind of methods to reduce the cost of tracing is a natural part of our method which is similar to the old idea of checkpoint/restart. In our method, by properly applying this new checkpoint/restart technique for concurrent programs, any redundant path that can be identified by the trace information is eliminated from the test.

Another data flow coverage criterion proposed by Kojima [Kojima09] also considers the order of data dependent operations which affect the values of shared variables.

Order of Use-define Operations

“Use-define” coverage is a coverage criterion based on data flow. The extension of use-define for concurrent programs was presented by [Lu07] [Yang03] [Yang98]. The use-define will be discussed in section **3.13 Use-Define**.

Order of Synchronization Operation

Synchronization coverage [Bron05] covers different orders of synchronization events from different threads for evaluating concurrent completeness. Its goal is to check whether the synchronization statements have been properly tested. For example,

the *tryLock* method of the Lock interface in Java 1.5 is used to check whether a lock is available. It does not block, but may succeed or fail depending on whether another thread is holding the lock.

The use-define coverage [Yang98] and the synchronization coverage [Bron05] are not suitable criteria for detecting race conditions because:

- Use-define coverage: consider only read/write operations to shared variables.
- Synchronization coverage: consider only locks.

For detecting race conditions, we need to consider both locks and read/write operations to shared variables.

Order of Operations Causing Potential Concurrent Errors

Another work by C. Park, K. Sen, P. Hargrove, and C. Iancu [Park11], known as active testing, generates a set of tuples that represents potential concurrent errors, by performing imprecise dynamic analysis in an execution trace. The format of a tuple corresponds to a particular class of errors. In the later phase, the program is re-executed by actively controlling the thread schedule to confirm the concurrent errors. However, the set of tuples might be incomplete if some tuples were not executed in the previous execution. This situation happens when the executions of some tuples depend on the “then” or “else” statements of a branch whose conditional statement is affected by interleavings. This incomplete set of tuples might cause some false negatives for detecting race conditions. Race conditions can only be detected using dynamic methods if the execution trace contains the potential concurrent errors. Unfortunately in a concurrent program, a branch can take a different execution path not only due to different input values, but also due to different interleavings. Hence, depending on the branches and interleavings, an execution trace might or might not contain potential race conditions.

2.9.3 Partial Components

Some coverage criteria are derived from the existing ones by partially selecting only some program components. For example, such partial selection may pertain only to some threads [Takahashi08], variables, synchronization operations [Bron05], or operations based on temporal order relations [Factor96].

Other methods inspect some subset of interleavings by selecting only combinations from some parameters. The intuition behind the idea is that many errors can be exposed by considering interactions among a small number of parameters. The work from [Lei07] proposes an efficient method for generating test cases for combinatorial testing. This method is effective if we can predict the number of parameters that cause the error and they should be far less than the total number of parameters.

Chapter 3. Basic Terms and Definitions

This section discusses the basic terms and definitions that are used in this dissertation.

3.1 Concurrency Control Using a Lock Mechanism

Race conditions in a concurrent program can be eliminated by using a concurrency control mechanism. Several methods are used for concurrency control, but a lock mechanism is one of the most commonly used methods. In this dissertation, we discuss only concurrency control using a lock mechanism. A lock mechanism is used to enforce exclusive access to a shared variable by lock-unlock operations. A lock mechanism prevents other threads from accessing the locked shared variable (resource) concurrently. The execution section which is protected by a lock is called a "critical section" (see Figure 18).

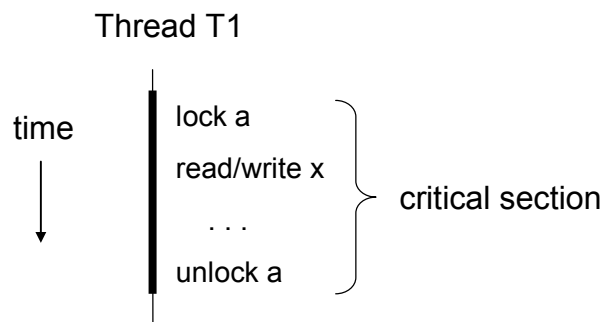


Figure 18. An example of a critical section

A thread is allowed to enter into a critical section when it acquires a lock and exits the critical section after it releases the lock. Which lock to be acquired is specified by the parameter "a" of the lock operation "lock a". Likewise, "unlock a" operation specifies the lock "a" to be released. In the case of multiple hierarchical locks (such as two-phase locks), a critical section has to be defined for each lock. These critical sections may overlap to each other.

3.2 Race Conditions

When a shared variable is concurrently accessed by multiple threads, the final value of the shared variable is not deterministic. Figure 19 shows an example where two threads *T1* and *T2* run concurrently and access a shared variable *x*.

Thread T1	Thread T2
<i>// x is a shared variable</i> integer x = 10;	
<i>// a is a local variable within T1</i> integer a;	<i>// b is a local variable within T2</i> integer b;
a = x;	b = x;
a = a + 1;	b = b - 1;
x = a;	x = b;

Figure 19. An example of two threads *T1* and *T2* run concurrently and access a shared variable *x*

The value of *x* after threads *T1* and *T2* have been executed is not determined to the same unique value. Depending on the interleaving of instruction executions, the value may be 10, 9, or 11. We say in such a situation that a “race condition” or simply “race” occurs. This is not a desirable phenomenon. A technique for avoiding races is concurrency control.

We define a race condition by referring to locking discipline from [Savage97]. “Every shared variable must be protected by some locks.” Such locks are called “consistent locks” for accessing a shared variable. The consistent locks are acquired by any threads before accessing the shared variable.

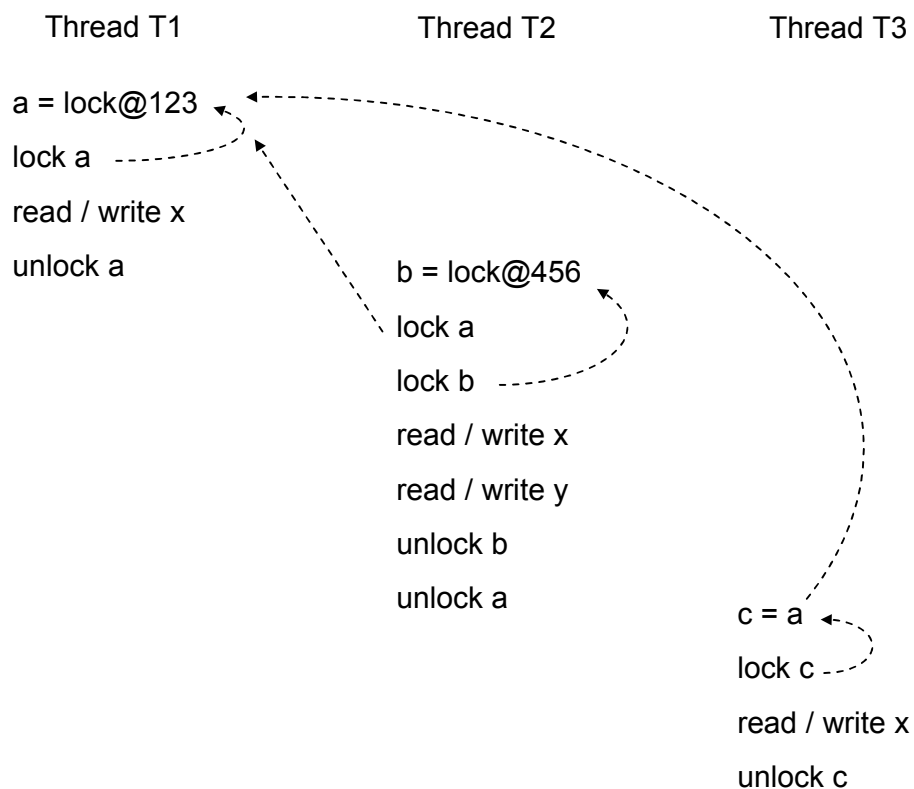


Figure 20. Examples of consistent locks

Figure 20 shows examples of consistent locks:

- Consistent lock for accessing the shared variable x is `lock@123`.
- Consistent lock for accessing the shared variable y is `lock@456`.

In concurrency control using a lock mechanism, a race condition exists when a thread is accessing a shared variable without acquiring consistent locks.

Detecting race conditions is mostly the task of checking consistent locks for accessing shared variables. A race detector called Eraser [Savage97] proposes an efficient algorithm for checking consistent locks in the execution of a concurrent program. In concurrency control using a lock mechanism, it is the responsibility of programmers that a proper lock operation is performed before accessing a shared variable, and that the lock is released after the access to the shared variable has been completed. If this rule is properly followed, the accesses to the shared variable are said to be "well-formed". In other words, an access is said to be well-formed if processes or threads acquire a consistent lock for the shared resource before accessing

it, and then eventually followed by an unlock operation to release the corresponding lock. There are various reasons why access to a shared variable may not be well-formed, for example:

- Programmers forget to write the lock, or they may write an incorrect lock before accessing shared variables.
- Programmers make an incorrect prediction about the execution path, resulting in the lock not being properly set.
- Programmers may intentionally omit a lock for performance reasons when race conditions are acceptable, for example by using a *volatile* variable in Java.

In those cases, the access to shared variables is not well-formed and it might cause a race condition. An example is shown in Figure 26(c) and Figure 26(d) where the "else-statements" in line 15 for thread *T1* access the shared variable *x* without acquiring any locks.

3.3 Total Replay

We define a term called “total replay” [Setiadi10] for testing and debugging concurrent programs. Total replay executes all possible different interleavings and interrupt timings within the scope given by an execution trace that contains an error. Formally, we define as follows:

- Let T be the information from the execution trace of a program execution r that contains an error E . Let S be the set of possible different executions in the scope of trace T ; that is, those executions that start with the same set of input values but with different interleavings and interrupt timings. If R is the set of all possible executions for the program, then S is a part of R . Since the information T obtained from the trace execution holds the conditions under which the error E had occurred in the execution r , then we can guarantee that $r \in S$. Therefore for reproducing the error E , we need to test only S (just the ones within the scope of T), and there is no need to test any other execution.
- When we replay an execution p based on the trace information T , we can guarantee that $p \in S$, but there is no guarantee that p and r are the same. When no trace information is available, then $S = R$. In this case it will become an entire program test. If S is small compared to R , then we are getting closer to a debugging of a particular error. When we can replay all the possible executions in S , then we call it as “total replay”. Most existing replay systems can replay only a

subset of S . Therefore there is no guarantee that the execution r is replayed. In contrast to the total replay, the existing replay systems may be called “selected replay”.

3.4 Dynamic Access

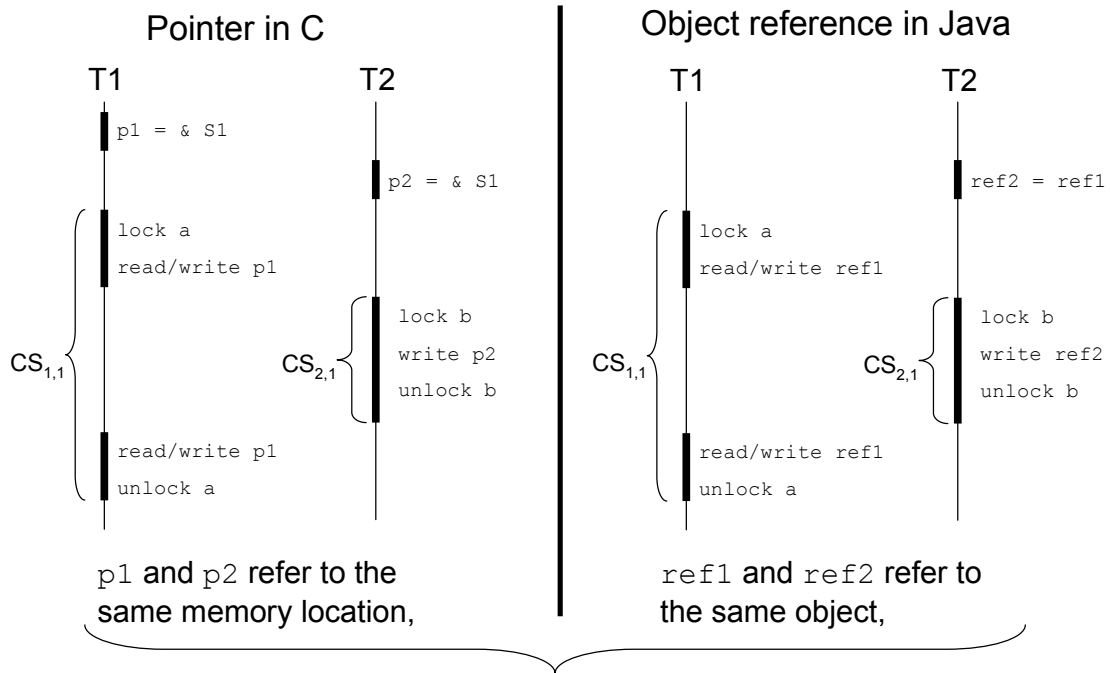
There are some situations where access to a shared variable is not determined at compile time. A shared variable or data access through a reference variable such as a pointer in C, an object reference in Java, or an index of an array, or a file name cannot be determined until an actual execution takes place. The reference variable itself can be detected when it is shared, but trace information is needed to actually determine which particular variable or data is accessed. We further elaborate the situation below.

3.4.1 Reference Variable

A reference variable is a variable that refers to an object in Java programming language. This is similar to a pointer in C programming language. Race conditions might arise when several shared reference variables actually refer to the same data. When checking for race conditions, we should not compare the reference variable's name. Instead we need to compare the actual data referred from an execution trace, which is the memory location of the pointer for C language, or the object for Java language. Again dynamic methods are needed because static methods cannot cope with such a situation. Figure 21 shows an example where two different pointers refer to the same memory location and two different reference variables refer to the same object. On the other hand, even when the same reference variable is shared between threads, the variable or data referred by them may not necessarily be shared. Those situations cannot be coped with static methods.

Pointer in C	Object Reference in Java
<pre>St S1,S2,S3; //st is a structure definition st *p1, *p2; // shared variable // p1 and p2 refer to the same memory</pre>	<pre>A ref1, ref 2; // object reference ref1 = new A(); // ref1 and ref2 refer to the same object</pre>

<pre>// location p1 = & S1; p2 = & S1;</pre>	<pre>ref2 = ref1;</pre>
--	-------------------------



A race condition will occur when critical section CS_{1,1} and CS_{2,1} are interleaved.

Figure 21. Examples of reference variables

We will show how to generate test cases for detecting race conditions caused by reference variables in section **5.3.6 Generating Test Cases to Check Consistent Locking for Access through Reference Variables**. Similar situations also happen for file references, for example `file_name = fopen(c:\¥¥data¥¥...)`. In such a situation, we can treat them in a similar way to reference variables.

3.4.2 Array

When an array element is shared, the value of the index to specify a particular element is not known until the actual execution (see Figure 22). In a static method, to be safe, the entire array should be considered to be shared. Thus the detection precision is lowered. A dynamic method is again required. Similarly in the case of a file name, the actual file may not be known until the actual

execution.

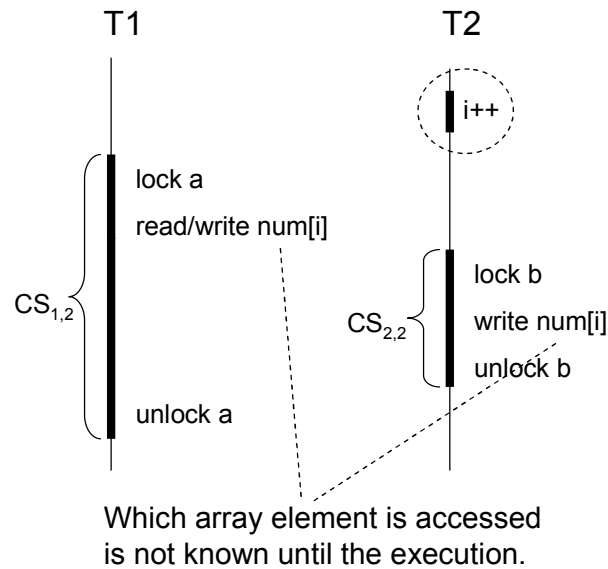


Figure 22. Sharing of an array element

3.5 Conditional Statements/Branches and Loops

Interleavings, timings, and execution paths are all related. A concurrent program can have different execution paths caused by conditional statements/branches, loops, thread interleavings, interrupt timings, and thread communications. Different execution path might cause the program to have a different access-manner to shared variables, i.e. exercising different sequences of lock/unlock. Control flows at conditional statements/branches can be affected by:

- Input values: The test case generation for all branches caused by different input values has been an issue in program testing, for instance [Visser04]. This is not our main concern in this dissertation. In our setting, the input values are known.
- Thread interleavings and interrupt timings: Thread interleavings and interrupt timings may affect the values of shared variables which may in turn affect the result of conditional statements/branches. A simple example is shown in Figure 23.

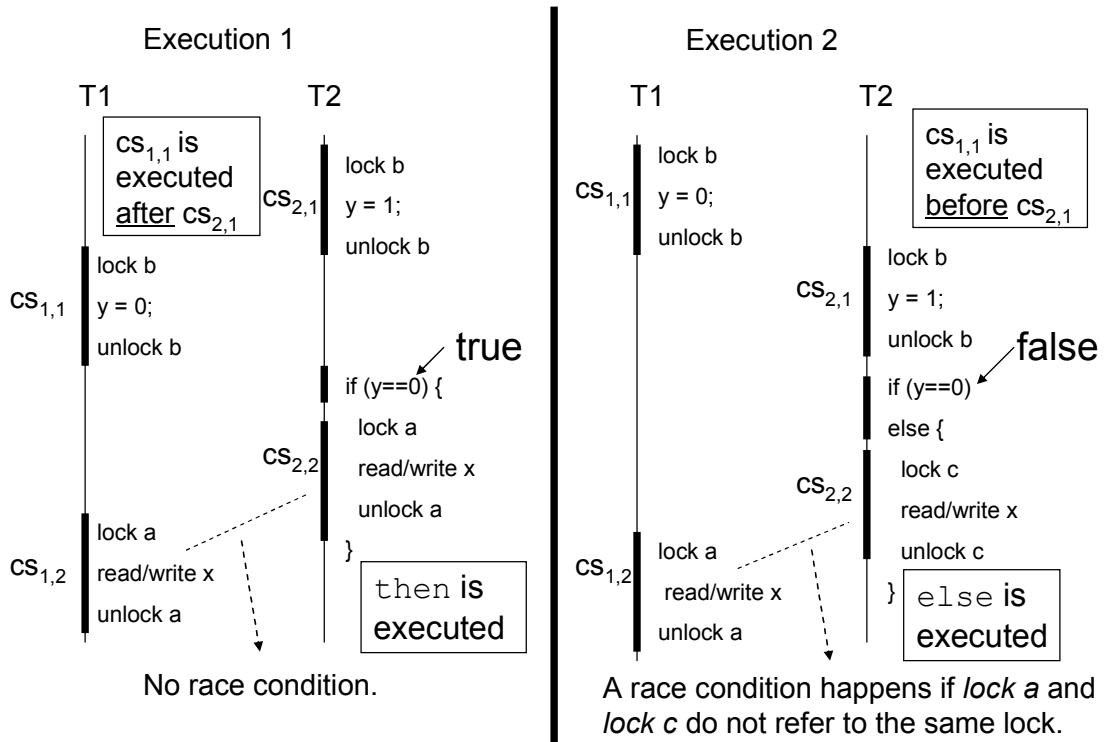


Figure 23. An example of a branch that is affected by interleavings

We use the term “branch outcome” to refer to the truth value within a conditional statement of a branch during a program execution, that is whether *true* or *false*. Let $b_{i,j}$ be the j -th branch from the execution trace of thread T_i . We define “branch-path” for a thread as execution sequence of branch name and its branch outcome. For an execution with N threads, let $nb(T_i)$ be the number of branches for thread T_i . The branch-path for concurrent program is the collection of branch-path from all the threads.

$$\begin{aligned}
 & b_{1,1}[true|false] b_{1,2}[true|false] b_{1,3}[true|false] \dots b_{1,nb(T1)}[true|false] \\
 & b_{2,1}[true|false] b_{2,2}[true|false] b_{2,3}[true|false] \dots b_{2,nb(T2)}[true|false] \\
 & \dots \\
 & b_{N,1}[true|false] b_{N,2}[true|false] b_{N,3}[true|false] \dots b_{N,nb(TN)}[true|false]
 \end{aligned}$$

For example, branch-path for a concurrent program execution with three threads could be $b_{1,1}true, b_{1,2}false, b_{1,3}true, b_{2,1}false, b_{2,2}false, b_{3,1}false, b_{3,2}false$.

Similarly loop-path at the execution of a thread is represented as:

$$l_{1,1}[1^{st} \text{ iteration}] l_{1,1}[2^{nd} \text{ iteration}] l_{1,1}[3^{rd} \text{ iteration}] \dots$$

A “race-equivalent” (will be explained in section **3.10 Race-Equivalent**) group can be defined using these branch-path, loop-path and “access-manner” (will be explained in section **3.9 Access-Manner**). Some program executions might be repeated even without a loop, for example because of *goto* statement. The existing work by [Huang11] can detect such repetitions by identifying some events that are mapped to the same lexical statements in the source code. Those repetitions might not affect the reproduction of race conditions. In such a case, different executions with different number of repetitions could be considered as one race-equivalent group.

3.6 Model for Concurrent Program Execution Traces

A concurrent program execution trace contains a sequence of operations from all the threads. An operation in a thread is modeled as a triplet of:

location : *operation* : *operand*, where

- *location* is *thread_name:file_name:line_of_code*. The thread name or the file name is omitted in some cases for simplicity when there is no ambiguity.
- *operation* is the *read* or *write* operation on a shared variable.
- *operand* is the name of the shared variable.

Figure 24 shows an example of a concurrent program and its flow graph. Let us assume that the following read and write sequence *S* is obtained from an execution trace of the first test:

T1:1 read x, T2:10 write x, T1:1 read y, T1:1: write n, T1:2 read n, T2:11 write y, T1:3 ..., T1:7 read y, T2:12 read x.

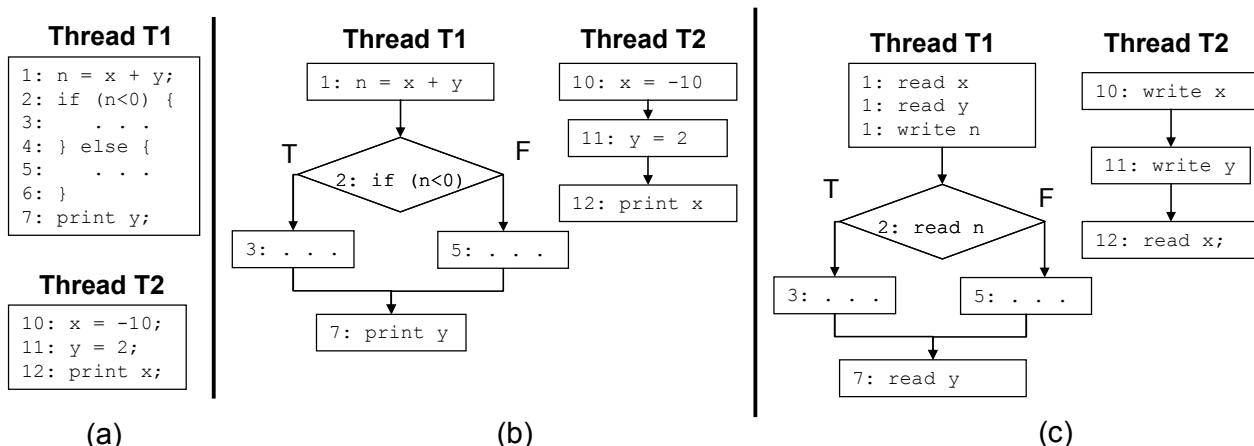


Figure 24. (a) An example of a concurrent program. (b) Flow graph. (c) Flow graph for read and write operations

3.7 Execution Paths

A concurrent program consisting of threads $T_1, T_2, T_3, \dots, T_p$, where p is the number of threads. An execution path is defined for a thread and a concurrent program:

- An execution path P_i of a thread T_i is a sequence of operations executed by the thread T_i . For the execution of the program shown in Figure 26(a) and Figure 26(b), we have:

$P_1 = \{10: \text{if}(\), 11: \text{lock } a, 12: \text{read } x, 13: \text{unlock } a\}$

$P_2 = \{20: \text{lock } a, 21: \text{lock } b, 22: \text{read } y, 23: \text{write } x, 24: \text{unlock } b, 25: \text{unlock } a\}$

- An execution path of a concurrent program is a sequence of operations executed by all threads, taking into account the global order among threads. Figure 26 shows four possible examples of concurrent execution paths for the concurrent program in Figure 25.

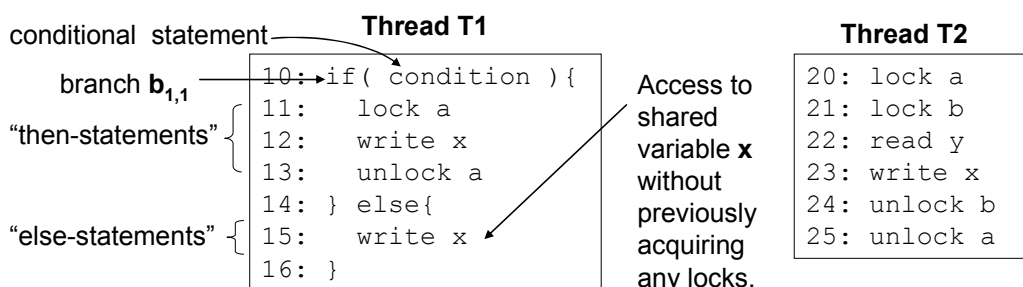


Figure 25. An example of a conditional statement

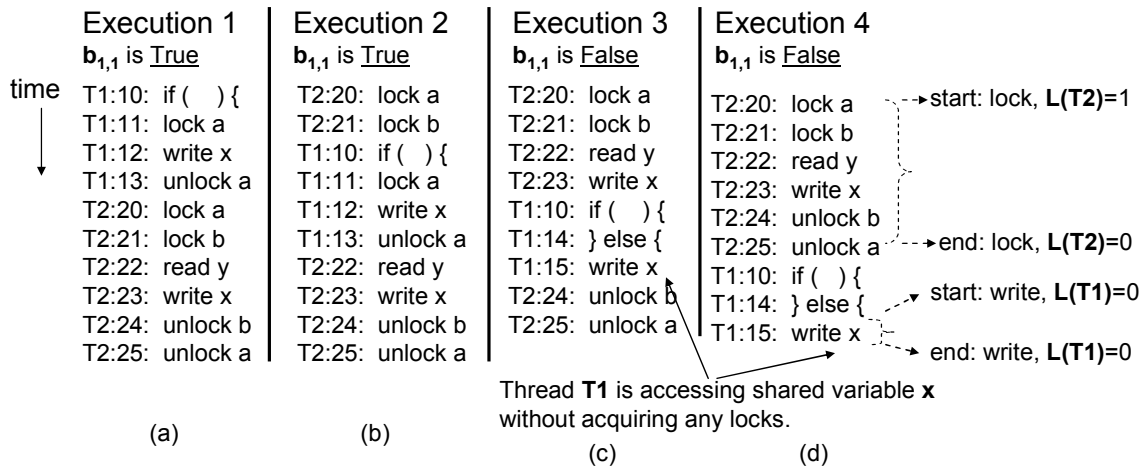


Figure 26. Examples of different concurrent execution paths for program in Figure 25

We define *PATHS* as a set of execution paths P_i 's.

$PATHS = (P_1, P_2, P_3, \dots, P_p)$, where p is the number of threads.

Note that *PATHS* does not take into account the global ordering among threads. For the example in Figure 26(a) and Figure 26(b), we have:

$PATHS = \{ P_1, P_2 \} = \{ \{10: \text{if} (), 11: \text{lock } a, 12: \text{read } x, 13: \text{unlock } a\}, \{20: \text{lock } a, 21: \text{lock } b, 22: \text{read } y, 23: \text{write } x, 24: \text{unlock } b, 25: \text{unlock } a\} \}$

3.8 Interleaving and Branching

We denote by $b_{i,j}$ the j -th branch of thread T_i in the execution path of thread T_i . The truth value of a conditional statement in a branch can be affected by both input values and interleaving because interleaving might affect shared variables, which may in turn affect the conditional statement. Figure 26(a) and Figure 26(b) show some possible concurrent execution paths for the program in Figure 25 when the conditional statement in the branch $b_{1,1}$ is *true*, whereas Figure 26(c) and Figure 26(d) show the concurrent execution paths when the conditional statement is *false*.

Let \rightarrow denotes the “happens-before” relation as follows: If a is an event in process P_i , and b is an event in process P_j , then event $a \rightarrow$ event b if and only if event a happens before event b . In the example of Figure 26, the order of $T1:10$ and $T2:23$ affects the truth value of the branch $b_{1,1}$. The branch is *true* in executions 1 and 2 when $T1:10 \rightarrow T2:23$, and *false* in executions 3 and 4 when $T2:23 \rightarrow T1:10$. We will later explain how to identify operations that affect a branch.

3.9 Access-Manner

Partial order reduction is in general performed with respect to the concerned properties. Reduction is possible only when the concerned properties hold in the reduced state space of the target system. Since our main concerns are race conditions, we can perform partial order reduction with respect to the sequence of lock/unlock and read/write operations to shared variables. We define a notation called “access-manner” to capture the following two properties for detecting race conditions:

- (1) The currently effective locks when performing read/write operations to shared variables.
- (2) The order of their lock operations performed.

The knowledge of access-manner is sufficient to detect race conditions. We use access-manner to check whether the access to a shared variable is performed correctly under a lock. Here it is assumed that the target system adopts the locking scheme to concurrency control.

In order to define an access-manner, we use notation $L(T_i)$ as the number of active locks acquired by thread T_i at a particular time. $L(T_i)$ is 0 at the beginning of the execution of the thread T_i . During an execution of a program, $L(T_i)$ is incremented and decremented by the following rules:

- Incremented by 1 when a thread successfully acquires a lock (i.e. has completed a lock instruction).
- Decrement by 1 when the thread T_i releases the lock which is currently being acquired (i.e. has completed an unlock instruction). $L(T_i)$ is not decremented if a thread is trying to release a lock which is not currently acquired. Hence, $L(T_i)$ cannot be negative.

We define an access-manner as a sequence of operations in which a thread has acquired a lock, has accessed a shared variable, and has released the corresponding lock. Access-manners are defined in the execution path of each thread. There could be several access-manners within the execution path of a thread. An individual access-manner is a sequence of lock/unlock and read-write operations to shared variables within an execution path of a thread. We classify access-manners based on their sequences of

lock/unlock and read/write operations to shared variables as follows:

- A usual access-manner:

An access-manner which starts and ends with the following conditions:

- Start : acquiring a lock, a lock operation which causes $L(Ti)$ to become 1.
- End : releasing the corresponding locks, an unlock operation which causes $L(Ti)$ to become 0.

Figure 27 shows an example of $L(Ti)$ for a usual access-manner using three locks. In between the start and end, the thread is accessing shared variables.

- An unusual access-manner:

An access-manner which starts or ends by the following conditions:

- Start : accessing a shared variable without acquiring any locks, or when executing only an unlock operation without acquiring a lock. This might happen because programmers forget to acquire locks.
- End : when an execution trace has terminated.

Such an unusual access-manner might potentially cause race conditions should another thread be accessing the same shared variable.

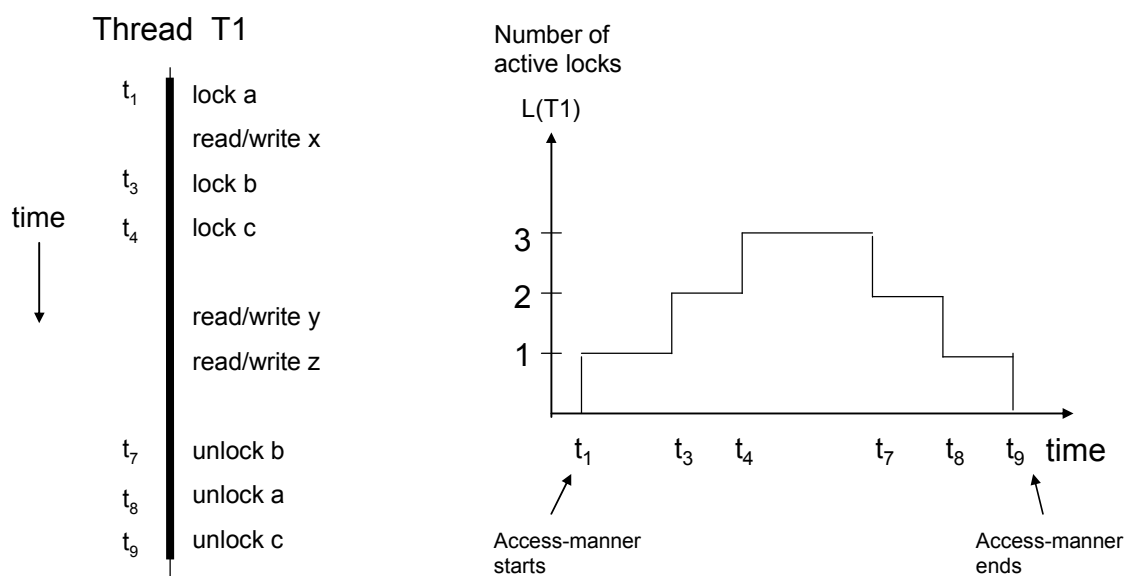


Figure 27. An example of $L(Ti)$ for a usual access-manner with three locks

An individual access-manner must end before another individual access-manner starts; thus they cannot overlap. Throughout this dissertation, an access-manner should be understood to mean a usual access-manner.

Two individual access-manners are the same if they have the same sequence of lock/unlock and read-write operations to shared variables. We define M_i to be a set of access-manners for the execution path of thread T_i , that is a collection of distinct individual access-manners without considering their order. We also define a concurrent set of access-manners $MANNERS = \{M_1, M_2, M_3, \dots, M_N\}$ as a collection of sets of access-manners from all the threads within a concurrent execution path of a concurrent program. When two concurrent execution paths of a concurrent program have the same $MANNERS$, each thread will have the same set of access-manners.

When two different concurrent execution paths of a concurrent program have the same $PATHS$, each thread in the two execution paths will exercise exactly the same sequence of lock/unlock and read-write operations to shared variables, hence they will also have the same set of access-manners. Therefore, two concurrent execution paths with the same $PATHS$ will certainly have the same $MANNERS$. The concurrent execution path in Figure 26(a) and the execution path in Figure 26(b) have the same $PATHS$, hence they will also have the same $MANNERS$:

$$\begin{aligned}
 M_1 &= \{11:\text{lock } a, 12:\text{write } x, 13:\text{unlock } a\} \\
 M_2 &= \{20:\text{lock } a, 21:\text{lock } b, 22:\text{read } y, 23:\text{write } x, 24:\text{unlock } b, 25:\text{unlock } a\} \\
 MANNERS &= \{ \{11:\text{lock } a, 12:\text{write } x, 13:\text{unlock } a\}, \{20:\text{lock } a, 21:\text{lock } b, 22:\text{read } y, 23:\text{write } x, 24:\text{unlock } b, 25:\text{unlock } a\} \}
 \end{aligned}$$

With the knowledge of access-manners and the accumulation previous execution traces, we can accomplish two things:

- (1) If the current sequence of lock/unlock and read/write operations is found to be the same as the previous logged one, then we do not need to repeat the detection of race conditions because the same situation has already been tested. This is true for any execution paths including loops.
- (2) In exploring execution paths, any execution paths having the same sequence of lock/unlock and read/write operations are grouped into the same group. They constitute a “race-equivalent” group. We will use access-manner to define the equivalency in terms of race conditions among different executions of a concurrent program (will be discussed in section **3.10 Race-Equivalent**). However, note that belonging to the same race-equivalent group does not necessarily imply that the future computation will be the same.

As explained in section **3.9 Access-Manner**, two concurrent execution paths with the same *PATHS* will certainly have the same *MANNERS*. Therefore, two concurrent execution paths of a concurrent program that have the same *PATHS* will certainly be race-equivalent. For examples, the execution path in Figure 26(a) and the concurrent execution path in Figure 26(b) have the same *PATHS*, so they are race-equivalent. We can see that lock *a* is a consistent lock for accessing shared variable *x* in both concurrent execution paths. Different race-equivalent groups can be created by taking a different concurrent execution path in which at least one thread changes its individual access-manner. A branch might lead to a different concurrent execution path which, in turn, can produce different individual access-manners that can affect consistent locking. As shown in the concurrent execution paths in Figure 26(c) and Figure 26(d), there is a race condition because there is no consistent lock for access to shared variable *x* in thread *T1*:

$$M1 = \{(15:\text{write } x)\}$$

$$M2 = \{(20:\text{lock } a, 21:\text{lock } b, 22:\text{read } y, 23:\text{write } x, 24:\text{unlock } b, 25:\text{unlock } a)\}$$

$$MANNERS = \{ \{(15:\text{write } x)\}, \{(20:\text{lock } a, 21:\text{lock } b, 22:\text{read } y, 23:\text{write } x, 24:\text{unlock } b, 25:\text{unlock } a)\} \}$$

To detect this race condition, we need only check the concurrent execution path in Figure 26(c) or the one in Figure 26(d) because they are race-equivalent. The same inconsistent locking can be detected.

When a branch changes the execution path of a thread, it might not necessarily produce different consistent locking. In this situation, the same thread in the two concurrent execution paths might not exercise exactly the same sequence of lock/unlock and read-write operations to shared variables, but they will still have the same *MANNERS*, and so we can also classify them as race-equivalent. This is particularly useful in the case of loops because we do not need to test all the iterations. It is sufficient to test only a partial execution trace from several iterations for checking race conditions because the execution of loop iterations can have the same access-manners.

In Figure 29, thread *T1* in execution 1 and execution 2 has different access-manners, hence concurrent execution paths 1 and 2 are not race-equivalent. When there is an active lock that was acquired outside the loop, then the first iteration will have different access-manners from those in the second iteration because they start

from different active locks, as shown in concurrent execution paths 1 and 2 in Figure 29. On the other hand, concurrent execution paths 2 and 3 in Figure 29 are race-equivalent because each thread in the two executions has the same *MANNERS*:

$$M1 = \{ (1:\text{lock } a, 3:\text{write } x, 4:\text{unlock } a), (3:\text{write } x), (4:\text{unlock } a) \}$$

$$M2 = \{ (20:\text{lock } a, 21:\text{read } x, 22:\text{unlock } a) \}$$

$$MANNERS = \{ \{ (1:\text{lock } a, 3:\text{write } x, 4:\text{unlock } a), (3:\text{write } x), (4:\text{unlock } a) \}, \{ (20:\text{lock } a, 21:\text{read } x, 22:\text{unlock } a) \} \}$$

The second iteration for the loop accesses the shared variable x without acquiring any lock, a fact that can be detected in either concurrent execution path 2 or 3. When there is no active lock at the end of a loop, the rest of the iterations will have the same set of access-manners. The rest of these iterations are called “equivalent iterations” in terms of consistent locking because they have the same set of access-manners.

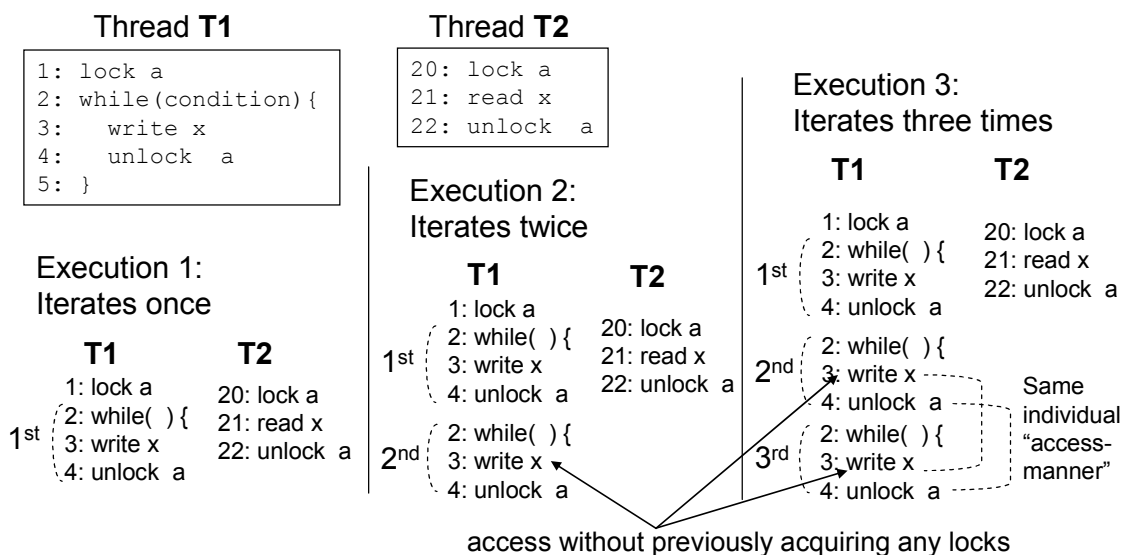


Figure 29. An example of set of access-manners for a loop

A different read/write sequence that affects the values of shared variables is called an “execution-variant”. Section 4.2 **Approach** explains how to derive execution-variant effectively using an existing method. Further to the discussion above, the problem for detecting a race condition can be stated as follows:

Given a concurrent program that has an execution-variant V_{error} containing an error in its concurrent set of access-manners $MANNERS_{error}$, find the V_{error} , or another execution-variant V , which has the same concurrent set of access-manners as

$MANNERS_{error}$. Since each thread in V and V_{error} will have the same set of access-manners, then the same inconsistent locking and improper lock-unlock sequences in V_{error} will also be detected in V .

3.11 Concurrent-Pair of Access-Manners

We define the term “concurrent-pair” of access-manners for the purpose of checking race conditions in a concurrent execution. Two access-manners $M1$ and $M2$ are a concurrent-pair, denoted by $pair(M1, M2)$, if there exists a different interleaving that can change the order of occurrence between one of the operations from $M1$ and one of the operations from $M2$. Let’s assume an access-manner $M1$ in a thread $T1$, and an access-manner $M2$ in a thread $T2$. The access-manners $M1$ and $M2$ are a concurrent-pair of access-manners if the following three conditions hold:

- Different threads: The threads $T1$ and $T2$ are different.
- Not blocked by a thread creation: The thread $T1$ is not created by the thread $T2$ after the access-manner $M2$ ends, or the thread $T2$ is not created by the thread $T1$ after the access-manner $M1$ ends.
- Not blocked by a synchronization message: The thread $T1$ does not wait for a message from the thread $T2$ before the access-manner $M1$ starts, or the thread $T2$ does not wait for a message from the thread $T1$ before the access-manner $M2$ starts.

Figure 30 is an example of an execution for the source code in Figure 32. It shows some concurrent-pairs of access-manners. The number of concurrent-pairs of access-manners depends on the number of access-manners and how they are distributed among threads.

