Guided Testing of Concurrent Programs Using Value Schedules

by

Jun Chen

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Computer Science

Waterloo, Ontario, Canada, 2009

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Testing concurrent programs remains a difficult task due to the non-deterministic nature of concurrent execution. Many approaches have been proposed to tackle the complexity of uncovering potential concurrency bugs. Static analysis tackles the problem by analyzing a concurrent program looking for situations/patterns that might lead to possible errors during execution. In general, static analysis cannot precisely locate all possible concurrent errors. Dynamic testing examines and controls a program during its execution also looking for situations/patterns that might lead to possible errors during execution. In general, dynamic testing needs to examine all possible execution paths to detect all errors, which is intractable.

Motivated by these observation, a new testing technique is developed that uses a collaboration between static analysis and dynamic testing to find the first potential error but using less time and space. In the new collaboration scheme, static analysis and dynamic testing interact iteratively throughout the testing process. Static analysis provides coarse-grained flow-information to guide the dynamic testing through the relevant search space, while dynamic testing collects concrete runtime-information during the guided exploration. The concrete runtime-information provides feedback to the static analysis to refine its analysis, which is then feed forward to provide more precise guidance of the dynamic testing. The new collaborative technique is able to uncover the first concurrency-related bug in a program faster using less storage than the state-of-the-art dynamic testing-tool Java PathFinder. The implementation of the collaborative technique consists of a static-analysis module based on Soot and a dynamic-analysis module based on Java PathFinder.

Acknowledgements

First of all, I would like to dedicate my wholehearted thanks to my supervisor Dr. Steve MacDonald and Dr. Peter Buhr. No words can fully express my gratitude for your abundant support and guidance in the past five years. I am forever in your debt.

I would like to greatly thank my Ph.D. committee members: Dr. Jo Atlee, Dr. Ondrej Lhotak and Dr. Patrick Lam as well as my external examiner Dr. Scott Stoller. Thank you very much for carefully reading my thesis and providing useful comments and advices.

I would like to greatly thank my parents for their support, helpful advices and absolute trust. I would like to give plenty of appreciations to all my friends from PLG lab: Brad Lushman (with his accordion music), Ashif Harji (with his technical support), Ghulam Lashari (with his cheerful attitude), Maheedhar Kolla (with his swami-knowledge), Azin Ashkan (with her kindness) and Mona Mojdeh (for generous party-hosting). Many thanks to my friends inside and outside of school, Guangrang Zhu, Xin Guan, Xiwen Hou, Qian Jia, Zhirong Li, Zhengrong Li, Wei Zhou, Yi Zhang, JingChi Chen, Zhuliang Chen, YunFeng Lin, Yinwen Chen, Wei Li, Jie Zhang, Haitian Zhao, and Shuang Li.

Dedication

This is dedicated to my parents.

Contents

List of Tables

List of Figures

List of Abbreviations

CCFG	Concurrent Control Flow Graph
CFG	Control Flow Graph
DPOR	Dynamic Partial Order Reduction
JPF	Java PathFinder
MHP	May Happen in Parallel
MM	Concurrent Monitor-Entry Pair
PEG	Parallel Execution Graph
POR	Partial Order Reduction
RW	Concurrent Read/Write Pair
SLOC	Line of Code for Source Program
WM	Concurrent Wait/Monitor-Entry Pair
WN	Concurrent Wait/Notify Pair
WW	Concurrent Wait/Wait Pair

Chapter 1

Introduction

Concurrent programs are gaining importance because of the increasing availability of multi-thread/core/processor computers. However, concurrent programs are much harder to debug than sequential programs. First, concurrent semantics introduce new types of bugs. For example, incorrect use of synchronization and mutual exclusion semantics can lead to concurrency-related bugs such as a race condition [11] and a deadlock [11]. Second, these concurrent bugs are notoriously hard to locate and fix, because they may only trigger an error under certain rarely-executed thread interleavings resulting from nondeterministic execution. Traditional testing methods, such as systematically inserting print statements, are no longer sufficient to pinpoint a bug because every run may follow a different thread interleaving. Moreover, an inserted print statement can lead to a probe effect [29] hiding the bug.

1.1 Problem Definition

The problems this thesis addresses are discussed in this section using the simple concurrent Java program in Figure 1.1, which has two concurrent threads. Each thread tries to access some variables, e.g. object a and its field $a.val$ are accessed by both threads. The program generates a *DivisionByZeroException* if the read of $a.val$ at line 2 for $Thread2$, reads the value 0 given by the assignment statement at line 1 of $Thread1$.

One solution for identifying concurrency bugs is to analyze the program statically. The advantage of static-analysis is the ability to bypass the nondeterminism. The static analysis targets some programming patterns that generally indicate potential concurrency bugs. For example, accesses to shared variables, such as $a.val$ in the sample program, should normally be protected by locks to ensure proper mutual exclusion. The static analysis can report that all accesses to $a.val$ in the sample program have no such protection and potentially produce errors. The disadvantage of static analysis is that it cannot

Thread 1

```
1  a.val = 0;
2  c.val = 3;
3  a = b;
4  a.val = 0;
```

Thread 2

```
1  a.val = 1;
2  s = a.val;
3  d = 3 / s;
```

Figure 1.1: Simple Java Example

definitely indicate which situations actually lead to an error. Moreover, due to the coarse-grained nature of static analysis, it produces false positives. For the sample program, a static analysis using flow insensitive alias-analysis [68] reports the concurrent accesses between the write of $a.val$ at line 4 of $Thread1$ and all accesses to $a.val$ in $Thread2$ should be protected by mutual exclusion because they are believed to access the same memory address. However, the $a.val$ accessed at line 4 of $Thread1$ is actually a different variable from the $a.val$ accessed in $Thread2$ because of the assignment to a of object b at line 3 of $Thread1$.

To eliminate false positives from the analysis report, the program needs more detailed analysis [67] or to be run concretely. This requirement is the major motivation behind using dynamic testing over static analysis. A naive way of testing the program is to run it just once. However, because concurrent programs execute nondeterministically at runtime, it is possible that the execution follows an interleaving such as,

$Thread1.1 \rightarrow Thread2.1 \rightarrow Thread2.2 \rightarrow Thread2.3$
in which the read of $a.val$ in $Thread2$ reads in the value given at line 1 of $Thread2$. In this interleaving, the program does not produce an error, and the hidden bug is missed.

Another approach is to run the program multiple times to expose hidden bugs. The assumption is that given a sufficient number of runs, an interleaving that triggers a concurrency error eventually occurs. However, there is no guarantee a concurrent program follows all of the different interleavings. Therefore, it is possible that all the test runs follow only error-free interleavings and miss the error. A more sophisticated approach tries to manipulate the program execution in a way that an error-triggering interleaving has a higher chance of being executed. Nevertheless, these approaches still offer no guarantee that a bug-inducing interleaving is eventually tested.

A way to ensure the testing process does not miss any concurrency bug is to systematically derive and test all interleavings of a program. Given N concurrent threads, and B_i atomic blocks in each thread i, the total number of possible thread interleavings is:

$$possible\ interleavings = \frac{(\sum_{j=0}^{N} B_j)!}{\prod_{i=0}^{N} B_i!} \tag{1.1}$$

For the concurrent program in Figure 1.1, assuming every statement is an atomic operation, 35 distinct interleavings are possible. It is important to note that the number of possible interleavings grows factorially with respect to the number of atomic operations in each thread. Therefore, for any practical program, the total number of possible interleavings is beyond computation, making it impractical to derive and test all interleavings.

However, a closer examination of the problem shows that not all interleavings are relevant for testing. Two interleavings that only differ in the execution order of the write of $c.val$ at line 2 in $Thread1$ and the read of $a.val$ at line 2 in $Thread2$ are irrelevant to the testing, because permutations on instructions accessing distinct variables produce the same runtime state. In other words, for those two interleavings, after $c.val = 3$ of $Thread1$ and $s = a.val$ of $Thread2$ are executed, the program always has the same value for the shared variable $a.val$. If a permutation triggers an interleaving that does not lead to the testing of a new concurrent behaviour, both the permutation and triggered interleaving are superfluous. However, two interleavings that differ in the order of the write of $a.val$ at line 1 of $Thread1$ and write of $a.val$ at line 1 of $Thread2$ produce different program states, because the order of conflicting accesses to the same variable are permuted. Two instructions are in conflict if they access the same shared variable and at least one is a write. Therefore, during systematic testing, it is essential to differentiate interleavings that are relevant from those are not. Leaving out irrelevant interleavings significantly reduces the computational cost of testing a concurrent program. In Figure 1.1, there are 4 instructions in $Thread1$ and 3 in $Thread2$, leading to 35 different interleavings. If accesses to disjoint variables are filtered out, only the permutations among instructions from line 1 of $Thread1$, line 1 and 2 of $Thread2$ need to be considered. Hence, the number of interleavings that need to be tested is reduced from 35 to 3.

Although static analysis generates an imprecise bug report that contains false positives, the potential bugs presented in the report are good indicators of possible errors in a program. For example, the report of the lack of mutual-exclusion protection for accesses on $a.val$ by static analysis serves as a hint to the dynamic testing-tool that permutations on accesses to $a.val$ should be considered relevant. On the other hand, dynamic execution of the program should be able to determine that the access to $a.val$ at line 4 of $Thread1$ accesses a different variable from the accesses to $a.val$ in $Thread2$. Therefore, permutations between the read of $a.val$ from line 4 of $Thread1$ and accesses to $a.val$ from $Thread2$ become irrelevant. Moreover, if static analysis shows that a program contains additional accesses to other shared variables that are believed to be consistently protected by a lock, it makes sense for a dynamic testing-tool to test permutations on unprotected accesses to $a.val$ first because they are more likely to produce errors than permutations on the lock entries. Thus, static analysis can provide hints on how to prioritize the search order for a dynamic testing-tool.

Based on these observations, this thesis address the problem of combining static and

dynamic-testing techniques to improve the overall process of testing concurrent programs. Static analysis can inform the dynamic testing-tool on instructions that should or should not be considered for permutations. Static analysis can also help the dynamic testing-tool to efficiently test these permutations at runtime, taking advantage of what has already been tested. For example, coarse-grained alias-analysis can lead to permutations based on incorrect assumptions of alias relationships. However, if static analysis can also instruct the dynamic testing-tool on how to validate those assumptions, many superfluous interleavings can be pruned out.

1.2 Contributions and Scope

This thesis addresses the issues discussed in the previous section. A new collaboration scheme is presented between static analysis and dynamic testing. This collaboration scheme tries to leverage the strengths of static and dynamic techniques to mitigate their respective weaknesses. Overall, the new technique aims to guide the testing process to reach the first concurrency-related error in a program faster using less memory than the current state-of-the-art dynamic model checking tool, such as Java PathFinder (JPF).

This thesis makes the following contributions:

- It introduces a static-analysis module that assists the dynamic model checker. This module extracts May Happen in Parallel (MHP) instructions that are relevant to permutations performed in later dynamic testing. The collected information includes MHP accesses to *may-alias* variables and other error-prone cases such as *barging* and *wait-notify* pairs. This module then applies a sequence of static analyses to refine the set of MHP instructions whose permutations are relevant for uncovering concurrency bugs. Some of these static analyses have been customized and extended to improve their applicability. Besides informing the dynamic testing-tool about relevant instructions for testing, this static-analysis module also provides guidance on how to validate the relevance of a permutation by incrementally computing a set of interleavings that can be used to fulfill such a permutation.

- It introduces a novel accessibility-based partial-order reduction-technique. In this technique, a dynamic testing-tool carries out permutations according to the MHP alias-pairs computed from the static analysis. Limiting permutations to these MHP instructions ensures that only instructions believed to access shared variables are permuted. Value-schedule-based testing aborts the testing of superfluous interleavings introduced because of imprecise static analysis, using the guidance provided by static analysis.