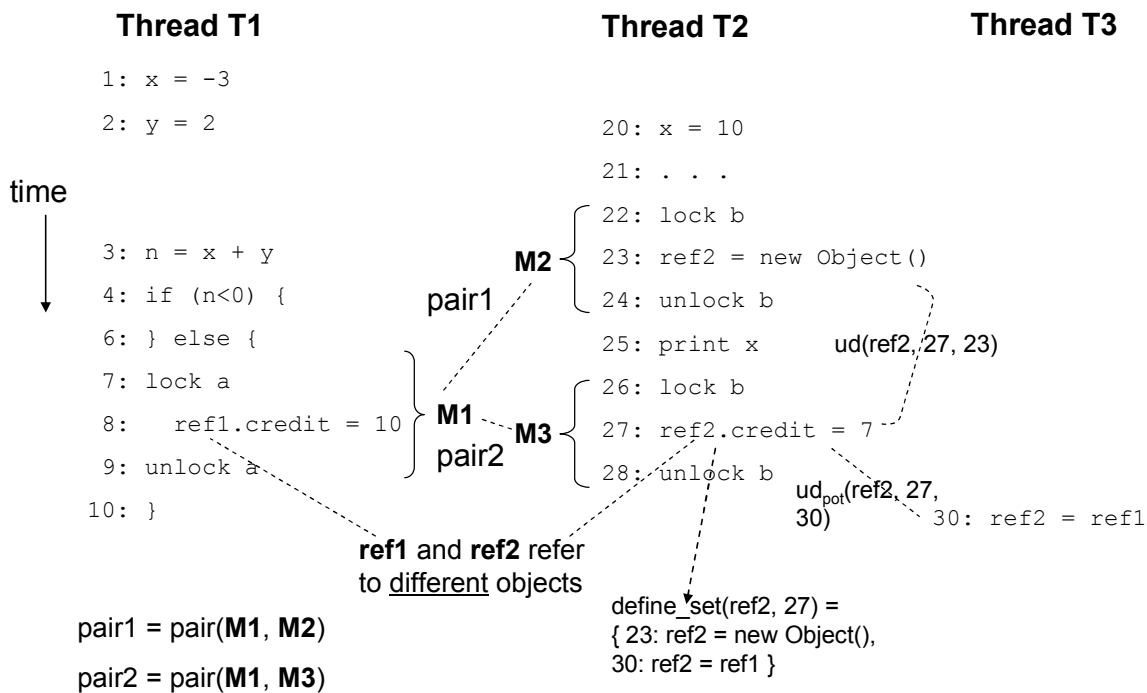


Figure 30. Examples of some concurrent-pairs of access-manners in an execution trace

We have to check race conditions for each concurrent-pair of access-manners. When a *use* operation has more than one member in its *define_set*, its value might be affected by different interleavings. For each concurrent-pair of access-manners, we have to check race conditions for all the combinations of the *define_set* of the lock variables and reference variables. The occurrence of race conditions might be affected in the event that any lock variables refer to different lock objects or any reference variables refer to different objects. A race condition can occur in Figure 30 between the concurrent-pair of access-manners *M1* and *M3*. This happens when the reference variables *ref1* and *ref2* refer to the same object, and the lock variables *a* and *b* refer to different lock objects.

There is no need to check different interleavings between a concurrent-pair of access-manners that satisfies the following two conditions, because the consistent locking will be the same:

- The concurrent-pair of access-manners has been checked for race conditions in the previous test execution.
- Different interleavings will not change the value of lock variables and reference variables.

In this way, we can reduce the number of test cases. On the contrary, if any different interleavings might affect the lock variables or reference variables, then they have to be tested because the consistent locking might be affected accordingly.

3.12 No-Race

We define a term named “no-race” for a concurrent-pair of access-manners. No-race means two access-manners can be interleaved without race conditions. This is the essential definition for “no-race”. When the concurrency control of the target system is based on a lock mechanism, we say that a concurrent-pair of access-manners $pair(M1, M2)$ is no-race when one of the following conditions is satisfied:

- No common shared variables.
No common shared variables between the two access-manners.
- Protected by consistent locks.
Every read/write operation to a shared variable is always protected by consistent locks (see about consistent locks in section 3.2 Race Conditions).
Algorithm 1 explains in more details on how to decide whether a concurrent-pair of access-manners is no-race.

Algorithm 1. Deciding whether a concurrent-pair of access-manners is no race

Definitions:

- $M1$: an access-manner
- $vars(Mi)$: set of variables within an access-manner Mi .
- $shared(M1, M2)$: set of variables which are shared between access-manner $M1$ and $M2$.
- $write(Mi)$: set of shared variables that are written within an access-manner Mi .
- $read(Mi)$: set of shared variables that are read within an access-manner Mi .
- $activeLocks(x, t)$: set of active locks when a variable x is read or written at a particular time t .

- $consistentLocks(x)$: set of consistent locks for accessing a variable x .

Input: a concurrent-pair of access-manners $pair(M1, M2)$.

Output: deciding whether the input $pair(M1, M2)$ is no-race.

Step 1. Check whether there are common shared variables between the two access-manners

If $((vars(M1) \cap vars(M2)) = \emptyset)$ {

 Report as no-race and terminate this algorithm.

}

Step 2. Check whether every access to a shared variable is protected by consistent locks.

Step 2.1 For every shared variable x in $shared(M1, M2)$

 Assuming the shared variable x is accessed at time : $t_1, t_2, t_3, \dots, t_N$

$consistentLock(x) = activeLocks(x, t_1) \cap activeLocks(x, t_2) \cap activeLocks(x, t_3)$

$\cap \dots \cap activeLocks(x, t_N)$

Step 2.2 For every shared variable x in $shared(M1, M2)$

If $consistentLock(x) \neq \emptyset$

 Report as no-race and terminate this algorithm.

Step 3.

 Do not report as no-race.

Once a concurrent-pair of access-manners is found to be no-race, and there are no other assignments to reference variables or lock variables, except during the initialization, then no further check is needed for other interleavings among

operations inside the two access-manners. Figure 31 shows an example of a no-race where access to shared variable x is protected by the consistent lock a .

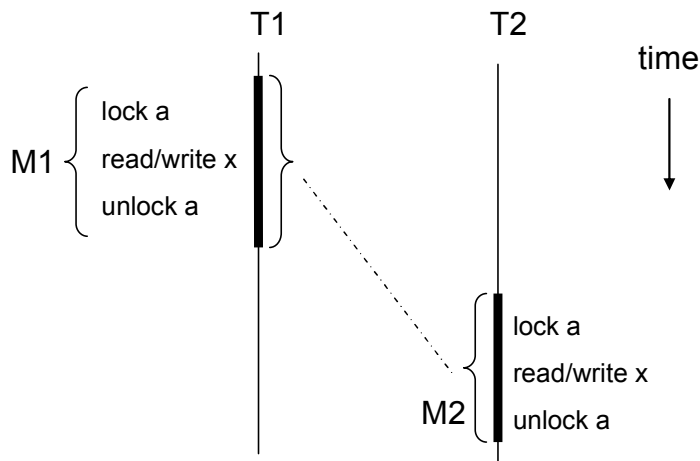


Figure 31. An example of concurrent-pair of access-manners which is no-race

We note that no-race guarantees only whether accesses to shared variables are protected by consistent locks. The order of them, of course, affects the values of the shared variables. In the example in Figure 31, the final value of shared variable x depends on whether access-manner $M1$ is executed before or after the execution of access-manner $M2$.

3.13 Use-Define

A “use-define” is a relation consisting of a usage “use” of a variable and the definition “define” of the variable.

- A *use* means a read operation on a variable.
- A *define* means a write operation of some value to a variable.

A use-define is a triplet:

$$ud(var, use_location, define_location) \quad (1)$$

There must be no other write operations to the variable in between the *use* and *define* operations. The use-define was initially defined for sequential programs. R. Caballero, C. Hermanns, and H. Kuchen [Caballero07] utilize use-define for measuring test coverage but that definition does not apply to concurrent programs. In this

dissertation, we call this use-define for sequential programs the “conventional use-define”. Yang, A.L. Souter, and L.L. Pollock [Yang98] extend the definition of use-define to the usage and definition of shared variables in concurrent programs. Below are the differences:

- Sequential program: the *use* and *define* operations are located in the same thread.
- Concurrent program: the *define* operation might be located in a different thread to the *use* operation. The interleaving in a particular execution decides which thread actually defines the value.

A set of use-defines is obtained from an execution trace. We use the set of use-defines to find operations which affect conditional statements in branches or reference variables in access-manners. We also define “potential use-define” for the same *use* of the variable when there could be another interleaving which satisfies the following two conditions:

1. There is another *define* operation which occurs before the *use* operation. We assume the *use* operation can be executed after the *define* operation, i.e. not blocked by a thread creation or a wait-notify message.
2. There is no other *define* operation to the variable between the *define* operation in condition 1 and the *use* operation.

A potential use-define is denoted by:

$$ud_{pot}(var, use_location, define_location) \quad (2)$$

Figure 33 is one of the possible execution traces for the source code in Figure 32. Its use-defines and potential use-defines are as follows:

- Use-defines: $ud(x, 3, 1)$, $ud(y, 3, 2)$, $ud(x, 25, 20)$, $ud(n, 4, 3)$, $ud(ref2, 27, 23)$
- Potential use-defines: $ud_{pot}(x, 3, 20)$, $ud_{pot}(x, 25, 1)$, $ud_{pot}(ref2, 27, 30)$

Let $setUD(V)$ be the set of use-defines in an execution-variant V . An execution-variant V satisfies a use-define $ud(var, use_location, define_location)$ if the use-define is included in the $setUD(V)$. In other words, it satisfies the following condition:

$$ud(var, use_location, define_location) \subseteq setUD(V) \quad (3)$$

Let $define_set(var, use_location)$ be the set of possible *define* operations for the variable var at the location $use_location$. Below are some examples of *define sets* in Figure 33:

- $define_set(x, 3) = \{ 1: x = -3, 20: x = 10 \}$
- $define_set(y, 3) = \{ 2: y = 2 \}$
- $define_set(n, 4) = \{ 3: n = x + y \}$
- $define_set(x, 22) = \{ 1: x = -3, 20: x = 10 \}$

If a *define set* contains only one *define* operation from the same thread, then we can guarantee that its values will not be affected by different interleavings.

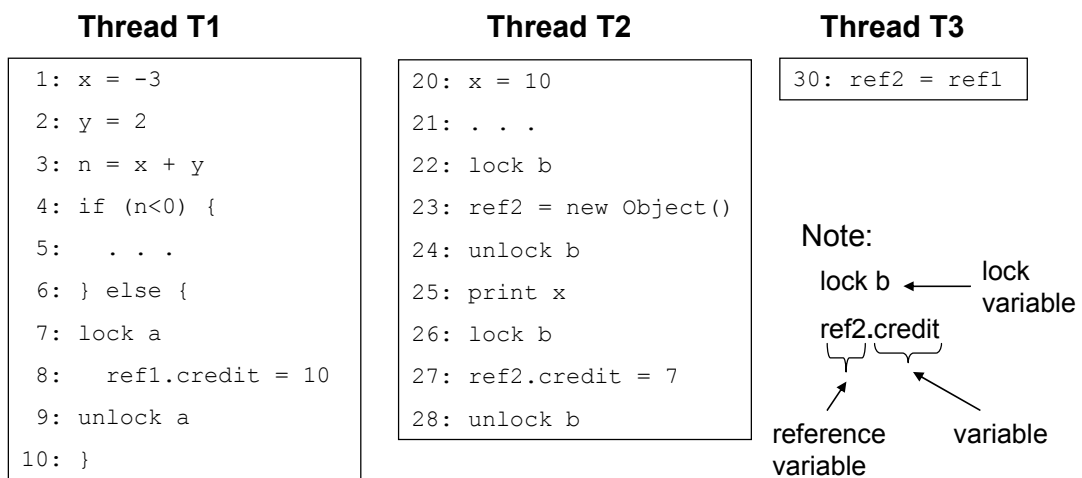


Figure 32. An example of a concurrent program

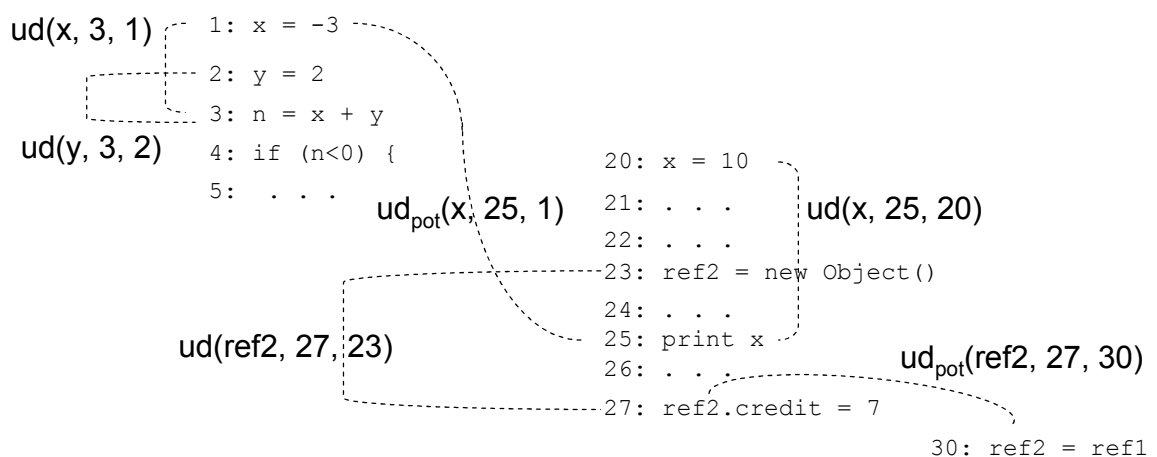


Figure 33. An example of execution traces and some of its use-defines

Chapter 4. Setting for the Proposed Method

4.1 Requirements

Testing and debugging of concurrent programs seem to be a matured area at least from a theoretical point of view. A large body of research exists for methodologies of testing and debugging of concurrent programs. In particular, many methods are proposed to reduce the necessary test cases of concurrent programs. However their applications to real systems are still limited to special cases. Practices of testing and debugging remain in many cases ad-hoc and rudimentary.

The purpose of this research is to develop a practical debugging methodology for normal concurrent programs. The presumed debugging situation is, as suggested in Introduction, where all logical errors have been resolved or where target systems have already been deployed and in service, and then errors have been found. Those errors are most likely caused by a different timing. It is well known that debugging such errors is extremely difficult. We set the requirements for the methodology to solve this situation as follows:

- 1) Exhaustiveness: We mandate path coverage and exhaustive path exploration. This is necessary to assure the reliability of this debugging.
- 2) Minimum interleavings: The number of necessary test cases is primarily determined by interleavings. This number should be kept minimum.
- 3) Practicality: For real systems, applications must be efficient. To meet this requirement, debugging should be dynamic. As discussed so far, static debugging methods suffer false positive error detections and do not fit well to debugging situations where some errors have been found and that those errors are the target of detecting their true causes.
- 4) Effectiveness: The errors found are presumably extremely difficult ones in finding their true causes. In order to find their true causes, tracing is one of the most effective approaches. Our method assumes the effective use of tracing.

- 5) Efficient tracing: Tracing is an effective debugging approach, but it incurs a large overhead. There must exist some means to mitigate this overhead. One of the useful and old techniques is a checkpoint/restart scheme that restarts the execution of the program from a recorded checkpoint instead of the program's initial start point so that the repeated executions of the same initial portion of the program are avoided. The usefulness of this checkpoint/restart scheme has also been recognized for debugging concurrent programs. The "prefix-based testing" [Hwang95] is one of such approaches.
- 6) Efficient race detection and deadlock detection: The amount of work for race detection and deadlock detection processing should be minimized. Previous race detection results should be utilized whenever possible.

These requirements cover not only the theoretical foundations but also practical considerations.

4.2 Approach

There exist several research results which the requirements stated in section 4.1 Requirements can be developed based on or extended from. One of the older ones is Reachability Testing Method of a concurrent program, which uses a partial order reduction technique and tracing [Hwang95]. In a similar line of development, we can find research results such as Dynamic Partial-Order Reduction by Flanagan, C. and Godefroid, P. which uses backtracking to identify program execution points where alternative paths in the state space need to be explored [Flanagan05], and Algorithmic Debugging by Caballero, R., et al. which is discussed for sequential programs but is claimed to be extended to parallel programs easily [Caballero07].

This section explains an existing method for generating test cases for concurrent programs using the reachability testing method [Hwang95] [Carver04] [Lei06]. This is a dynamic method that uses partial order reduction for reducing test cases. The reachability testing method in [Hwang95] performs an efficient exploration of different sequences of read/write operations which affect values of shared variables as test cases. Only read and write operations are modeled. Using the idea behind the partial order reduction, it groups and ignores different interleavings that do not affect

any values of shared variables. It uses a dependency relation between two read/write operations to determine whether the order of those operations affect the value of a shared variable. Two operations are dependent if the following conditions are satisfied:

- The two operations are concurrent, i.e. from different threads.
- The two operations are accessing the same variable.
- One of the two operations is a write operation to the variable.

Any two operations that do not satisfy the conditions above are called as independent. Figure 34 shows the comparison between the exhaustive method and the reachability testing method. It gives the basic idea for reducing the number of different interleavings for independent operations.

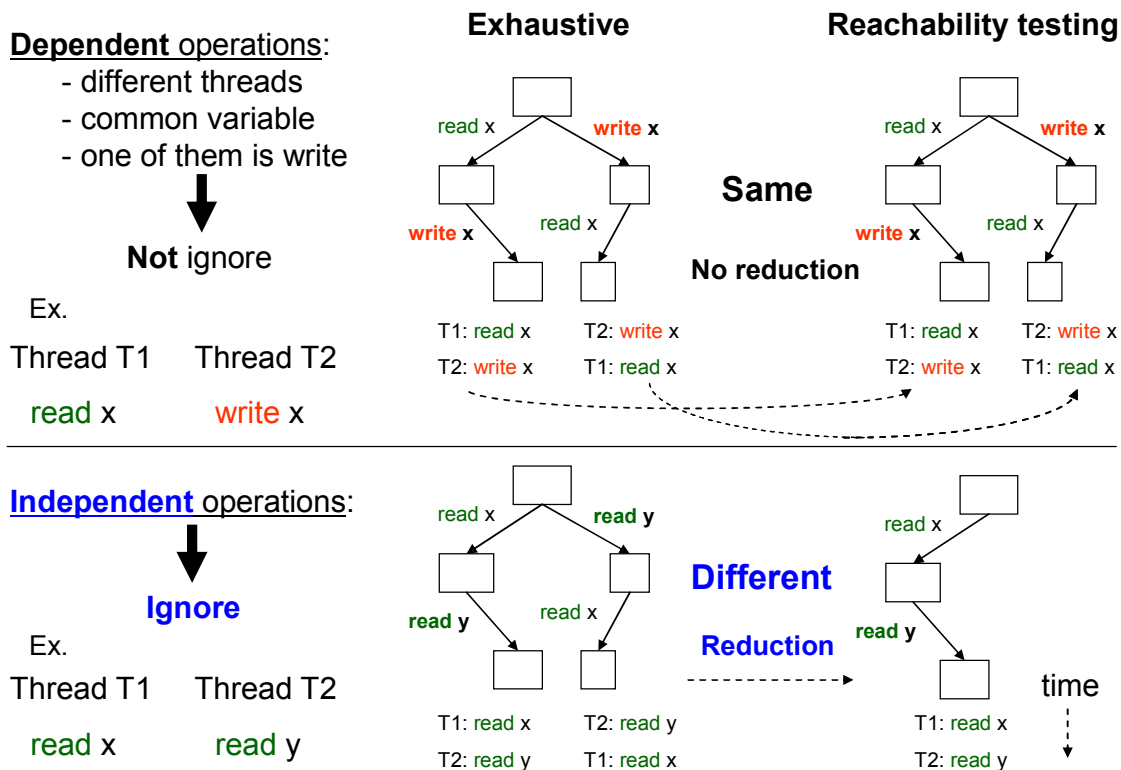


Figure 34. Comparison between the exhaustive method and reachability testing method

This reachability testing uses the previous execution trace to derive different read/write sequences that affect values of shared variables. Assume that S is a read/write sequence from an execution of a concurrent program. The concept of reachability testing is defined as follows:

1. Use S to derive other read/write sequences, called “execution-variants”, that

- produce different values of shared variables.
2. Perform deterministic testing based on the result from step 1 using tracing.
 3. For each new execution-variant from step 2, repeat step 1 and 2 until no more execution-variants are found.

This general approach is common in many methods that use a partial order reduction and trace. Test cases are generated systematically using a variant graph where an execution variant is a different execution whose state is different from the previous ones. The reachability testing method performs an efficient exploration of execution-variants by grouping and ignoring different interleavings that do not affect values of shared variables, using the idea of partial order reduction. Test cases are generated systematically using a variant graph. A variant graph derives different read/write sequences from the previous execution trace. A different read/write sequence that affects the values of shared variables is called an execution-variant. Execution-variants are used as test cases in reachability testing. G. H. Hwang, K. C. Tai, and T. L. Huang introduced an algorithm to create a variant graph from an execution trace of a concurrent program [Hwang95]. The general steps for creating a variant graph are as follows:

- 1) Starting from the initial node, set all the indices and the versions to zero.
- 2) Create a child node for read and write operation by changing one index from a thread. Increase the version if the operation is write.
- 3) If the version for each operation do not conform with the initial trace, then label the child node as “variant” (V). Label the parent node of the variant node as “check point” (CP).
- 4) Continue until the indices in all the threads are explored. Do not explore/create a child node for the node which is labeled as a variant (V), this will be done later by executing it as a test case.

Algorithm 2 shows how to create a variant graph from an execution trace of a concurrent program.

Algorithm 2. Creating a variant graph

Definitions:

- $S(j)$ is a read/write sequence for thread T_j .

- $S(j, i)$ is the i -th operation in the sequence of thread T_j .

Each node N in the execution-variant graph contains the following two vectors:

- index vector: $(id_1, id_2, \dots, id_p)$, where p is the number of threads and id_j indicates the i -th operations in a thread T_j when node N is generated. The index vector is initialized to zero and increased by one after each read or write operation in the thread T_j .

- version vector: $(ver_1, ver_2, \dots, ver_q)$, where q is the number of shared variables and ver_k is the version number of variable V_k when node N is generated. The version for variable V_k is initialized to zero and increased by one after each write operation to the variable V_k .

Input: read/write sequence.

Output: variant graph.

Step 1. Initialize the variant graph.

Create an initial node and label it as “unmarked”. Set its index vector to $(0, 0, \dots, 0)$ and version vector to $(0, 0, \dots, 0)$.

Step 2. Derive different read/write sequences.

2.1 Select an “unmarked” node, say N .

For each j , $1 \leq j \leq p$, where p is the number of threads

If $id_j <$ the length of $S(j)$,

Then construct a child node N' of N according to steps 2.2 – 2.5.

2.2 Set the index vector of N' to that of N except that the j -th element is $id_j + 1$.

2.3 Set the version vector of N' to that of N .

2.4 Let var_k be a shared variable in the operation $S(j, id_j + 1)$ and ver_k is the version number of variable var_k in $S(j, id_j + 1)$.

2.5 **If** $S(j, id_j + 1)$ is a write operation to shared variable var_k ,

Then increase the ver_k of N' by 1.

Step 3. Identify an execution-variant.

3.1 Let ver_k' be the k -th element of the version vector of N' .

3.2 **If** $ver_k' \neq ver_k$

Then label N' as “marked” and execution-variant (V).

Else If the variant graph already contains a node with the same index and version vector as N' .

Then label N' as “marked”

Else label N' as “unmarked”

Step 4. Repeat **step 2** until all nodes in it are labeled “marked”. Do not create child

nodes for the nodes which are labeled as execution-variant (V), as this will be done later by executing them as test cases.

Note that we first need to identify all shared variables from source code before creating a variant graph. If we do not consider all shared variables, then later we might need to reconstruct the variant graph when other variables are found to be shared. It is not enough just to identify shared variables from the execution trace because maybe not all shared variables can be detected from a particular execution trace. Unfortunately, it is not always possible to identify precisely all shared variables from source code: in the case that threads are dynamically created according to input data, for example, it is necessary to consider all potential shared variables. If some variables are not actually shared, they will lead to redundant nodes in a variant graph, but they will not produce redundancy in test cases because they will not lead to any new execution-variants.

Figure 35 is an example of a variant graph constructed using Algorithm 2 for the execution trace in section **3.6 Model for Concurrent Program Execution Traces**. Lined boxes in a variant graph represent possible read/write sequences where they access the same values of the shared variables as in the previous execution. A dotted box in a variant graph represents an execution-variant (V) in which some read or write operations access values of shared variables different from the previous execution as a result of a different interleaving. There are seven execution-variants $V1$, $V2$, $V3$, $V4$, $V5$, $V6$, and $V7$ in Figure 35. Execution variants are identified as candidate test cases in the reachability testing method. Test cases can be started from the corresponding check point (cp) to avoid executing unnecessary interleavings.

Figure 35 shows two equivalent read/write sequences surrounded by dotted lines. They are equivalent in terms of the read/write sequence, in the sense that every operation will read or write the same versions of shared variables. The reachability testing method [Hwang95] performs reduction by considering only one of them as an execution-variant

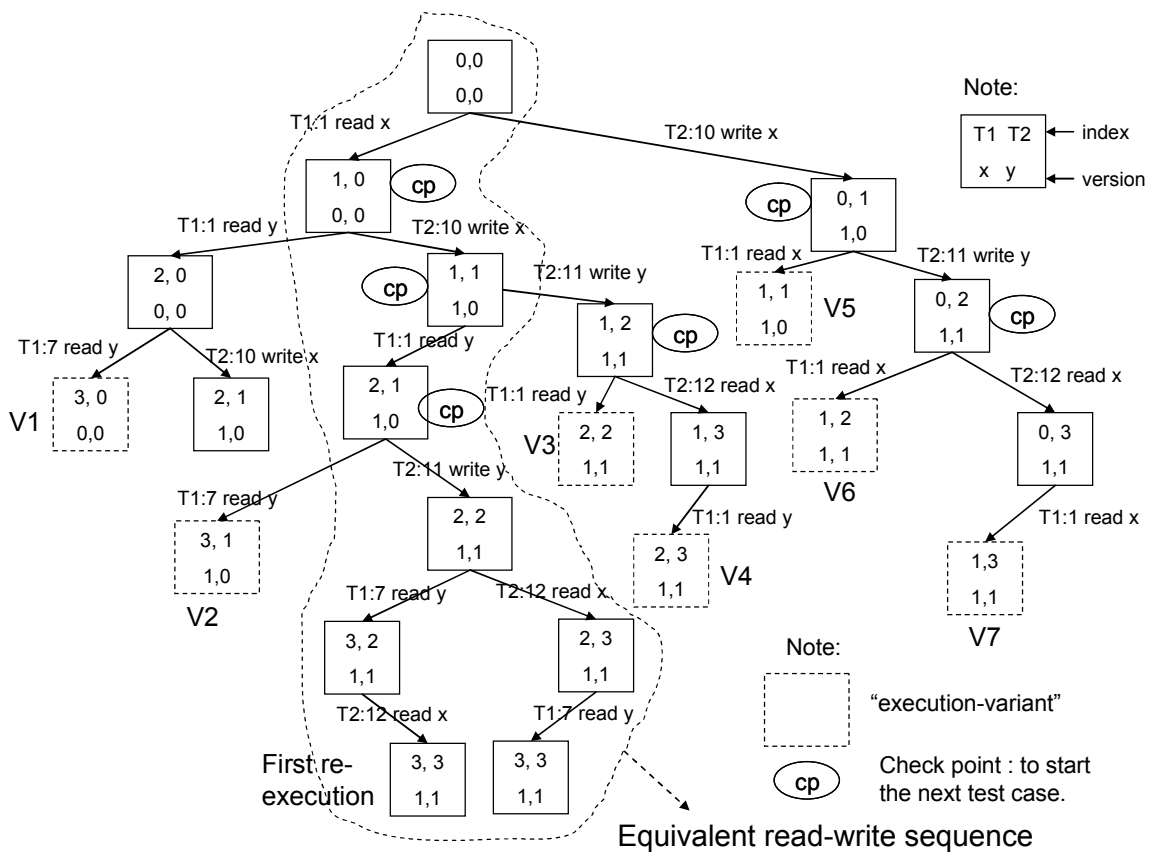


Figure 35. An Example of a variant graph from an execution trace

The reachability testing is efficient for exploring different possible value of shared variables caused by different interleavings, but has some redundancies for the purpose of checking race conditions:

- Generate test cases with the same branch coverage
 - The current development of variant graphs are not necessarily complete because they focus mostly on read and write operations. Branch operations and interrupts are not well considered, so it often produces test cases that result the same path coverage which do not change the access-manner to shared variables.
- Generate infeasible test cases
 - The current development does not consider lock sequences. Enforcing them in deterministic replay environment might cause suspension, which will not be allowed in the real situation.
- Generate infinite test cases
 - In the case of infinite loop, it might generate infinite test cases because the

current model considers the index of operation. A trivial solution would be just to limit the length of the execution trace. If the iteration of the loop does not change the access-manner to shared variables it is not necessary to test all the loop iterations for checking race conditions.

Our method extends the idea of partial order reduction to a dynamic testing or debugging by ignoring the order of irrelevant interleavings that do not affect branch outcome. Furthermore, we improve the reduction precision and increase the number of reductions by exploiting the trace information. The trace information can give more precise information concerning branching. The existing methods can identify all interleavings which may affect shared variables whereas our method identifies only those interleavings which affect branch outcomes. Not all interleavings which may affect shared variables necessarily affect branch outcomes, thus redundant interleavings are included in these interleavings. Those redundant ones are further reduced in our method.

In this research, we exploit several new ideas to further improve the debugging effectiveness and efficiency. First, we further reduce the number of interleavings considering the fact that not all shared variables affect the truth values of branches. We improve the reduction precision and increase the number of reductions by exploiting the trace information. The trace information can give more precise information concerning branching. Many of existing methods identify all interleavings that may affect shared variables whereas our method identifies only those interleavings which affect branch outcomes. Not all interleavings which may affect shared variables necessarily affect branch outcomes, thus redundant interleavings are included in these interleavings. Those redundant ones are further reduced in our method.

We can also reduce the amount of work required to detect race conditions and deadlock. Assuming that the target concurrent program adopts a locking mechanism, it is known that the knowledge of the order of the currently effective locking is sufficient to detect race conditions and deadlocks involving those variables that are under the locking mechanism. We define the order of the currently effective locking as the lock structure. Then we can say that any execution sequence having the same lock structure belongs to the same equivalent group. If

that structure has already been tested against race conditions and deadlocks, it is not necessary to repeat those tests again. Furthermore, any part of executions that maintain the same lock structure can be reduced to the same one.

Regarding the interrupt timing, it is not fully considered in the model of the variant graph proposed by reachability testing [Hwang95]. If we assume interrupt can happen anytime and we create a new node for all the interrupt timings, then the graph might become unlimited. In order to support checking interrupt timings, we propose to change interrupt as a thread as described in section 6, so that the existing model for variant graph can be still be applied.

Chapter 5. Proposed Method

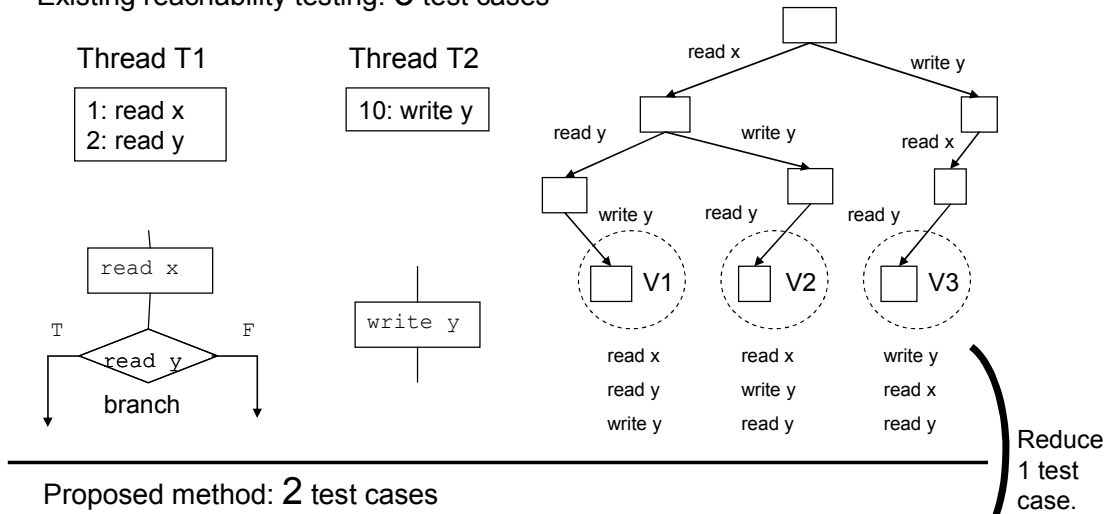
This chapter proposes new methods to effectively reproduce race conditions by reducing the followings:

- Number of test cases:
 - Avoid testing redundant interleavings (section 5.1)
 - Avoid testing infeasible interleavings (section 5.2)
- Memory required for generating test cases (section 5.3).
- Effort involved in checking race conditions (section 5.4).

5.1 Avoid Testing Redundant Interleavings

Figure 36 shows the idea of proposed method for reducing test cases. It avoids redundant interleavings by grouping.

Existing reachability testing: 3 test cases



Proposed method: 2 test cases

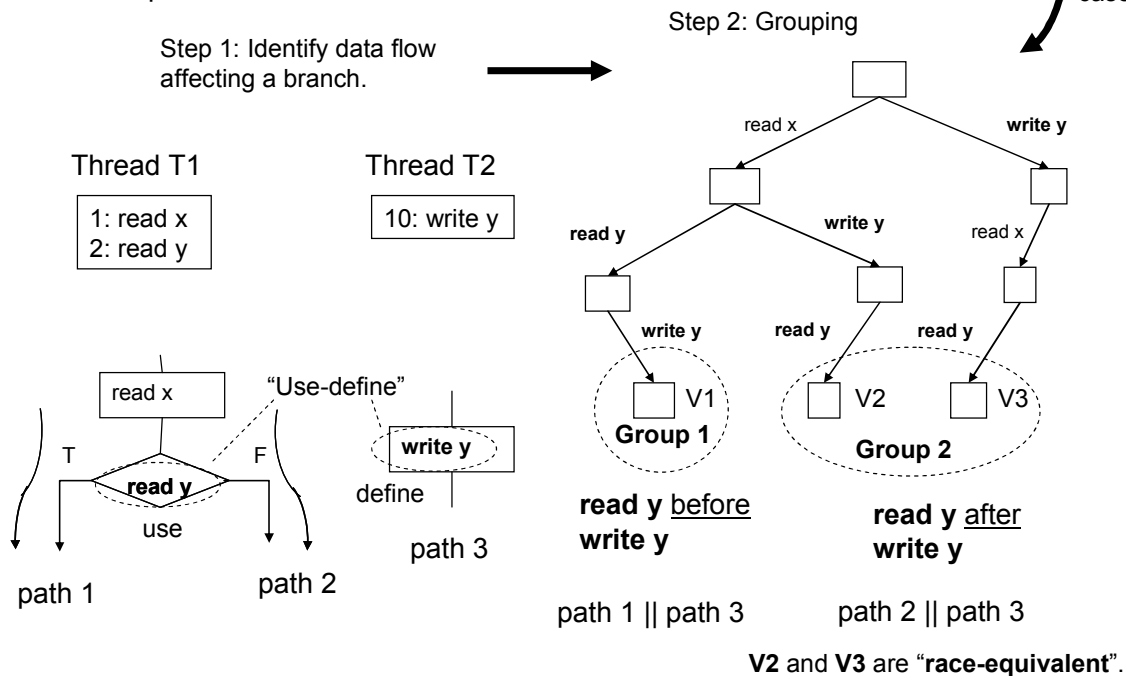


Figure 36. Reducing test cases by avoiding redundant interleavings

5.1.1 Creating Different Race-Equivalent Groups

This subsection explains our proposed method for the reduction of the number of different interleavings required to detect race conditions. The number of different interleavings is reduced by trying to create only interleavings that lead to a different race-equivalent group by:

- Changing a control flow by changing a branch outcome (see Figure 39).

- Changing a lock sequence (see Figure 40). Similarly we can also change the assignment to a reference variable to create a different race-equivalent group (will be explained in subsection **5.3.6 Generating Test Cases to Check Consistent Locking for Access through Reference Variables**).

There can be a chain of reactions from a change of interleaving and/or interrupt timing that can cause a different race-equivalent group (see Figure 37). The set of different interleavings and interrupt timings which disconnect this chain constitutes a race-equivalent group.

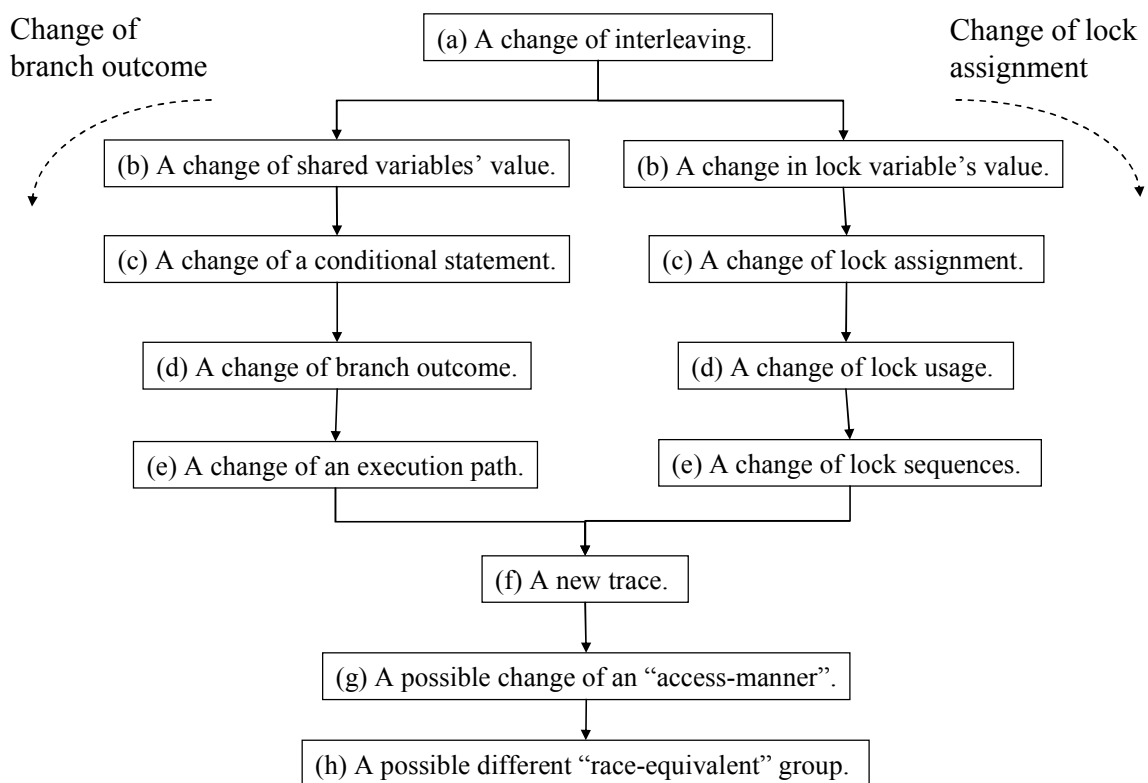


Figure 37. Chain of reactions that can cause a different race-equivalent group

5.1.2 Creating a Different Race-Equivalent Group by Changing a Control Flow

We create different race-equivalent groups efficiently by considering only different execution paths. The basic idea in this research is that, for exploring possible different execution paths, it is sufficient to create and test only those

interleavings that might change the control flow. Figure 38 shows an example how interleavings and a branch can affect the occurrence of a race condition. Throughout the following explanation, we will discuss only change of control flow through branches, but the same principle can also be applied for loops.

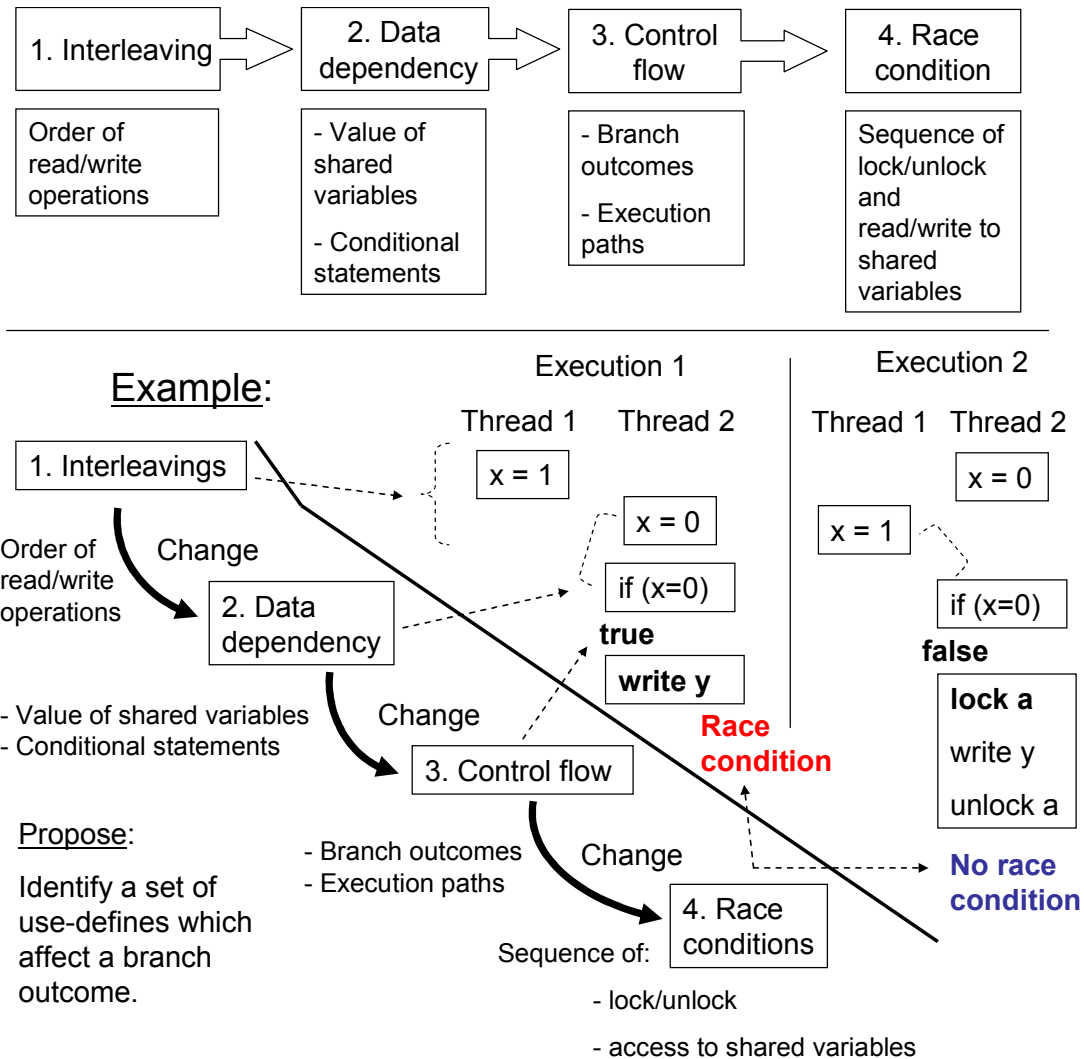


Figure 38. Interleavings and a branch affecting the occurrence of a race condition

Different execution-variants might lead to the same branch outcome for a particular branch b . Hence, in exploring different concurrent execution paths caused by the branch b , we can reduce test cases by grouping those execution-variants and testing only one member from each group. We name such a group a “branch-affect” group. A branch-affect group for a branch b contains a set of execution variants that

would cause the same branch outcome for the branch b , which is either *true* or *false*.