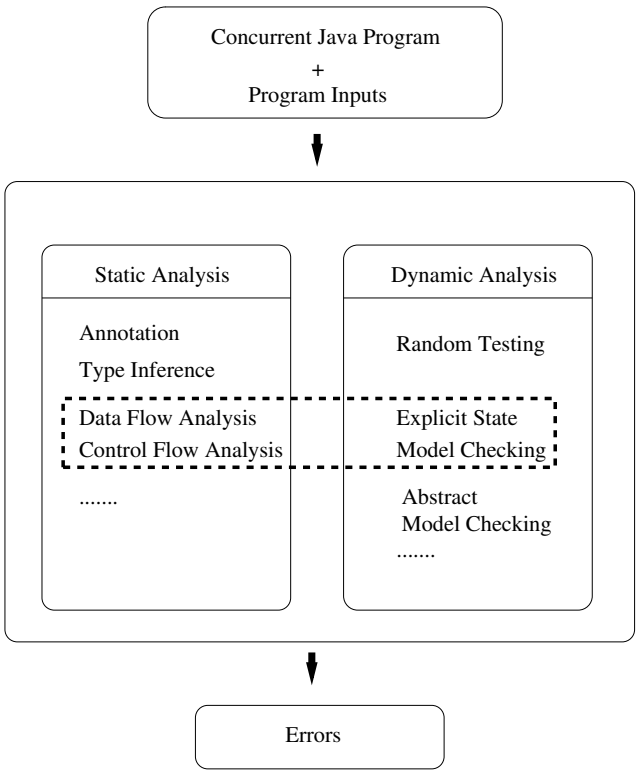


Figure 1.2: Scope of Work

- It introduces a novel search heuristic called *prioritized permutation*, to guide the dynamic testing-tool to reach an error quickly. This heuristic guides testing to check permutations that are most likely to trigger an error first. This technique is implemented using the static analysis and value-schedule-based testing technique. Different MHP pairs are prioritized according to the estimated likelihood they can lead to an error. During testing, permutations are performed first on MHP pairs that have a higher-priority value. Again, the relevance of such permutations is checked using the value-schedule-based technique.

- A collaboration between static analysis and dynamic testing is implemented. Precise program facts collected during dynamic execution are used to refine the initial static-analysis results. Then, these refined interleaving-specific static-analysis results are used by the dynamic testing-tool in further testing. The relevant information exchanged between the static analysis and the dynamic testing-tool includes runtime alias information and the precise type of the caller at a polymorphic-method call-site.

The scope of this thesis is illustrated in Figure 1.2 by the dashed box. **This work centers on combining static analysis with dynamic-based explicit-state model checkers to improve the efficiency of testing a concurrent program to uncover**

the first concurrency-related bug in the program. It is important to note that the proposed testing technique depends on fixed inputs, so any results only apply for this single case. This limitation is an inherited drawback of dynamic-based testing-tools. There has been work, such as [74], that focuses on automatically deriving different test inputs to increase the coverage of the dynamic testing. In this thesis, the generation of test inputs is assumed to have been computed by other tools and/or given by users. Moreover, for the program under testing, it is assumed that static analysis can always determine the exact number of threads created at runtime.

The proposed techniques in the thesis are implemented and evaluated for concurrent Java programs written without using the *java.util.concurrent* package introduced in Java 1.5. The static analyses used in testing employs various data and control-flow analyses, such as alias analysis and MHP analysis. An explicit-state model-checker for concurrent Java programs provides basic mechanisms for deriving and testing different interleavings. Moreover, the explicit-state model-checker dictates the memory model that is used for executing interleavings.

1.3 Thesis Organization

This thesis is organized as follows. Chapter 2 presents background knowledge for this research. First, a background description of different concurrency bugs is presented. Second, different techniques for testing concurrent programs are discussed. Chapter 3 explains the components used to implement the work presented in this thesis, and the experimental setup used to evaluate the different techniques. Chapter 4 introduces the static analysis that forms the starting point of the proposed testing technique. Chapter 5 presents the core of the new testing technique: *value-schedule-based testing*. The discussion centers on how initial static-analysis results can be used to improve model checking. Chapter 6 presents a practical implementation of the value-schedule-based testing that uses precise runtime information to further improve testing. Finally, conclusions and future work are given in Chapter 7.

Chapter 2

Background and Related Work

This chapter has two parts. First, different types of concurrency bugs are reviewed. Second, different existing techniques for uncovering concurrency bugs are discussed. The main rationale behind the different techniques is presented. Moreover, emphasis is placed on the strengths and weaknesses of each approach.

2.1 Concurrency Errors

This section presents concepts essential to concurrent-program testing. Definitions of different concurrency errors are given and illustrated through examples. Finally, some algorithmic limitations for debugging and verifying concurrent programs are discussed. In the following discussion, the notation $tx.y$ denotes statement y in thread x.

2.1.1 Deadlock

A deadlock occurs when one or more processes are waiting for an event that will not occur [11]. Deadlocks result from the incorrect use of a mutual exclusion and synchronization mechanisms such as locks and wait/post. An example of mutual-exclusion deadlock is given in Figure 2.1(a). In this example, if the locks are acquired in the interleaving sequence, $t1.1 \rightarrow t2.1 \rightarrow t2.2 \rightarrow t1.2$, neither $Thread1$ nor $Thread2$ can proceed because the second lock sought by each thread is held by the other thread. An example of synchronization deadlock using *wait/post* is shown in Figure 2.1(b). In this example, $Thread1$ cannot run to completion due to a logical error that missed the post, i.e., a is always negative.

Thread 1

```
1  lock(a);
2      lock(b);
3      //......
4      unlock(b);
5  lock(a);
```

Thread 2

```
1  lock(b);
2      lock(a);
3      //......
4      unlock(a);
5  unlock(b);
```

(a) Mutual-Exclusion Deadlock

Thread 1

```
1  a.wait();
```

Thread 2

```
1  if (a.val > 0)
2      a.notify();
```

(b) Synchronization Deadlock

Figure 2.1: Examples of Deadlock

2.1.2 Race Condition

A race condition occurs when there is missing mutual exclusion (data race) or synchronization (general race), i.e., two or more tasks race along assuming that mutual exclusion or synchronization has occurred [53, 33].

A data race happens when shared variables have unprotected accesses allowing a value to be read/written before being completely modified. In Figure 2.2(a), all accesses to variable $a.val$ by $Thread1$ are protected by lock a, but a different lock, b, is used to protect variable $a.val$ by $Thread2$. The inconsistency in lock protection leads to a race condition on the accesses to the variable $a.val$; e.g., $Thread2$ can read an inappropriate intermediate value of $a.val$ being calculated by $Thread1$.