The idea for grouping the execution-variants comes from the fact that if two execution-variants have the same data flow affecting a branch b , then the branch b will have the same branch outcome in those two execution-variants. Formally we define as follows:

- Let $BranchRelUD(b, V)$ be the set of use-defines affecting the conditional statement of a branch b in an execution-variant V .
- If $BranchRelUD(b, V1) = BranchRelUD(b, V2)$, then the branch b will have the same branch outcome in execution-variant $V1$ and $V2$.

Thus they can be grouped into the same branch-affect group. Two or more execution-variants in the same branch-affect group for a branch b are redundant with respect to exploring the different concurrent execution paths caused by the branch b . In Figure 39, the execution-variant $V2$ and $V3$ are in the same branch-affect group and they all cause the branch outcome to become *false*.

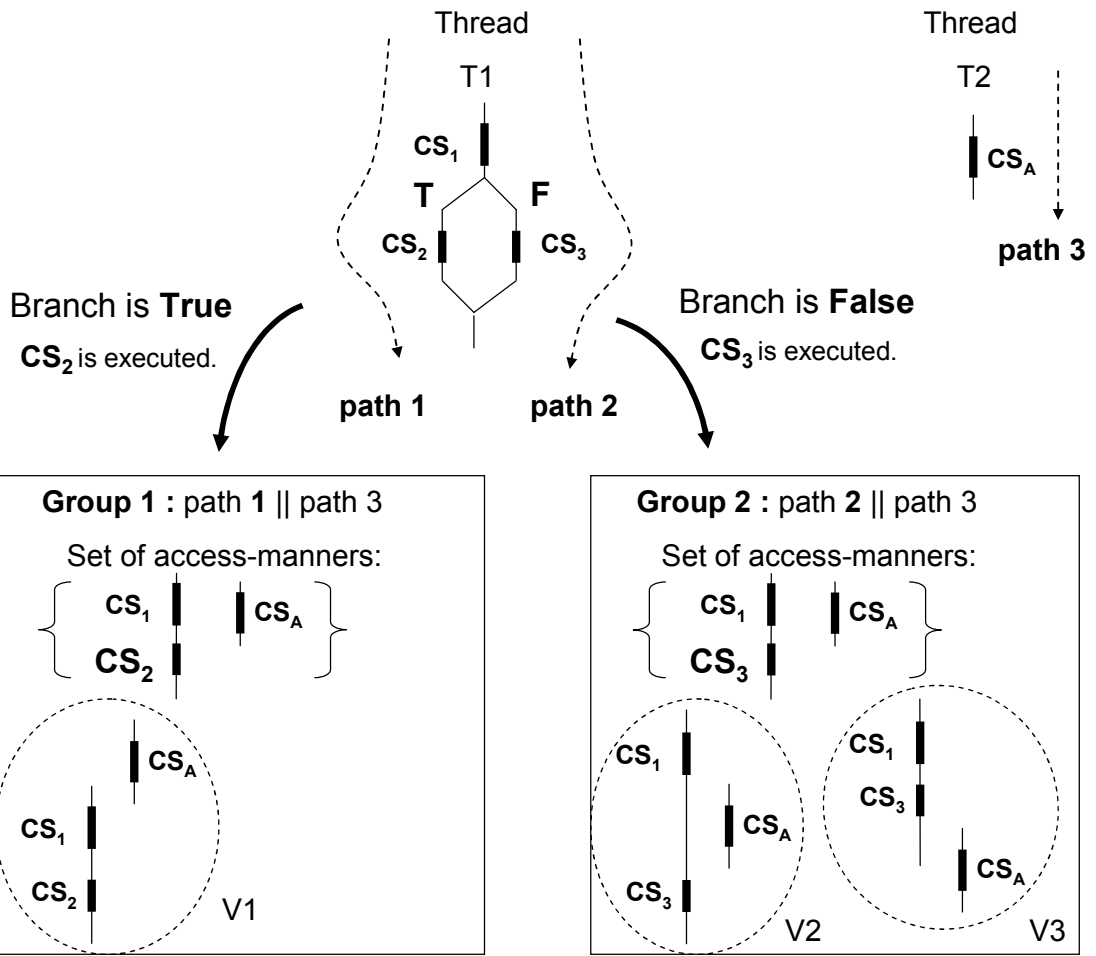


Figure 39. Examples of grouping by changing a branch outcome

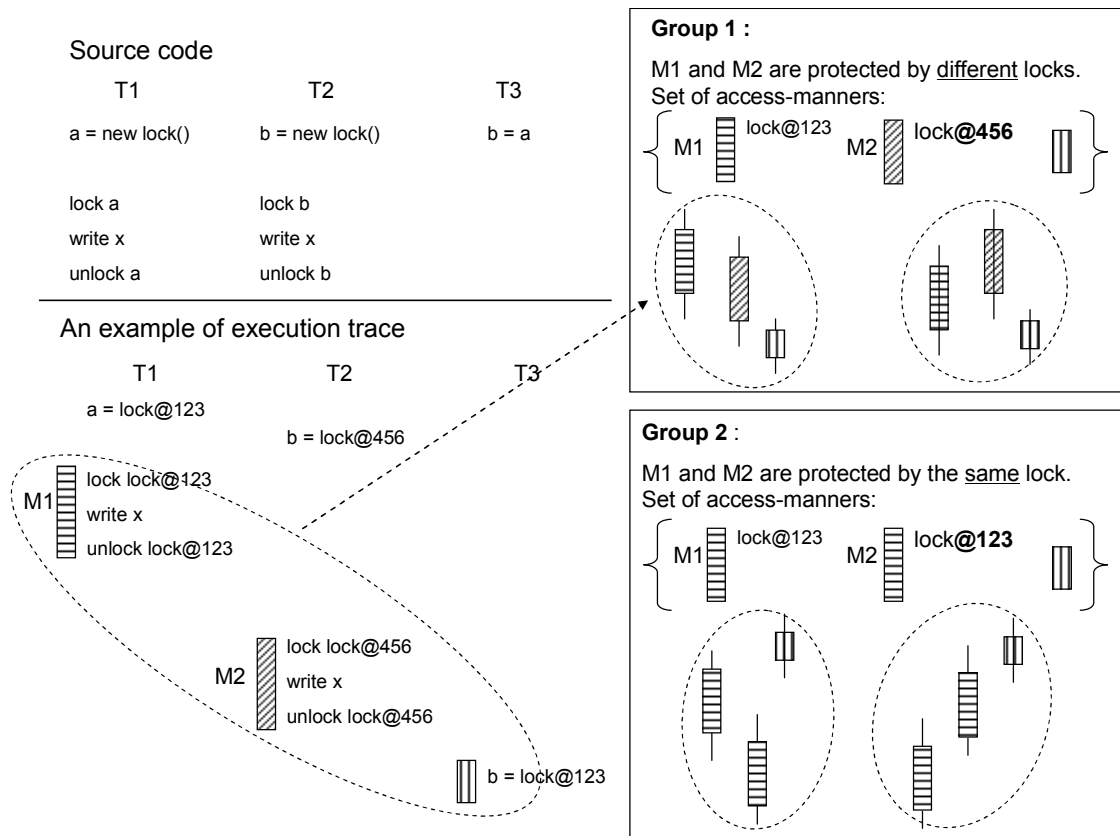


Figure 40. Creating a different race-equivalent group by changing a lock sequence

5.1.2.1 Determining the Set of Operations that Affect Branch Outcomes

In order to identify branch-affect groups, we first need to determine the set of operations that affect the conditional value of a particular branch b . We propose a data dependency analysis method using use-define (see section 3.13 Use-Define) to identify operations that affect the conditional statement of the branch b from an execution trace. This method analyzes data dependency among read/write operations to shared variables related to the conditional statement of the branch b . Based on this analysis, we can determine which operations are affecting the conditional statement.

The set of use-defines can be obtained by analyzing the execution trace or source code. Since the method proposed in this dissertation iteratively generates different interleavings based on previous execution traces, it is sufficient to use the use-define set obtained only from the execution trace. The use-define set obtained by

the static analysis of source code may contain redundant elements. Information from the source code can be used as a supplement if execution traces do not contain complete information for obtaining the use-define set. In this dissertation, we assume that the execution trace contains enough information to obtain the set of use-defines consisting of triplets of variable names, read or write operations, and locations. Figure 42 shows an example of a use-define set for the program example in Figure 41.

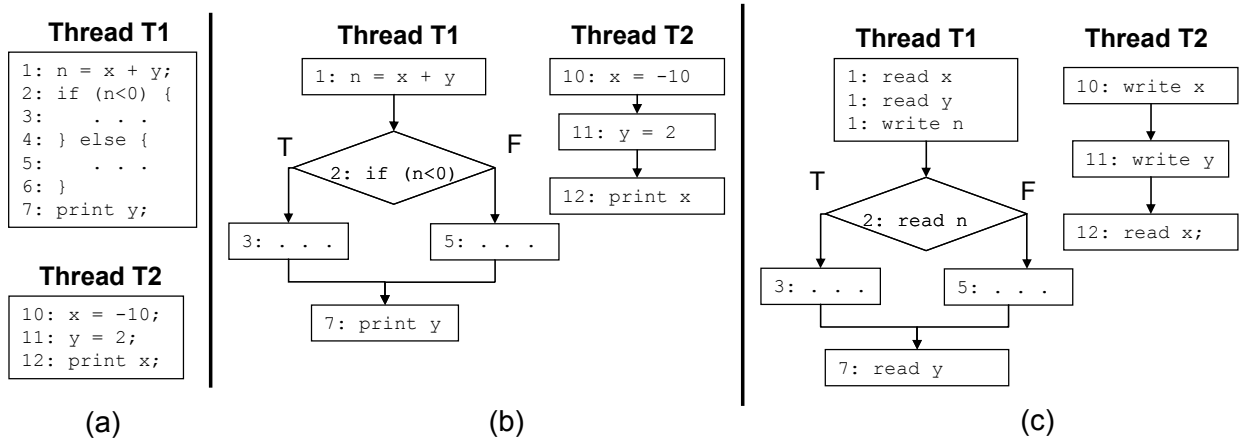


Figure 41. (a) An example of a concurrent program (b) Control flow graph (c) Control flow graph for read and write operations

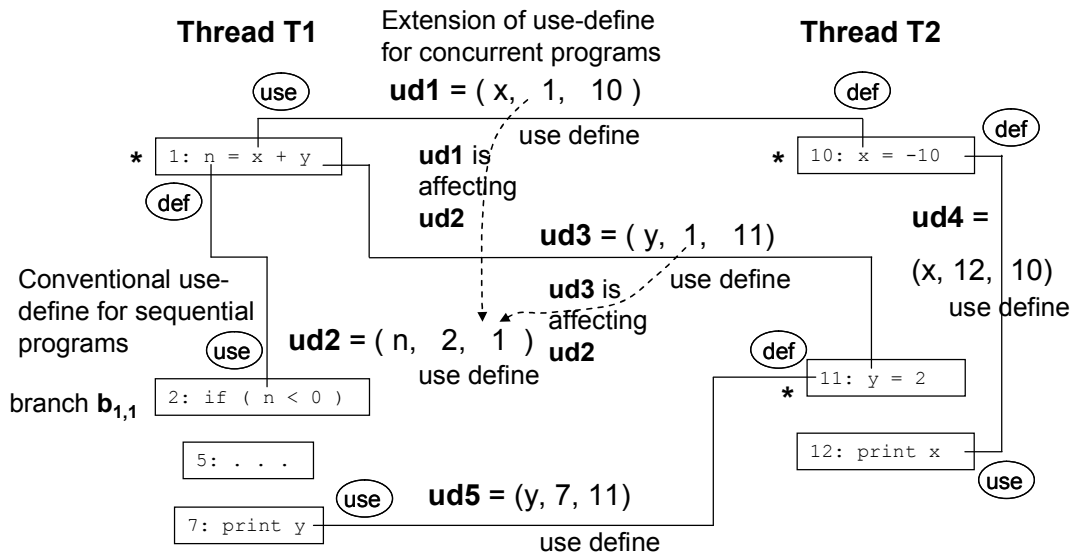


Figure 42. Examples of use-defines for the concurrent program in Figure 41

To detect a conventional use-define, we identify the variable in a thread's execution trace and check if it forms a conventional use-define. To detect an extended

use-define, we first need to identify shared variables from the execution trace. A variable is shared if it is accessed by more than one thread. In the example of Figure 42, we see that the variable x and y are shared variables because they were accessed by more than one thread. For each access to a shared variable in a thread, we check if it forms an extended use-define with another thread. In the example of Figure 42, the read operation on shared variable x in line 1 and the write operation on shared variable x in line 10 form an extended use-define. There are several examples of use-define in Figure 42, as follows:

- Conventional use-define: $ud2 = (n, 2, 1)$, $ud4 = (x, 12, 10)$
- Extended use-define for concurrent programs: $ud1 = (x, 1, 10)$, $ud3 = (x, 1, 11)$, $ud5 = (y, 7, 11)$

Since a wait-notify mechanism can change data flow, it might cause some infeasible use-defines. This situation could happen, for example, when there is a “wait” command without the corresponding “notify” command. In this example, the *use* or *define* after the wait command will not be executed, so the use-define becomes infeasible. C. Yang, A.L. Souter, and L. L. Pollock [Yang98] [Yang97] describe some complications that synchronization causes during data flow analysis. Some infeasible use-defines might be included in a use-define set, but they will not be executed and will not be used for grouping execution-variants. The infeasible use-define pairs will cause redundancy in the use-define set, but they will not cause redundancy in test case generation.

Data Dependency Relation with Use-define

For identifying operations which are affecting a control flow, we define a dependency relation named as use-defines. A use-define $ud2$ depends on another use-define $ud1$, if the definition for the variable in use-define $ud2$ is using the variable in the use-define $ud1$. Formally we define as follows:

Let: $ud1 = (var1, use1_location, def1_location)$
 $ud2 = (var2, use2_location, def2_location)$

The $ud2$ depends on the $ud1$ iff the following two conditions are satisfied:

$$def2_location = use1_location$$

This also means that variable $var2$ depends on variable $var1$.

An example of a dependency relation between use-defines is shown in Figure 42. Since the *def_location* of use-define *ud2* is the same as the *use_location* of use-define *ud1*, then use-define *ud2* depends on use-define *ud1*. This means that there is data flow from the variable *x* to the variable *n*, because the definition of variable *n* in line 1 uses the variable *x* in line 10. In a similar way, the use-define *ud2* depends on the use-define *ud3*.

Algorithm 3 shows how to find the members of $BranchRelUD(b)$ using the dependency relation of use-define.

Algorithm 3. Finding a set of use-defines affecting branch outcomes

Input:

- An execution variant V .
- A branch b .

Output:

- $BranchRelUD(b, V)$: a set of use-defines affecting branch outcomes of branch b .

Step 1. Initialization.

1.1 $SetUD$: set of use-defines from the execution variant V .

1.2 $BranchRelUD(b, V)$: use-defines from $SetUD$ where the variables are used in the conditional statement of the branch b .

Step 2. Find all related use-defines.

2.1 **For** each use-define ud in $SetUD$, where

ud is not included in $BranchRelUD(b, V)$, and

ud does not contain any operations from the same thread as the branch b after the execution of the branch b .

2.1.1 If any use-defines in $BranchRelUD(b)$ depend on ud .

Then

Add ud to $BranchRelUD(b, V)$.

Repeat **Step 2.1** until **Step 2.1.1** no longer adding any use-defines to $BranchRelUD(b, V)$.

2.2 Remove use-defines for local variables from $BranchRelUD(b, V)$.

Terminate this algorithm.

When Algorithm 3 no longer finds use-defines that satisfy the conditions in

step 2.1.1, it means that all use-defines related to the conditional statement of the branch b have been included in $BranchRelUD(b, V)$. When we consider different effects caused by interleavings, we need to consider only different interleavings of read and write operations on shared variables. Hence, we can consider only the set of use-defines which is affecting the shared variables (Step 2.2). The example in Table 2 illustrates how Algorithm 3 finds $BranchRelUD(b_{1,1})$ for the program example in Figure 42.

Table 2. An example of finding a set of operations that is affecting branch outcomes using Algorithm 3

Step	Description
1.1	$SetUD = \{(x,1,10), (n,2,1), (y,1,11), (x,12,10), (y,7,11)\}$.
1.2	$BranchRelUD(b_{1,1}, V) = \{(n, 2, 1)\}$.
2.1.1	The use-define $(n, 2, 1)$ depends on the use-define $(x, 1, 10)$ and $(y, 1, 11)$. $BranchRelUD(b_{1,1}, V) = \{(n,2,1), (x,1,10), (y,1,11)\}$. Go to step 2.1.
2.1.1	No more use-defines that satisfy the conditions in step 2.1.1.
2.2	Remove use-defines for local variables. $BranchRelUD(b_{1,1}, V) = \{(x,1,10), (y,1,11)\}$. Algorithm terminates.

Grouping Execution-Variants Which Causing the Same Branch Outcome

We define Algorithm 4 for creating branch-affect groups for a branch. Execution-variants in the same branch-affect group for a branch b will have the same branch outcome for the branch b .

Algorithm 4. Creating a set of branch-affect groups for a branch

<p>Input: Execution-variants from a variant graph.</p> <p>Output: A set of branch-affect groups $G(b)$ for a branch b. $G(b) = \{g1(b), g2(b), g3(b), \dots\}$, where $g1(b)$, $g2(b)$, $g3(b)$ are the first, second, and third branch-affect groups for the branch b in the execution trace.</p> <p>Step 1. Find $BranchRelUD(b)$ using Algorithm 3.</p> <p>Step 2. For each execution-variant V in the variant graph.</p> <p>2.1 Take a sequence of operations S within the execution-variant V, where S starts from the root node of execution variant V and ends at the operation within the conditional statement of branch b.</p>

2.2 Check which of the $BranchRelUD(b)$ members are in the sequence S using the following rules:

Assume $ud(var, use_location, def_location)$ is a member of $BranchRelUD(b)$.

The $def_location$ is executed before the $use_location$ in sequence S

No other definition to the variable var in between $def_location$ and $use_location$ within the sequence S .

2.3 **If** the use-define members from step 2.2 are already exist in the current branch-affect group.

Then Add the execution variant V to the corresponding existing branch-affect group.

Else Create a new branch-affect group into $G(b)$ and add the execution variant V as its member.

As shown in the example in Figure 44, execution-variants $V3$ and $V4$ can be grouped together into the same branch-affect group with respect to the branch $b_{l,l}$ because they have the same set of use-defines affecting the branch b . A similar situation also applies for the execution-variants $V6$ and $V7$, as shown in Figure 44. Table 3 shows the complete groups for the examples in Figure 44.

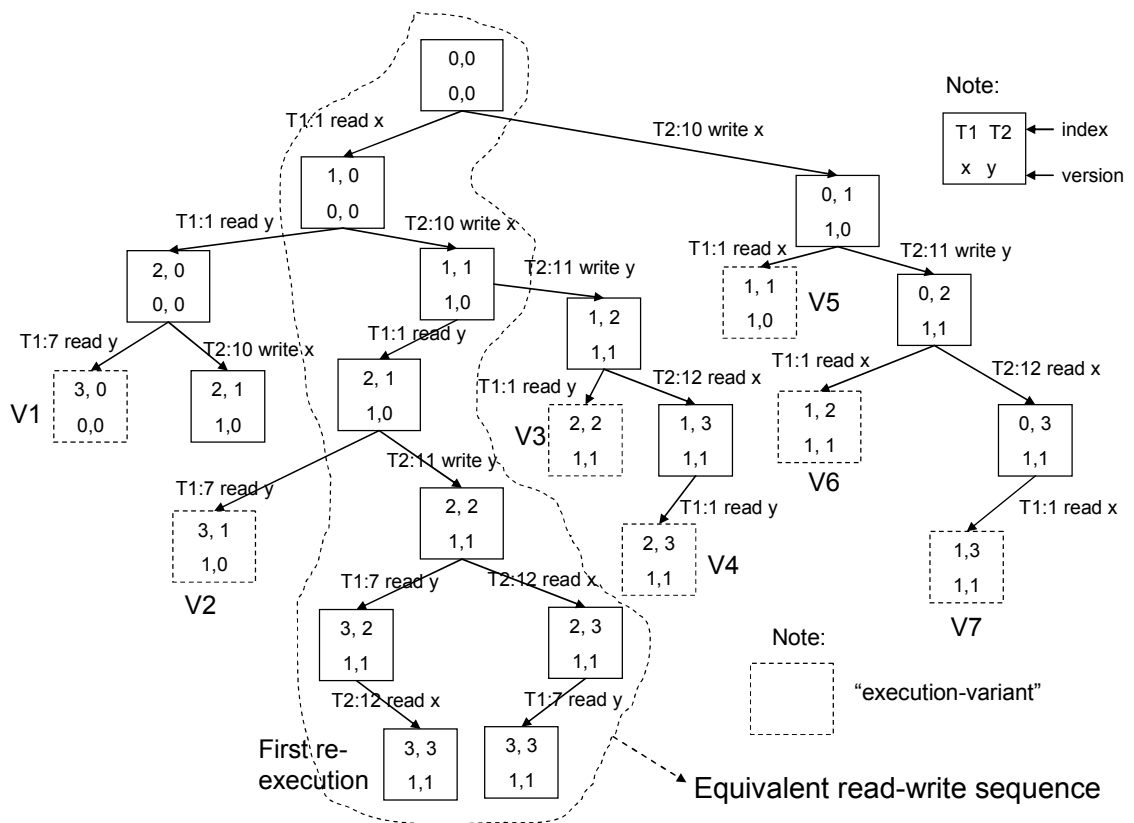


Figure 43. An example of a variant graph from an execution trace

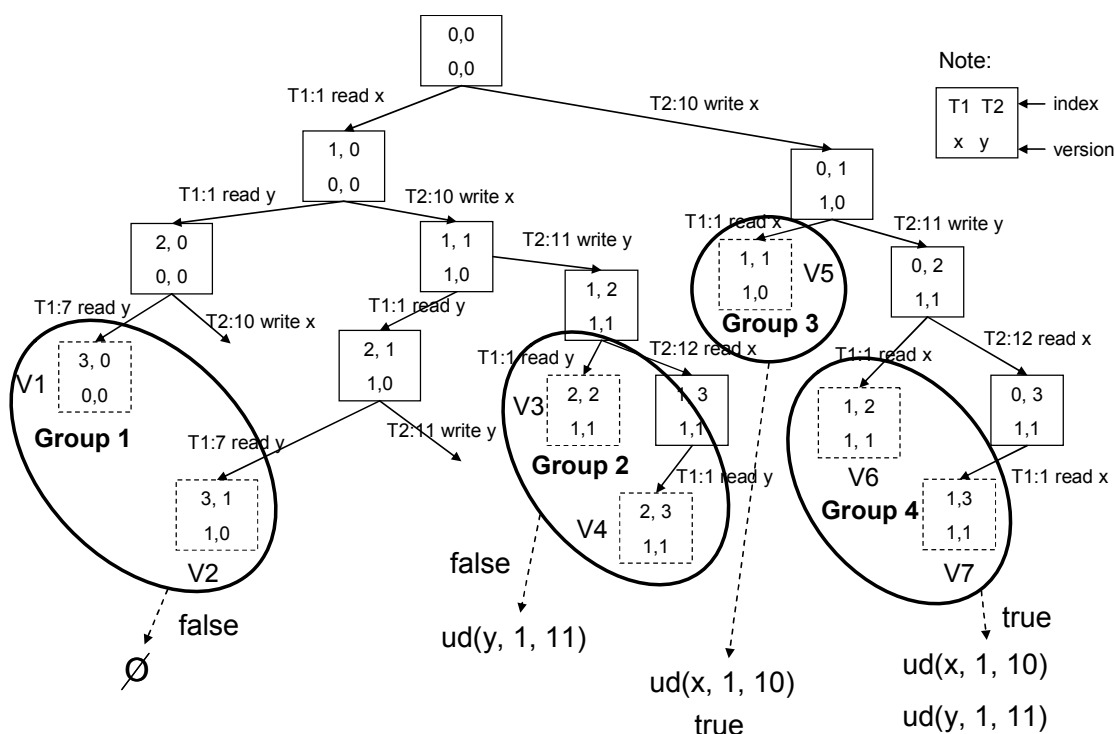


Figure 44. Examples of branch-affect groups for the variant graph in Figure 43

Table 3. An example of a branch-affect table

Branch	Members of branch-affect groups	Set of use-defines from $BranchRelUD(b_{l,i})$
$b_{l,i}$	$g1(b_{l,i}) = \{V1, V2\}$	$\{\emptyset\}$
	$g2(b_{l,i}) = \{V3, V4\}$	$\{ud(y, 1, 11)\}$
	$g3(b_{l,i}) = \{V5\}$	$\{ud(x, 1, 10)\}$
	$g4(b_{l,i}) = \{V6, V7\}$	$\{ud(x, 1, 10), ud(y, 1, 11)\}$

As mentioned in section 3.10 **Race-Equivalent**, two different concurrent execution paths with the same set of execution paths $PATHS$ will be race-equivalent. To explore different race-equivalent groups, it is necessary to find different sets of execution paths $PATHS$. Since the execution path of a thread is affected by branches, we introduce a “branch-condition” table to measure the progress of a test. A “branch-condition” table contains a list of all possible sets of execution paths $PATHS$. Each row in a “branch-combination” table represents the condition values of if-statements and the number of iterations for loops in a concurrent execution path, so each row represents a possible set of execution paths $PATHS$. Each different loop iteration will lead to a different execution path, so we need to consider all loop iterations. However, if loop iterations have the same set of access-manners, then there is no need to

check all of the iterations because they will be race-equivalent. A “branch-combination” table is an accumulation from each execution of a test case. It is possible that not all branches can be identified from the execution trace of the first test case. If new branches are found during the execution of the next test case, they should be added to the “branch-combination” table. At the beginning, all rows are marked as “untested”, except for the one corresponding to the execution in the first test case.

An example of a “branch-condition” table is shown in Figure 45. We need to test all the feasible sets of execution paths *PATHS*; that is, in order to find the inconsistent locking for read/write operations to shared variables that have caused errors, all the rows in a “branch-combination” table need to be tested. Algorithm 5 is the complete algorithm of the proposed method. This algorithm integrates the existing reachability testing in step 1.2, with the deterministic testing and race detection in step 4.

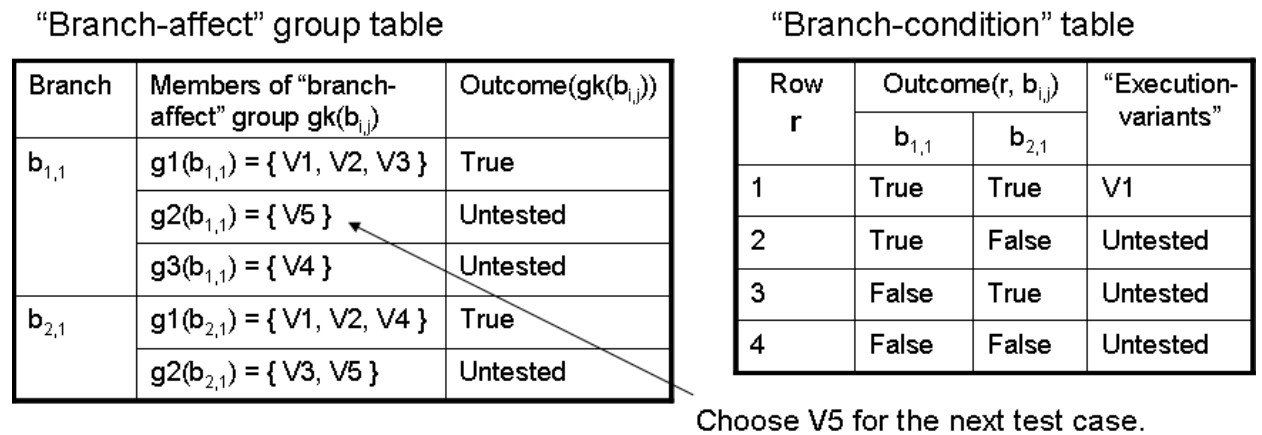


Figure 45. Branch-affect group table and branch-condition table for the first test case

Algorithm 5. Complete algorithm for generating test cases and checking race conditions

<p>Definitions:</p> <ul style="list-style-type: none"> - $Outcome(gk(b_{i,j}))$ is the truth value for an if-statement or the number of iterations for a loop of a branch-affect group $gk(b_{i,j})$ - $Outcome(r, b_{i,j})$ is the truth value or the number of iterations of the branch $b_{i,j}$ for row r in a “branch-condition” table. <p>Input: a concurrent program and its input.</p> <p>Output: test cases and race-detection results.</p>
--

Step 1. Initialization:

1.1. Re-execute the concurrent program taking trace using the same input as in the execution in which the error occurred.

1.2. Create the corresponding variant graph from the execution trace using Algorithm 2.

1.3. Create a “branch-condition” table based on the execution trace from step 1.1.

1.4. For each branch of the variant graph in step 1.3, classify each execution-variant into branch-affect groups using Algorithm 4.

Step 2. Conditions for termination.

2.1 Terminate this algorithm if at least one of the following conditions is satisfied:

- Condition 1: all rows in the “branch-condition” table have been tested,
- Condition 2: all branch-affect groups have been marked as “tested”. Note that the algorithm terminates with the second condition if there exists any infeasible set of concurrent execution paths for the given input.

Step 3. Select the next test cases *TestCases*:

3.1 $TestCases = \{ \emptyset \}$

3.2 **For** each untested row r in “branch-condition” table

3.2.1 $Candidates = \{ \emptyset \}, firstGroup = true.$

3.2.2 **For** each branch $b_{i,j}$.

If ($firstGroup == true$).

Then $Candidates =$ all members of branch-affect groups of branch $b_{i,j}$ where $Outcome(gk(b_{i,j})) == Outcome(r, b_{i,j})$

$firstGroup = false$

Else $Candidates = Candidates \cap$ all members of the branch-affect groups of the branch $b_{i,j}$ where $Outcome(gk(b_{i,j})) == Outcome(r, b_{i,j})$

3.2.3 Select one execution-variant from $Candidates$ and add it to $TestCases$.

3.2.4 **If** step 3.2.3 does not produce any test cases.

Then choose a member from an untested branch-affect group and add it to the $TestCases$.

Step 4. Test cases execution.

4.1 Execute the execution-variants from the $TestCases$ using deterministic testing with tracing.

4.2 Check the execution trace from step 4.1 using an existing race detector and report any errors.

4.3 Derive new execution-variants from the execution trace in step 4.1, update the

variant graph and “branch-condition” table.

4.4 Classify the new execution-variants into branch-affect groups.

Step 5. Repeat from **step 2**.

Race-equivalent means two concurrent execution paths of a concurrent program have the same consistent locking for accessing shared variables, and also share the same proper/improper lock-unlock sequences. When a variant graph produces execution-variants, our algorithm groups them into race-equivalent groups. Our method achieves test case reduction by testing only one member of each race-equivalent group.

A step-by-step example of Algorithm 5 is shown in Table 4 and Table 5. We assume that there is a concurrent program with two threads $T1$ and $T2$. Thread $T1$ has one branch $b_{1,1}$ and thread $T2$ has one branch $b_{2,1}$. The branches $b_{1,1}$ and $b_{2,1}$ are if-statements. The steps in Table 4 are deduced from the analysis shown in Figure 45. The steps in Table 5 are deduced from the analysis shown in Figure 46.

Table 4. Step-by-step example of Algorithm 5

Step	Description
1	Let us assume that step 1 results a variant graph with five execution-variants. The execution for the first test case is $V1$ which makes $b_{1,1}$ and $b_{2,1}$ <i>True</i> . Assume that the branch-affect group has been calculated using Algorithm 4 and the “branch-condition” table is as exemplified in Figure 45.
2	Not all rows in the branch-condition table have been tested, so proceed to Step 3.
3	Step 3.2.3 does not produce any test cases.
3.2.4	Since Step 3.2.3 does not find any test cases, $V5$ is chosen as a test case from untested branch affect group $g2(b_{1,1})$.
4.1	Execute $V5$ using deterministic testing and obtain execution trace.
4.3	When we derive the execution trace from step 4.1, we find the new execution-variant $V6$
4.4	The new execution-variant $V6$ is classified into $g2(b_{1,1})$ and $g1(b_{2,1})$, see Figure 46.
5	Repeat from step 2

Table 5. Step-by-step example of Algorithm 4 (continued)

Step	Description
2	Not all rows in the branch-condition table have been tested, so proceed to Step 3.
3	$TestCases = \{ \emptyset \}$, for each untested row r in the branch-condition table The 2 nd row: $Candidates = \{V1, V2, V3\} \cap \{V3, V5\} = V3$. The 3 rd row: $Candidates = \{V5, V6\} \cap \{V1, V2, V4, V6\} = V6$. The 4 th row: $Candidates = \{V5, V6\} \cap \{V3, V5\} = V5$. $TestCases = \{V3, V6, V5\}$
4	No need to do step 4 because there are some test cases from step 3.
5.1	Execute the members of $TestCases$.
5.2	No new execution-variants can be derived from the trace in step 5.1.
2	All rows in the “branch-condition” table have been tested, so the algorithm terminates.

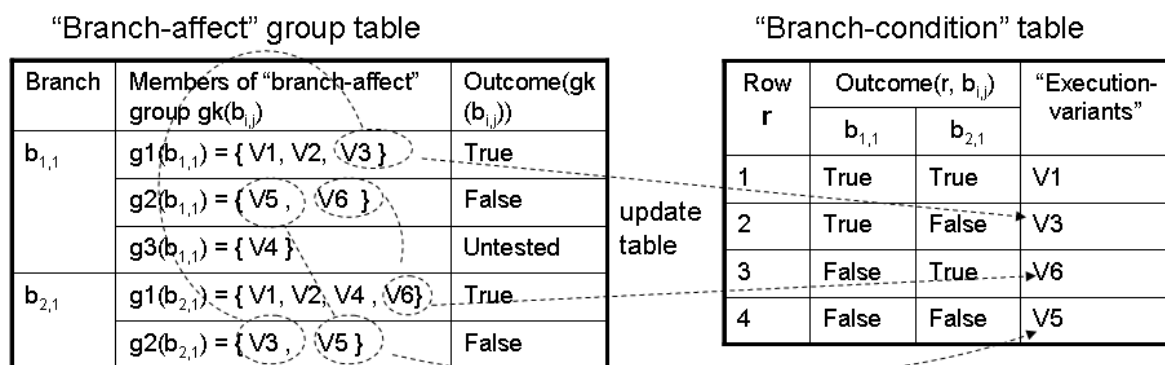


Figure 46. Branch-affect group table and branch-condition table when Algorithm 5 terminates

There is no need to test the branch-affect group $g3(b_{1,1})$ because all the rows in the “branch-condition” table in Figure 46 have been completed. Our algorithm requires only the testing of four execution-variants from the total of six execution-variants.

5.2 Avoid Testing Infeasible Interleavings

Generating different interleavings for test cases must consider the existence of locks in order to avoid deadlock. Enforcing infeasible interleavings in deterministic replay environment might cause suspension, which will not be allowed in the real situation. Avoid generating infeasible test cases reduces the number of interleavings.

We extend the existing variant graph [Hwang95] by considering

synchronization dependencies to eliminate redundancy. The extended model for variant graphs utilizes trace information about lock-unlock and wait-notify operations. For wait-notify, we use a simple model with the following assumptions:

- A thread that is waiting for a notification can receive a notification from any threads.
- A notification is sent to all threads.
- In general, a notification will be accepted and processed by particular threads. In this simple model, we assume only waiting threads will accept and process the notification, otherwise the incoming notification will be lost.

We extend the node in a variant graph to include flags for “lock” and “wait” besides the existing “index” and “version”. “Index” will also be incremented for lock-unlock and wait-notify operations. In this way, different orders of wait-notify will be considered in test case generation, thus avoiding false negatives. We add the following rules in the extended variant graph for handling lock-unlock and wait-notify operations:

- Lock-unlock:
 - If the operation is “lock”, set the lock flag for the corresponding lock to 1.
 - If the operation is “unlock”, reset the lock flag for the corresponding lock to 0.
- Wait-notify:
 - If the operation is “wait”, set the wait flag for the corresponding thread to 1.
 - If the operation is “notify”, reset the wait flags for all threads to 0.
 - Since in our model we assume a notification is sent to all thread, so the wait flags are reset for all threads.

When expanding an extended variant graph, a node is infeasible if any one of the following conditions holds:

- The wait flag for the corresponding thread is 1.
- The operation is lock and the lock flag is 1.

Figure 47 shows an extension of a variant graph which adds lock-unlock and wait-notify operations for the concurrent program in Figure 41(a). The extended variant graph in Figure 47 identifies some infeasible interleavings caused by the lock-unlock and wait-notify operations.

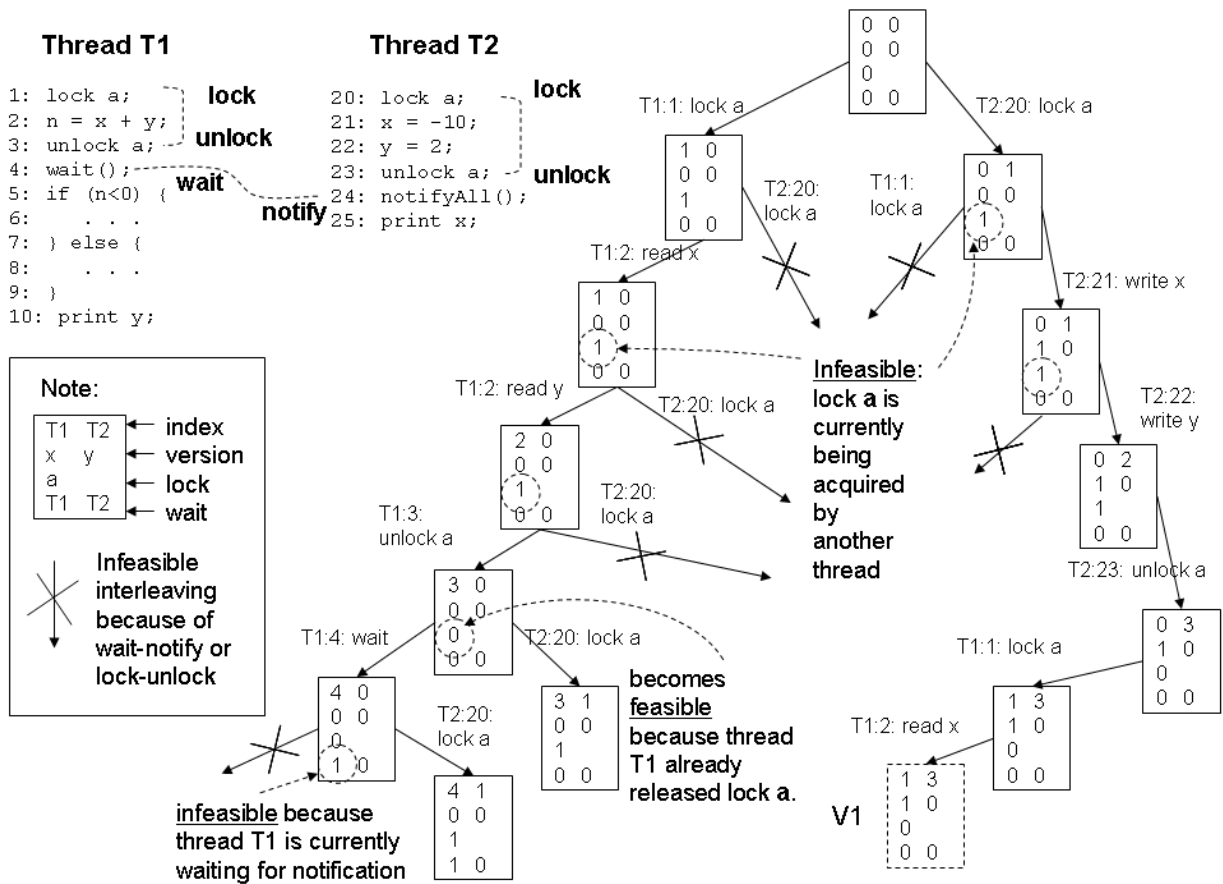


Figure 47. An example of the extension of a variant graph

5.3 Reduce Memory Required for Generating Test Cases

This section explains how to reduce memory required for generating test cases by proposing a concurrent dependency graph. Figure 48 illustrates the general idea for reducing the required memory.

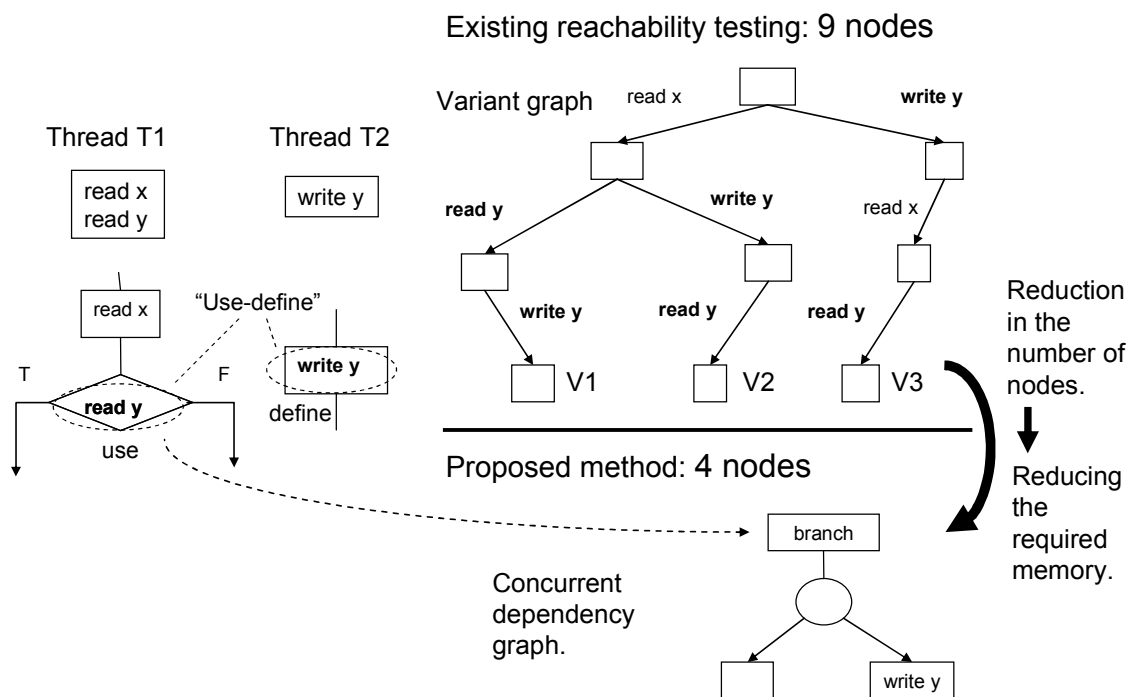


Figure 48. General idea to reduce memory required for generating test cases

5.3.1 System Overview

This new method introduces “concurrent dependency graphs”, instead of variant graphs. Variant graphs are the major instruments for representing and analyzing the execution development of a concurrent program in the reachability testing method. Table 6 shows the comparison between the existing variant graph and the proposed concurrent dependency graph.