A general race happens when there is insufficient synchronization to ensure the deterministic execution order of statements predefined by the program specification. In Figure 2.2(b), if the program specification requires $t1.1$ to happen before $t2.1$, then $Thread1$ and $Thread2$ contain a general race on accesses to variable $a.val$ because there is no synchronization to enforce this ordering. Moreover, adding mutual exclusion locks around $t1.1$ and $t2.1$ in Figure 2.2(b) does not enforce the required execution order of these two assignments; wait/post or a similar synchronization is needed. The main difference between a data race and a general race is that data races capture violations of an atomic region while general races capture violations of a particular execution order among statements.

Thread 1

```
1  lock(a);
2      a.val = a.val * 2;
3      if (a.val < 0){
4          a.val = 0;
5      }
6      a.val = 3;
7  unlock(a);
```

Thread 2

```
1  lock(b);
2      b.val = a.val;
3  unlock(b);
```

(a) Data Race

Thread 1

```
1      a.val = 0;
```

Thread 2

```
1      a.val = 1;
```

(b) General Race

Figure 2.2: Examples of Data Race

From a program's execution-trace, it is possible to discover a data race by locating a pair of concurrent accesses to a shared variable within multiple mutual-exclusion regions. A mutual-exclusion region can be derived from the program by identifying locking mechanisms. Then, a shared variable accessed by multiple threads holding an inconsistent set of locks at runtime indicates a potential data race.

A general race is harder to detect than a data race. It is impossible to precisely claim that a pair of concurrent accesses constitutes a general race without knowing the program specification on the execution order of those two accesses. It is unrealistic to assume such specifications are always available. Therefore, every pair of accesses to a shared variable from two threads may indicate a potential general race, even with proper mutual exclusion. One possible solution is to derive all possible interleavings of a program to identify the value assignments responsible for an error. However, such exhaustive testing is usually infeasible.

A manifestation of a race condition is called an *atomicity violation* [50]. An atomicity violation happens when a concurrent execution produces a sequence of read/write accesses to shared variables that could never happen in any sequential execution of a concurrent program, where sequential execution means no pre-emption or time-slicing, i.e., each thread is started and runs until it blocks or terminates, but threads may start/restart in any order. Both data and general races can lead to atomicity violations. For example, in Figure 2.2(a), the read of $a.val$ in $Thread2$ can potentially read in values given by any of

Thread 1

```
1  //a.val initialized to 0
2  lock(a);
3      a.val = 1;
4  unlock(a);
5  lock(a);
6      a.val = a.val + 1;
7  unlock(a);
```

Thread 2

```
1  lock(a);
2      r = a.val;
3  unlock(a);
```

Figure 2.3: Examples of Atomicity Violation

the three writes to *a.val* in *Thread*1 depending on the runtime interleaving. However, if the program is executed sequentially and *Thread*1 starts first, it is impossible for *Thread*2 to read in the *a.val* assigned by the first or second write to *a.val* in *Thread*1.

An atomicity violation can also happen when accesses to shared variables are consistently protected by the same lock. An example of such an atomicity violation is shown in Figure 2.3. In this example, all accesses to shared variable *a.val* are consistently protected by lock *a*. However, the two writes to *a.val* in *Thread*1 are not carried out as a single atomic operation. It is possible for *Thread*2 to execute in between the two mutual exclusion blocks of *Thread*1, and read the intermediate value, 1, for *a.val*. Again, if the program is executed sequentially and *Thread*1 starts first, *Thread*2 reads in the value written by the second write to *a.val* in *Thread*1, but never the first write to *a.val*. The atomicity violation in Figure 2.3 can be considered as a general race among lock acquisition operations in *Thread*1 and *Thread*2.

A testing tool that is capable of detecting data races and general races should be able to uncover atomicity violations as well.

2.1.3 Non-deterministic Execution

For a sequential program, given a fixed program input, the program always follows the same execution path. However, because of the non-deterministic nature of concurrent execution, concurrent programs might take different interleavings in different executions even with the same input values. Therefore, an error caused by a concurrency bug might not show up consistently across different runs. Traditional testing techniques, such as running the program multiple times with various sets of input, are insufficient. Hence, the ability to deal with the non-deterministic nature of concurrent execution becomes an essential feature of any testing tool for concurrent programs.

2.2 Concurrent Debugging Techniques

Because of the complexity of uncovering bugs, attempts have been made to restrict existing program languages to remove certain kinds of concurrency bugs, such as data races, from programs [4, 75]. However, in this thesis, the work focuses on testing tools that find concurrency bugs in programs written in existing program languages. The effectiveness of a testing tool is measured by two metrics: soundness and precision. A testing tool is sound if it guarantees all errors are found (no false negatives), otherwise it is unsound. Precision measures the true bugs in the final bug-report. A testing tool has higher precision if the percentage of false alarms is low (few false positives) in the report.

In general, existing techniques for uncovering concurrency bugs can be grouped into two categories: static and dynamic based techniques. The static-based techniques rely on various static analyses to identify potential concurrency bugs in a program. The dynamic-based techniques require program execution to expose concurrency bugs. The main advantage of static analysis is that it does not have to deal with the non-determinism issue. Therefore, it is capable of locating subtle bugs that may only occur in some obscure interleavings. However, it delivers a bug report with a high false-positive rate due to its coarse-grained nature. On the other hand, dynamic techniques achieve high precision in reporting concurrency bugs because dynamic techniques only report bugs that actually happen. However, dynamic techniques may not trigger all behaviours of a concurrent program, and thus tend to produce unsound reports. The strengths and limitations of static and dynamic techniques have led researchers to propose testing tools that make use of both approaches.

In this section, different existing testing-techniques are reviewed. Comparisons are drawn when it is appropriate.

2.2.1 Static Analysis

In this section, static analysis methods for uncovering concurrency errors are presented. Then, some existing static-analysis-based testing-techniques are discussed to illustrate how different static analyses can efficiently collaborate with each other to uncover concurrency bugs.

Alias analysis, also referred to as points-to analysis, is a data-flow analysis for determining storage locations that may be accessed in two or more ways [48]. Concurrent accesses to a shared-memory location are the source of many concurrency errors. Therefore, it is essential to know which memory addresses are shared among threads. However, the difficulty is that the same memory location may have different symbolic names in different methods and threads. In Java, aliasing of the same memory location may be the result of parameter passing for methods.

Two commonly-used alias-analysis methods are the Steensgaard equivalent-based analysis [68] and the Andersen subset-based analysis [2]. In the equivalent-based approach, two pointer-type variables are considered to be equivalent if they are associated by an assignment edge. The equivalent-based approach makes no distinction on the direction of alias assignments. In contrast, the subset-based approach makes this distinction. The alias information is propagated and merged from the right side of an assignment to the left. Therefore, the alias relationships of the operand on the left of the assignment operator merges with the alias relationships of the operand on the right of the assignment operator. Moreover, this subset relationship persists in all future modifications.

Alias analysis for concurrent programs with simple structures, such as parbegin/-parend, was attempted in [62, 59]. The alias analysis for concurrent programs proposed by these two papers is very similar to its sequential counterpart with an extension that supports the propagation of alias information across fork/join edges. To cope with the large number of alias relationships generated from the program analysis, BDD-style compression can be used to reduce the space overhead [5].

Escape analysis is an extension of alias analysis. Escape analysis tries to locate variables that have become accessible outside of a method or a thread object. Escape analysis is relevant for two reasons. First, a variable that does not escape a method is a local variable; a local variable is never shared among threads. Therefore, accesses to a local variable are safe from race conditions. Second, if a variable is accessed by only one thread, it cannot be involved in any type of race condition. Escape analysis helps to identify the set of variables that should be checked for possible race conditions.

There are some differences between escape and alias analysis. Alias analysis resolves different pointers to the same node on the points-to graph if those pointers point to the same memory location. On the other hand, escape analysis is only concerned with whether a variable is referenced outside of the method or thread being analyzed (a yes/no question). Equivalent-style escape analysis has been presented in [9, 61]. More precise flow-sensitive escape-analyses are given in [76, 14].

The essential problem in testing concurrent programs is to determine which statements from different threads may happen in parallel. Only the MHP statements can execute in nondeterministic order and trigger concurrency-related errors. The task of identifying MHP statements requires static analysis to analyze the control flow of the program while taking into consideration the synchronization and mutual exclusion mechanisms. Alias analysis also plays a role in this process by matching up mutual exclusion statements that operate on the same object. For example, only mutual exclusion regions protected by different locks are considered to be MHP regions.

The Control Flow Graph (CFG) for sequential programs is extended to facilitate MHP analysis. A simple Parallel Execution Graph (PEG) [64] matches up the synchronization

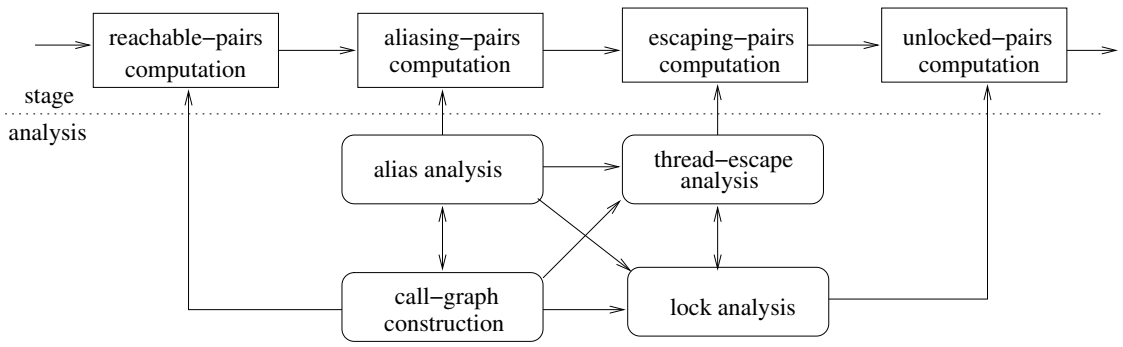


Figure 2.4: Testing based on Combined Static Analysis Methods [50]

statements to mark out the MHP regions. A more advanced Concurrent Control Flow Graph (CCFG) was proposed to capture the interleavings among potential concurrent accesses to shared variables by adding interference edges among accesses [41, 54].

Different types of static analyses identify a particular set of program statements that may be relevant to testing concurrency. It makes sense to combine different types of static analyses to improve the overall performance of bug detection.

Chord is a tool that detects data races using multiple phases of static analysis [50]. Each phase of analysis refines a given set of potential race-pairs to achieve better precision. Figure 2.4 shows the architecture of *Chord*, which consists of four stages.

In the first stage, every two accesses to variables from different threads are placed into a pair. Each of these pairs represents a possible data race. These accesses include accesses to fields, static variables and array elements. The result of this pairing is a set of access pairs. Each pair is made up of two accesses to a common field of the same class. These pairs are conservatively considered by *Chord* as original race-pairs. In the second stage, the original race-pairs are filtered using a simple alias analysis. Two accesses in the original race-pair set can form a real data-race only if both events access the same variable. Also in the second stage, a *k-object sensitive* alias-analysis [47] is performed to further filter out race pairs that do not access the same object. In the third stage, escape analysis is performed to filter out race pairs that contain thread-local variables. In the fourth stage, the remaining race pairs are filtered using a lock-set based algorithm [50]. Race pairs that involve accesses to variables consistently protected by the same lock are not a possible data-race set.

An extension to *Chord* is introduced in [49]. Another static analysis phase is added to categorize alias references in the program into disjoint *must-not-alias* sets. For example, if every element of an array of *Object* type is initialized to new objects in a loop, then concurrent accesses to the fields of different elements of that array do not lead to data races because the owner objects of those fields have been proven to *must-not-alias*.