Table 6. Comparison between the existing variant graph and the proposed concurrent dependency graph

	Existing: variant graph	Proposed: concurrent dependency graph
Purpose	Generate all different interleavings affecting values of shared variables.	Generate only different interleavings affecting race conditions.
Size	Bigger	Smaller

Usually only fewer different interleavings affecting race conditions.

Figure 49 shows the overview of the proposed method to avoid redundancy in test case generation. The whole procedure for testing is shown as follows:

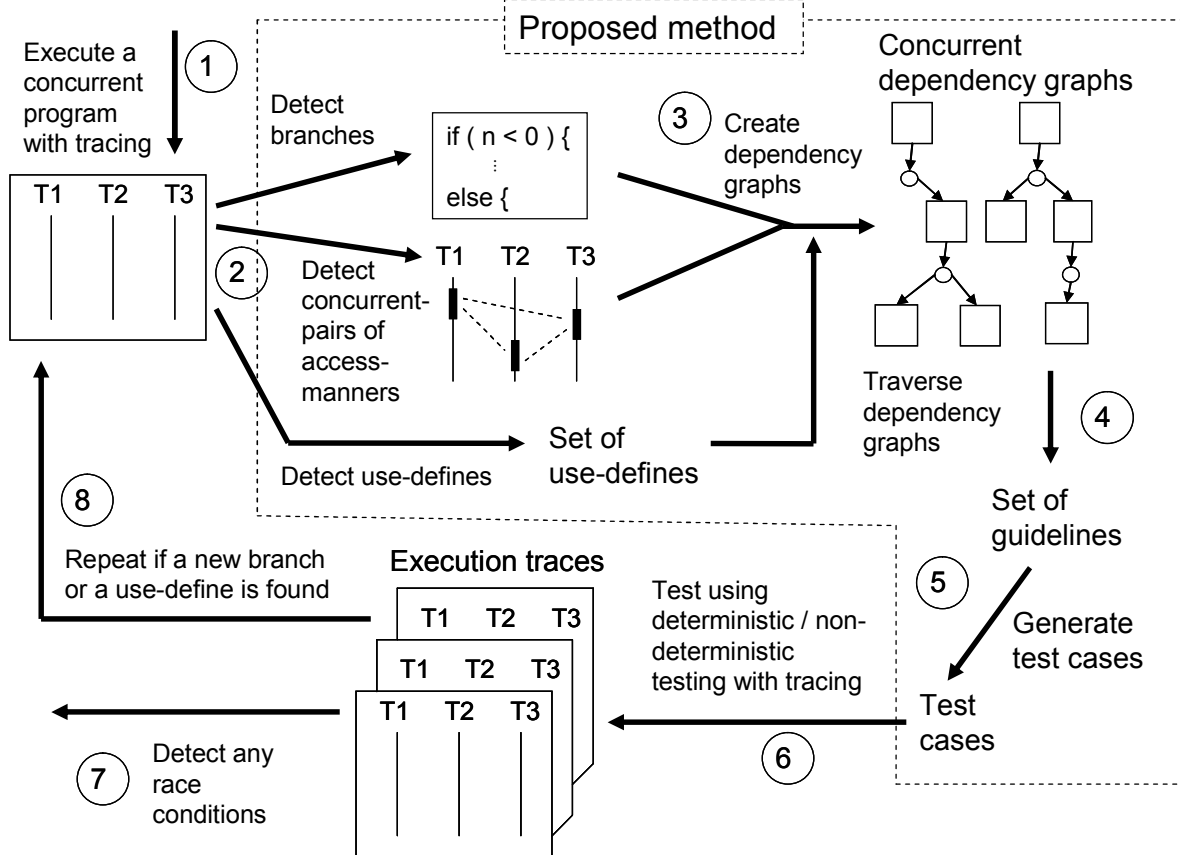


Figure 49. Overview of the proposed method

- 1) Execute a concurrent program with a trace.
- 2) Detect branches, concurrent-pairs of access-manners, and a set of use-defines from the execution trace.
- 3) Create concurrent dependency graphs from branches and concurrent-pairs of access-manners. A concurrent dependency graph represents data flow relations among operations that might affect race conditions.
- 4) Determine a set of “guidelines” for generating test cases. A “guideline” is a set of use-defines obtained by traversing the concurrent dependency graphs from the previous step (will be explained in subsection 5.3.3 Traversing a Concurrent Dependency Graph).
- 5) Generate test cases based on the set of guidelines from step 4. The idea is to generate only those test cases necessary to avoid redundancy and that do not affect

- race conditions.
- 6) Execute the test cases using an existing deterministic/non-deterministic testing method by taking a trace.
 - 7) Detect any race conditions using an existing race detector and report them to programmers.
 - 8) If a new branch or a new use-define is found in the execution trace in step 6, repeat step 3 to step 8 for the new branch or the new use-define.
 - 9) The test is completed if neither a new branch nor a new use-define is found in step 6.

5.3.2 Concurrent Dependency Graph

We newly propose a concurrent dependency graph for identifying data dependencies of shared variables or reference variables. A concurrent dependency graph is a directed graph representing use-define relations in an execution of a concurrent program. A conventional dependency graph depends only on data flow, but a concurrent dependency graph depends on data flow and interleavings. A concurrent dependency graph contains all possible data dependencies for different interleavings. Which data dependency actually occurs in a particular execution would depend on the interleaving during the execution. Figure 50 shows an example of a concurrent dependency graph.

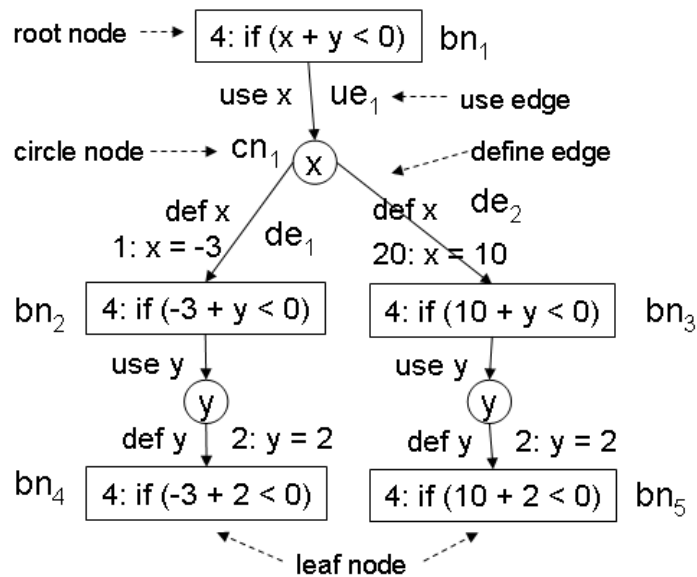


Figure 50. Components of a concurrent dependency graph

Let us take an example of the shared variable x in the root node. There are two write operations that can define its value depending on the interleavings. One is the write operation in line 1 while the other one is in line 20. The components of a concurrent dependency graph are as follows:

■ Node:

● Box node (bn):

◇ Root node: represents one of the following:

- A conditional statement in a branch (see the example in Figure 50), or
- An access-manner (see the example in Figure 58).

A root node does not have an incoming edge.

◇ Non-root node: derived from a root node or another non-root box node. Algorithm 6 explains how to derive non-root nodes. A non-root node has one incoming and one outgoing edge.

◇ Leaf node: a box node whose statement does not contain any variables. When a variable is used without being defined, then there will be no corresponding leaf node. A leaf node does not have an outgoing edge. In Figure 5, nodes bn_4 and bn_5 are leaf nodes.

The maximum number of outgoing edges from a box node is 1.

● Circle node (cn): represents a selection of “define” operations for a variable.

■ Edge:

● “Use-edge” (ue): represents a read operation to a variable. This edge goes out from a box node and comes into a circle node. It is labeled by the program statement that reads the variable.

● “Define-edge” (de): represents a write operation to a variable. This edge goes out from a circle node and comes into a box node. It is labeled by the program statement that writes to the variable.

Table 7 lists the definitions in a concurrent dependency graph.

Table 7. Definitions in a concurrent dependency graph

Definitions	Examples (refer to Figure 50)
$variable(ue)$: the variable used by a use edge ue .	$variable(ue_1) = x$
$variable(de)$: the variable defined by a define edge de .	$variable(de_1) = x$
$variable(bn)$: the set of variables in the statement of a box node bn .	$variable(bn_1) = \{x, y\}$
$variable(cn)$: the set of variables in the statement of a circle node cn .	$variable(cn_1) = \{x\}$
$def_edge(cn)$: the set of define edges for a circle node cn .	$def_edge(cn_1) = \{de_1, de_2\}$
$parent(cn)$: the parent node of a circle node cn .	$parent(cn_1) = bn_1$
$parent(cn) = \{bn \mid \text{where a use edge } ue \text{ exists in which}$ $ue \text{ is the outgoing edge of } bn,$ $ue \text{ is the incoming edge of } cn,$ $variable(bn) \cap variable(cn) \neq \emptyset \}$	
$child(bn)$: the child node of bn .	$child(bn_1) = cn_1$
$child(bn) = \{cn \mid \text{where a use edge } ue \text{ exists in which}$ $ue \text{ is the outgoing edge of } bn,$ $ue \text{ is the incoming edge of } cn,$ $variable(bn) \cap variable(cn) \neq \emptyset \}$	
Note: The child node of a box node is a circle node that represents the “use” of a variable within the statement of the box node. A box node can only have one circle node as its child node.	
$child(cn)$: the set of child nodes of cn .	$child(cn_1) = \{bn_2, bn_3\}$
$child(cn) = \{bn \mid \text{where for every } bn, \text{ a define edge } de \text{ exists in which}$ $de \text{ is an outgoing edge of } cn,$ $de \text{ is an incoming edge of } bn \}$	
Note: A circle node cn does not have any child nodes if the variable for cn is used without being defined.	

A concurrent dependency graph is created by deriving child nodes starting from their root node. Algorithm 6 explains how to derive child nodes from a box node, while Figure 6 is an illustration of Algorithm 6.

Algorithm 6. Deriving child nodes from a box node

Input : - A box node bn_{input} as a parent node.
 - A set of use-defines and potential use-defines.

Output : - The input parent node is connected to a newly-created circle node cn as a child node.
 - The circle node cn is connected to newly-created box node(s) as its child node(s).

Step 1. Create a circle node cn for the input box bn_{input} .

- 1.1 Choose a variable var from the statement inside the bn_{input} .
- 1.2 Create a new circle node cn and label it as var .
- 1.3 Create an outgoing use-edge ue from the bn_{input} to the circle node cn created in step 1.2.
- 1.4 Label the *use* edge ue with the variable chosen in step 1.1.

Step 2. Create child nodes for the circle node cn .

- 2.1 Find *define* operations for $variable(cn)$ from the set of use-defines.
- 2.2 For every *define* operation in step 2.1, create one *define* edge de .
 - 2.2.1 For each *define* edge de in step 2.2, create a box node bn .
 - 2.2.1.1 Make the de the incoming edge for the bn .
 - 2.2.1.2 The box node bn contains the statement from the bn_{input} with the variable var substituted by the define statement in step 2.2.

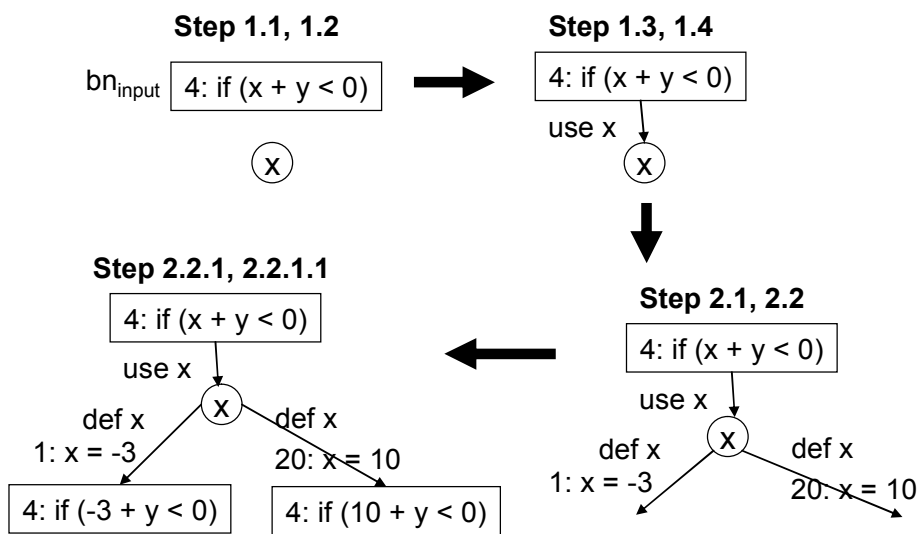


Figure 51. Step-by-step illustration for Algorithm 6

Algorithm 7 explains how to construct a concurrent dependency graph. It derives a box node using Algorithm 6 until all the derived child nodes reach leaf nodes.

Algorithm 7. Constructing a concurrent dependency graph

Input: - A set of use-defines and potential use-defines from an execution trace.
 - A root node.

Output: A concurrent dependency graph dg .

Step 1. Initialization: include the root node in the concurrent dependency graph dg .

Step 2. For every box node bn in dg that does not have an outgoing edge.
 2.1 Create child nodes bn using Algorithm 6.

Step 3. Repeat **step 2** until no more new edges or new boxes are created.

Figure 52 shows a concurrent dependency graph constructed using Algorithm 7 for the branch of the thread $T1$ in Figure 33.

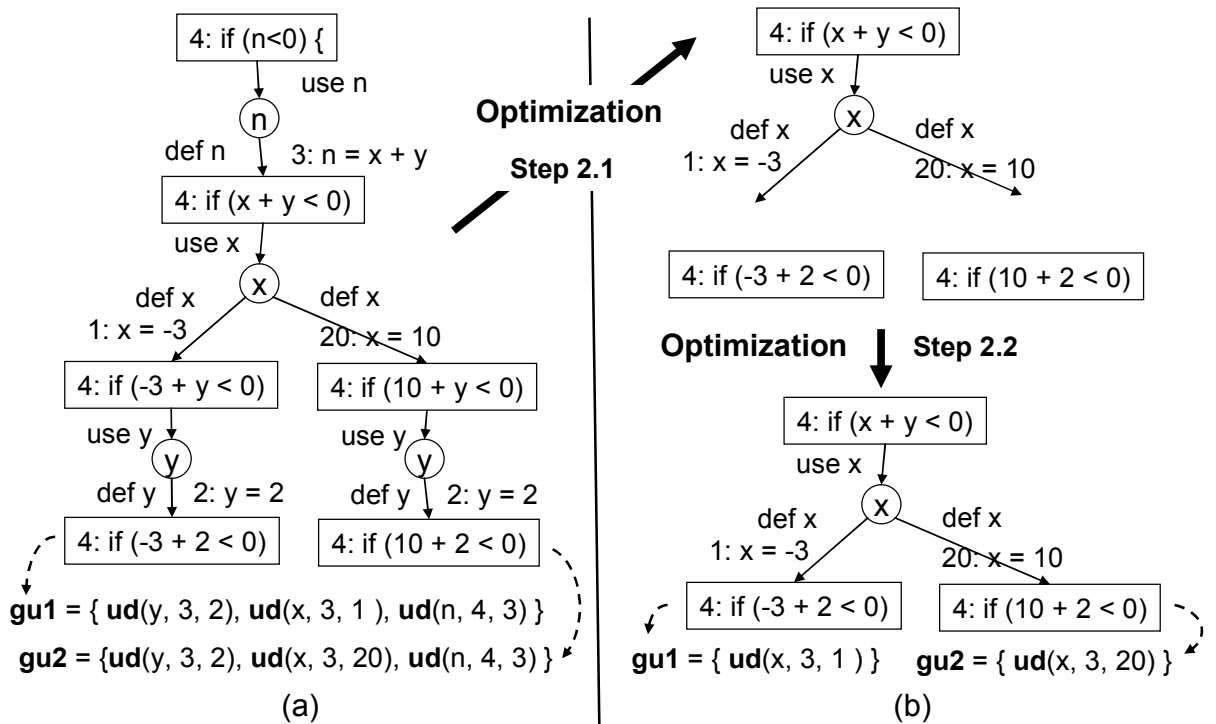


Figure 52. (a) An example of a concurrent dependency graph. (b) and its optimized version

Only variables with a *define set* of more than one member within a concurrent dependency graph can create different execution-variants. Therefore, any variables with only one member in their *define set* are redundant with respect to exploring different execution-variants. Algorithm 8 describes how to optimize a concurrent dependency graph by removing such a redundancy. Figure 52(b) shows an example of an optimized dependency graph.

Algorithm 8. Optimizing a concurrent dependency graph

Input: A concurrent dependency graph dg .

Output: An optimized concurrent dependency graph dg .

Step 1. For each circle node cn in the concurrent dependency graph dg .

1.1 **If** cn has only one outgoing edge.

Then

1.1.1 Remove the parent node of cn and all edges connected to cn .

1.1.2 Make the incoming edge of $parent(cn)$ the incoming edge of $child_node(cn)$.

Note: **step** 1.1.2 is not applicable if the $parent(cn)$ is a root node, because a root node does not have an incoming edge.

The optimized graph is more efficient because it is smaller and thus requires fewer steps to traverse. The next subsection explains how to traverse a dependency graph.

5.3.3 Traversing a Concurrent Dependency Graph

A race condition can occur because different interleavings affecting branch outcomes can lead to different sequences of lock/unlock and read/write operations to shared variables. This subsection explains how to generate different interleavings in order to explore different branch outcomes. We use the term “guidelines” as a set of use-defines for generating a test case. The guidelines determine the data dependency for creating a test case. An execution-variant V satisfies a guideline if all members of the guideline are included in the set of use-defines of the execution-variant V . In other words, the following condition must be satisfied:

$$\text{all members of guideline} \subseteq \text{setUD}(V) \quad (4)$$

Algorithm 9 explains how to traverse the paths in a concurrent dependency graph to obtain a set of guidelines. Table 8 is an example of a set of guidelines obtained by applying

Algorithm 9 to the concurrent dependency graph in Figure 52(a).

Algorithm 9. Traversing a concurrent dependency graph

Input: A concurrent dependency graph dg .
Output: A set of guidelines for generating test cases.
Step 1. Initialization.
 Let the output set of guidelines = $\{ \emptyset \}$
Step 2. Start from the root node of the input concurrent dependency graph dg , do a “Depth First Search” (DFS).
 2.1 When the DFS visits a leaf node, extract the set of use-defines from the root node to the leaf node and add them as a guideline to the set of guidelines as the output.
 2.2 Repeat **step 2.1** until all leaf nodes in the concurrent dependency graph dg have been visited.

Table 8. A set of guidelines from the concurrent dependency graph in Figure 52(a).

No.	Guideline
1	$gu1 = \{ ud(y, 3, 2), ud(x, 3, 1), ud(n, 4, 3) \}$
2	$gu2 = \{ ud(y, 3, 2), ud(x, 3, 20), ud(n, 4, 3) \}$

One test case will be created for each guideline, so there will be two test cases based on Table 8. The use-define $ud(y, 3, 2)$ and $ud(n, 4, 3)$ are the same for both guidelines. They are redundant because the concurrent dependency graph in Figure 52(a) is not optimal. In order to distinguish between these two test cases, only the use-defines on variable x matter. Table 9 is an example of a set of guidelines obtained by applying Algorithm 9 to the optimized concurrent dependency graph in Figure 52(b). It shows that only the use-defines on variable x are necessary to distinguish between those two guidelines.

Table 9. A set of guidelines from the concurrent dependency graph in Figure 52 (b)

No.	Guideline
1	$gu1 = \{ ud(x, 3, 1) \}$
2	$gu2 = \{ ud(x, 3, 20) \}$

5.3.4 Generating Test Cases from a Concurrent Dependency Graph

This subsection explains an efficient test case generation using a set of guidelines from a concurrent dependency graph [Setiadi14]. We recall some definitions from the work by T. E. Setiadi, A. Ohsuga, and M. Maekawa [Setiadi13] in the subsection on a Model for Concurrent Program Execution Traces about the sequence of operations in an execution of a concurrent program. These are as follows:

- S is a sequence of read/write operations from an execution trace.
- $S(j)$ is a sequence of read/write operations in thread T_j .
- $S(j, i)$ is the i -th operation in the sequence of operations in thread T_j .

The task for generating test cases can be stated as follows:

Given a concurrent dependency graph dg derived from an existing sequence of read/write operations $S1$ and the following set of guidelines obtained from the concurrent dependency graph dg :

- $gu1 = \{ ud(var, use, def1) \}$
- $gu2 = \{ ud(var, use, def2) \}$

Supposing that the existing sequence of read/write operations $S1$ satisfies the guideline $gu1$, create another sequence of read/write operations $S2$ that satisfies the guideline $gu2$.

Let:

- $S(a,j) =$ the *use* operation in the guideline $gu2$.
- $S(a,j-1) =$ one operation in the thread T_a before the *use* operation $S(a,j)$.
- $S(b,k) =$ the *def2* operation in the guideline $gu2$.
- $S(b,k-1) =$ one operation in the thread T_b before the *def2* operation $S(b,k)$.

The solution for the $S2$ depends on whether the *use* operation is located in the same thread as *def2* operation or not:

- Case 1: the *use* operation is in the same thread as the *def2* operation, i.e. they are located in the same thread T_b , $S(b,j) = use$ operation and $S(b,k) = def2$ operation (refer to Algorithm 10).
- Case 2: The *use* operation is in a different thread to the *def2* operation (refer to Algorithm 11).

Figure 53 illustrates the examples of these two cases.

Algorithm 10. Generating test cases if the *define* operation is in the same thread as the *use* operation

<p>Step 1. Select the next operation non-deterministically.</p> <p>Step 2. If the operation selected in step 1 is the <i>def2</i> operation $S(b, k)$,</p> <p style="padding-left: 40px;">Then</p> <p style="padding-left: 80px;">2.1 The next operations are from thread Tb until the <i>use</i> operation $S(b, j)$.</p> <p style="padding-left: 80px;">2.2 Select the next operations non-deterministically until the concurrent program terminates.</p> <p style="padding-left: 80px;">2.3 Terminate this algorithm.</p> <p style="padding-left: 40px;">Else</p> <p style="padding-left: 80px;">2.1 Repeat from step 1.</p>

Algorithm 11. Generating test cases if the *define* operation is in a different thread to the *use* operation

<p>Step 1. Initialization:</p> <p style="padding-left: 40px;">- All threads are not blocked.</p> <p>Step 2. Select the next operation non-deterministically from any non-blocked threads.</p> <p>Step 3. Check whether the operation selected in step 2 is one operation before the <i>use</i> operation or before the <i>def2</i> operation.</p> <p style="padding-left: 40px;">3.1 If the operation selected in step 2 is $S(b, k-1)$</p> <p style="padding-left: 80px;">Then</p> <p style="padding-left: 120px;">3.1.1 Thread Tb is blocked.</p> <p style="padding-left: 40px;">3.2 If the operation selected in step 2 is $S(a, j-1)$</p> <p style="padding-left: 80px;">Then</p> <p style="padding-left: 120px;">3.2.1 Thread Ta is blocked.</p> <p>Step 4. If thread Ta and thread Tb are blocked</p> <p style="padding-left: 40px;">Then</p> <p style="padding-left: 80px;">4.1 Execute <i>def2</i> and <i>use</i> consecutively as the next operations.</p> <p style="padding-left: 80px;">4.2 Select the next operations non-deterministically until the concurrent program terminates.</p> <p style="padding-left: 80px;">4.3 Terminate this algorithm.</p> <p style="padding-left: 40px;">Else</p> <p style="padding-left: 80px;">4.1 Repeat from step 2.</p>

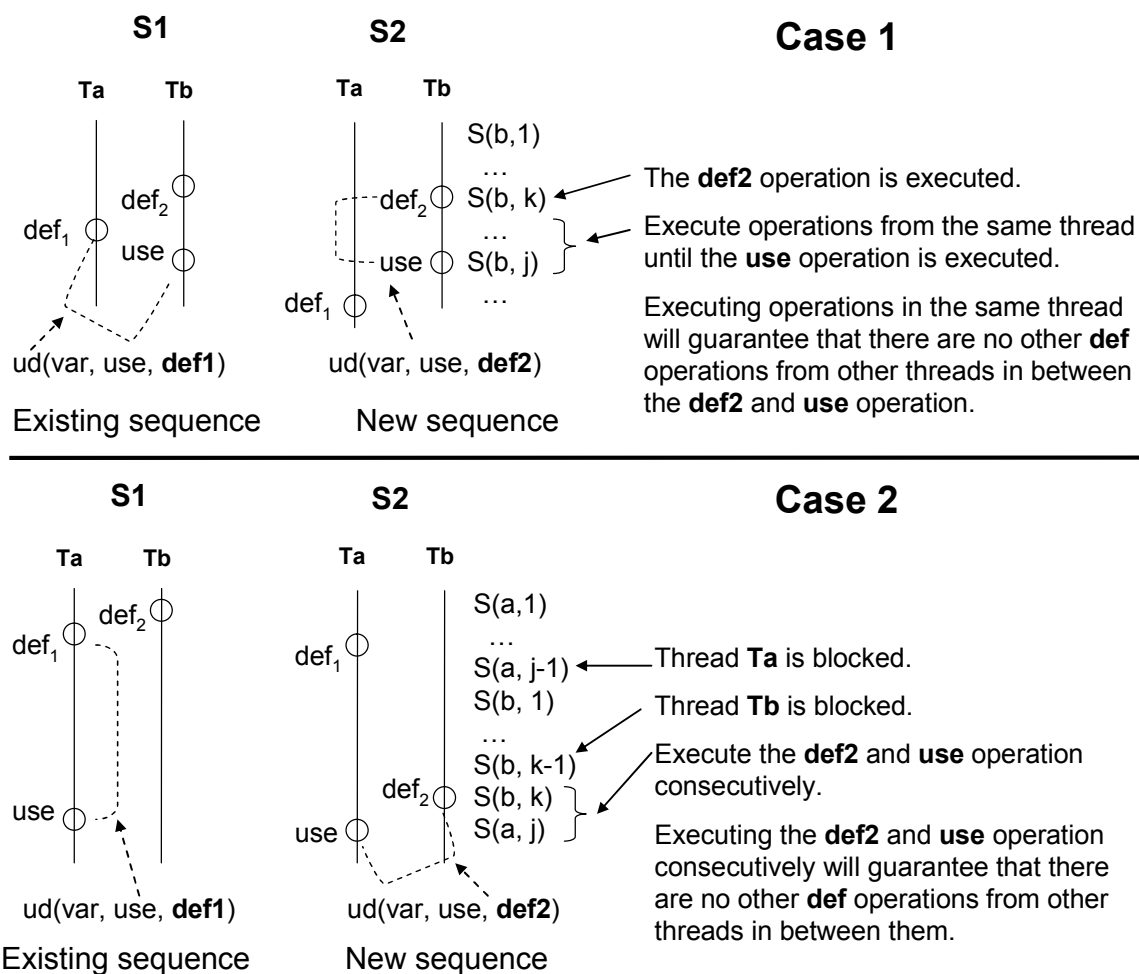


Figure 53. An example of test case generation for different cases

An example of case 2:

- From Figure 33: *S1* is T1:1: $x = -3$, T1:2: $y = 2$, T1:3: $n = x+y$, T1:4:if($n < 0$), T1:5:..., T2:20: $x = 10$, T2:21:..., T2:22:..., T2:23: $ref2 = \text{new Object}()$, T2:24:..., T2:25:print x , T2:26:..., T2:27: $ref2.credit = 7$, T3:30: $ref2 = ref1$
- Figure 52: Let *dg* be the concurrent dependency graph derived from the existing sequence *S1*.
- From Table 9: the set of guidelines = { $gu1 = \{ ud(x, 3, 1) \}$, $gu2 = \{ ud(x, 3, 20) \}$ } is derived from the concurrent dependency graph *dg* in Figure 7(b).

This example falls into case 2 because the *use* and *def2* in *gu2* are in different threads. Figure 54 illustrates the test case generation. The sequence for *S2* is T1:1: $x = -3$, T1:2: $y = 2$, T2:20: $x = 10$, T1:3: $n = x+y$, T1:4:if($n < 0$), T1:5:..., T2:21:..., T2:22:..., T2:23: $ref2 = \text{new Object}()$, T2:24:..., T2:25:print x , T2:26:..., T2:27: $ref2.credit = 7$, T3:30: $ref2 = ref1$.

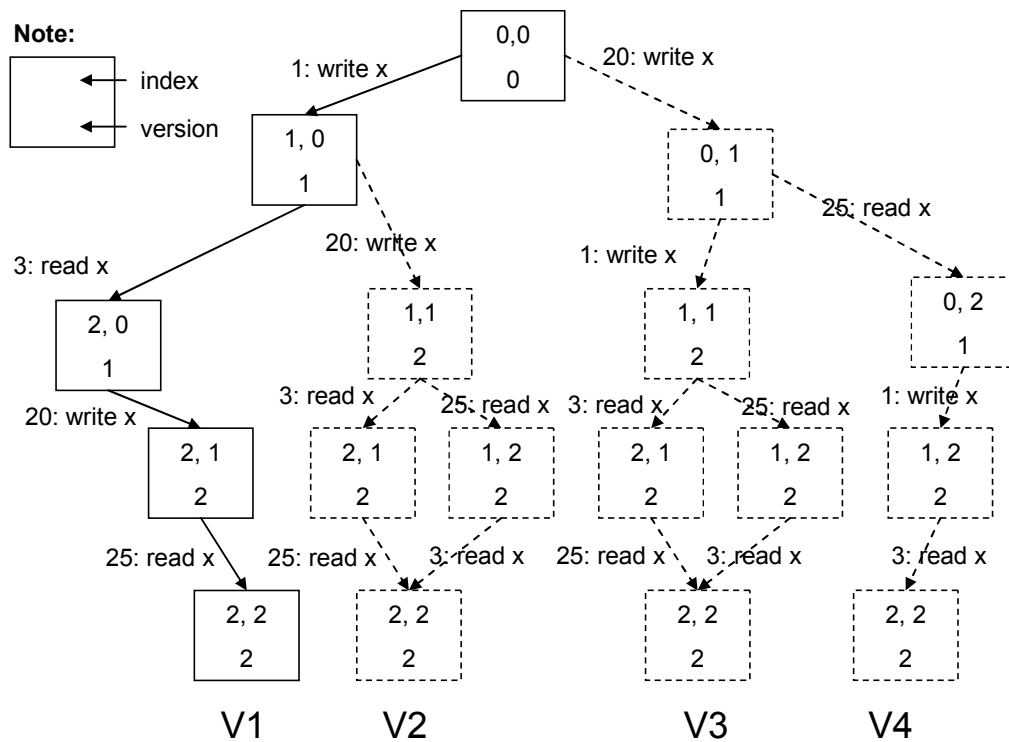


Figure 55. Example of a variant graph.

The variant graph in Figure 55 generates four test cases, but some of them are redundant. From the set of guidelines in Table 8 or Table 9, our proposed method identifies that only two test cases are required. Table 10 shows different values of variables when executing different execution-variants. The execution-variants $V1$ and $V3$ have the same truth value for the branch in line 4. It is sufficient to test only one of them with respect to exploring different execution paths caused by the branch. They differ in the values of the variable x in line 25, but the truth value of the branch in line 4 is the same. A similar situation happens for the execution-variants $V2$ and $V4$. Suppose that the execution-variant $V1$ is executed when the program is first tested. The execution-variant $V2$ can be created from $V1$ by replacing the use-define $ud(x, 3, 1)$ with $ud(x, 3, 20)$.

Table 10. Different values of variables among different execution-variants

Execution-variant	3: read x	3: read y	3: write n	4: if ($n < 0$)	25: read x
$V1$	-3	2	-1	<i>True</i>	10
$V2$	10	2	12	<i>False</i>	10
$V3$	-3	2	-1	<i>True</i>	-3
$V4$	-3	2	12	<i>False</i>	10

Figure 54 shows how to generate only the required test cases based on the guideline from the proposed concurrent dependency graph.

5.3.6 Generating Test Cases to Check Consistent Locking for Access through Reference Variables

The basic premise suggested in the method that we have proposed so far is that covering an execution path is sufficient to detect a race or no-race condition by checking consistency locking in that execution path, independently of variable values. In some cases, this might not be sufficient, since the value of the lock object itself may depend on the data flow and, theoretically, on the interleaving, as illustrated in Figure 56(a). This situation may be considered as a race condition.

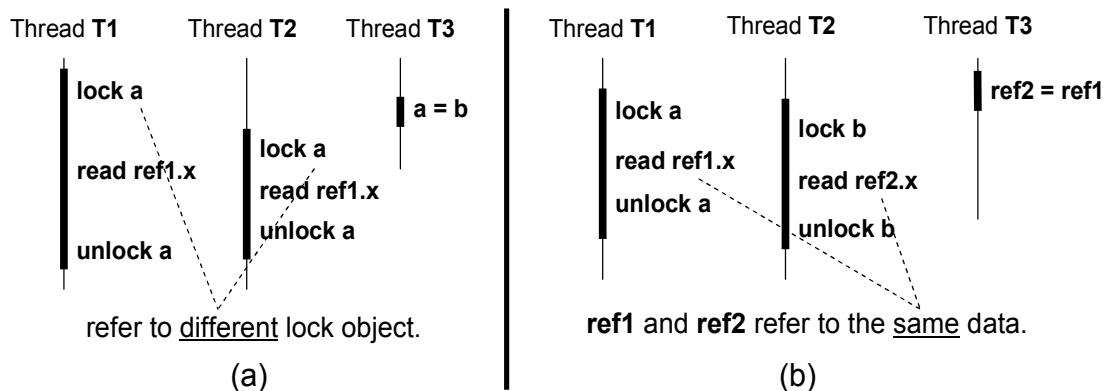


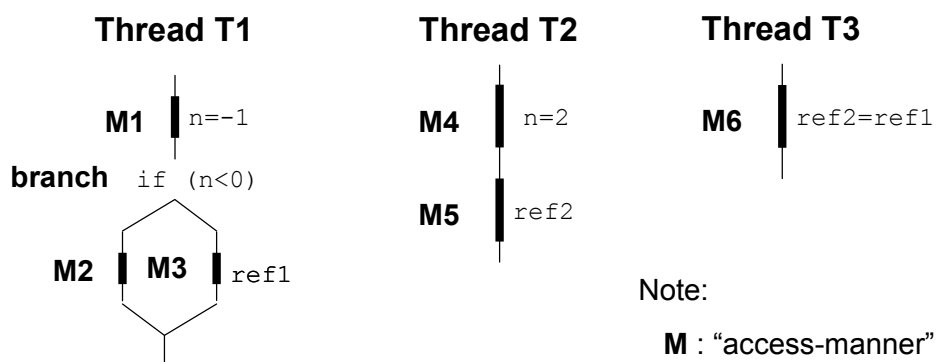
Figure 56. Example of lock variables (a) and reference variables (b)

Similar problems may also arise when different shared reference variables (a

pointer in C or an object reference in Java) actually refer to the same data as illustrated in Figure 56(b). Threads acquire a consistent lock for accessing *ref1* and *ref2*, but actually they are referring to the same data depending on the interleaving of the assignment $ref2 = ref1$ in the thread *T3*. On the other hand, even when the same reference variable is shared between threads, the actual data referred to may not necessarily be shared.

This subsection shows that our proposed concurrent dependency graph can also generate test cases for detecting race condition caused by lock variables or reference variables. A more complicated example involving a branch is illustrated below:

- In Figure 57, the truth value of the branch depends on the order of executions of the access-manner *M1* and *M4* as seen in Figure 57(a) and Figure 57(b). In the event that the branch takes a different execution path, the error might not be detected.
- The reference variables *ref1* and *ref2* can refer to the same or different objects depending on the order of executions of the access-manners *M5* and *M6*, as shown in Figure 57(b) and Figure 57(c). A race condition arises in execution 3 in Figure 57(c) in the event that the access-manner *M3* and access-manner *M5* are not protected by the same lock. A race condition cannot be detected in execution trace 1 or 2, but can be detected in execution trace 3.



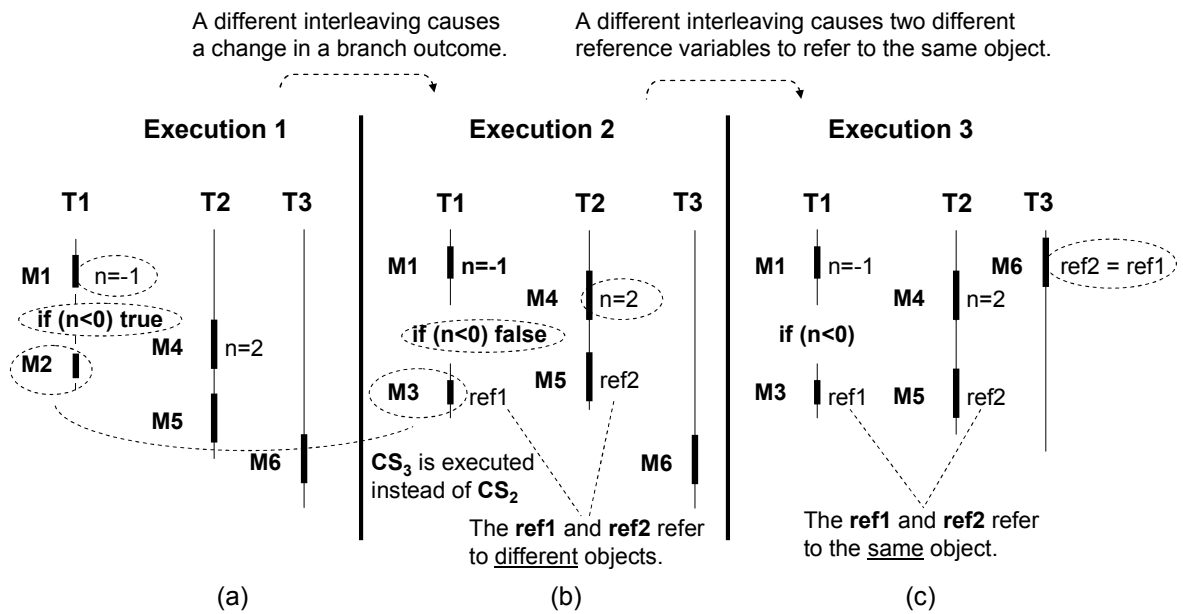


Figure 57. Examples of three executions with different interleavings

5.3.7 Generating Test Cases: Traversing a Concurrent Dependency Graph of an Access-Manner

This subsection explains how to generate different interleavings to check whether accesses through reference variables in an access-manner have consistent locking. In Figure 30, the *define_set* for the read operation to *ref2* in *M3* for *pair2* contains two members, hence its value might be affected by different interleavings. Figure 58 shows an example of a concurrent dependency graph for the access-manner *M3* in Figure 30. The root node contains the statements from the access-manner *M3*. We will show an example of how to traverse the concurrent dependency graph of the access-manner *M3* in Figure 58 to generate test cases.

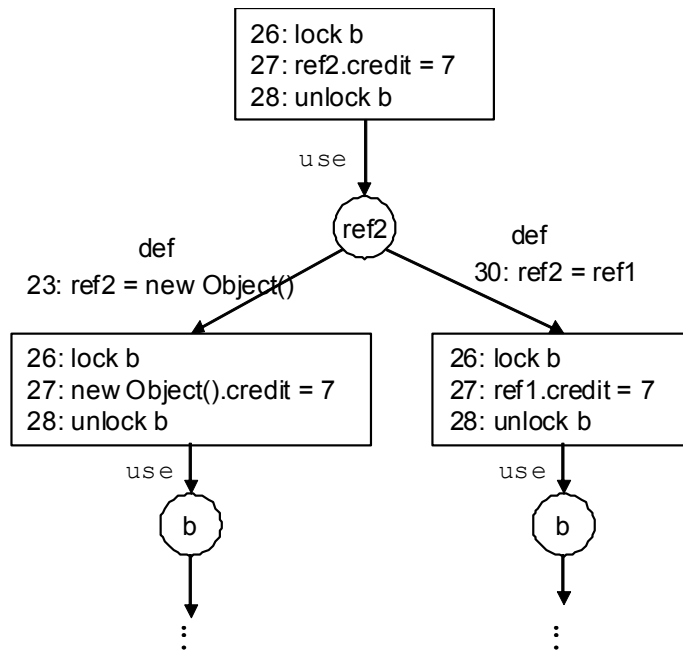


Figure 58. An example of a concurrent dependency graph for the access-manner $M3$ in Figure 30

Table 11 shows the results of traversing the concurrent dependency graph in Figure 58 by applying Algorithm 9. Let us assume that the execution in Figure 30 is obtained when the program is first tested, and we call it execution-variant $V1$. Its interleaving satisfies the use-define $ud(ref2, 27, 23)$. The execution-variant $V2$ is used as the next test case as shown in Figure 59. Its interleaving satisfies the use-define $ud(ref2, 27, 30)$. The next subsection explains how to create the execution-variant $V2$ effectively from the concurrent dependency graph in Figure 58.

Table 11. A set of guidelines for generating test cases for testing pair2 in Figure 11

No.	Guideline	execution -variant	Test result
1	$gu1 = \{ ud(ref2, 27, 23) \}$	$V1$	No race condition, because $ref1$ and $ref2$ refer to different objects.
2	$gu2 = \{ ud(ref2, 27, 30) \}$	$V2$	Race condition for accessing $ref1$, if lock a and lock b refer to different lock objects.

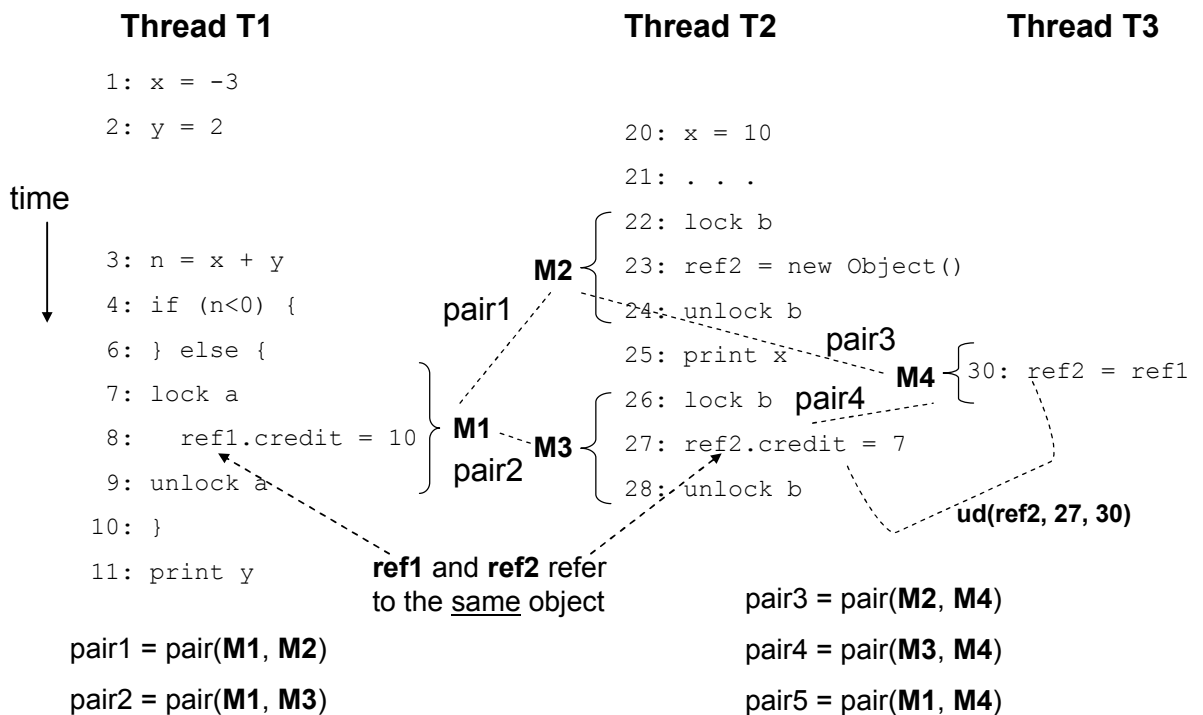


Figure 59. An example of a test case execution for execution-variant $V2$

5.3.8 Generating Test Cases for Checking Consistent Locking of an Access-Manner

Based on Table 11, the execution-variant $V2$ can be generated from execution-variant $V1$ by changing the *define* operation for the *use* operation of variable *ref2* in line 21.