There are also static analyses for identifying potential deadlocks in concurrent pro-

grams. Jlint [3] performs control-flow analysis over a concurrent program to produce a *lock-order graph*, in which locks are represented as nodes, and acquisitions of locks are represented by edges labelled with the names of the threads that attempt the acquisitions. Then, the graph is analyzed to uncover circular acquisitions in the lock-order graph. [77] extends the deadlock analysis to accommodate deadlocks that may exist in library code by merging the *lock-order graphs* of public methods in classes. Alias analysis is applied in both [3] and [77] to prune out cycles formed by acquisitions of non-aliased locks. Nevertheless, both techniques still report many false positives because of the imprecise CFG and alias analysis. Moreover, to reduce the complexity of analysis, acquisition cycles that only contain less than a certain number of nodes are identified. Therefore, both approaches are unsound.

The precision of the static-based techniques depends on the precision of different static analyses making up the tool. For example, the precision of alias analysis affects the precision of escape and MHP analysis. Moreover, the precision of escape and MHP analysis affects precision in the testing tool, such as *Chord*. However, many precise static analyses are computational intractable. As shown in [40], precise path-sensitive and alias analysis is undecidable. In addition, any path and synchronization-sensitive analyses, either control-flow analysis such as MHP analysis or data-flow analysis such as alias analysis, is undecidable [58]. As a result, static analyses have to introduce many approximations. A conservative approximation introduces false positives, while an unsafe approximation, such as those in deadlock analysis [77], introduces false negatives. The approximations in individual analysis propagate and manifest as they are combined, and affect the precision and soundness of testing tools.

To deal with the imprecision of static analysis, techniques have been proposed where a programmer annotates a program with relevant program facts, such as the set of locks that should be held at given program points [22, 26, 27, 28]. However, the need for manual insertions of annotations can easily introduce human errors into the analysis. Other static analyses include KISS [57], which translates a concurrent program into different sequential programs. Each sequential program represents a particular runtime interleaving. Then, the sequential programs are checked with a theorem prover for correctness. To make the testing tractable, this technique only computes sequential programs for interleavings that have less than two context switches.

2.2.2 Unsound Dynamic Testing

Dynamic testing-techniques uncover concurrency bugs by running programs. The chance that a concurrency bug is uncovered during testing depends on whether an interleaving with a runtime error is executed. Therefore, the soundness of a dynamic technique is largely determined by its ability to trigger different thread interleavings of a program. A

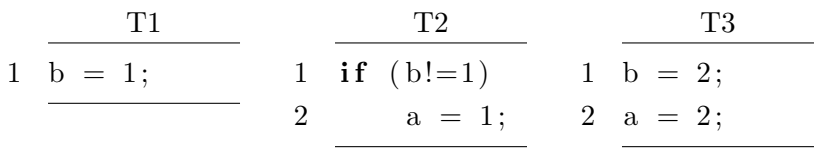


Figure 2.5: Unsound Dynamic Testing

dynamic-testing technique is sound only if interleavings that trigger all possible concurrent behaviours are tested.

The task of triggering interleavings that cover all possible concurrent behaviours of a program is non-trivial. When all possible interleavings are not tested, techniques have been proposed to improve the coverage of all tested interleavings. The aim is to increase the chance that an error-triggering interleaving is tested. Techniques have also been proposed to identify concurrency bugs from a limited number of interleavings even if those interleavings do not directly lead to runtime errors. The ability to identify concurrency bugs from an error-free interleaving requires dynamic techniques to recognize patterns that generally lead to bugs. After potential interleavings are derived, some additional techniques, such as those in [51], can be used to determine whether the interleavings could manifest into runtime errors. Nevertheless, these techniques are still unsound because they cannot derive concurrency bugs that are only observable from interleavings. For example, in Figure 2.5, three threads $T1$, $T2$ and $T3$, access two shared variable a and b. A sequential run of the program in the order $T1 \rightarrow T2 \rightarrow T3$ exposes a data race on b because b is accessed multiple times in different threads. However, examining the trace of this interleaving cannot expose the potential data race on a because the branch statement in $T2$ evaluates to false and skips the access to a. Thus, a is considered to have no data race because it is accessed only once in the execution. On the other hand, a sequential execution such as $T3 \rightarrow T2 \rightarrow T1$ enables the assignment to a in $T2$, and exposes the data race on a.

In the following sections, unsound dynamic testing-techniques are discussed.

Capturing Execution Patterns

Eraser [65] is an on-the-fly dynamic testing-tool that detects data races that violate mutual exclusion at runtime by capturing execution patterns that may lead to data races. Eraser is based on a dynamic lockset-algorithm that requires every shared variable to be consistently protected by the same set of locks at runtime to be free of data races. Therefore, at runtime, Eraser is responsible for two tasks: identifying shared variables and tracking the lockset of each shared variable. Eraser classifies a variable as a shared variable if the following two conditions are met:

- the variable has been accessed by at least two threads;
- there is at least one write access from a thread that is not the thread that first accessed the variable.

After a variable is determined to be shared, Eraser tracks the locks that are held by the accessing threads when that variable is accessed. A data-race error occurs when an access is inconsistently protected by the same set of locks as previous accesses. In other words, a data race occurs when a join operation on the current lockset and the locksets from all previous accesses produces an empty set. It is important to note that Eraser does not require error-triggering interleavings to uncover potential errors. Instead, Eraser uncovers errors by monitoring error patterns, such as the inconsistent lockset, which may lead to errors in some interleavings.

Eraser suffers from false positives. For example, an error is triggered for an unprotected access to a shared variable when only one thread is active. This false positive is caused by imprecision in the bug pattern. Different approaches, such as that in [78, 56], have been suggested to recognize the execution context, such as the number of active threads, to remedy the false-positive issues. However, this approach may lead to false negatives, if an interleaving is not tested in which more than one thread is active during the time that access occurs. Eraser is sensitive to conditional interleavings. For example, an execution branch may be skipped due to a particular commit order of two concurrent writes. Any potential data races in that branch are omitted as well, yielding a possible false negative result. Despite these shortcomings, Eraser remains an easy to understand but powerful tool for data-race detection. In [8], the AspectJ language is extended with new pointcuts to capture lockset refinement at runtime. Combined with Lipton's reduction theorem [45], Eraser can be used to detect more complicated atomicity violations on the fly [24].