- The **guideline** for the current execution-variant $V1$: { *ud(ref2, 27: ref2.credit = 7, 23: ref2 = new Object())* }
- The **guideline** for the target execution-variant $V2$: { *ud(ref2, 27: ref2.credit = 7, 30: ref2 = ref1)* }

Generating the execution-variant $V2$ applies to case 2 because the *use* operation is in a different thread from the target “def” operation. Therefore, Algorithm 11 applies for this case.

- *def_{base}* : 23: *ref2 = new Object()*
- *def_{target}* : 30: *ref2 = ref1*
- *use* : 27: *ref2.credit = 7*

Figure 59 shows an example of the execution trace that satisfies the guideline *gu2*.

5.4 Reducing the Effort Involved in Checking Race Conditions

Effort involved in checking race conditions can be reduced by utilizing previous check results. Suppose we have the first execution with the set of execution paths *PATHS1* which is already checked. Then we execute the next test case which results an execution with the set of execution paths *PATHS2*. The effort for checking the set of execution paths *PATHS2* can be reduced as follows:

- In case *PATHS1* and *PATHS2* are in the same race-equivalent group: No need to check race conditions for *PATHS2*.
- In case *PATHS1* and *PATHS2* are not in the same race-equivalent group: Check only some parts of execution traces affected by a new test case. Figure 60 shows the idea for reducing the effort involved in checking race conditions:
 - The $pair(CS_1, CS_A)$ and $pair(CS_2, CS_A)$, indicated by **, exists in the previous execution, hence they do not require checking for race conditions.
 - However, the $pair(CS_4, CS_A)$, indicated by *, does not exist in the previous execution, hence it requires checking for race conditions.

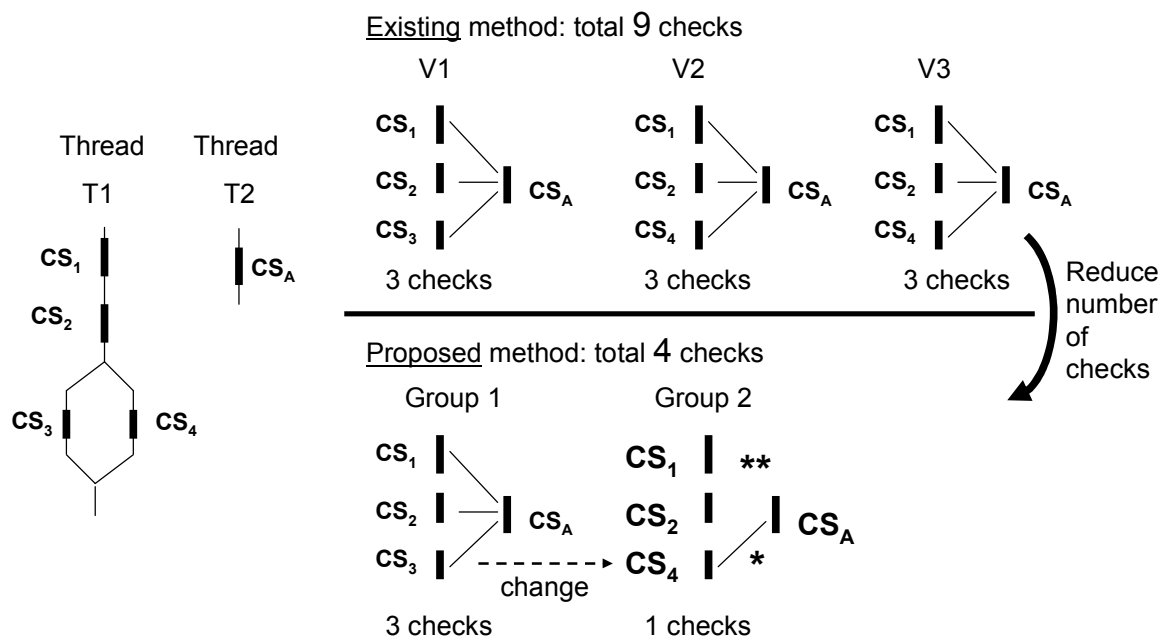


Figure 60. Reducing the effort involved in checking race conditions

5.4.1 Executions in the Same Race-Equivalent Group: No Need to Check Race Conditions

It is possible to reduce the amount of work for checking errors in different execution paths by grouping different execution paths from different executions with the same set of manner-manners of access to shared variables into the same race-equivalent group. All members of the same group are said to be race-equivalent. All members within the same group will have the same set of access-manners. By this, only one execution path from each race-equivalent group need to be checked. This method reduces the task for checking errors in different execution paths by eliminating all the execution paths belonging to the same race-equivalent group except one. During the test iteration, the already checked race-equivalent group is recorded to avoid repeating checking the combination of execution paths belong to the already checked race-equivalent group.

Determining no-race can contribute to reduce the computation effort of finding race conditions and deadlocks. This reduction is applied during exploring different execution paths due to different interleavings. The reduction is possible by followings:

1. Logging and detecting for race conditions for the set of access-manners appeared in the past.
2. As the execution path of the target system progresses, execution traces for other test cases are logged.
3. If a new execution trace has the same set of access-manners as one of the logged, then we do not need to repeat race conditions detections because the same sequence of lock/unlock and read/write operations to shared variables has already been tested. In other words, they are in the same race-equivalent group.

This is true for any execution paths including loops. If it is found that looping does not change sequences of lock/unlock and read/write operations to shared variables, we do not need to check race conditions for each execution of the loop. One test is enough for the entire loop. In exploring execution paths due to different interleavings, any execution paths having the same sequences of lock/unlock and read/write operations to shared variables can be grouped into the same

race-equivalent group. However, the fact that some execution paths belong to the same race-equivalent group does not necessarily imply that the future computations of them will be the same.

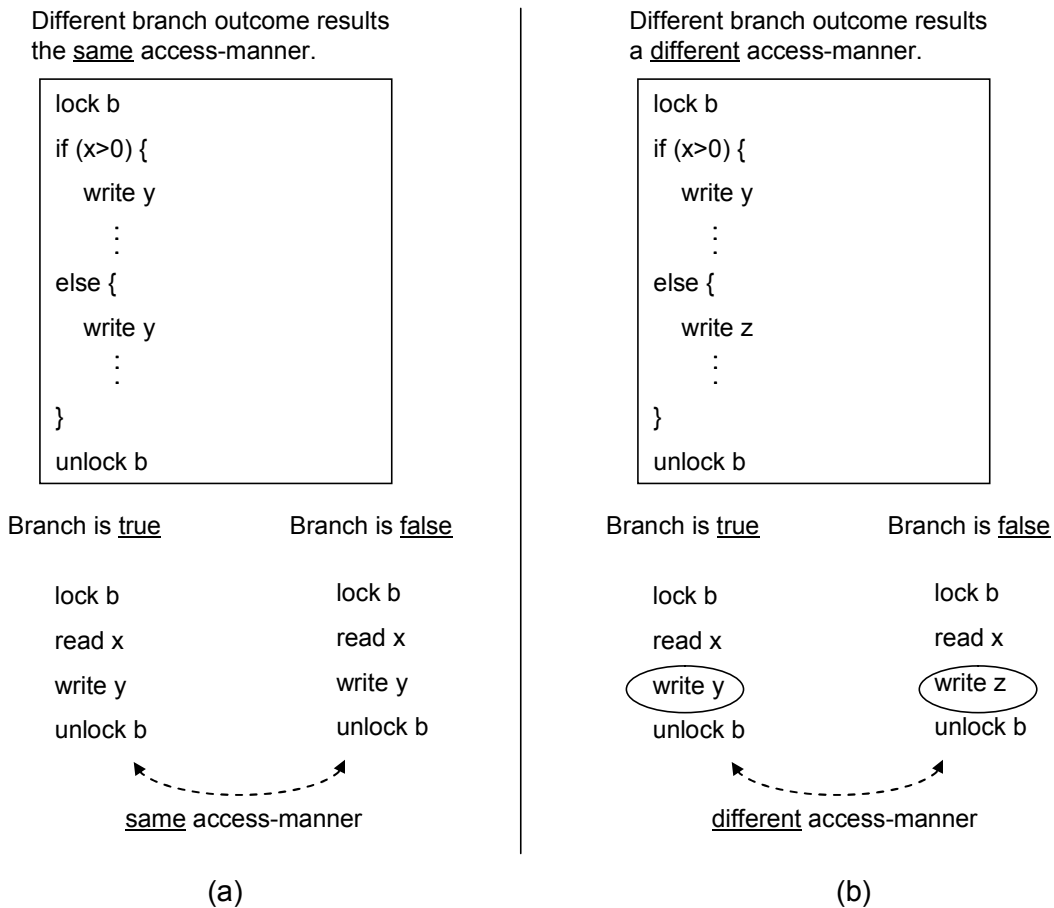


Figure 61. Examples of the same and a different access-manner caused by a branch

If the branch path *b true* and the branch path *b false* have the same access-manner to shared variables, then two executions with different execution paths caused by different branch outcomes for the branch *b* are said to be equivalent (see an example in Figure 61(a)).

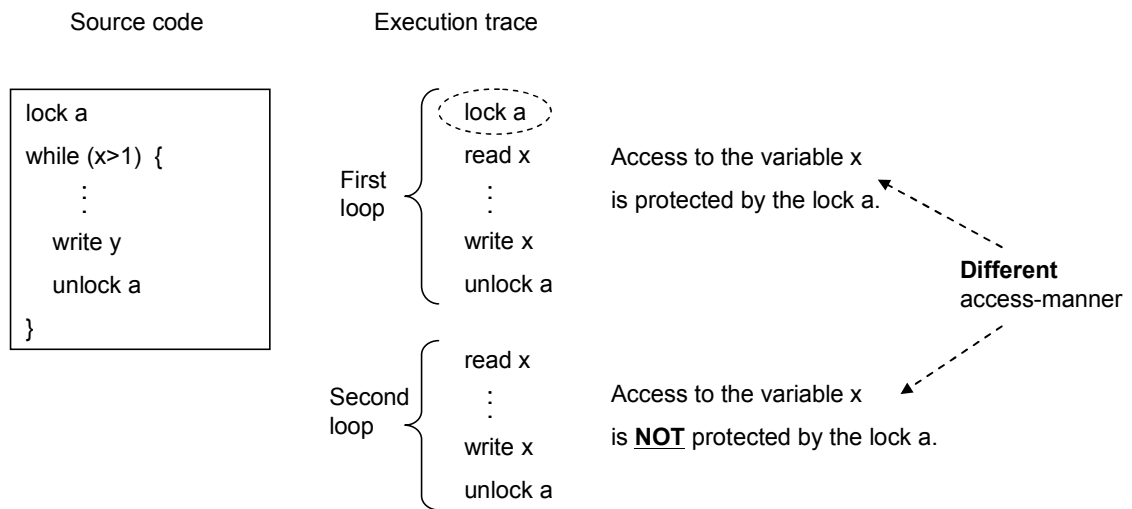


Figure 62. Different access-manners caused by a loop

Similarly, if the access-manner to the shared variables are all the same in $l_1, l_2, l_3, l_4, l_5, l_6, \dots, l_\omega$, then different execution paths caused by the different loop iterations for a loop l will be in the same race-equivalent group. This property is useful to avoid checking a long or an infinite loop. In the case of an infinite loop, there might be infinite execution path, but we group different execution paths caused by loop if the access-manner to the shared variables is the same. In this case we need only to consider the combination of branches and loops. When there are only finite numbers of branches and loops, then their combinations will also be finite. Figure 62 shows an example of a loop in which the access-manner in the first iteration is different from the second one. The rest of the iterations will have the same access-manners as the second one. Figure 63 shows more complicated examples where the access-manners are affected by a branch and a loop.

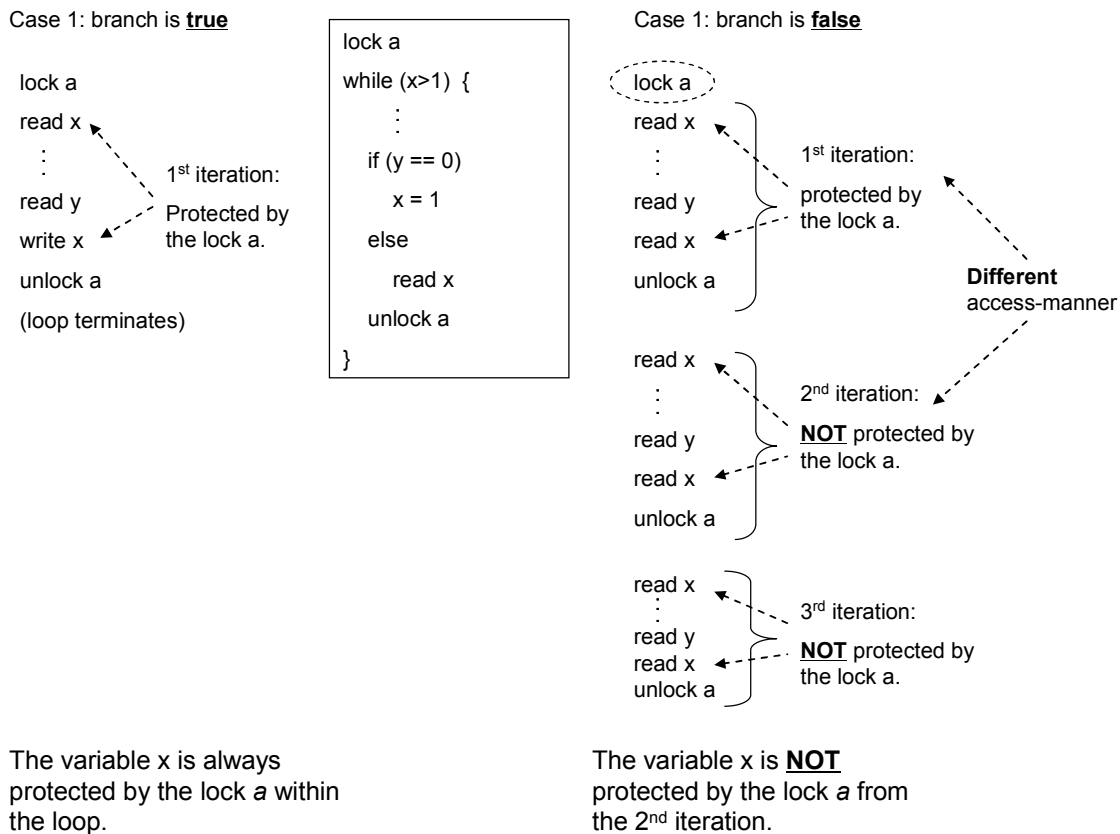


Figure 63. Different access-manner caused by a branch and a loop

Theorem 1. All race conditions can be detected within the race-equivalent groups.

Proof:

Race conditions happen when there exists a possible combination of execution paths from different threads which are not well-formed. We need to proof that for any possible combination of execution paths, there exist one race-equivalent group in which has the same access-manner to shared variables. Hence, if the possible combination of execution paths has race conditions, it will also be detected at the corresponding race-equivalent group.

Assume that P is a possible combination of execution paths and there is no race-equivalent group which has the same access-manner to shared variables as in P . Two concurrent executions with the same access-manner to shared variables for all their threads will be grouped into the same race-equivalent group. Therefore, the set

of race-equivalent groups will contain all possible combinations of access-manner to shared variables from all threads.

Since P is a possible combination of execution paths, there exist one race-equivalent group in which the access-manner to shared variables for the first, second, third, ... , N -th thread are the same. Then P should be in one of race-equivalent group. This contradicts our assumption that there is no race-equivalent group which has the same access-manner to shared variable as in P . Therefore, for any possible combination of execution paths, there exists one race-equivalent group in which has the same access-manner to shared variables. Q.E.D. ■

5.4.2 Executions in a Different Race-Equivalent Group: Check Only Some Parts of Execution Traces Affected by A New Test Case

When a new test case is executed, only concurrent-pairs of access-manners whose access-manners are affected by the new test case have to be re-checked for race conditions. In this way, the effort for checking race conditions is reduced. The following discussion explains how to identify the access-manners which are affected by a new test case.

5.4.2.1 Conditional Statements in a Branch

A different interleaving might change branch outcomes which can, in turn, change the sequences of lock/unlock and read/write operations to shared variables. In the event that a test case is created based on a conditional statement of a branch, then only the access-manners affected by the change of the branch outcomes have to be re-checked for race conditions. Let $op(br, true)$ be the set of operations executed only when the conditional statement in a branch br is *true* and let $op(M)$ be the set of operations within an access-manner M . When the outcome of the branch br changes from *true* to *false*, we have to check only race conditions in concurrent pairs of access-manners involving access-manner M , where $op(br, false) \cap op(M) \neq \emptyset$. Also, when the outcome of branch br changes from *false* to become *true*, a similar rule applies. For example, let us assume a test case is created based on the branch in line 4 in Figure 59. If the branch has changed its outcome from *true* to *false*, then the

access-manner affected by the test case is $M1$. Therefore, we have to check only those race conditions for the concurrent-pairs related to the access-manner $M1$; these are $pair1$, $pair2$, and $pair5$.

5.4.2.2 Assignment of Lock Variables or Reference Variables within an Access-Manner

Different interleavings might change the assignment of lock variables or reference variables within an access-manner. If a test case is created based on an access-manner M_a , then we have to check only those race conditions for the concurrent pair of access-manners $pair(M1, M2)$ where $M1 = M_a$ or $M2 = M_a$. The test cases in the example of Table 11 are created based on the access-manner $M3$ from Figure 30. Only $pair2$ and $pair4$ have to be re-checked using a race detector because they are related to the access-manner $M3$. On the other hand, since $pair1$, $pair3$ and $pair5$ are not related to the access-manner $M3$, they are not affected by the test case. Hence, there is no need to re-check race conditions among them (see Figure 59).

When a loop contains an access-manner, each iteration can generate a concurrent-pair of access-manners. In the case of an infinite loop, the number of concurrent-pairs of access-manners can be infinite. However, in some cases the concurrent-pairs generated in each iteration could be the same as in the previous one. In such cases, there is no need to check for all the iterations. In this way, the effort involved in checking race conditions during the test can be reduced. We will show an example of this in subsection **6.6.3 Experiment 3: jNetMap**.

Chapter 6. Implementation and Experiments

This chapter describes an implementation of the proposed method in Java and shows some results of experiments.

6.1 Lock Mechanism in Java

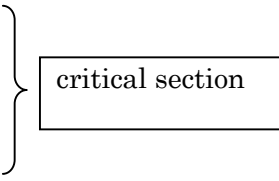
In Java, the lock mechanism is implemented as follows:

- Lock objects
- Synchronized methods
- Synchronized statements

6.1.1 Lock Objects

A lock object is an actual object that represents a lock. One example of an implementation class is *ReentrantLock*. A lock is acquired by calling the *lock()* method and released by calling the *unlock()* method. The execution between them becomes a critical section.

```
...
private Lock scoreLock = new ReentrantLock( );
...
public void method1() {
    ...
    try {
        scoreLock.lock( );
        ...
    } finally {
        scoreLock.unlock( );
    }
}
```



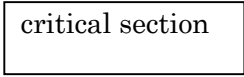
6.1.2 Synchronized Methods

A synchronized method is a method which has a “synchronized” keyword in its method declaration. There are two kinds of locks when using a synchronized method:

- Class lock: a synchronized method is defined as a static method.
- Object lock: a synchronized method is defined not as a static method. It uses the object instance as the lock object specified from an object reference or using the keyword *this* to specify its own object.

A thread that wants to execute a synchronized method must first obtain the lock. The lock is released after it returns from the synchronized method. The execution within the method becomes a critical section.

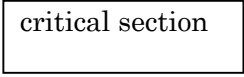
```
public synchronized void method2() {  
    . . .  
}
```



6.1.3 Synchronized Statements

A synchronized statement is similar to a synchronized method, but synchronized statements must specify an object as shown below.

```
public void method3() {  
    . . .  
    synchronized(this) {  
        . . .  
    }  
    . . .  
}
```



For the three mechanisms above, a lock is being acquired irrespective of which syntactic approach is used.

6.2 Interrupt as a Thread in Java Program

It is not always possible to translate existing programs to support

interrupt-as-a-thread principle for debugging/testing as suggested in this dissertation. In Java, however, it is rather easy. We can simply create a (new) thread for interrupt handling when interrupt comes. Preserving interrupt processing order, for example, a series of interrupt from a keyboard or a series of interrupt from the same ATM, can be achieved using joint point. In Java, invoking `t.join()` for a thread `t` suspends the caller until the target thread `t` completes [Lea99]. Therefore when a series of interrupts come, the execution of a later interrupt handling can be suspended until the previous interrupt handling thread completes its execution.

If we design interrupt handling based on an interrupt dispatcher, we can use Executor interface [Oaks04] for defining thread pools for interrupt processing. In fact, the recent `java.util.concurrent` package provides the code for such an implementation. An interrupt handling and the task required for it are executed in a newly created thread.

6.3 Tracing

We use AspectJ [Gradecki03] for tracing Java multi-threaded concurrent programs. It is an aspect-oriented extension to the Java programming language. AspectJ was chosen because of its flexibility to trace the necessary data from an execution of a program. Other means of tracing can also be used as long as they can capture the necessary information about lock sequences, access to shared variables, and branches.

We capture the necessary information from an execution of a program using the concept of “pointcut”, “advice”, and “reflection” in AspectJ. Note that they are specific terms for AspectJ. Here, we describe only the general idea of tracing using AspectJ:

- Pointcut: specify locations within an execution of a program where necessary information needs to be captured. We do not explicitly specify the locations in term of line of code; instead we specify wildcards so AspectJ will take a trace when any locks are acquired or released, or any shared variables are accessed.
- Advice: a piece of code to be executed for each *pointcut*.
- Reflection: getting trace information from program execution, for example about locks’ acquisition or operations on shared variables. *Reflections* are written

within an *advice*. Besides obtaining the shared variables or object references' name, it is also possible to know which actual data is referenced by specifying its object id in order to precisely determine where a race condition has actually happened.

- For each of the *pointcut*, we write a corresponding additional a piece of code to be executed, called advice to get the information about locks or shared variables using AspectJ *reflection*. Within the advice, we use reflection to get the necessary information for tracing, such as a shared variable's name.

For detecting a branch or loop, the line of code (loc) in the source code is recorded when a variable is accessed, and then later compared to the source code to determine whether it is in an if-statement or a loop. The AspectJ codes necessary for tracing are written in AspectJ files, which are separated from the target programs. The AspectJ files need to be weaved with the target source code. The information necessary to be traced is a sequence of lock/unlock, read/write operations to shared variables, branches and loops. The pointcut definition for lock, call to synchronized method, read access and write access to shared variables are `call(void Lock.lock())`, `call(synchronized * *.*(..))`, `get(* *.*.)`, `set(* *.*.)`. The overhead incurred by tracing differs case by case depending on the occurrence of locks' acquisition and read/write operations to shared variables.

6.4 Deterministic Testing

For controlling a program execution, Java code instrumentation [Baur03] can be used. Thread switch is realized by unblocking the next thread in the schedule followed by blocking all other threads including the current thread. A lock object is assigned to each thread. Methods *wait* and *notifyAll* are used to implement block and unblock operations that suspend and resume execution of a thread.

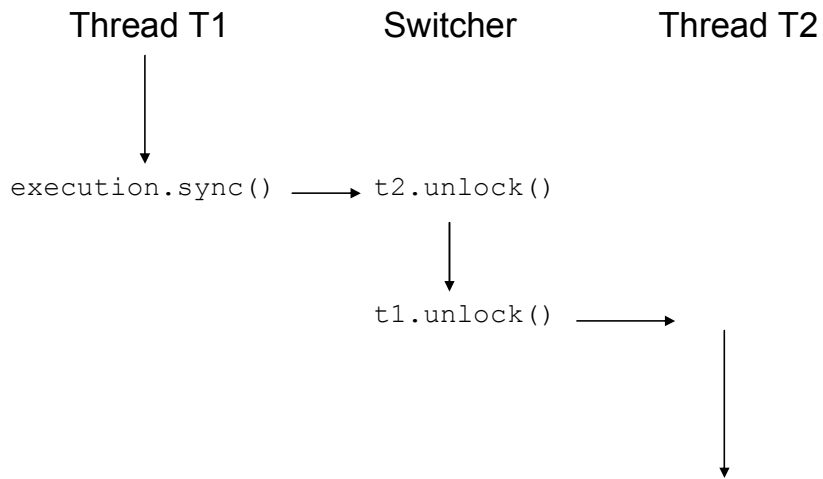


Figure 64. Control transfer from thread *T1* to *T2*

6.5 Implementation Diagram

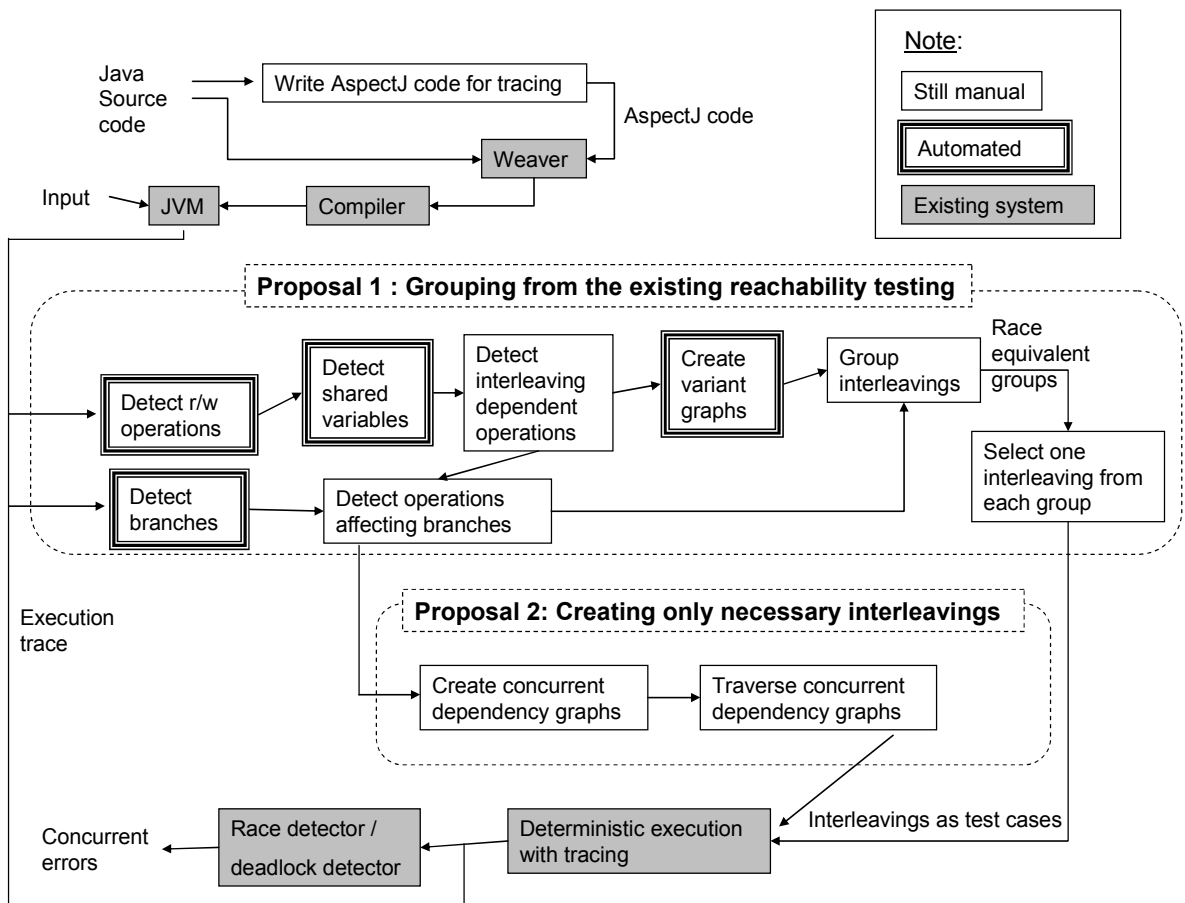


Figure 65. Implementation

6.6 Experiment Results: Test Case Reduction

We use some Java open source programs for network control and database management in the experiments, because these programs are usually designed to be multi-threaded. The effectiveness of the proposed method for detecting race condition depends on the structure of the program. Some concurrent programs have only read-shared variables [Savage97], for example BlueJ [BlueJ09] and Baralga [Baralga10]. The values of read-shared variables are only assigned once during initialization and they are not affected by different interleavings. Hence, they also do not have branches that are affected by different interleavings. The concurrent errors in such program are always reproducible because there will be no change in the sequence of lock/unlock and read/write operations to the shared variables in each thread. They can be easily detected using existing dynamic race detection tools. Debugging such programs is relatively easy by treating them as similar to sequential programs. In such easy situations, the effectiveness of the proposed method for detecting or reproducing race conditions is the same as the existing methods. The proposed method is superior in the case where race conditions are difficult to be detected or reproduced. Figure 66 shows the effectiveness of the proposed method compared to the existing methods.

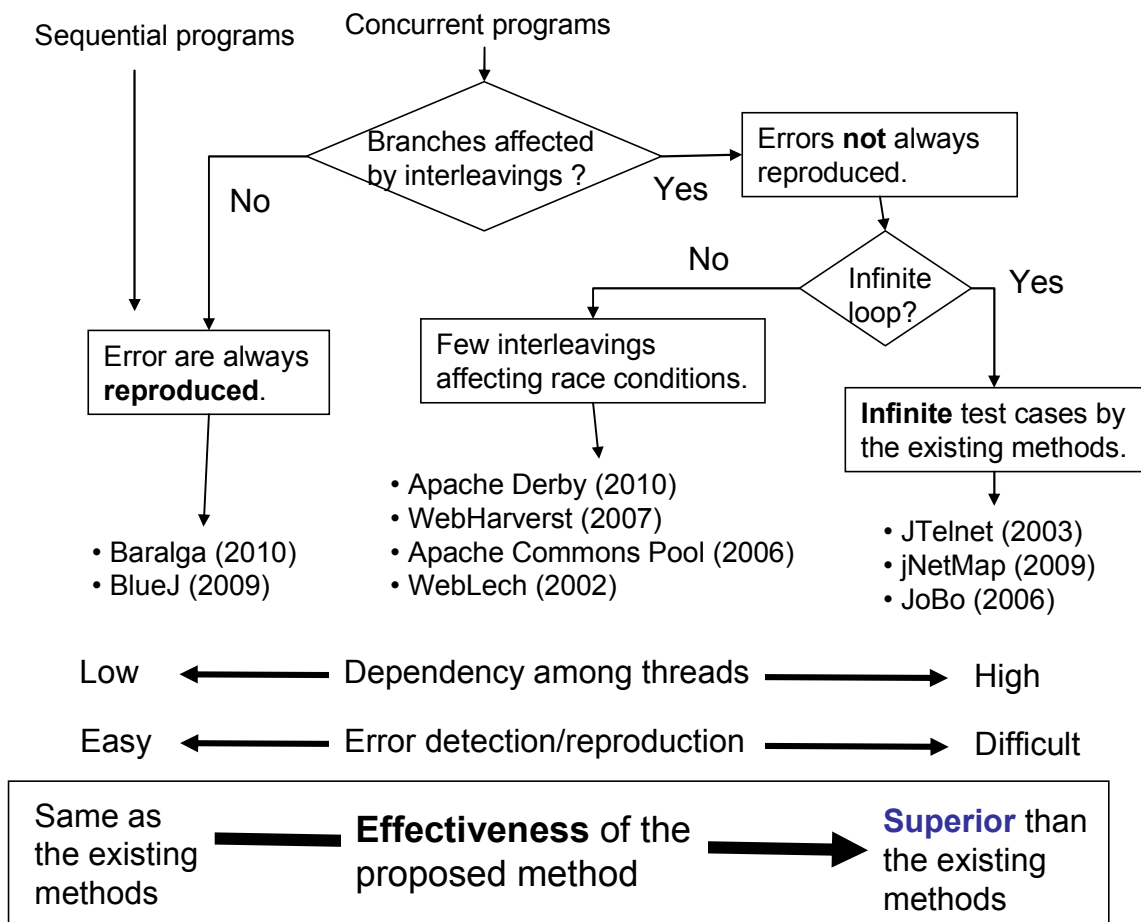


Figure 66. The effectiveness of the proposed method

The objective of the experiments is to show the efficiency of the proposed method for reducing the number of test cases in detecting race conditions. We compare the number of test cases against an existing test case reduction method based on the Thread-Pair-Interleaving (TPAIR) criterion [Lu07]. The results are summarized in Table 12. For a fair comparison, we allow only the same input for both methods. In these experiments, we measure the reduction in the number of different interleavings used for test case generation. We ignore different orders of read-shared variables. A read-shared variable is a variable that it is written during initialization only and becomes read-only thereafter [Savage97]. Its value is determined only by the input and it does not change during an execution of a program. As such, it can be ignored during test case generation because different interleavings do not affect its value.

Table 12. Summary of experiment results

	1. Apache Commons Pool	2. JTelnet	3. jNetMap	4. JoBo	5. Apache Derby
Program size (Kloc)	123	5	3	45	292
Trace size (KB)	35	1638	201	87500	72800
Number of threads	3	3	6	4	5
Number of shared variables	33	7	10	4	33
Number of branches executed from trace	17	329	31	121665	14164
Number of branches affected by interleaving	1	0	1	1	29
Number of test cases in TPAIR	23	66	Infinite	Infinite	1453539
Number of test cases in proposed method	2	1	4	1	58

6.6.1 Experiment 1: Apache Commons Pool

In Experiment 1, we use a generic object-pooling library called Apache Commons Pool [ApachePool06]. Some race conditions have been reported in related work [PLDI06] [Naik06]. Most of the race conditions are easy to detect in that they can be found by simply re-executing the program and using an existing dynamic race detector. Our proposed method is intended to find race conditions that are difficult to detect. This is because such race conditions are affected by branches and different interleavings. There are 160 race conditions reported at [PLDI06]. We observed 15% of them as being difficult to detect. One possible example is shown in Figure 67.

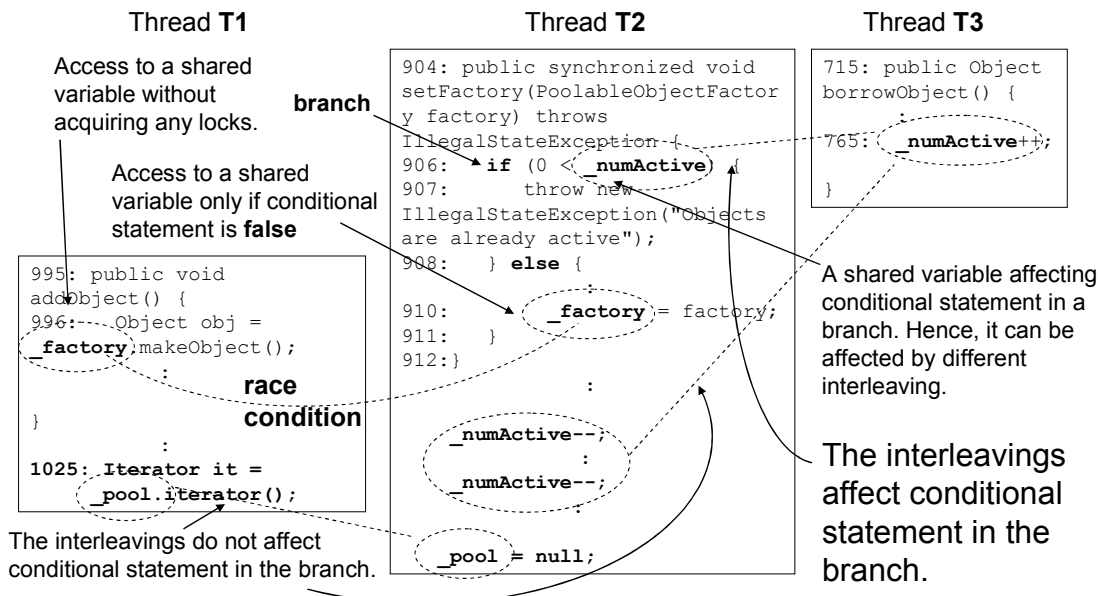


Figure 67. An example of a race condition that is difficult to detect

There is a race condition in Figure 67 between thread $T1$ and thread $T2$ when accessing the shared variable $_factory$, because thread $T1$ does not acquire any locks. However, it happens only when the conditional statement for the branch in thread $T2$ is false. Furthermore, the conditional statement depends on the value of shared variable $_numActive$ which is affected by the interleaving with thread $T3$. Figure 68 shows read and write accesses to the shared variables for the execution of the first test case, in which the race condition is not reproduced. Using Algorithm 3, we calculate the following:

$$BranchRelUD(b, V) = \{ (_numActive, 906, 765) \}.$$

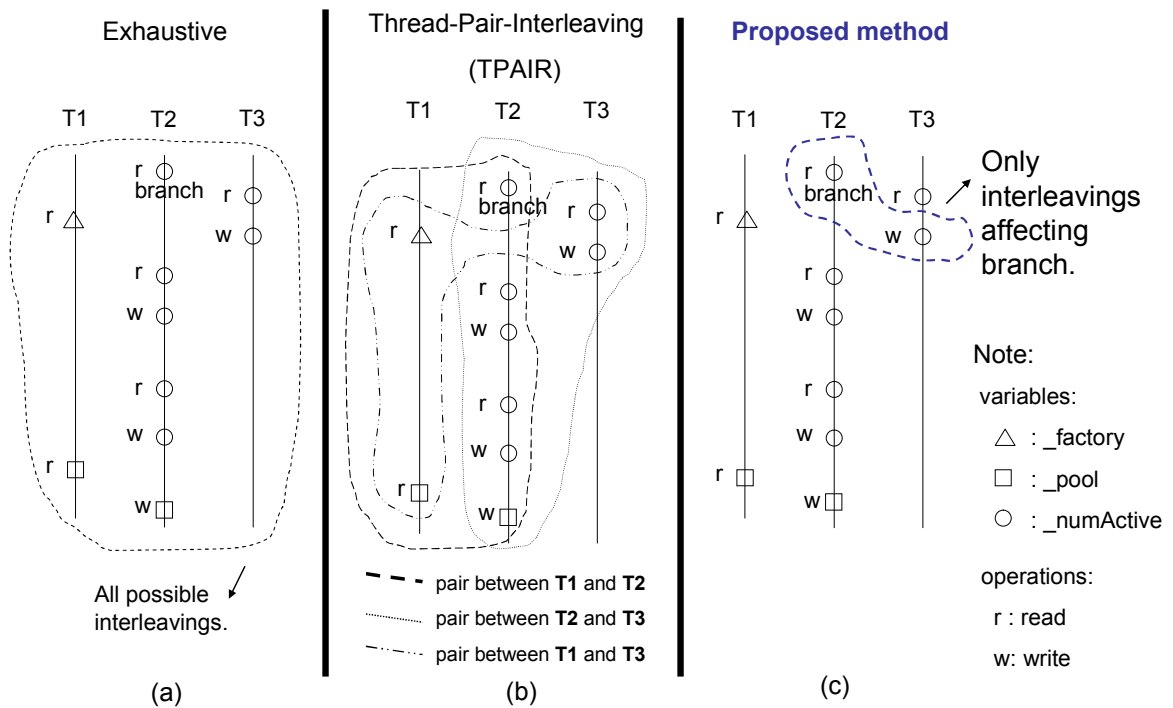


Figure 68. A comparison of exhaustive, TPAIR, and the proposed method

Our proposed method generates two test cases based on Table 13. Group $g2(b)$ will cause the conditional statement to become *false*, so the error will be reproduced.

Table 13. Grouping of test cases for experiment 1

Groups	Set of use-defines affecting branch b
$g1(b)$	$\{ (_numActive, 906, \dots) \}$
$g2(b)$	$\{ (_numActive, 906, 765) \}$

We compare our proposed method against an existing test case reduction method based on the Thread-Pair-Interleaving (TPAIR) criteria [Lu07]. Instead of generating different interleavings among all threads, TPAIR generates only different interleavings for every pair of threads to reduce the number of test cases. This reduction is based on the fact that most concurrency bugs are caused by the interaction between two threads, instead of all threads, as explained in the previous error detection work [Savage97] [Lu06]. This also happens for the race condition between thread $T1$ and thread $T2$ when accessing shared variable $_factory$ in Figure 67. Its reproduction depends on the branch in thread $T2$ whose conditional statement is affected by the interleaving between thread $T2$ and thread $T3$. However, not all different interleavings between those two threads will affect the reproduction of the race condition. For example, shared variable $_pool$ is affected by the interleaving between thread $T1$ and thread $T2$, but the race condition when accessing the shared variable $_pool$ will always be reproduced. Hence, it can always be detected by a race detector independent of the interleaving between those two threads. In this experiment, the reachability testing method produces 147 test cases, the TPAIR method produces 23 test cases, and our proposed method produces only 2 test cases for detecting the race condition.

In order to evaluate the feasibility, we performed several experiments by increasing the number of shared variables accesses for the same target program. Figure 69 indicates the increase in the number of test cases when the number of read/write operations to shared variables is increased. In order to reproduce the race condition, Figure 69 shows that our proposed method produces fewer test cases than test generation based on the existing TPAIR. In addition, error detection by TPAIR can be guaranteed only if the errors are caused by interleaving between two threads. In contrast, our proposed method can reproduce errors caused by interleavings from any number of threads. This is because our proposed dependency graph considers data flow from any threads that affect the conditional statement in a branch.

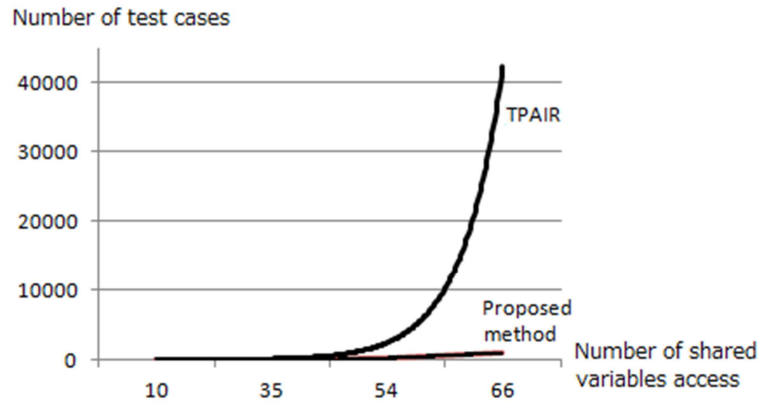


Figure 69. Comparison of numbers of test cases

6.6.2 Experiment 2: JTelnet

The JTelnet [JTelnet03] is a telnet client written in Java. Among the 7 shared variables, 6 of them are read-shared. Based on the data flow analysis, one branch is affected by a shared variable. This experiment shows that some interleavings will change the values of shared variables, but they might not affect the reproduction of race conditions. In such circumstances, the existing reachability testing and TPAIR methods will generate some test cases, but our proposed method generates no test case. The results are summarized in Table 14.

Table 14. Summary of experiment results for JTelnet

Method	Number of test cases	Description
Existing TPAIR	66	Test cases generated by TPAIR will affect only the values of shared variables in thread AWT-EventQueue-0, but will not affect any conditional statements for branches in thread <i>T2</i> (Figure 70)
Proposed method	1	Branches in thread <i>T2</i> are only affected by operations in the same thread. Therefore, the proposed method does not produce any other test cases because their outcomes will not be affected by a different interleaving.

Thread T-AWT-EventQueue-0 Updating GUI

```

public void paint(Graphics g) {
    :
317: g.setColor(new Color(screenbg[yloc][xloc].
    getRGB() ^ 0xFFFFFF));
318: g.fillRect(3+xloc*charOffset, 2+yloc*
    lineOffset, charOffset, lineOffset);
319: g.setColor(new Color(screenfg[yloc][xloc].
    getRGB() ^ 0xFFFFFF));
320: g.drawChars(screen[yloc], xloc, 1, 3*xloc*
    charOffset, topOffset+yloc*lineOffset);
    :

```

shared variable: xloc

Thread T2 Receiving input from socket

```

while (true) {
    try {
        if ((read=sIn.read(buf)) >= 0) {
81:     if (xloc >= columns) {
            :
            :
114:    screen[yloc][xloc] = (char) c;
115:    screenfg[yloc][xloc] = fgcolor;
116:    screenbg[yloc][xloc] = bgcolor;
117:    xloc++;
            :

```

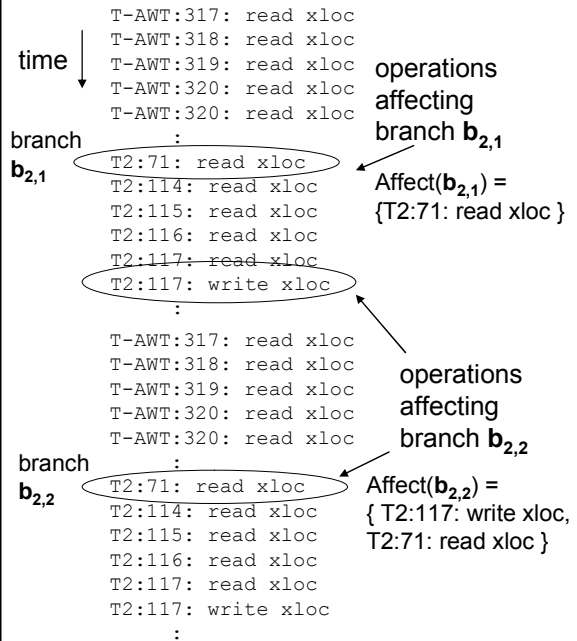


Figure 70. The source code of the JTelnet and its execution trace

6.6.3 Experiment 3: jNetMap

The jNetMap [jNet09] is a network client to monitor devices in a network. This program detects PCs and a router in a network. Among the 10 shared variables, 9 of them are read-shared variables. Based on data flow analysis, the one non read-shared variable affects one branch. The source code and its execution trace are shown in Figure 71 and Figure 72. The results are summarized in Table 15.

Table 15. Summary of experiment results for jNetMap

Method	Number of test cases	Description
Existing TPAIR	Infinite	There is an infinite loop affecting the read and write sequence which causes infinite test case generation because it considers different values of shared variables as different test cases.
Proposed method	4	There are two test cases from the branch-affect group for branch $b_{2,1}$ and two test cases from the branch-affect group for branch $b_{2,2}$. All these groups are listed in Table 16. The same set use-defines affects branches $b_{2,2}$, $b_{2,3}$, $b_{2,4}$ and the rest of the branches within the loop 1 for iteration 2, 3, 4, and so on. In this example, the test cases for the branch $b_{2,2}$ do not change the branch outcomes, i.e., they are always <i>false</i> . Therefore, branches within the loop 1 will always have the same outcome, so there is no need to test for infinite iterations in loop 1.

Thread T2

```

276: while (true) {
      :
279:   if (pingInterval <= 0) {
280:     synchronized (t) {
          t.wait();
        }
283:   } else {
284:     Thread.sleep((int)
          (60000*pingInterval));
285:   }
286:   pingInterval =
      parseFloat(interval.getText());
      :
      :
    }

```

shared variable: pingInterval

Thread T-AWT-EventQueue-0

```

108: FileOutputStream out = null;
109: ObjectOutputStream obj = null;
      :
112: pingInterval =
      parseFloat(interval.getText());
      :
114: File conf = new
File(System.getProperty("user.home")+
"/.jNetMap.conf");
115: out = new FileOutputStream(conf);
116: obj =new ObjectOutputStream(out);
      :
      :
123: obj.writeFloat(pingInterval);
      :
224: notifyAll();
      :

```

Figure 71. The source code of the jNetMap

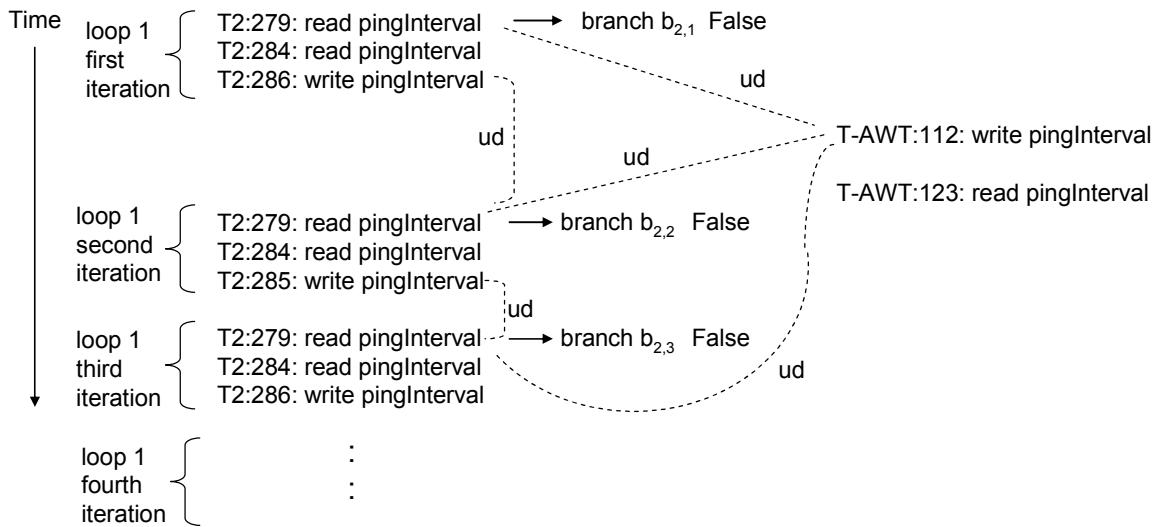


Figure 72. Execution trace of jNetMap

Table 16. Branch-affect groups for jNetMap

Branch-affect groups	Set of use-defines
$g1(b_{2,1})$	$ud(pingInterval, T2:279, T-AWT:112)$
$g2(b_{2,1})$	$ud(pingInterval, T2:279, \dots)$
$g1(b_{2,2})$	$ud(pingInterval, T2:279, T-AWT:112)$
$g2(b_{2,2})$	$ud(pingInterval, T2:279, T2:286)$

6.6.4 Experiment 4: JoBo

JoBo [JoBo06] is a web crawler for downloading complete websites to a local computer. In this experiment, we downloaded a website from Yahoo [Yahoo] and saved it into a local computer. The program has four threads and four shared variables and is 14 kloc in size. Among the four shared variables, one of them is non read-shared. Similar to the previous experiment using jNetMap, this experiment shows that the proposed method generates a finite number of test cases, while existing methods generate an infinite number of test cases.

Figure 73 shows the source code of JoBo. Based on data flow analysis, there is one branch affected by the shared variables. The first iteration in the loop has the same set of access-manners as the second iteration, whereas the third iteration has a different

access-manner (see the execution trace in Figure 74). The results are summarized as follows:

Table 17. Summary of experiment results for JoBo

Method	Number of test cases	Description
Existing TPAIR	Infinite	There exists an infinite loop that causes infinite test case generation.
Proposed method	1	All possible concurrent paths have been checked from the first re-execution trace, no more test cases are required.

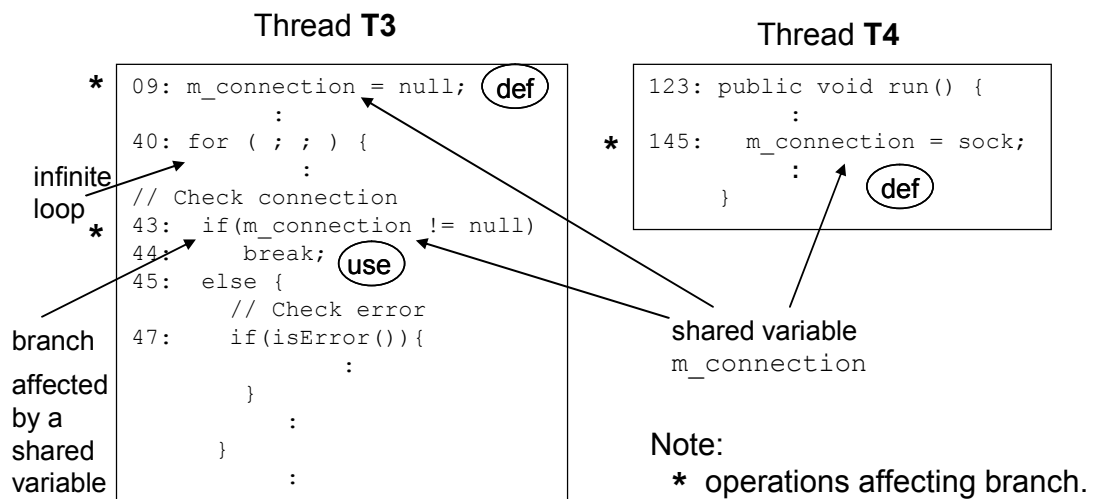


Figure 73. The source code of JoBo

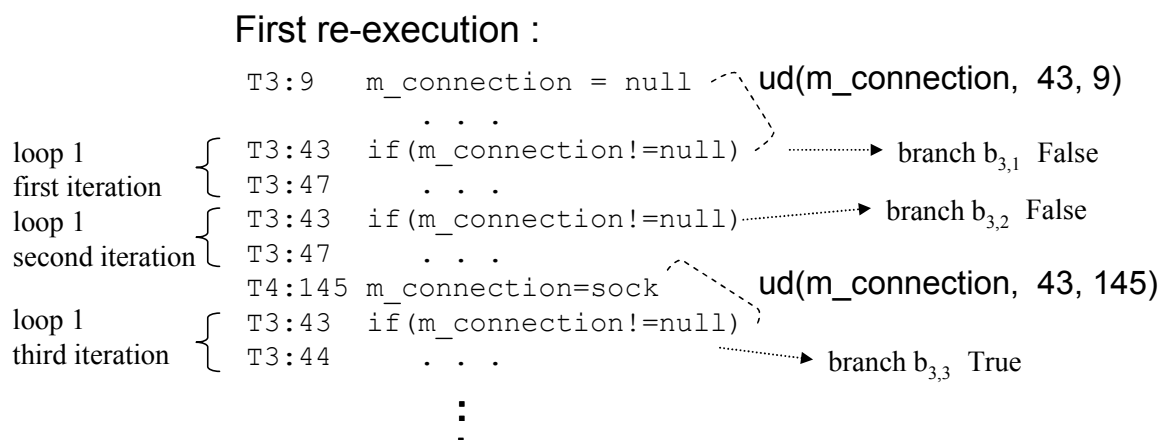


Figure 74. Execution trace of JoBo

6.6.5 Experiment 5: Apache Derby

Apache Derby [ApacheDerby10] is a database written in Java. It has a higher degree of concurrency because it has more non read-shared variables. In such a program, our proposed method proves its significance because there are more potential concurrent errors that are difficult to reproduce. One of the examples is shown in Figure 75.

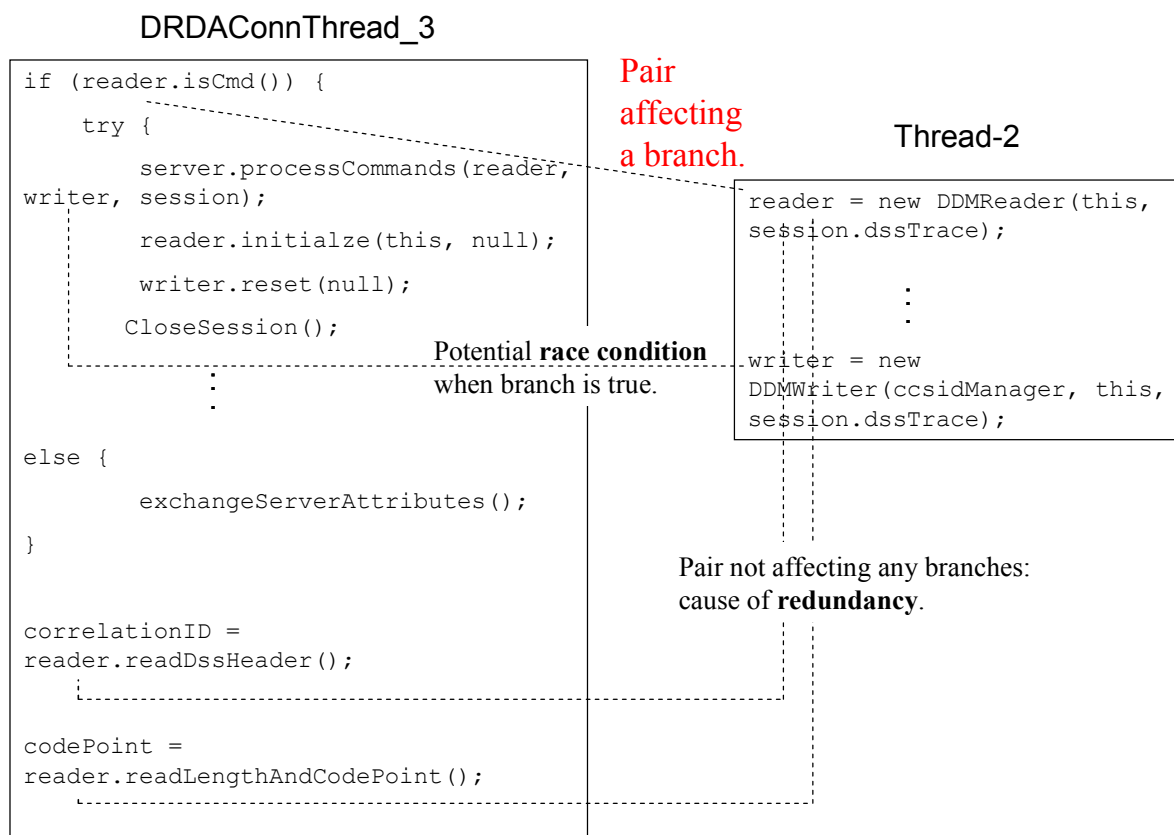


Figure 75. The source code of Apache Derby

6.7 Experiment Results: Memory Reduction

In this section, we show the effectiveness of our proposed new method in reducing the memory required for generating test cases. The work by T. E. Setiadi, A. Ohsuga, and M. Maekawa [Setiadi13] requires a variant graph from the existing reachability testing method. The effectiveness of our proposed new method is demonstrated by comparing the size of our proposed concurrent dependency graph

against that of the variant graph. We discuss three experiments using the following multi-threaded Java open source programs:

1. jNetMap [jNet09] is a network client for monitoring devices, such as PCs and routers, in a network.
2. Apache Commons Pool [ApachePool06] is a generic object-pooling library from Apache.
3. Jobo [JoBo06] is a web spider for downloading complete websites to a local computer.

Table 18 shows that the concurrent dependency graph proposed in this dissertation is smaller in size than the variant graph in the existing reachability testing method.

Table 18. Comparison of the experiment results for existing variant graph and the proposed concurrent dependency graph

No	Target programs	Number of nodes		Memory size (in bytes)	
		Existing*	Proposed**	Existing*	Proposed**
1	jNetMap	Infinite	8	Infinite	320
2	Apache Commons Pool	990	4	71,280	288
3	Jobo	Infinite	4	Infinite	160
4	JTelnnet	42	1	1,680	40

Note:

* **Existing** variant graph from reachability testing method

** **Proposed** concurrent dependency graph

6.7.1 Experiment 1: jNetMap

There is an access to a shared variable in an infinite loop affected by another thread. This causes an infinite sequence of read/write operations and creates a variant graph of infinite size. Figure 76 shows only some parts of the variant graph from the reachability testing method. Here we explain only one example that caused a redundancy.

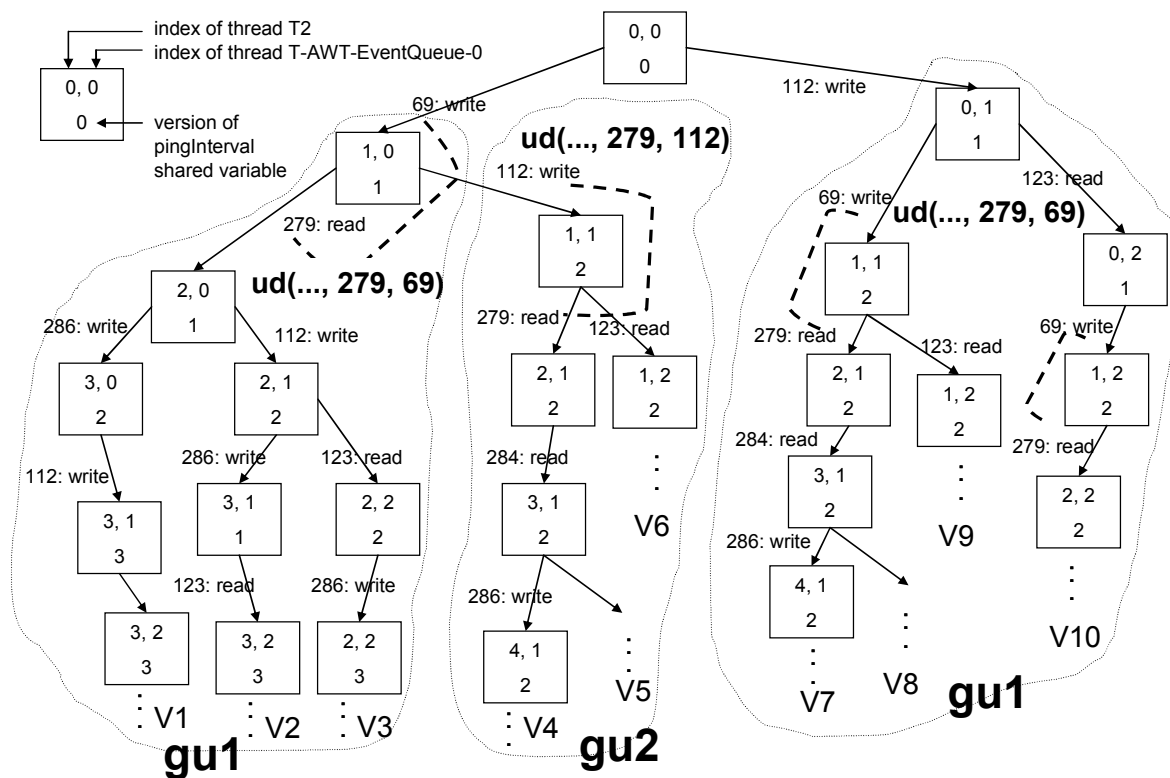


Figure 76. Variant graph for the execution of jNetMap

Figure 77 shows the execution trace of the first execution. The reachability testing method considers all different interleavings between the two threads that can affect the values of shared variables. On the other hand, our proposed method considers only different interleavings that can possibly change the outcome of the conditional statement in line 279, so it generates fewer test cases. In this experiment, only the conditional statement in line 279 might cause different sequences of lock/unlock and read/write operations to shared variables.

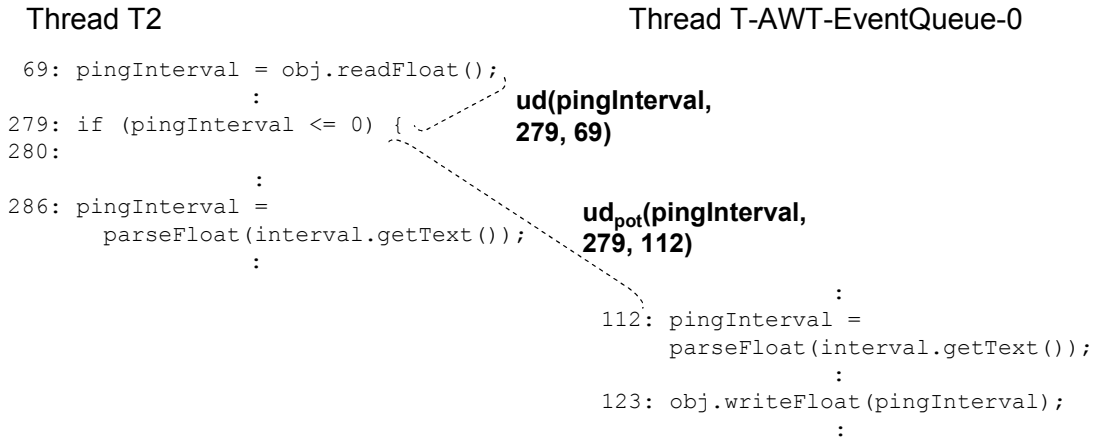


Figure 77. Execution trace of the first test execution of jNetMap

Figure 78 shows a concurrent dependency graph for the branch from the execution trace analysis of the first execution. The traversal of the concurrent dependency graph in Figure 78 results in a set of guidelines in Table VII for generating test cases. Table VII shows the set of guidelines for producing two test cases based on the traversal of the concurrent dependency graph in Figure 78.

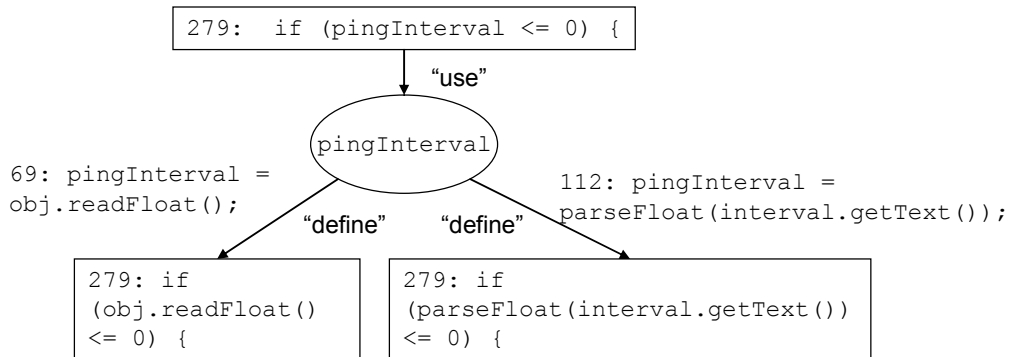


Figure 78. An example of a concurrent dependency graph for the execution of jNetMap

Table 19. A set of guidelines from the concurrent dependency graph in Figure 78

No.	Guideline
1	$gu1 = \{ ud(pingInterval, 279, 69) \}$
2	$gu2 = \{ ud(pingInterval, 279, 112) \}$

The branch outcomes for the conditional statement in line 279 are determined by the assignment from the write operation in either line 69 or 112. For a comparison

with the existing reachability testing method, we created a variant graph in Figure 76 based on the execution trace in Figure 77.

We refer to the source code in Figure 79 to explain the cause of redundancy. The truth value of the branch in line 279 is affected by the order of interleavings between the assignment of the shared variable *pingInterval* in line 69 and 112. The other *read* and *write* operations to the shared variable *pingInterval* in line 123, 284, and 286 do not affect the truth value of the branch in line 279, so different interleavings among them are redundant. For exploring different execution paths caused by the branch in line 279, we have to consider only whether an execution-variant satisfies the $ud(\text{pingInterval}, 279, 69)$ or $ud(\text{pingInterval}, 279, 112)$. In other words, we can group those execution-variants into two groups and it is sufficient to test only one of each group.

<p style="text-align: center; margin: 0;">Thread T2</p> <pre style="margin: 0;"> 69: pingInterval = obj.readFloat(); : 276: while (true) { 279: if (pingInterval <= 0) { 280: : 283: } else { 284: Thread.sleep((int) (60000 * pingInterval)); 285: } 286: pingInterval = parseFloat(interval.getText()); : </pre>	<p style="text-align: center; margin: 0;">Thread T-AWT-EventQueue-0</p> <pre style="margin: 0;"> : 112: pingInterval = parseFloat(interval.getText()); : 123: obj.writeFloat(pingInterval); : </pre>
--	---

Figure 79. The source code of jNetMap

6.7.2 Experiment 2: Apache Commons Pool

The reachability testing method uses a variant graph with 990 nodes to generate 216 test cases. However, most of them do not affect the occurrence of the race condition. As shown in the work by T. E. Setiadi, A. Ohsuga, and M. Maekawa [Setiadi13], only two test cases are actually required. Figure 80 shows that we require a concurrent dependency graph with only 4 nodes to generate those two required test cases.

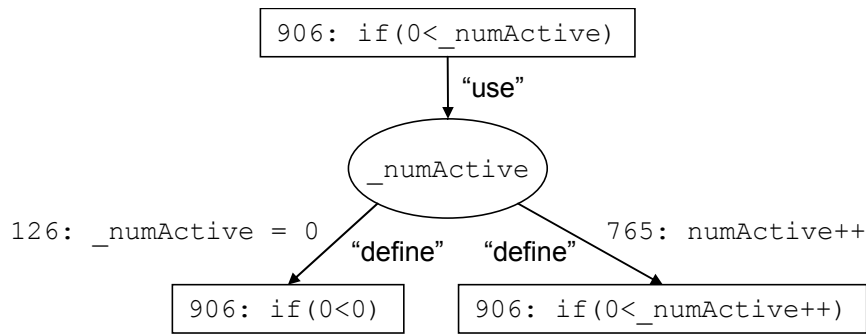


Figure 80. An example of a concurrent dependency graph for Apache Commons Pool

Figure 81 shows the execution trace of the test program containing race conditions. The reachability testing method considers all different interleavings that affect the values of shared variables among the three threads in Figure 81. Our proposed method generates fewer test cases because it considers only those interleavings that can possibly affect the conditional statement in line 906. Figure 80 shows a concurrent dependency graph from the execution trace in Figure 81.

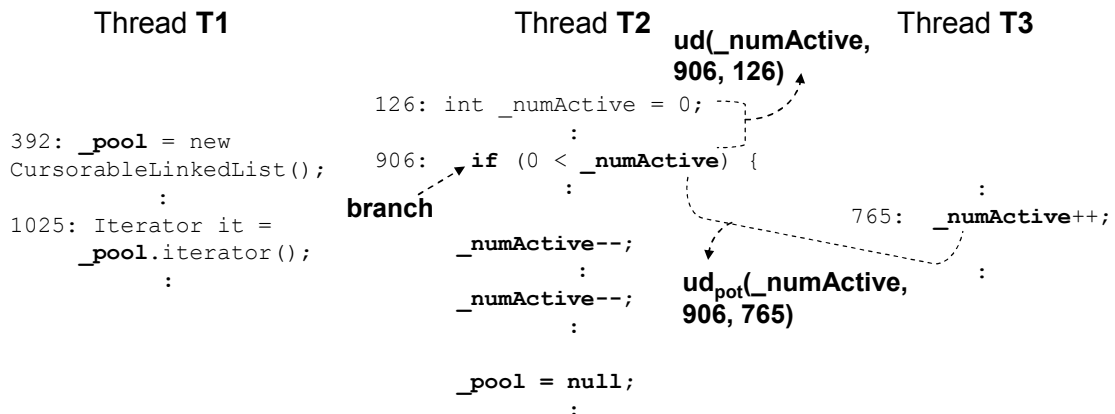


Figure 81. Execution trace of the experiment using Apache Commons Pool

Based on the set of guidelines in Table 20, our proposed method generates only 2 test cases. Figure 82 shows a piece of code to explain the cause of redundancy in the reachability testing method. The conditional statement in line 906 depends only on the values of the shared variable `_numActive` affected by the interleavings with the assignment in line 765 of the thread `T3`. The access through the reference variable `_pool` depends on interleavings, but it does not affect the conditional statement in line 906. Hence, different interleavings that are affecting the reference variable `_pool` are redundant. Figure 83 shows the concurrent dependency graph for the reference variable

_pool.

Table 20. A set of guidelines from the concurrent dependency graph in Figure 80

No.	Guideline
1	$gu1 = \{ ud(_numActive, 906, 126) \}$
2	$gu2 = \{ ud(_numActive, 906, 765) \}$

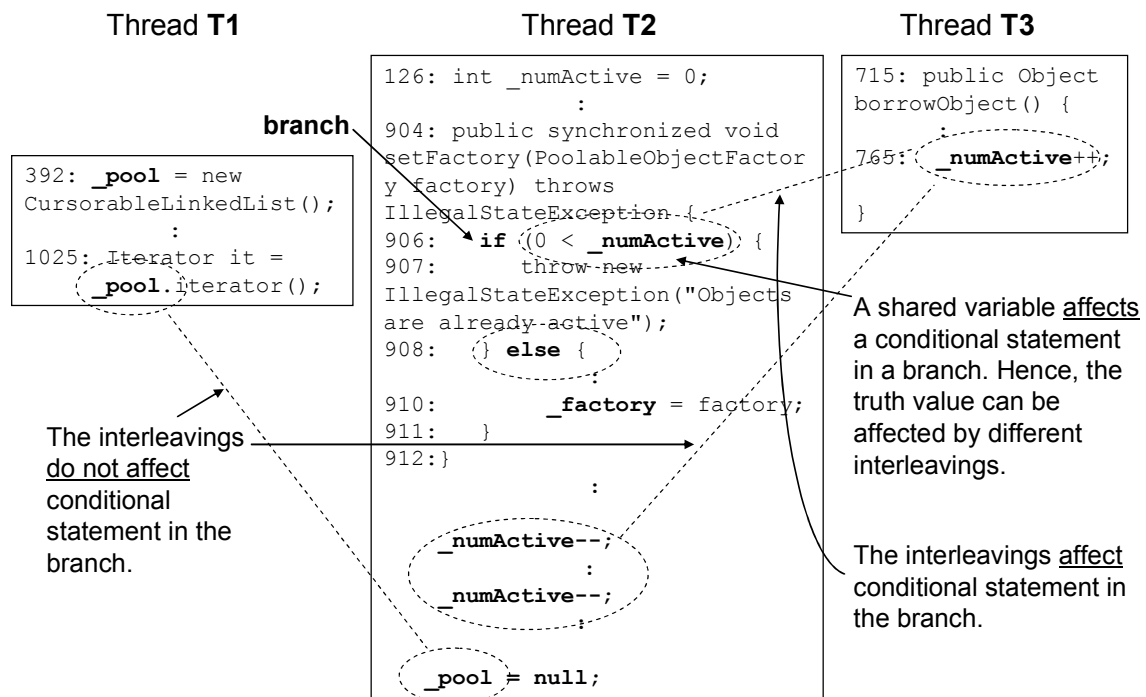


Figure 82. An example of a test program using the Apache Commons Pool library

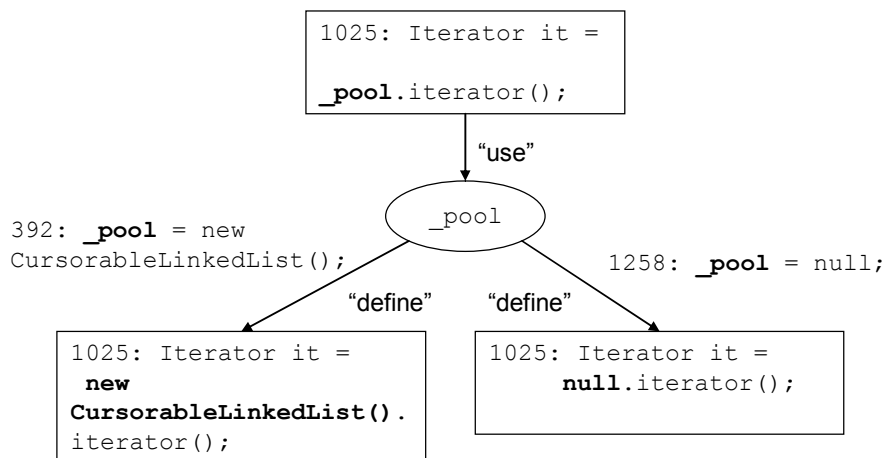


Figure 83. Concurrent dependency graph for the reference variable *_pool*

6.7.3 Experiment 3: JoBo

In this experiment, we downloaded a website from Yahoo [Yahoo] and saved it in a local computer. Similar to Experiment 1, there is an access to a shared variable within an infinite loop. This shared variable is affected by another thread, thus causing an execution trace of infinite length accessing the shared variable. The reachability testing method produces a variant graph of infinite length and infinite number of test cases because of the infinite length of execution trace. However, actually only two test cases are required as shown in the work by T. E. Setiadi, A. Ohsuga, and M. Maekawa [Setiadi13].

Figure 84 shows the execution trace of the first execution. Note that loop 1 is an infinite loop. The infinite loop in the thread $T3$ is accessing a shared variable. For each access to a shared variable in the loop iteration, its value can be affected by the assignment from the thread $T4$. Therefore, the reachability testing method generates infinite test cases because it produces a different test case for each iteration in the infinite loop. Our method identifies that only some of the iterations are sufficient for checking consistent locking, because the concurrent-pair of access-manners generated for each iteration is the same as in the previous one.

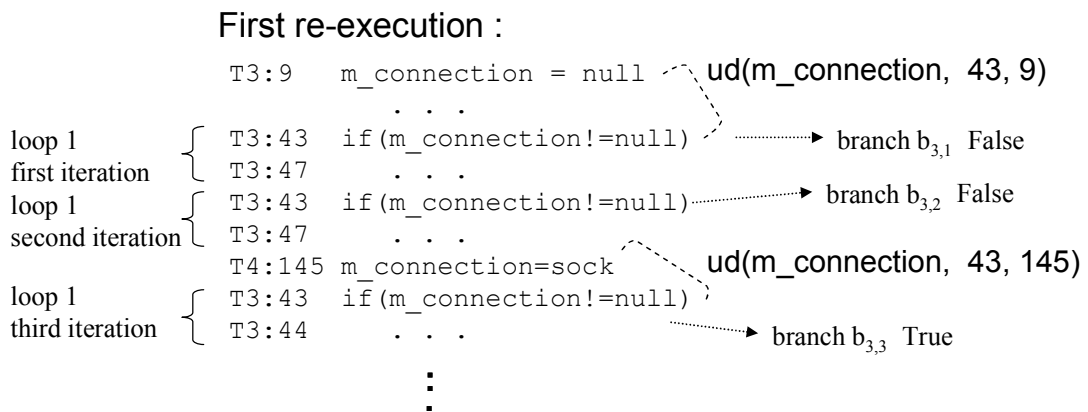


Figure 84. Execution trace of the first test

Figure 85 shows a concurrent dependency graph for the branch from the execution trace analysis of the first test execution in Figure 84. Based on the traversals of the concurrent dependency graph in Figure 85, our proposed method produces the set of guidelines in Table 21. We then generate two test cases based on Table 21.

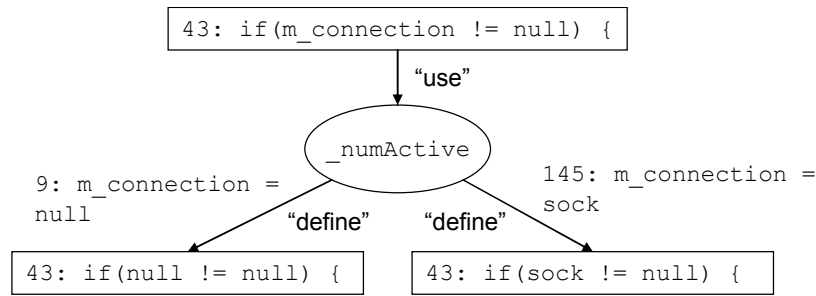


Figure 85. An example of a concurrent dependency graph for JoBo

Table 21. A set of guidelines from the concurrent dependency graph in Figure 85

No.	Guideline
1	$gu1 = \{ ud(m_connection, 43, 9) \}$
2	$gu2 = \{ ud(m_connection, 43, 145) \}$

Figure 86 shows the piece of code that affects the test case generation. There is an infinite loop in the thread *T3* accessing a shared variable. From the execution trace of the first execution, the reachability testing method produces a variant graph with infinite nodes. For each node, an execution-variant can be created by making a different order of interleavings for an assignment from the thread *T4*, hence causing an infinite number of test cases.

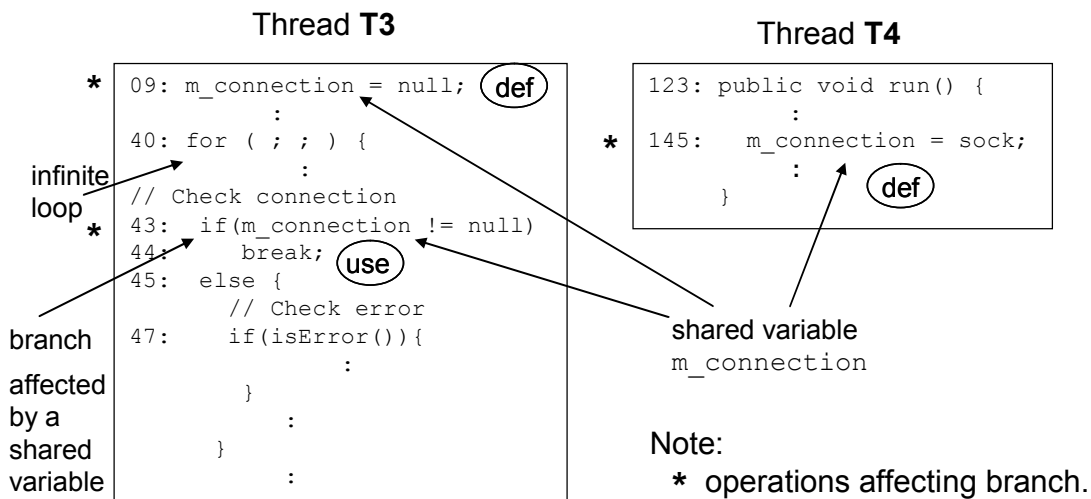


Figure 86. The source code of JoBo

The first and second loop iterations of the execution trace in Figure 84 satisfy

the first use-define in the guideline *gu1*, whereas the third iteration satisfies the second use-define in the guideline *gu2*. The first iteration of the infinite loop has the same concurrent-pair of access-manners as the second iteration, whereas the third one has a different concurrent-pair of access-manners. All possible different concurrent-pairs of access-manners in the iterations of loop 1 have been explored, from the first iteration until the third one. Therefore our proposed method does not need to test all the infinite loop iterations, because the remaining loop iterations will not produce different sequences of lock/unlock and read/write operations to shared variables.

Chapter 7. Discussions

The proposed method is intended to be used for debugging multi-threaded concurrent programs as complement for dynamic race detector tools. Specifically in the case when the exact timing information when the error happened is unknown. Dynamic race detector tools detect potential race condition from a particular execution trace. When there is only limited information from the execution trace when the error occurred, then it is possible that the actual execution path might not be exactly reproduced because different interleavings caused different branch outcome. Hence, the dynamic race detector tools might not detect the existence of error. In order to reproduce the error, one has to replay the concurrent program many times by changing the timings/interleavings.

7.1 Applicability

The proposed method is applicable for the following program characteristic, error types, and environments:

7.1.1 Program Characteristics

- Concurrent programs that are using lock mechanisms, for example Java, C, and C++.
 - Applicable for procedural or object-oriented languages. Our detection for race conditions only concerns about the sequence of lock/unlock and read/write operations to shared variables. The lock/unlock and read/write operations to shared variables can be called from another function or method. It justifies the correctness of the program by checking program execution whether all read/write operations to shared variables are protected by consistent locks.
- Program structure: It manages to detect/reproduce concurrent errors caused by interleavings and various program structures such as branches, loops, interrupts, pointers, reference variables, file references, and arrays.

Program Structure

Reasons why errors are difficult to be detected/reproduced.	Interleavings cause different execution paths.		Interleavings cause variables to refer to different data.
Program structures.	<ul style="list-style-type: none"> - Branch: if, switch. - Loop: for, while. 	Interrupt	<p style="text-align: center;">Variable → data</p> <hr style="border-top: 1px dashed black;"/> <ul style="list-style-type: none"> - Pointer (in C) → memory address - Reference variable (in Java) → object - File reference → file - Lock variable → lock object - Index of an array → element of an array - Iterator → element of an array
Solution by the proposed method to detect/reproduce errors.	Use dependency graphs to determine interleavings that can change the conditional statement.	Apply the concept of interrupt as a thread. Hence checking interrupt timing become checking interleavings, so the proposed method can be applied.	Use dependency graphs to determine interleavings that can change the data.

Array

When an array is shared, the actual element that is shared depends on the index of the array. The value of the index to specify a particular element might not be known until the actual execution. The index could be specified by a variable whose value can depend on input and interleaving. Depending on interleavings, the particular element specified during the execution might be different even though the same execution path

is executed. To be safe, programmers can take a conservative approach by considering all elements in an array to be shared when we are checking for race conditions. Unfortunately, locking an entire array would decrease concurrency because other threads have to wait to access different elements. To increase concurrency, sometimes programmers divide the values of the index into several groups and use separate locks for each group consistently during programming. For an array, programmers need to specify whether the array has to be accessed as a whole, or it can be accessed individually for each element. In the later case, the proposed method will generate test cases based on the index of array or the iterator.

Concurrent Programs with Interrupts

The proposed method can also apply to concurrent programs with interrupts by treating an interrupt as a thread. Interrupt handling programs might necessarily access shared variables or locks, which might cause race conditions or deadlocks. To ensure program correctness, it is necessary to check accesses to shared variables and lock consistency when an interrupt or event is processed. Based on the origin of the interrupt, we classify two types of interrupts:

- Internal interrupts: caused by an illegal CPU execution, such as buffer overflow, divide by zero, memory protection violation, etc. For this type of interrupt, it is natural that the program thread processes the interrupt handling.
- External interrupts: caused by a device other than CPU in the timing that is independent of (no relation to) the program thread progress. Most real-time applications are composed of processes that deal with interrupts from external sources such as signals from sensors, network interfaces, and I/O devices.