Improving Interleaving Coverage

ConTest was developed for testing concurrent Java programs [21]. ConTest inserts *sleep* and *yield* statements at the bytecode level of the original program. By manipulating the thread scheduling of the program being tested, ConTest increases the probability of exposing potential errors that could only occur in some subtle interleaving. In its implementation, ConTest instruments only those bytecodes that may produce or affect concurrent events. Some examples of such bytecodes are: *getfield/setfield* to shared variables, invocation of thread ordering-functions (such as $join()$, $start()$, $wait()$ and $notify()$), and bytecodes that manage monitor entry and exit. The delay instructions can also be inserted at the source-code level using aspect-oriented programming [16]. At runtime, every inserted instruction is invoked according to a predefined probability, and

the lengths of wait times caused by these instructions are randomly decided. This mechanism is useful for detecting concurrent errors such as deadlocks. Despite the insertion of delay instructions, there is no guarantee of triggering an interleaving. Therefore, in order to obtain satisfactory coverage of program behaviours, ConTest would have to run the program with all possible interleavings.

The value substitution proposed in [6] is an approach that randomly changes the matching of concurrent read/write accesses of a shared variable to explore subtle interleavings. During execution, every write event is saved along with the value written. The value written by a write access w may potentially reach a read access r if it happens before r. Intuitively, runtime value substitution is the process of selecting a value from the many values written in the past and assigning one to a read access. At runtime, when a read is encountered, value substitution randomly assigns a write value that may reach the read. Therefore, runtime value substitution is also referred to as a way of "choosing among alternative pasts".

Runtime value-substitution provides a mechanism in which each possible interleaving has an equal opportunity to be tested. However, the current implementation of runtime value substitution in [6] provides no guarantee of finding all feasible interleavings. First, runtime value substitution provides no mechanism to explore every possible write partner for each read. Second, a write event that happens before a read event during execution does not guarantee the existence of a feasible path where these two accesses match up. Therefore, a runtime selection of a read/write pair without detailed knowledge of control dependencies may produce a circular dependency error during variable assignments [6]. Third, runtime substitution can only match up a read access with writes that actually happen before it at runtime. Therefore, runtime substitution cannot pair up a read access to a write access that may happen in parallel with that read but actually happened after it in the runtime due to the runtime scheduling.

Thread Interleavings and Value Schedules

Although neither the runtime delay insertion nor the value-substitution approach can deliver complete coverage of all possible execution interleavings, the differences between these two techniques underline important views of thread interleaving from the perspective of a testing tool. Delay insertion manipulates the execution flow (context switches) of a program to explore runtime interleavings. Each distinct interleaving is viewed as a schedule for executing instructions in threads. In contrast, the value substitution directly manipulates concurrent variable assignments to simulate the effect of an interleaving. The runtime value substitution technique treats each interleaving as a schedule that fulfills a sequence of variable assignments.

It is straightforward to observe that many thread interleavings generated by the delay

insertion technique contain the same set of reads and writes such that a particular read access always matches up with the same write event. These interleavings are considered to be equivalent because they always bring the program into the same state. Such a set of equivalent thread interleavings is referred to as an equivalence class with respect to a distinct value-schedule. A value schedule is formally defined as "a set (equivalence class) of all thread interleavings such that 1) all schedules in the class agree on the set of critical events, and 2) the value consumed by any critical read event is generated by the same critical write event for all the schedules in the class." [6]

2.2.3 Sound Dynamic-Testing

In this section, theoretically sound dynamic-testing techniques are introduced. These techniques systematically trigger thread interleavings that cover all possible concurrent behaviours or value schedules of a program. However, they are not "practically sound" because the exhaustive testing of all concurrent interleavings is impractical (see Equation 1.1 in Chapter 1).

As discussed, many distinct interleavings may correspond to a single value-schedule. Therefore, the testing tool still preserves soundness by only testing one interleaving for each value schedule. Various dynamic testing-tools make different attempts to reduce the number of redundant interleavings tested for each value schedule. Even with these reductions, the attempt to cover all value schedules remains an impractical task for most programs. To increase the chance that a concurrency bug is found before computational resources run out, dynamic tools make use of various program facts to guide the testing tool to trigger value schedules that are more likely to induce errors first.

Richtest introduces an algorithm called reachability testing [12, 42, 36, 43] that can derive a complete set of relevant and feasible execution sequences starting from any arbitrary execution trace. Richtest consists of the following phases: collecting an execution trace in which each event is time-stamped with a Lamport clock; collecting potential write partners for the read events from the trace; deriving a set of variance sequences for each trace; and a prefixed-point replay on every newly derived variance sequence before letting the program run nondeterministically to its completion. Intuitively, Richtest discovers a new value schedule by changing the write partner of a read access in an existing execution trace, and replaying the execution trace until the point where the change occurs. A pair of read/write accesses may potentially match up if the Lamport timestamps indicate a partial (may happen in parallel) ordering between two accesses, and they operate on the same memory location. After the prefix is replayed and changes are applied, the program is run nondeterministically beyond the point of modification. Note that each run tests a distinct value schedule that differs from the original trace by a least one read/write pair.

Richtest has three main problems that prevent it from scaling up to large programs.

First, instrumentation, post-execution analysis, and derivation for every value schedule requires a prohibitive amount of computational resources for any practical program. Second, Richtest offers no mechanism to prioritize the testing of interleavings. Every interleaving is tested in the same order as it is discovered. In the worst case, an error that dominates a large number of accesses to shared variables could be left undiscovered for a long time while Richtest spends resources testing other error-free value-schedules. Third, two interleavings are run independently even if the most of orderings fulfilled by them are the same.

Model checking is another way to test for soundness in concurrent programs. The interest in model checking comes from the success of model checking in the field of hardware design. The main idea behind model-checking software is to construct a model of the software and perform exhaustive checking on this model to ensure the absence of error states. The model of a program is usually presented using state automaton. The possible program states are represented as nodes in the automata, while the possible program-state changes are represented as transition edges between nodes. Different model checkers represent the program states and transitions in different detail. For example, in the traditional approach, only the specification of the program is abstracted. In contrast, some new model checkers allow modelling of a program at the program-statement level. Differences between those approaches are presented along with their application in testing concurrent programs.