This dissertation proposes to change an interrupt handling processing into a thread. When an interrupt occurs, a corresponding interrupt handler is executed as a different thread. In this way, interrupt timing's problem is translated into a synchronization and/or interleaving's problem. Thus, testing interrupt timing problems can be handled in the same way using the proposed method in Chapter 5 as concurrent programs.

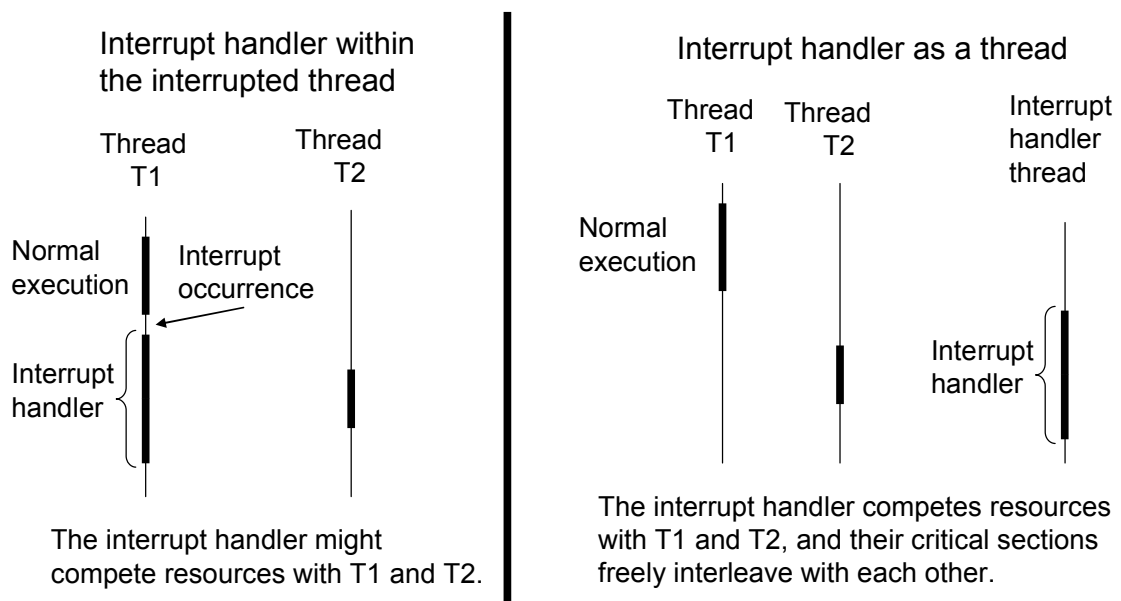


Figure 87. Interrupt as a thread

For realizing interrupt-as-a-thread, there are two basic approaches. In the first method, a separate thread, namely, an interrupt processing thread, is assigned to each interrupt signal. Thus interrupt processing thread directly receives an interrupt signal and then handles it. In the second method, the central interrupt dispatcher process receives all interrupt signals and then dispatches its processing to an interrupt processing thread. In either of these two methods, interrupt processing threads can be newly created every time interrupt is received, or prepared beforehand at the system start up.

Advantage of Making an Interrupt as a Thread

There are three advantages in making an interrupt as a thread.

- Easier for testing: the problem of checking interrupt timings is reduced to the problem of checking interleavings among the threads including the threads for interrupt processing. The exact timing of interrupt occurrences is no longer need be concerned.
- Deadlock avoidance: A deadlock occurs when the interrupt handler tries to acquire resources (ex. a lock) that is currently occupied by the interrupted thread. If we design a system following the principle of interrupt-as-thread, we can avoid the deadlock caused by a competition of resources between the interrupted thread and the interrupt handler, because the thread scheduler can

switch control to the interrupted thread and continue the execution until the resource is released.

- Easier for programming: In order to avoid deadlocks, or to in order to avoid that resources are locked for a long time during interrupt processing, one traditional programming style for a critical section is to entirely inhibit interrupts during its entire execution. This is no longer needed by making an interrupt as a thread because the delay is guaranteed to be short, namely, the time to trigger an execution of an interrupt handler thread.

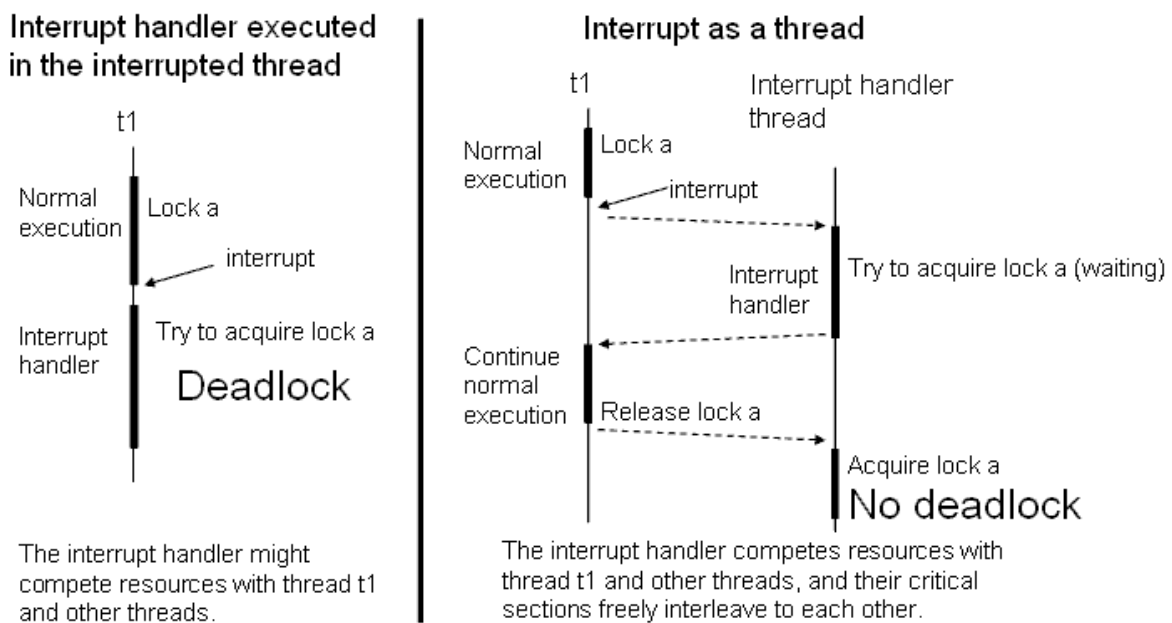


Figure 88. Deadlock can be avoided by following the interrupt-as-thread principle.

Disadvantage of Making an Interrupt as a Thread

The disadvantages of making an interrupt as a thread are:

- Less responsive: since the interrupt handler runs as a thread, its execution will depend on the thread scheduling and might be preempted by others.
- Processing order: since the threads for interrupt processing are under the control of thread scheduler, their order of processing may not be the same as the occurrences of the interrupts. This may cause problems when a series of interrupts are expected to be processed as a stream, for example processing a series of input signals from the same device.

Preservation of Interrupt Processing Order

Depending on the nature of processing, there is a need to preserve the interrupt order. Take an example in which an interrupt comes from each “different” ATM for account processing such as cash deposit or withdrawal. In this case, the precise timing and order of the interrupts may not be so important. In fact, these requests may compete for resources to each other. On the other hand, in the case where a series of interrupt come from the same ATM, the order need be preserved.

It is difficult to control the order of execution as in the original program only by controlling scheduling. Another alternative solution is by using some graph model such as Petri Net model to preserve the interrupt order. When the order of interrupts does not need to be preserved, we can create/use different nodes for handling different interrupts. When interrupt order needs to be preserved, we can assign all the interrupts to be handled by the same node so that the next interrupt will be executed after the current interrupt handling finishes.

7.1.2 Error Types

- Race condition
- Deadlock

The proposed method can also be applied to detect/reproduce deadlocks as well using an existing deadlock detector. In fact, detecting a deadlock is easier because it considers only the sequence of lock/unlock (see Figure 89(b)).

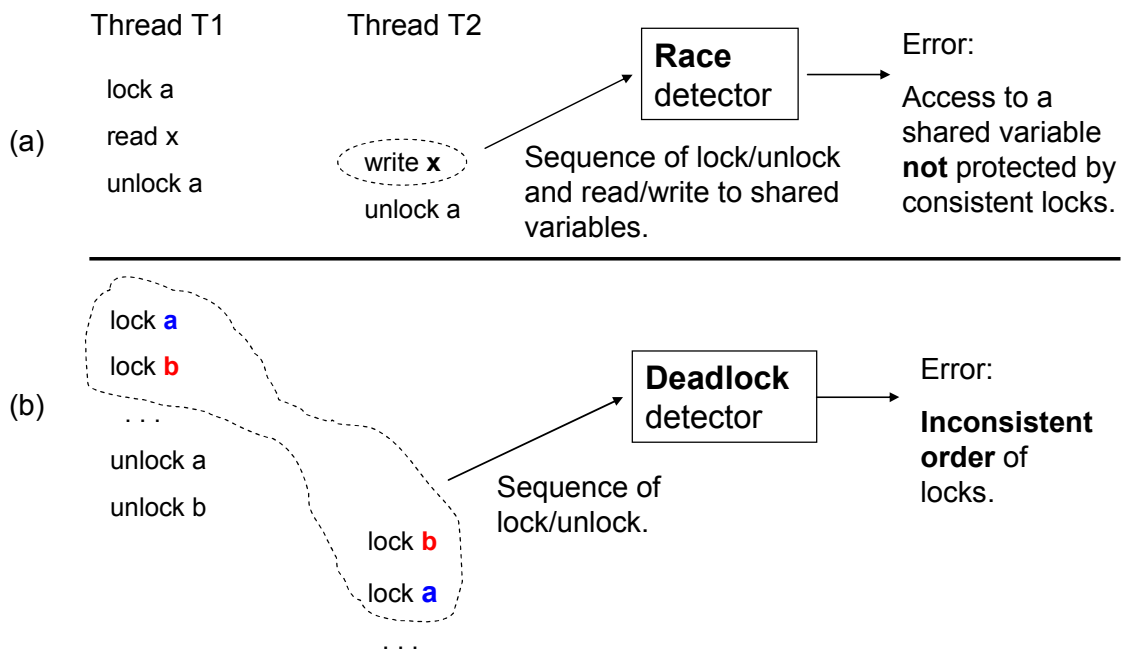


Figure 89. Comparison between race detection (a) and deadlock detection (b)

7.1.3 Execution Environment

- Availability of source code.
 - Some information from source code is required for tracing, for example variable names, class names, line of code, etc and of course source code is required for fixing the bugs.
- Tracing capabilities to record lock/unlock and read/write operations to variables.
- Deterministic testing: using specialized virtual machine or instrumentation for controlling interleaving, i.e. thread switches.
- No bugs in the compiler, virtual machine, or processor.
 - The proposed method is intended to check whether programmers have written their code correctly by using appropriate locks for accessing shared variables. Even though a source code is written correctly, concurrent errors might still occur if there is a bug in the compiler or Virtual machine that violates memory consistency. Such concurrent errors caused by the compiler or virtual machine might not be detected by the proposed method (see Figure 90). The read/write operations can be re-ordered for optimization purpose by the compiler, virtual machine, or processor. However, the happen-before relationship for

lock/unlock to the same lock object must be guaranteed to be correct, otherwise it will give false alarms. For programs using lock mechanisms, such happen-before relation must be guaranteed.

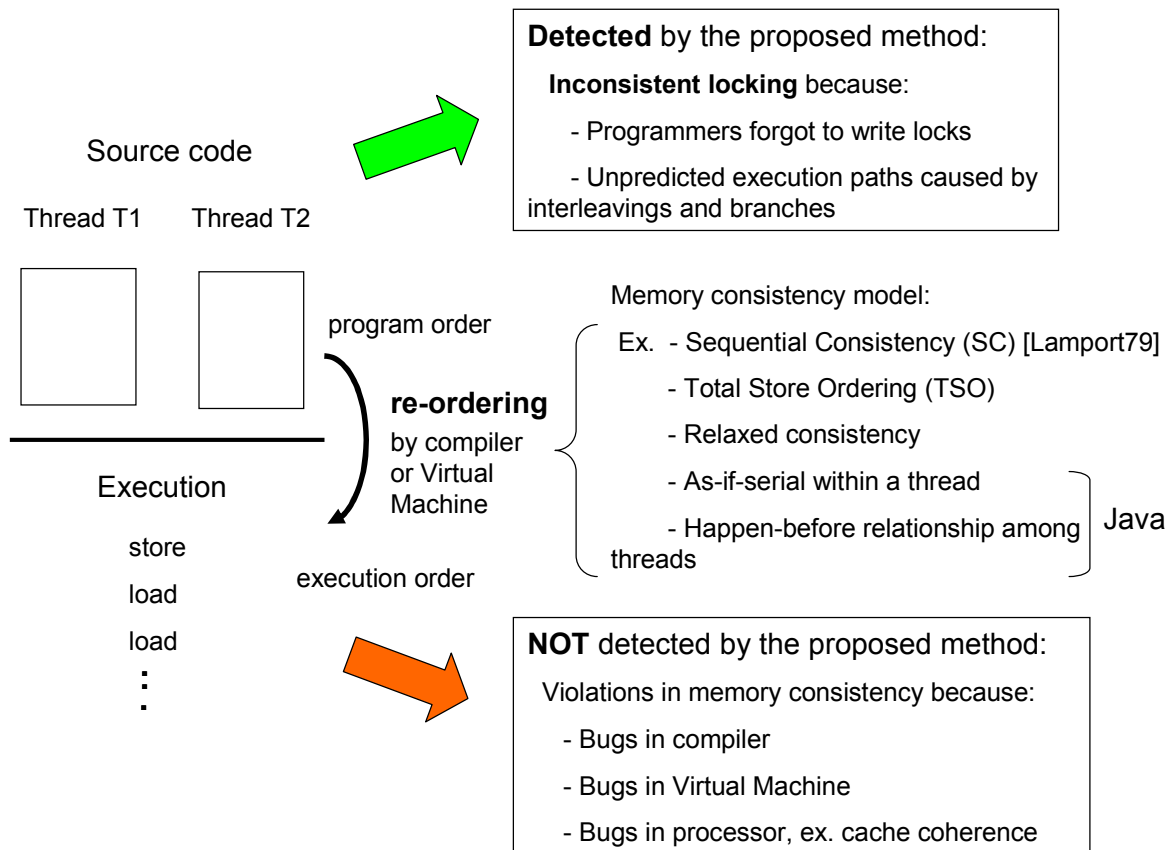


Figure 90. Memory consistency

7.2 Limitations

- Intentional race conditions.
 - If a concurrent program is designed with some race conditions that are intentional, for example, to speed up the process, then the proposed method will report them as false positives. Such writing of code is not usually recommended. Such a situation happens when programmer are certain that the program will behave correctly even though a race condition occurs. It requires manual analysis and currently beyond the scope of our proposed method.
- Synchronization other than lock mechanism, for example barrier [Nishiyama04].
- Real Time.
 - The proposed method focuses on detecting errors caused by interleavings of

threads. However, it cannot measure exact timing; for example it cannot check a case such as, whether after a particular interrupt, the next interrupt must come within 2 seconds. Therefore the proposed method cannot be applied as in the current form for checking the correctness of critical real time applications.

- Time interval.
 - Using commands to “wait” for a fixed period of time, for example `wait(100ms)`, will cause some interleavings to become infeasible. Some commands in the same thread after a wait command would not be interleaved immediately with other threads because the thread is suspended for a period of time. For example, in an extreme situation, other threads might have finished, so the waiting thread continues its own execution without interleaving with any other threads. Since our current method does not consider the usage of wait command for a fixed period of time, our algorithm might generate some interleavings that are infeasible. However, we consider using a command to wait for a fixed period of time to be a bad programming practice.

7.3 Efficiency

The efficiency of the proposed method to reproduce errors is measured by how much it can reduce the necessary test cases while still maintaining to cover all the race-equivalent groups. The proposed method is efficient in reducing the number of test cases by considering only different interleavings that are affecting race conditions. The efficiency of the proposed method depends on the structure of the target programs. It performs efficiently in a concurrent program which has complex sequences of lock/unlock in branches. Such complex structures often make it difficult to reproduce concurrent errors because different execution paths caused by different interleavings often execute different sequences of lock/unlock and read/write operations to shared variables.

Our proposed method significantly reduces the number of test cases by the following means:

- Grouping together different interleavings that do not affect consistent locking using the concept of race-equivalence.
- Testing only one member of each group.

The debugging efficiency is primarily measured by the number of test cases. The minimum number of test cases required is the number of race-equivalent groups. In order to improve the efficiency for reproducing the errors, we extend the past work [in particular, Hwang95] for reachability testing of concurrent programs. Many existing methods try to identify all interleavings which may affect shared variables, whereas our method can identify only those interleavings which affect branch outcomes. The advantages of the proposed method are an improvement in efficiency by the following means:

- Reduction of test cases that do not change execution path.
- Reduction of the amount of work for checking race condition in execution paths with the same set of access-manners to shared variables.
- Creation of a new execution path can be identified and created by combining the branch-paths found in the previous trace without further replaying the program.

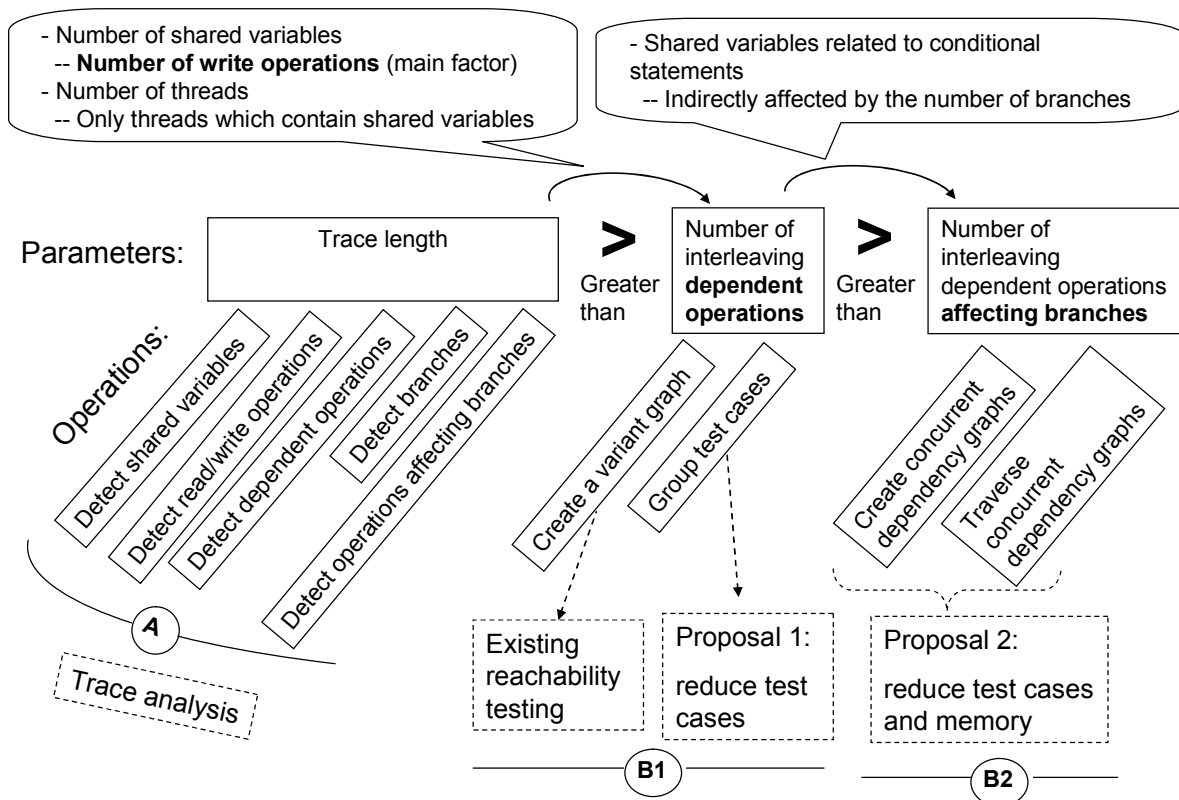
The proposed method performs efficiently for concurrent program in which branch outcome and loop are not much affected by different interleavings. This happens when the conditional statement for the branch or the conditional for the loop are affected by only few shared variables and few dependent concurrent operations that can change their values. We performed data flow analysis for some case studies in experiment section and found that two of them, that is WebLech [Weblech02] and WebHarverst [Wbhv07], satisfy this condition. For those two case cases, experimental results shows that our proposed method is superior and achieves the reduction as much as 90% of test cases compared to those partial order reductions.

Similar to reachability testing, our method is also exhaustive in the way that the test cases are created systematically and all the possible combination of execution paths from the threads can be explored if the number of interleavings is bounded. In the presence of infinite loop or in reactive system that does not terminate, the number of possible interleavings might be unlimited and causes unlimited execution paths. Even in such situation, our proposed method groups different execution paths with the same access-manner to shared variables in to the same race-equivalent group and test only one of them. Therefore the number of test cases is bounded by the number of branches. Since we test only one member from

each race-equivalent group, it is possible that the error was reproduced from different execution path, but it has the same access-manner to shared variable as in the execution path when the actual error occurred. This should be sufficient for the purpose of race detection because the same cause of error, i.e. the lock inconsistency, exists in both execution paths.

7.4 Complexity

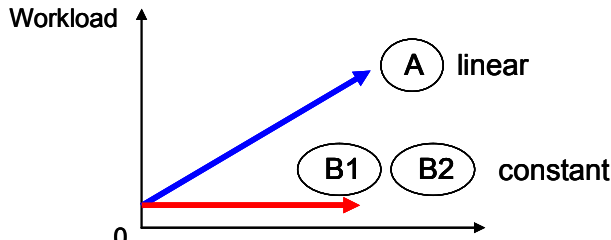
The complexity of the proposed method is in factorial order, which is required for creating/grouping variant graphs and creating/traversing concurrent dependency graphs (refer to operations B in Figure 91). Other operations are for obtaining information from execution traces which are in linear order complexity (refer to operations A in Figure 91).



Case 1:

- Number of threads = 1, or
- Number of interleaving dependent operations = 0

(same as sequential programs)



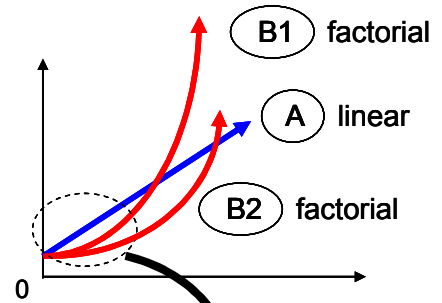
A: trace length

B1: number of interleaving dependent operations

B2: number of interleaving dependent operations affecting branches

Case 2:

- (other than case 1)



Experiment

Actual workload from the experiment

Target program	Time taken (in seconds)	
	Operation A	Operation B2
Apache Common Pool	24.3	0.5
jTelnet	421.5	0.3
jNetMap	38.8	0.3
JoBo	88.1	0.2

Trace analysis Test case generation

Figure 91. Complexity and the actual workload of the proposed method

The calculation for the complexity will be derived as follows.

Let:

- m : number of threads, $m \geq 2$.
- n : number of interleaving dependent read/write operations, $n \geq 0$.
 - $n = 0\%$ means there will be no concurrent errors.
- p : percentage of n which are affecting branches.
 - $p = 0\%$ means interleavings are not affecting any branches.
- n_{T1} : number of interleaving dependent read/write operations in thread $T1$.
- interleaving dependent read/write operations: read/write operations to shared variables in which the value of the shared variables can be affected by the interleavings of the read/write operations.

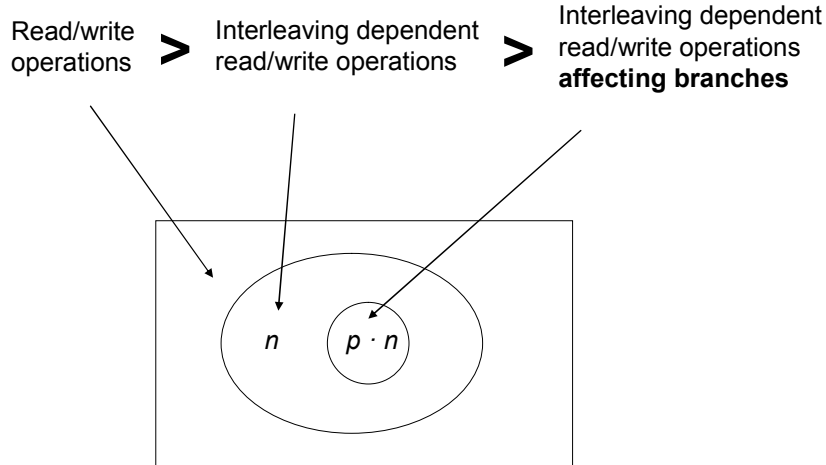


Figure 92. Classification of read/write operations

$$\text{Number of possible interleavings} = \frac{n!}{(n_{T1})!(n_{T2})!(n_{T3})!\dots(n_{Tm})!} = n! \cdot \prod_{i=1}^m \frac{1}{(n_{Ti})!} \quad (7-1)$$

See some examples in **Appendix A**.

Take the worst case where operations are distributed equally among threads, hence each thread will have $\frac{n}{m}$ number of operations. Therefore $n_{T1} = n_{T2} = n_{T3} = n_{Tm} = \frac{n}{m}$

(7-2)

In the worst case (7-2), the number of test cases for the existing reachability testing will become (7-1)(7-2):

$$\prod_{i=1}^m \frac{1}{(\frac{n}{m})!} = \frac{n!}{((\frac{n}{m})!)^m} \quad (7-3)$$

Stirling's approximation [Hazewinkel01]:

$$n! \cong \sqrt{2\pi n} \left(\frac{n}{e}\right)^n \quad (7-4)$$

Approximate equation (7-3) using Stirling's approximation (7-4):

$$\frac{n!}{((\frac{n}{m})!)^m} \cong \frac{m^{0.5m} \cdot m^n}{(\sqrt{2\pi n})^{(m-1)}} \quad (7-5), \text{ see } \mathbf{Appendix B} \text{ for the detail proof.}$$

In most cases, the number of operations (n) is much larger than the number of threads (m), i.e. $n \gg m$. Take the largest order from equation (7-5) to measure the complexity:

- Existing reachability testing method: $O(m^n)$ (7-6)
- Proposed method: since our proposed method only concerns with the interleavings affecting branching, the complexity: $O(m^{p \cdot n})$ where $0\% \leq p \leq 100\%$ (7-7)

The order of complexity does not change, but since $0\% \leq p \leq 100\%$, the complexity of proposed method (7-7) is **less than** the existing method (7-6), or it is equal to the existing method in the worst case when $p = 100\%$.

From the experiment results, the value of p is between 0% and 33.3%.

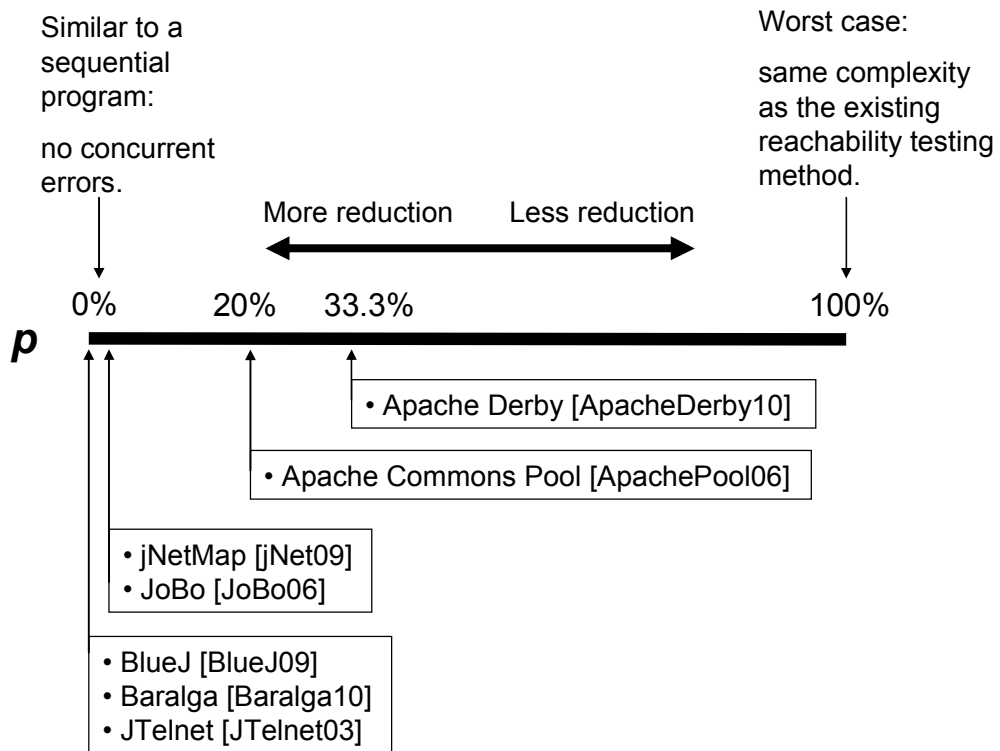


Figure 93. The percentage of operations affecting branches for several target programs

Figure 94 shows the operations affecting branches for the Apache Commons Pool.

Interleaving dependent r/w operations, $n = 10$.

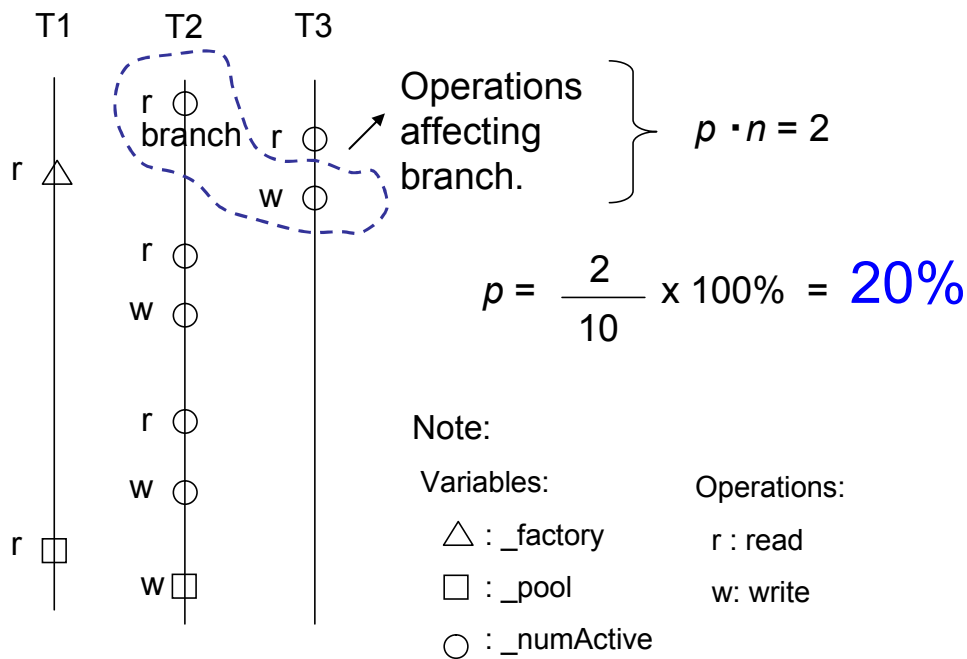


Figure 94. Operations affecting branches for the Apache Commons Pool

7.5 Correctness

Scope

- Target: concurrent programs with lock mechanism.
- Definition for a race condition: exist an interleaving where two threads are accessing the same data without protected by consistent locks.

Precondition

- For detecting/reproducing concurrent errors, the proposed method will have an execution trace with the same input as when the concurrent errors occurred.

(7-8)

- ✧ The execution trace contains the sequence of lock/unlock and read/write operations to shared variables.
- ✧ However, the concurrent errors might not be detected/reproduced in that execution trace because the interleaving might be different from the one when the concurrent errors occurred.

For detecting/reproducing the concurrent error, the proposed method will generate different interleavings from the execution trace stated in precondition (7-8). Test case reduction by the proposed method is achieved by:

1. Group the interleavings generated from the existing reachability testing into several race-equivalent groups. (7-9)
2. Test only one member from each race-equivalent group. (7-10)

Reasons why concurrent errors are difficult to be detected/reproduced (refer to Applicability subsection):

- Interleavings cause different execution paths. (reason 1)
- Interleavings cause variables to refer to different data. (reason 2)

Correctness: we have to prove there are no false alarms in the proposed method.

- **A. No false positives** : **must not** report any concurrent errors which actually do not exist. (7-11)
- **B. No false negatives** : concurrent errors **must** be detected/reproduced even though not all interleavings are tested because of reduction by the proposed method. (7-12)

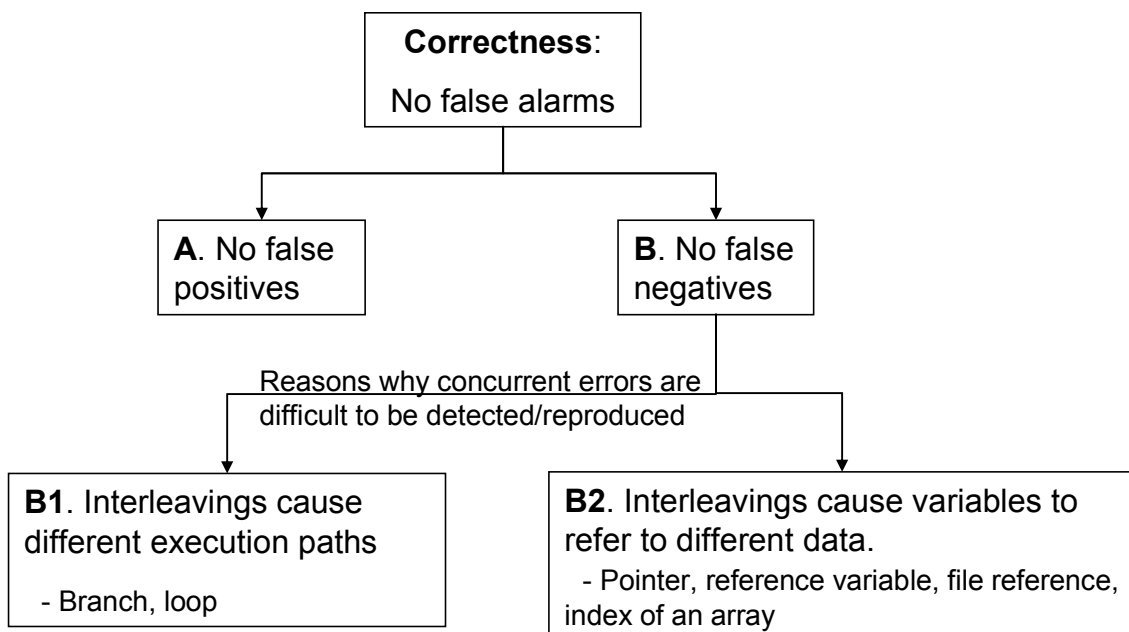


Figure 95. Proof for correctness

A. Proof for (7-11): No false **positives**

Proof that if a program does not contain concurrent errors \Rightarrow proposed method does **not** generate interleavings that contain concurrent errors.

Definition:

- V : execution variant. A different read/write sequence that affects the values of shared variables.
- Reachability: set of execution-variants from reachability testing method.
 - Reachability = $\{V1, V2, V3, \dots, VN\}$, N = the number of execution-variants.
- g_i : race-equivalent group. With $i = 1, 2, 3, \dots, M$. M = number of race-equivalent groups.

Lemma: Reachability testing does not contain false positives.

If a program does not contain concurrent errors \Rightarrow the reachability testing does not generate any interleavings that contain concurrent errors.

Since the reachability is just changing the order of interleavings and does not remove any locks, it will not cause any new concurrent errors that actually do not exist.

Assume there is no concurrent errors \Rightarrow in a program, then from the proposed method
Lemma above:

For $\forall V$ where $V \in \text{Reachability} \Rightarrow V$ does not generate interleavings contain concurrent errors.

Proof that all members in that race-equivalent groups do not contain concurrent errors.

Proof: For all $i = 1, 2, 3, \dots, M$, $\forall V \in g_i \Rightarrow V$ does not contain concurrent errors.

Since the proposed method only groups the interleavings generated by the existing reachability testing (7-9). Hence, no matter how from the algorithm reachability testing method, so if $V \in g_i \Rightarrow V \in \text{Reachability}$

From the Lemma, for the grouping is, it will not create any new concurrent errors after grouping if the reachability testing does not generate interleavings that $\forall V$ where $V \in \text{Reachability} \Rightarrow V$ does not contain concurrent errors.

By implication, we can conclude that: $\forall V \in g_i \Rightarrow V$ does not contain concurrent errors.

If a program does not contain concurrent errors \Rightarrow proposed method does not generate

interleavings that all members in race-equivalent groups do not contain concurrent errors.

■ QED.

B. Proof for (7-12): No false **negatives**

Suppose Assumption: there is an interleaving that contains a concurrent error in a program, let's say V_{error} .

~~We have to~~ Prove that there is a race-equivalent group, let's say gI , in which:

- The V_{error} is a member of the gI . $V_{error} \in gI$ (7-13)

- All members in the gI contain the same error as in the V_{error} . $\forall V \in gI \Rightarrow V$
contains the same concurrent error as in the V_{error} . (7-14)

Lemma:

Our proposed method will test one interleaving from each race-equivalent group. Hence, the concurrent error will be detected when one of the members from the race-equivalent group gI is tested. ■ QED.

Lemma: The reachability testing does not contain false negatives.

Assume that a concurrent program contains an interleaving, V_{error} , that contains concurrent errors $\Rightarrow V_{error} \in Reachability$.

In other words, $\exists V$ where $V \in Reachability$ and $V = V_{error}$

Given an execution trace as stated in precondition (7-8), the reachability testing will generate different interleavings which contain the concurrent errors. See Appendix C for the proof.

Proof for (7-13):

Proof that if a concurrent program contains an interleaving that contains concurrent errors, let's say V_{error} , then the V_{error} will exist in one of the race-equivalent groups.

$\exists gi$ where $i = 1, 2, 3, \dots$, number of race equivalent groups, so that $V_{error} \in gi$

From (7-9), the proposed method groups all interleavings generated from the existing reachability testing into several race-equivalent groups. From the lemma above, the existing reachability testing will generate the interleaving that contains the error, V_{error} , so it will be grouped into one of the race-equivalent groups. ~~We name the race-equivalent group that contains the V_{error} as gI .~~ ■ ■ QED.

Proof for (7-14):

B1 Different execution paths	B2 Different data
<ul style="list-style-type: none"> ● The proposed method identifies the set of use-defines that is affecting conditional statements in branches (<i>BranchRelUD</i>). ● Then it groups the interleavings with the same <i>BranchRelUD</i> into the same race-equivalent group. ● Interleavings with the same <i>BranchRelUD</i> will have the same branch outcomes. 	<ul style="list-style-type: none"> ● The proposed method identifies the set of use-defines that is affecting variables (<i>VarRelUD</i>). ● Then it groups the interleavings with the same <i>VarRelUD</i> into the same race-equivalent group. ● Variables within the interleavings with the same <i>VarRelUD</i> will refer to the same data.
↓ Implies	
The proposed method groups the interleavings with the same branch outcomes into the same race-equivalent group.	The proposed method groups the interleavings in which the variables' accesses refer to the same data into the same race-equivalent group.
↓ Implies	
For all members in the same race-equivalent group: The same thread will have the same sequence of lock/unlock and read/write operations to shared variables.	For all members in the same race-equivalent group: The variables' accesses will refer to the same data.
↓ Implies	
For all members in the same race-equivalent group: The thread that contains the concurrent error, let's say thread T_{error} , will have the same sequence of lock/unlock and read/write operations to shared variables. Hence the same concurrent errors exist in the thread T_{error} of all members in the same race-equivalent group. (7-15)	For all members in the same race-equivalent group: The variable that causes the error, let's say var_{error} , will refer to the same data. Hence the same concurrent errors exist when the var_{error} is accessing the data for all members in the same race-equivalent group. (7-16)

From (7-10), (7-15) and (7-16)

We will test one interleaving from each race-equivalent group (7-10). When the race-equivalent group that contains the V_{error} is tested, the same concurrent error will be detected no matter which member is selected. This is because all members in the same race-equivalent group will contain the same concurrent errors (7-15) (7-16). ■ QED

Figure 96 is an example for the case **B1**. The interleavings no.1 and no. 2 will be in the same race-equivalent group. Both will contain the same error. No matter which one is chosen (no. 1 or no. 2), the concurrent error will be detected.

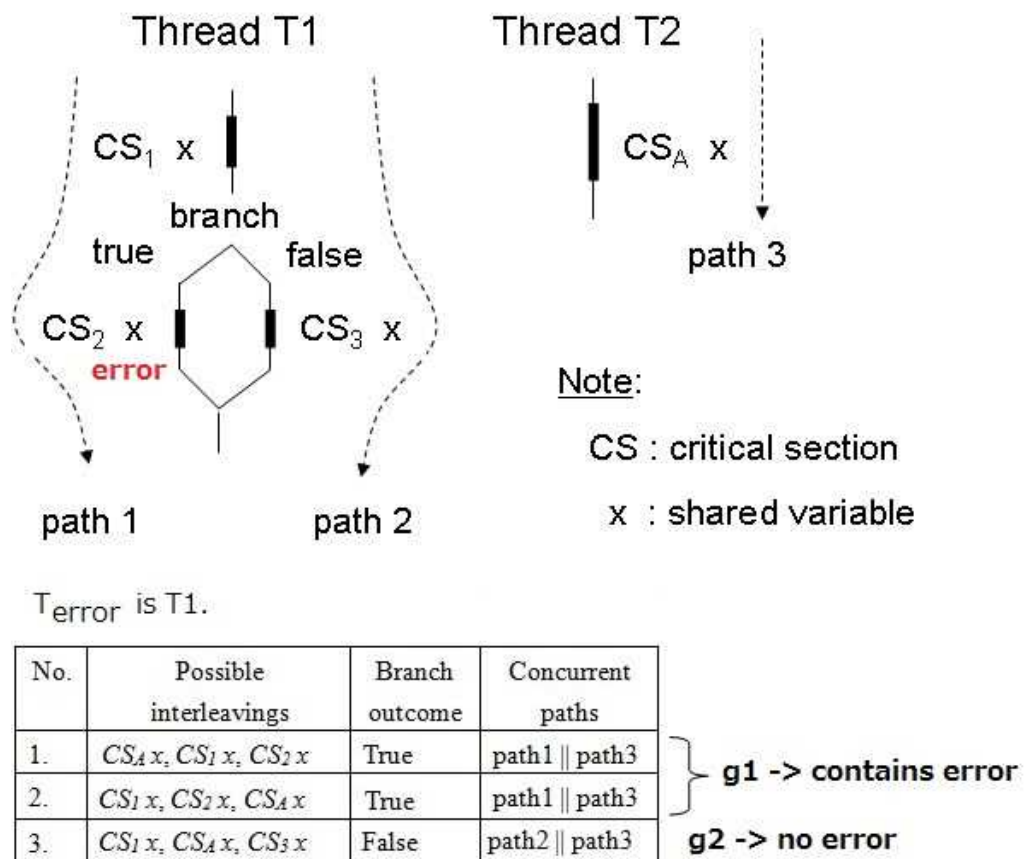


Figure 96. Example of a concurrent program with an error

7.6 Future Work

7.6.1 Correctness Criteria

Translating interrupt as a thread might reduce responsiveness as it would depend on scheduling. By measuring the responsiveness of interrupt and use it as one of the criteria to decide the correctness of multi-threaded concurrent program, our method can be applied for real time system.

7.6.2 Target Program

Currently, our proposed method is applied to the actual target program written in Java language. Another existing work from [Yu08] proposed prototyping for software testing and showed its benefits. Applying our method to a prototyping language could be one direction for further research.

7.6.3 Scope

Also for the future work, the method can be extended not only for debugging, but also apply it for testing all possible executions. In the current proposal, we concentrate on checking the possible interleavings and interrupt timings restricted for fixed values of the input variables. There are some existing systems for test generation for branch coverage. Some generates input data that exercises a selected branch [Prather87], [Gupta00] based on execution based approach. It is necessary to investigate whether we can utilize the existing method to extend our proposal for testing and also the possibility to help reducing false positives.

7.6.4 Reduction of the Load of Execution Trace

A checkpoint/restart scheme can reduce the load of execution trace. A checkpoint method allows a program to resume from a checkpoint, thus eliminating re-executing of the same portion of program code up to the checkpoint each time an execution trace is taken. This method is called “prefix-based testing” [Hwang95]. It allows starting non-deterministic testing from a specific program state other than the initial state. Our proposed method can also take the advantage of prefix-based testing. Here are the steps:

- Put check point at every parent node of the execution variant node in the

execution variant graph.

- For testing an execution variant, start from its parent node (not from beginning) and then execute it non-deterministically.

By employing a check point system, we avoid repeating the same execution up to the execution variant node. The efficiency can be improved by performing interleaving gradually step-by-step, accumulating the intermediate result to be utilized for the next step. By continuing from the last check point, the next debug step can be done in a minimum effort.

7.6.5 Reduction of the Need for Executing Test Cases

The verification of an execution path does not necessarily require the execution of the path. A new execution path can be identified and created by combining the branch-paths found in the previous execution trace without further testing the program. We use this technique to further reduce the need for executing the test cases.

An execution path from a thread contains sequence of branch-paths from each branch execution. A branch-path is an execution from one branch to the next branch in the execution trace of a thread. For each execution, the truth value of the branch-path could be either *true* or *false*. Suppose we obtain the following information from a trace:

- 1) Initial execution trace:
 - Thread *T1* creates branch-path *A*, thread *T2* creates branch-path *P*.
 - There is an execution path where branch-path *A* is concurrent with branch-path *P* (Figure 97(d)).
- 1) Execution trace from the test case:
 - Thread *T1* creates branch-path *B* instead of the branch-path *A*, thread *T2* creates branch-path *Q* instead of branch-path *P*.
 - There is an execution path where branch-path *B* is in concurrent with branch-path *Q* (Figure 97(c)).

Then the two new execution paths can be created by combining the information from

the initial execution trace and the execution trace from the test case without further executing the program:

- Branch-path *A* is in concurrent with branch-path *Q* (Figure 97(b)).
- Branch-path *B* is in concurrent with branch-path *P* (Figure 97(a)),

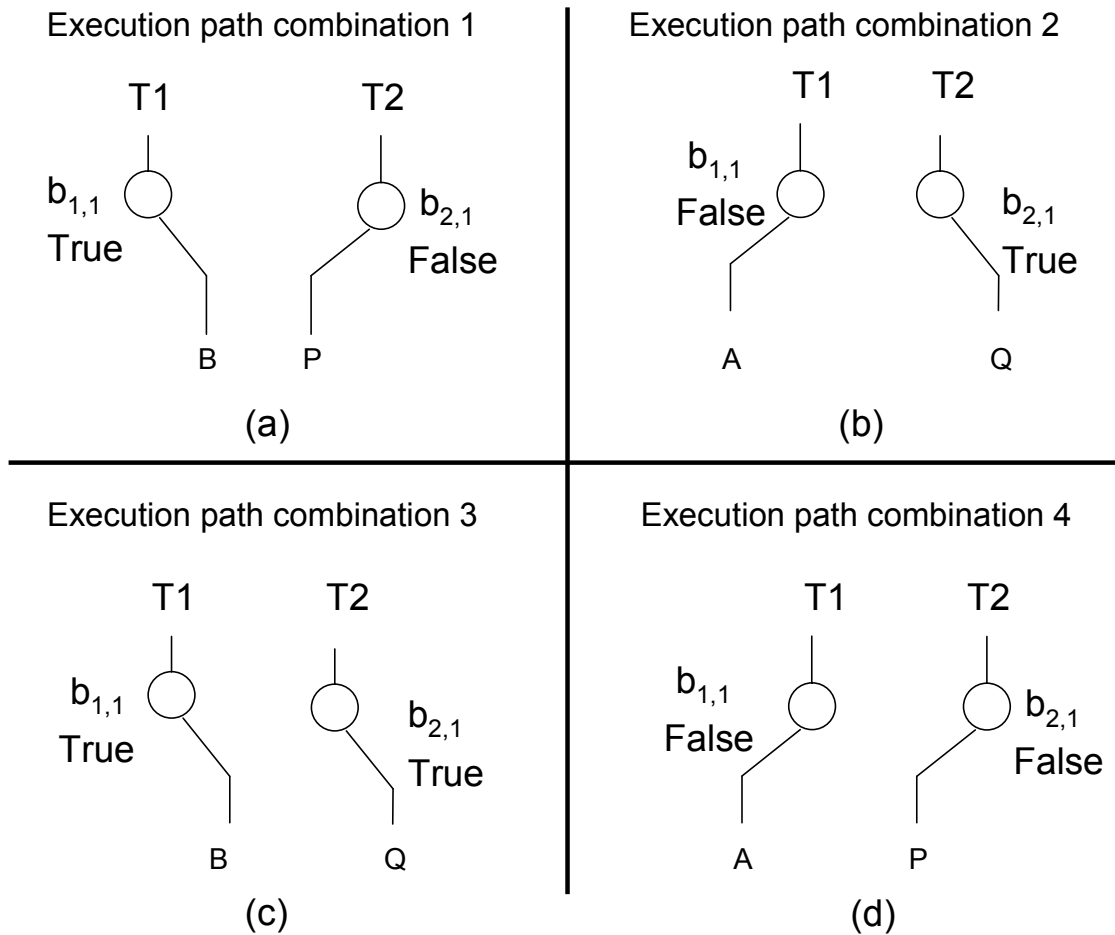


Figure 97. Example of execution paths combinations

The possible number of combinations for the branch-paths will be maximum when there is no nested branch. For n branches with no nested branch, the possible number of all combinations is 2^n . Figure 97 shows an example of two threads with one branch each, since there are two possible execution-paths for each thread, we have four possible combinations of execution-paths.

Since our program executions are limited by a fixed input for debugging purposes, only those possible execution paths under the given values of the input

variables will be executed. It means that not all branches will change the execution path. The test is finished when all the possible combination of execution-paths have been checked.

Algorithm 12 shows how to create a new execution path by utilizing information from execution trace.

Algorithm 12. Creating a new execution path by utilizing information from execution trace

```
Input:
- A test case
- Branch-paths history: contains previously executed branch-paths

Output:
- New execution-paths labeled as either “no race condition” or “potential race condition”

Step 1 If the test case changes the branch outcome then {
    create new execution paths by combining the new branch-path with the
    existing ones in the branch-paths history
    Step 1.1 for each new execution path {
        check the new execution path using existing race detector tool
        if no race was detected then {
            label the new execution path as "no race"
        }
        else {
            label the new execution path as "potential race"
        }
        add the new branch-path into branch-paths history
    }
}
```

The number of test cases can be reduced because we can ignore the next test cases that lead into the new execution path labeled as "no race". The new execution paths labeled as "potential race" might be a false positive which will become a race if exists a test case that could lead into the combination of execution-paths. This should be explored in the next test cases.

Chapter 8. Conclusions

In this dissertation, we have proposed efficient methods for reproducing multi-threaded concurrent program errors. Debugging concurrent multi-threaded programs is notoriously difficult because the exact timing that caused the error is normally unknown. Gathering trace information while executing a program using even the same input values can cause schedule and timings to be different that could lead the program into a different execution path, for example due to branching or loop, so the error cannot always be reproduced and detected by existing error detection tools.

In order to reproduce an error in a multi-threaded concurrent program, this research aims at realizing a deterministic replay which we call it "total replay". Total replay reproduces all possible executions caused by different thread interleavings and interrupt timings as test cases, whereas existing deterministic replay often reproduce only a selected execution. We focus on detecting errors, particularly race conditions, caused by interleavings of threads and different interrupt timings. It is intended to reproduce all possible execution paths within the scope determined by the limited information obtained from an execution trace. Even though the input values are fixed, the range of execution reproduction is still very large due to a wide range of different schedules and interrupt timings.

In order to realize total replay efficiently, we propose some methods for reducing the number of test cases. We observed that executions from different interleavings with the same combination of execution path between threads have the same access-manner to shared variables and data, so regarding the detection of race condition we can classify them into the same race-equivalent group. The non-existence of race condition in multi-threaded concurrent programs can be ensured by checking the lock consistency from all possible combinations of execution paths between threads. In that sense, interleavings that do not change execution path in a thread produce redundancy with respect to checking race conditions. Since an execution path in a thread is affected by branches, our

proposed method identifies only those interleavings that affect branch outcomes by utilizing data flow from the trace information to identify such redundancy. For our purpose, we extend the definition and notation of use-def chain to cover usage and definition of shared variables in multi-threaded. We first identify the set of operations that affect the conditional statement of a branch. Based on this analysis, which interleavings affect the branch outcomes can be determined.

The originality of the proposed method is as follows:

1) Reducing test cases.

- Grouping different interleaving that have the same locking consistency: The existing methods try to identify all interleavings which may affect shared variables whereas our method identifies only those interleavings which affect sequence of lock/unlock and read/write operations to shared variables. Different execution paths with the same locking consistency are grouped into the same race-equivalent group and tested only once. This significantly reduces the number of interleavings necessary for testing.
- Avoiding infeasible test cases: Infeasible test cases caused by synchronization mechanisms, such as a *wait-notify* mechanism, are identified and eliminated.

2) Reducing memory space required for generating test cases.

Our method exploits data dependency to generate only those test cases that might affect sequences of lock/unlock and shared variables. Our new proposed method requires smaller sized graphs for generating test cases compared to the existing reachability testing method. This means the required memory space is reduced.

3) Reducing the effort involved in checking race conditions.

Our method identifies only the parts of the execution trace whose sequences of lock/unlocks and shared variables might be affected by a new test case. Race conditions are then checked again only for those affected parts. For other unaffected parts, we can reuse the results from previous executions, thereby reducing the effort involved in checking race conditions.

We conducted some experiments on several real world Java open source programs to demonstrate the effectiveness of our proposed method. The

experimental results suggest that redundant interleavings can be identified and removed that lead to a significant reduction of test cases.

9. Glossary

A

access-manner - A sequence of operations in which a thread has acquired a lock, has accessed a shared variable, and has released the corresponding lock (section 3.9 Access-Manner, page 45).

advice – An Aspect-oriented term referring to a piece of code to be executed in a *pointcut* (section 6.3 Tracing). See also *pointcut* (section 6.3 Tracing).

Aspect-oriented programming - A programming paradigm that aims to increase modularity by allowing the separation of concerns.

AspectJ - An aspect-oriented extension to the Java programming language. We use it for tracing Java multi-threaded concurrent programs (section 6.3 Tracing, page 115).

B

branch-affect group - A branch-affect group for a branch *b* contains a set of execution variants that would cause the same branch outcome for the branch *b*, which is either *true* or *false*.

branch outcome - The truth value within a conditional statement of a branch during a program execution, that is whether *true* or *false*.

C

concurrency – “A condition that exists when at least two threads are making progress. A more generalized form of parallelism that can include time-slicing as a form of virtual

parallelism” [Oracle10].

concurrent dependency graph - A directed graph representing use-define relations in an execution of a concurrent program for identifying data dependencies of shared variables (subsection 5.3.2 Concurrent Dependency Graph, page 89).

concurrency control – A control mechanism for concurrent programs to avoid race conditions.

concurrent set of access-manners (MANNERS) - A collection of sets of access-manners from all the threads within a concurrent execution path of a concurrent program (section 3.9 Access-Manner, page 45).

consistent lock for a shared variable - A lock which is acquired by any threads before accessing the shared variable (section 3.2 Race Conditions, page 35).

D

define - A write operation of some value to a variable (section 3.13 Use-Define, page 55).

deterministic replay - Executing a concurrent program with exactly the same interleaving as previous execution (section 2.7 Deterministic Replay, page 24).

deterministic testing - Executing a concurrent program with the interleaving as specified by a test case (section 2.8 Deterministic Testing, page 25).

E

execution-variant - A different read/write sequence that affects the values of shared variables (section 4.2 Approach, page 59).

F

false alarm - a false positive or a false negative.

false positive - reporting errors which actually do not exist.

false negative - test results that do not indicate the presence of errors which are actually present.

G

guideline - A set of use-defines obtained by traversing a concurrent dependency graphs (subsection 5.3.3 Traversing a Concurrent Dependency Graph). It is used for constructing test cases.

N

no-race - A concurrent-pair of access-manners is said to be no-race if the two access-manners can be interleaved without race conditions (section 3.12 No-Race). See also concurrent-pair of access-manners (section 3.11 Concurrent-Pair of Access-Manners)

P

parallelism – A principle that large problems can often be divided into smaller ones, which are then solved simultaneously in parallel [Gottlieb89]. This dissertation discusses about concurrency instead of parallelism, see the definition about **concurrency**.

pointcut – An Aspect-oriented term to specify a location within an execution of a program where an *advice* has to be executed (section 6.3 Tracing). See also *advice* (section 6.3 Tracing).

R

race condition - A condition when there is a concurrent access to a shared variable which is not protected by consistent locks (section 3.2 Race Conditions, page 35).

race-equivalent - Two executions of a concurrent program are race-equivalent if they have the same set of access-manners (MANNERS) (section 3.10 Race-Equivalent, page 48).

race-equivalent group - A group contains concurrent execution paths that are race-equivalent (section 3.10 Race-Equivalent, page 48). See also race-equivalent (section 3.10 Race-Equivalent, page 48)

reachability testing method - One of testing methods for concurrent programs that performs an efficient exploration of different sequences of read/write operations which affect values of shared variables (section 4.2 Approach, page 59).

reference variable - A variable that refers to an object in Java programming language. This is similar to a pointer in C programming language (subsection 3.4.1 Reference Variable, page 38).

reflection – An Aspect-oriented term for getting information about program execution (section 6.3 Tracing, page 115).

S

shared variable - A variable which is accessed by more than one thread.

sequential consistency - A multiprocessing system had sequential consistency if "the results of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program." [Lamport79]

U

use - A read operation on a variable (section 3.13 Use-Define, page 55).

use-define - A relation consisting of a usage “use” of a variable and the definition “define” of the variable (section 3.13 Use-Define, page 55). See also “use” and “define” (section 3.13 Use-Define, page 55)

V

variant graph - A directed graph for deriving different read/write sequences from an execution trace (section 4.2 Approach, page 59).

W

well formed - An access to a shared variable is said to be well formed if all threads acquire consistent locks before accessing the shared variable, and then perform an unlock operation to release the corresponding locks (section 3.2 Race Conditions, page 35). See also "consistent lock" (section 3.2 Race Conditions, page 35).

10. References

- [1] [Abdelqawy12] Abdelqawy D., Kamel A. and Omara F. "A Survey on Testing Concurrent and Multi-Threaded Applications Tools and Methodologies". International Conference on Informatics & Applications (ICIA2012), Malaysia, pp.459-471. 2012.
- [2] [Adve91] Adve S. V., Hill M.D., Miller B.P., and Netzer R.H.B. "Detecting Data Races on Weak Memory Systems," Proceedings of the 18th Annual International Symposium on Computer Architecture (ISCA), pp.234–243. May 1991.
- [3] [ApacheDerby10] Apache Derby. 2010. Available: <http://db.apache.org/derby/>
- [4] [ApachePool06] Apache Commons Pool, 2006. Available at: <http://jakarta.apache.org/commons/pool/>
- [5] [Artho01] Artho C. "Finding Faults in Multi-Threaded Programs," Master thesis, ETH Zurich, Switzerland. March 2001.
- [6] [Beckman06] Beckman N. E. "A Survey of Methods for Preventing Race Conditions," Literature survey of Analysis of Software Artifacts. Carnegie Mellon University. 2006. Available at: http://www.cs.cmu.edu/~nbeckman/papers/race_detection_survey.pdf
- [7] [Ben03] Ben-Asher Y., Farchi E., and Eytani Y. "Heuristics for Finding Concurrent Bugs," Proceedings of the 17th International Symposium on Parallel and Distributed Processing, IEEE Computer Society. Washington, DC, USA. 2003.
- [8] [Ben06] Ben-Asher Y., Farchi E., Eytani, Y.; Ur S. "Noise Makers Need to Know Where to be Silent – Producing Schedules That Find Bugs," Second International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA). Nov 2006.
- [9] [Baur03] Baur M. C. "Instrumenting Java Bytecode to Replay Execution Traces of Multithreaded Programs," Formal Methods Group, Computer Systems Institute, Swiss Federal Institute of Technology (ETH Zurich). 2003.
- [10] [Bertolino94] Bertolino A. and Marre M. "Automatic Generation of Path Covers Based on the Control Flow Analysis of Computer Programs." IEEE Transactions on Software Engineering, Vol.20, No.12, pp.885-899. December 1994.
- [11] [Bochmann94] Bochmann G. V. and Petrenko A. "Protocol Testing: Review of Methods and Relevance for Software Testing," Proceedings of the 1994 ACM SIGSOFT international symposium on Software testing and analysis, Seattle, Washington, United States. 1994.
- [12] [Boyapati01] Boyapati C. and Rinard M. "A Parameterized Type System for Race-Free Java Programs." Proceedings of the 16th ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications (OOPSLA), Oct. 2001.
- [13] [Bron05] Bron A., Farchi E., Magid Y., Nir Y., and Ur S. "Applications of

- Synchronization Coverage,” Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming. Chicago, IL, USA, pp.206-212. 2005.
- [14][Baralga10] Baralga. 2010. Available: <http://baralga.origo.ethz.ch/>
- [15][BlueJ09] BlueJ. The interactive Java environment. 2009. Available: <http://www.bluej.org/>
- [16][Carver98] Carver R. H. and Tai K. C. “Use of sequencing constraints for specification-based testing of concurrent programs,” IEEE Transactions on Software Engineering, Vol.24, Issue 6. 1998.
- [17][Caballero07] Caballero R., Hermanns C., and Kuchen H. “Algorithmic Debugging of Java Programs”. Electronic Notes in Theoretical Computer Science 177, pp.75-89. 2007.
- [18][Carver04] Carver R. and Lei Y. "A General Model for Reachability Testing of Concurrent Programs." International Conference on Formal Engineering Methods, pp.76-98. Nov 2004.
- [19][Choi91] Choi J. D., Miller B. P., and Netzer R. H. B. “Techniques for Debugging Parallel Programs with Flowback Analysis,” ACM Transactions on Programming Languages and Systems, Vol. 13, No. 4, pp. 491–530. October 1991.
- [20][Chung01] Chung I. S. and Kim B. M. "A New Approach to Deterministic Execution Testing for Concurrent Programs,” IEICE TRANSACTIONS on Information and Systems, Vol.E84-D, No.12. 2001.
- [21][Jong98] Choi J. D. and Srinivasan H. “Deterministic Replay of Multithreaded Java Applications,” ACM SIGMETRICS SPDT98, Oregon. August 1998.
- [22][Jong02] Choi J. D. and Zeller A. "Isolating Failure-Inducing Thread Schedules," International Symposium on Software Testing and Analysis (ISSTA2002), Via di Ripetta, Rome – Italy. July 2002.
- [23][Chris01] Christiaens M. and Bosschere K. De. “TRaDe, A Topological Approach to on-the-fly Race Detection in Java Programs,” Proceedings of the Java Virtual Machine Research and Technology Symposium (JVM), Apr. 2001.
- [24][Clarke00] Clarke E. M., Grumberg O., and Peled D. A. Model Checking, The MIT Press. January 2000.
- [25][Cleaveland94] Cleaveland R., Parrow J., and Steffen B. “The Concurrency Workbench: A Semantics Based Tool for the Verification of Concurrent Systems,” ACM Transactions on Programming Languages and Systems. 1994.
- [26][Crummey91] Crummey J. M. “On-the-fly Detection of Data Races for Programs with Nested Fork-Join Parallelism,” Proceedings of Supercomputing. November 1991.
- [27][Dinning90] Dinning A. and Schonberg E. “An Empirical Comparison of Monitoring Algorithms for Access Anomaly Detection,” Proceedings of the 2nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP), pp.1–10. 1990.
- [28][Edelstein03] Edelstein O., Farchi E., Goldin E., Nir Y., Ratsaby G, and Ur S. "Framework for Testing Multi-Threaded Java Programs," Concurrency and Computation: Practice and Experience, John Wiley & Sons. 2003.
- [29][Engler03] Engler D. and Ashcraft K. "RacerX: Effective, Static Detection of

- Race Conditions and Deadlocks." Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP), pp.237–252. October 2003.
- [30][Eytani07] Eytani Y. and Latvala T. "Explaining Intermittent Concurrent Bugs by Minimizing Scheduling Noise," Lecture Notes in Computer Science. Hardware and Software, Verification and Testing, Vol.4383. 2007.
- [31][Factor96] Factor M., Farchi E., Lichtenstein Y., and Malka Y. "Testing Concurrent Programs: A Formal Evaluation of Coverage Criteria," Seventh Israeli Conference on Computer-Based Systems and Software Engineering (ICCSSE '96), Herzliya, ISRAEL. ISBN: 0-8186-7536-5. 1996.
- [32][Flanagan00] Flanagan C. and S. N. Freund. "Type-based Race Detection for Java." Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), pp.219–232. 2000.
- [33][Flanagan05] Flanagan C. and Godefroid P. "Dynamic Partial-Order Reduction for Model Checking Software". Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. ACM SIGPLAN. Vol.40, Issue 1, pp.110-121. January 2005.
- [34][Godefroid96] Godefroid P. "Partial-Order Methods for the Verification of Concurrent Systems - An Approach to the State-Explosion Problem", Lecture Notes in Computer Science. Springer-Verlag, Vol.1032. January 1996.
- [35][Godefroid97] Godefroid P. "Model Checking for Programming Languages using VeriSoft." Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Paris, France. 1997.
- [36][Gottlieb89] Gottlieb A. and Almasi G. S. "Highly parallel computing." Redwood City, Calif. Benjamin/Cummings. ISBN 0-8053-0177-1. 1989.
- [37][Gradecki03] Gradecki J.D. and Lesiecki N. Mastering AspectJ: Aspect-Oriented Programming in Java, John Willey & Sons (Asia) Pte Ltd. 2003.
- [38][Gupta00] Gupta N., Mathur A. P., and Soffa M. L. "Generating Test Data for Branch Coverage," In Proc. of the International Conference on Automated Software Engineering. 2000.
- [39][Hwang95] Hwang G.H, Tai K.C, Huang T.L. "Reachability Testing: An Approach to Testing Concurrent Software." International Journal of Software Engineering and Knowledge Engineering. 1995.
- [40][Havelund00] Havelund K. and Pressburger T. "Model checking JAVA programs using JAVA PathFinder," International Journal on Software Tools for Technology Transfer, Vol.2, No.4, pp. 366-381. 2000.
- [41][Hazewinkel01] Hazewinkel M. "Stirling Formula." Encyclopedia of Mathematics. Springer. ISBN 978-1-55608-010-4. 2001.
- [42][Henzinger04] Henzinger T., Jhala R., and Majumder R. "Race Checking by Context Inference." Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI). June 2004.
- [43][Holzmann91] Holzmann G. J. Design and Validation of Computer Protocols, Prentice Hall. 1991.
- [44][Hwang95] Hwang G. H., Tai K. C., and Huang T. L. "Reachability Testing: An Approach to Testing Concurrent Software." International Journal of Software Engineering and Knowledge Engineering, Vol.5, No.4, pp.493-510. 1995.
- [45][Huang11] Huang J., Zhou J., and Zhang C. "Scaling Predictive Analysis of

- Concurrent Programs by Removing Trace Redundancy,” ACM Transactions on Software Engineering and Methodology, Vol.22, Issue 1. 2011.
- [46][Jasaitis13] Jasaitis R., Prapuolenis J, and Bareisa E. "Distributed System Resource Racing Conditions Automated Testing Method," Technology Education Management Informatics (TEM) Journal, Vol.2, No.4, pp.283-290. 2013.
- [47][jNet09] jNetMap. June 2009. Available: <http://www.rakudave.ch/?q=jnetmap>
- [48][JTelnet03] Kristjansson D. JTelnet. 2003. Available: <http://mrl.nyu.edu/~kristja/jtelnet.html>
- [49][Kojima09] Kojima H., Kakuda Y., Takahashi J., and Ohta T. “A Model for Concurrent States and Its Coverage Criteria,” International Symposium on Autonomous Decentralized Systems, ISADS '09, pp.23-25. 2009.
- [50][Lu06] Lu S., Tucek J., Qin F., and Zhou Y., "Avio: Detecting Atomicity Violations via Access Interleaving Invariants," International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS). 2006.
- [51][Lamport78] Lamport L. “Time, Clocks, and the Ordering of Events in a Distributed System,” Communications of the ACM, Vol.21, No.7, pp.558–565. July 1978.
- [52][Lamport79] Lamport L. "How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs," IEEE Trans. Comput. C-28,9, pp.690-691. September 1979.
- [53][Lea99] Lea D. Concurrent programming in Java: Design Principles and Patterns, Second edition. Addison-Wesley. November 1999.
- [54][Lei04] Lei Y. and Carver R. “A New Algorithm for Reachability Testing of Concurrent Programs,” Proceedings of the 16th IEEE International Symposium on Software Reliability Engineering, pp.346 - 355. 2005.
- [55][Lei06] Lei Y. and Carver R.H. "Reachability Testing of Concurrent Programs." IEEE Transactions on Software Engineering, Vol.32, Issue 6, pp.382 - 403. June 2006.
- [56][Lu07] Lu S., Jiang W., and Zhou Y. "A Study of Interleaving Coverage Criteria," Proceedings of the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on the foundations of software. 2007.
- [57][Lei07] Lei Y., R.H. Carver, Kacker R., and Kung D. “A Combinatorial Testing Strategy for Concurrent Programs”. Software Testing, Verification & Reliability. Vol. 17, Issue 4, pp.207 – 225. December 2007.
- [58][Lee96] Lee E.K., Thekkath C., and Petal A. “Distributed Virtual Disks,” Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VII). 1996.
- [59][Dowell89] McDowell C. E. and Helmbold D.P. “Debugging Concurrent Programs”, ACM Computing Surveys, Vol.21, No.4, pp.593-622. 1989.
- [60][Mutilin06] Mutilin V. “Concurrent Testing of Java Components using Java PathFinder,” Proceedings of the Second International Symposium on Leveraging Applications of Formal Methods, Verification and Validation, pp.53-59. 2006.
- [61][Musuvathi07] Musuvathi M., Qadeer S., and Ball T. "CHESS: A Systematic Testing Tool for Concurrent Software," Microsoft Research Technical Report.

- MSR-TR-2007-149. 2007.
- [62][JoBo06] Matuschek D. JoBo: Web Spider. Dec 2006. Available at: <http://www.matuschek.net/jobbo-menu/>
- [63][Naik06] Naik M., Aiken A., and Whaley J. "Effective static race detection for Java," Proceeding PLDI '06 Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation, Vol.41, Issue 6, ISBN:1-59593-320-4. June 2006.
- [64][Park11] Park C., Sen K., Hargrove P., and Iancu C. "Efficient Data Race Detection for Distributed Memory Parallel Programs," SC11, November 12-18, 2011, Seattle, Washington, USA Copyright 2011 ACM 978-1-4503-0771-0/11/11
- [65][Pugh07] Pugh W. and Ayewah N. "Unit Testing Concurrent Software," Proceedings of the twenty-second IEEE/ACM international conference on Automated Software Engineering, Atlanta, pp.513-516. November 2007.
- [66][Netzer91] Netzer R. H.B. and Miller B. P. "Improving the Accuracy of Data Race Detection," Proceedings of the 1991 Conference on the Principles and Practice of Parallel Programming. 1991.
- [67][Nishiyama04] Nishiyama H. "Detecting Data Races using Dynamic Escape Analysis based on Read Barrier," Proceedings of the 3rd Virtual Machine Research and Technology Symposium (VM). May 2004.
- [68][Oaks04] Oaks S. and Wong H. Java Threads, Third Edition, O'Reilly. 2004.
- [69][Oracle10] Oracle Corporation. Multithreaded Programming Guide. Oracle Help Center. 2010.
<https://docs.oracle.com/cd/E19455-01/806-5257/6je9h032b/index.html>
- [70][PLDI06] PLDI Experimental Results. 2006. Available: http://www.cc.gatech.edu/~mnaik7/research/pldi06_results.html
- [71][Plexousakis05] Plexousakis D. "Concurrency Control". Univ. of Crete. CS460 Fall 2005.
- [72][Prather87] Prather R.E. and Myers J.P. Jr, "The Path Prefix Software Testing Strategy." IEEE Transactions on Software Engineering. Vol.SE-13, Issue 7. July 1987.
- [73][Praun01] Praun C. and Gross T. "Object Race Detection," Proceedings of the 16th ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications (OOPSLA), pp. 70–82. 2001.
- [74][Ramalingam02] Ramalingam G. "Context-Sensitive Synchronization-Sensitive Analysis is Undecidable". ACM Transactions on Programming Languages and Systems, Vol.22, No.2, pp 416–430. 2000.
- [75][Rahul08] Rahul V.P. and Boby G. "Tools and Techniques to Identify Concurrency Issues." MSDN Magazine. Microsoft. 2008. Available at: <http://msdn.microsoft.com/en-us/magazine/cc546569.aspx>
- [76][Savage97] Savage S., Burrows M., Nelson G., Sobalvarro P., and Anderson T. "Eraser: A Dynamic Data Race Detector for Multithreaded Programs," ACM Transactions on Computer Systems. 1997.
- [77][Sen06] Sen K. and Agha G. "Concolic Testing of Multithreaded Programs and Its Application to Testing Security Protocols," UIUC Technical Report. Department of Computer Science, January 2006.
- [78][Sen06_b] Sen K. and Agha G. "CUTE and jCUTE: Concolic Unit Testing and

- Explicit Path Model-Checking Tools," In CAV. Springer, pp.419-423. 2006.
- [79][Setiadi04] Setiadi T. E., Nakayama K., Kobayashi Y., and Maekawa M. "Identifying Candidate Invariant Conditions of Running Program". The 8th IASTED International Conference on Software Engineering and Applications. MIT, Cambridge, MA, USA. 2004.
- [80][Setiadi04_2] Setiadi T. E., Nakayama K., Kobayashi Y., and Maekawa M. "Analyzing Invariant Condition of Running Java Program". The 16th International Conference of Software Engineering and Knowledge Engineering (SEKE'04). Alberta, Canada. 2004.
- [81][Setiadi05] Setiadi T. E., Nakayama K., Kobayashi Y., and Maekawa M. "Interactive Environment for Smart Summarization of Execution Trace". International Symposium on Communications and Information Technology (ISCIT), IEEE International. Beijing, China. 2005.
- [82][Setiadi10] Setiadi T. E., Nakayama K., Ohsuga A., and Maekawa M. "Efficient Replay for Reproducing Concurrent Program Errors," International Journal of Computational Science Vol 4, No. 2. pp. 129-155. April 2010. <http://www.gip.hk/ijcs/Internet%20V4N2/Abstract/v4n2%20ab2.pdf>
- [83][Setiadi13] Setiadi T. E., Ohsuga A., and Maekawa M. "Efficient Execution Path Exploration for Detecting Races in Concurrent Programs," IAENG International Journal of Computer Science, Vol.40, Issue 3, pp.143–163, September 2013. http://www.iaeng.org/IJCS/issues_v40/issue_3/IJCS_40_3_02.pdf
- [84][Setiadi14] Setiadi T. E., Ohsuga A., and Maekawa M. "Efficient Test Case Generation for Detecting Race Conditions," IAENG International Journal of Computer Science, Vol.41, Issue 2, pp.112-130. May 2014. http://www.iaeng.org/IJCS/issues_v41/issue_2/IJCS_41_2_04.pdf
- [85][Sterling93] Sterling N. "Warlock: a static data race analysis tool." Proceedings of USENIX Winter Technical Conference, January 1993.
- [86][Stoller02] Stoller S. D. "Testing Concurrent Java Programs using Randomized Scheduling," Proceedings of the Second Workshop on Runtime Verification (RV), Vol.70, No.4, Electronic Notes in Theoretical Computer Science. © Elsevier, 2002.
- [87][Sebek02] Sebek F. "Instruction Cache Memory Issues in Real-Time Systems," Technology Licentiate Thesis. Computer Architecture Lab. Department of Computer Science and Engineering. Malardalen University. Vasteras, Sweden, ISBN 97-88834-38-7. October 2002.
- [88][Takahashi08] Takahashi J., Kojima H., and Furukawa Z. "Coverage Based Testing for Concurrent Software," The 28th IEEE International Conference on Distributed Computing Systems Workshops, pp.533-538. 2008.
- [89][Taylor92] Taylor R. N., Levine D. L., and Kelly C. D. "Structural Testing of Concurrent Programs," IEEE Trans. Soft. Eng, Vol.18, No.3, pp.206-215. 1992.
- [90][Total10] TotalView Technologies. Getting Started with Replay Engine. 2010. Available: http://www.totalviewtech.com/support/documentation/pdf/ReplayEngine1-7_GettingStarted.pdf
- [91][Visser04] Visser W., Pasareanu C., and Khurshid S. "Test Input Generation with Java PathFinder," Proceedings of ISSTA, Boston, MA. July 2004.
- [92][Weblech02] Weblech-0.0.3. WebLech: web site download/mirror tool. 2002.

- <http://weblech.sourceforge.net/>
- [93][Wbhv07] Web-Harvest. <http://web-harvest.sourceforge.net/> 2007.
- [94][Yahoo] Yahoo. Available at: <http://www.yahoo.com>
- [95][Yang98] Yang C., Souter A. L., and Pollock L. L., "All-du-path coverage for parallel programs." International Symposium on Software Testing and Analysis, pp.153-162, 1998.
- [96][Yuan05] Yu Y., Rodeheffer T., and Chen W. "RaceTrack: Efficient Detection of Data Race Conditions via Adaptive Tracking, " ACM Symposium on Operating Systems Principles (SOSP). 2005.
- [97][Yang98] Yang C., Souter A.L., and Pollock L.L., "All-Du-Path Coverage for Parallel Programs," International Symposium on Software Testing and Analysis, pp.153-162. 1998.
- [98][Yang03] Yang C.D. and Pollock L.L. "All-Uses Testing of Shared Memory Parallel Programs," Software Testing, Verification, and Reliability (SVTR) Journal, Vol.13, No.1, pp.3-24. 2003.
- [99][Yang97] Yang C.S. and Pollock L. "An Algorithm for All-Du-Path Testing Coverage of Shared Memory Parallel Programs," Asian Test Symposium, pp.263-268. 1997.
- [100][Yu08] Yu L. "Prototyping, Domain Specific Language, and Testing," Engineering Letters, International Association of Engineers (IAENG), Vol.16, Issue 1. 19 February 2008.

11. Appendices

Appendix A

Example of the calculation for the number of possible interleavings.

Let:

m : number of threads, $m \geq 2$

n : number of interleaving dependent read/write operations, $n \geq 0$

$n = 0\%$ means there will be no concurrent errors.

n_{T1} : number of interleaving dependent read/write operations in thread $T1$

interleaving dependent read/write operations: read/write operations to shared variables in which the value of the shared variables can be affected by the interleavings of the read/write operations.

$$\text{Number of possible interleavings} = \frac{n!}{(n_{T1})! \cdot (n_{T2})! \cdot (n_{T3})! \cdot \dots \cdot (n_{Tm})!} = n! \cdot \prod_{i=1}^m \frac{1}{(n_{Ti})!}$$

Examples:

$n = 3, n_{T1} = 2, n_{T2} = 1$

The number of possible interleavings is calculated using equation (7-1).

$$\frac{3!}{2! \cdot 1!} = 3$$

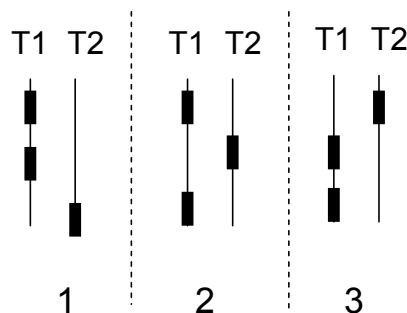


Figure 98. Example of possible interleavings for 2 threads and 3 operations

$n = 4, n_{T1} = 2, n_{T2} = 2$

$$\frac{4!}{2!2!} = 6$$

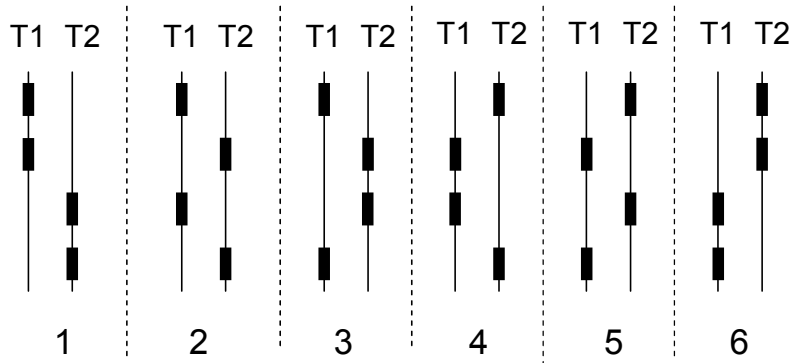


Figure 99. Example of possible interleavings for 2 threads and 3 operations

Appendix B

Approximation of equation (7-3) using Stirling's approximation (7-4) [Hazewinkel01] for calculating the computational complexity.

$$\begin{aligned}
 \frac{n!}{\left(\left(\frac{n}{m}\right)!\right)^m} &\cong \frac{\sqrt{2\pi n}\left(\frac{n}{e}\right)^n}{\left(\sqrt{2\pi\frac{n}{m}}\left(\frac{n}{m\cdot e}\right)^{\frac{n}{m}}\right)^m} = \frac{\sqrt{2\pi n}\left(\frac{n}{e}\right)^n}{\left(\sqrt{2\pi\frac{n}{m}}\right)^m \left(\frac{n}{m\cdot e}\right)^n} \\
 &= \frac{\sqrt{2\pi n}\left(\frac{n}{e}\right)^n}{(\sqrt{2\pi n})^m \left(\sqrt{\frac{1}{m}}\right)^m \left(\frac{1}{m}\right)^n \left(\frac{n}{e}\right)^n} = \frac{\sqrt{2\pi n}}{(\sqrt{2\pi n})^m \left(\sqrt{\frac{1}{m}}\right)^m \left(\frac{1}{m}\right)^n} \\
 &= \frac{1}{(\sqrt{2\pi n})^{(m-1)} \left(\sqrt{\frac{1}{m}}\right)^m \left(\frac{1}{m}\right)^n} = \frac{1}{(\sqrt{2\pi n})^{(m-1)} m^{-0.5m} \cdot m^{-n}} \\
 &= \frac{m^{0.5m} \cdot m^n}{(\sqrt{2\pi n})^{(m-1)}} \quad (7-5)
 \end{aligned}$$

Appendix C

Proof: Given an execution trace with the same input as when the concurrent errors

occurred, the reachability testing will generate different interleavings which contain the concurrent errors.

Reachability testing method generates different interleavings by:

- **Ignoring** the order of interleaving **independent** operations. (characteristic 1)
- **Considering** only the order interleaving **dependent** operations. (characteristic 2)

From the characteristic 2, the reachability testing method will generate all different interleavings that are affecting the values of shared variables. This overcomes the difficulties in detecting/reproducing the concurrent errors:

- A) Interleavings cause different execution paths. -> characteristic 2 affects the values of shared variables, then affecting conditional statements, then affecting the branches causing different execution paths.
- B) Interleavings cause variables to refer to different data. -> will be directly explored by characteristic 2.

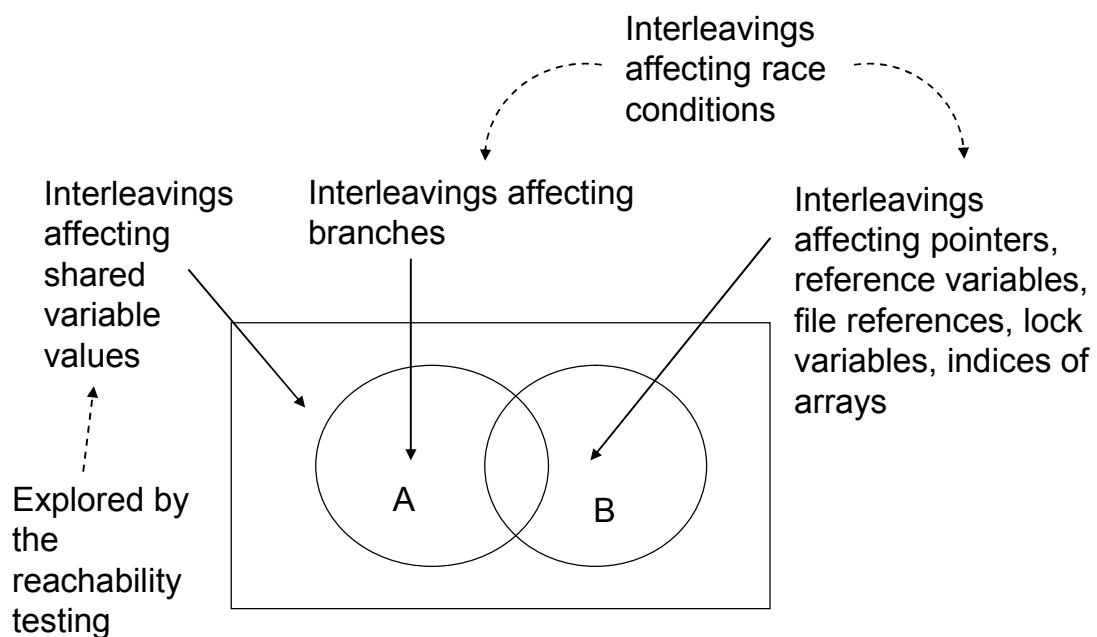


Figure 100. Type of interleavings in a concurrent program

About the Author

Theodorus Eric Setiadi. He received his Engineering Degree in Electrical Engineering and a Masters Degree in Computer System Engineering from the Institute of Technology, Bandung, Indonesia, in 2000 and 2002, respectively. He pursued his PhD degree at the Graduate School of Information Systems, University of Electro-Communications, Tokyo, Japan with the support from the Jinnai International Student Scholarship. His research interests are debugging systems and execution trace analysis. He has working experiences in developing and verifying software. He is now working as a technical consultant related to finance.