Abstract Model-Checkers

Traditional model checking requires the creation of an abstract model of a program. The abstract model consists of a set of high-level program states and transitions among these states. For example, if a programmer wants to verify that a variable may have a value greater than 0, the states of the variable can be modelled as positive, negative, and zero instead of all integer numbers. In this example, three high-level states represent all possible states of that variable of interest. Moreover, programmers can create a model only for a particular set of variables and program behaviours of interest. It is the responsibility of the programmer to ensure the abstract model is complete and correct for the behaviours being checked. For example, all possible transitions of a variable under investigation must be captured in the model and all possible states must be identified. In most cases, a model of a program has to be specified in some kind of modelling language, such as Promela in SPIN [35], SMV in NuSMV [15], or BIR in Bogus [60]. The state transitions specified in these languages translate to an equivalent Buchi finite-state automata. All possible transitions are exhaustively traversed to find possible violations of a set of specified properties. These properties are generally specified in Linear or Computational Temporal Logic [46]. Programmers can specify a statement such as "lock z should be acquired before a write to x" in temporal logic. The temporal logic property is also

translated into an automaton. Transitions in the program model trigger transitions in the temporal-logic model.

There are a few significant disadvantages of abstract-model checking. First, it is hard to extract a correct model out of a real-world-sized program. Furthermore, it is hard to ensure the extracted model correctly represents the original source code. Second, after the model is extracted, it is still difficult to translate these specifications into the required modelling language correctly. It takes a large amount of training for programmers to extract a model and apply the modelling language efficiently. Sophisticated model checkers, such as Bandera [17] and Zing [70], provide a way to automatically extract a model out of a program through slicing [37] and translate it into the appropriate target modelling language. However, Bandera is limited by the types of abstractions it can perform automatically. Currently, Bandera can only recognize a few simple abstractions such as whether a variable is positive. Third, there are some runtime operations, such as dynamic heap allocation, that cannot be specified in existing modelling languages, which further limits the applicability of abstract-model checking. Fourth, it is also hard to keep the model up-to-date as the program evolves.

Explicit-State Model-Checkers

Due to the complexity of correctly extracting a program model and specifying it in a proper model-checking language, model checkers such as JPF (Java PathFinder) [72] and Verisoft [30] perform model checking directly on the original program. Moreover, because JPF and Verisoft explore the program state by actually running the program, such model checkers are also called dynamic model-checkers.

JPF is a dynamic model-checker for concurrent Java programs. It is implemented as an independent virtual machine, that in turn runs on top of a local Java Virtual Machine. A program is checked by running it in JPF. Only the native method calls are delegated to the underlying Java Virtual Machine. Because program execution is controlled and monitored by JPF's thread scheduler, JPF can systematically explore all possible execution sequences of a program. Consequently, JPF is capable of detecting any kind of error that results from nondeterministic execution.

JPF achieves full interleaving coverage through two main mechanisms: backtracking and state matching. After every state transition, such as executing a bytecode, a snapshot of the program state is taken. A state includes the thread stack information and the data on the heap. Each state also maintains a candidate list that contains the program counter of all ready threads. The instructions indicated by these program counters are generally referred to as the *co-enabled* transitions of a state in model checking. JPF discovers a new program state by arbitrarily selecting one transition from the candidate list and executing it. This depth-first exploration continues until the program completes. Then, JPF starts

a backtracking process on the exploration path to a visited program state in which the candidate list has unexplored co-enabled transitions. JPF selects an available transition from the candidate list and performs another depth-first exploration from that transition. This recursive exploration continues until all possible interleavings are explored. Because JPF saves all encountered program states during exploration, it is referred to as a stateful explicit-state model-checker.

Verisoft offers stateless state-exploration in a fashion similar to JPF. Verisoft is a model checker for C programs with support for concurrent libraries such as *pthread*. Like JPF, Verisoft controls the execution of a C program by providing a custom thread-scheduler. A recursive depth-first search is performed on each program state with multiple co-enabled transitions. Stateless exploration is achieved by saving each executed transition on a stack instead of the program state that results from executing that transition. Then, when backtracking is required, Verisoft undoes instructions on the stack to bring the system back to the desired state. A stateless exploration reduces the amount of memory consumed by the model checker, but undo operations during backtracking require more computation than stateful approaches. Therefore, there is the classical trade-off between stateful and stateless exploration in terms of time and space.

However, both stateful and stateless approaches share a problem, called *state-space explosion*, where the number of states a model checker has to check can exceed the storage and computational capacity of the testing environment. With respect to a concurrent program, the state-space explosion is caused by an excessive number of interleavings in the program. To cope with the excessive consumption of memory and computational resources, solutions have been proposed to distribute the state exploration to multiple computers and process them in parallel [19].

Partial-Order Reduction

As discussed, only one interleaving is needed to test each equivalence class of interleavings for a value schedule. In other words, a testing tool should only explore transitions that could lead to new value schedules at each backtracking point. Sen [66] proposes a new approach to randomize the selection of the next-to-explore transition to improve the chance that a newly derived interleaving will cover a new value schedule. A more systematic way of identifying which transitions are relevant to generate new value schedules is referred as Partial Order Reduction (POR) [31].

POR aims to prune out permutations of transitions that can only lead to redundant tests of the same value schedule. There are two types of POR: *sleep-set* and *persistent-set*. The *sleep-set* ensures two co-enabled transitions are permuted if they access the same shared variables, and at least one of them is a write. The *persistent-set* ensures two

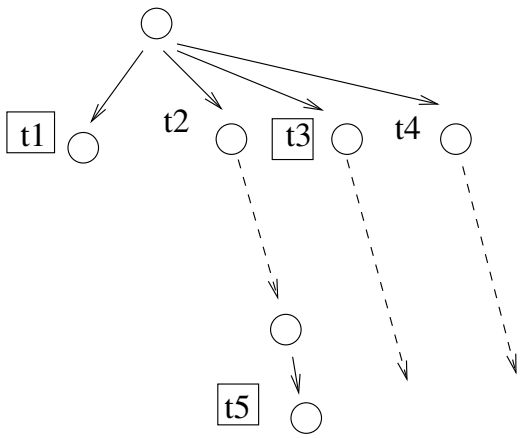


Figure 2.6: Illustration of Partial-Order Reduction

co-enabled transitions are permuted if one of the two transitions may access the same shared variable as a successor of a co-enabled transition.

For example, in Figure 2.6, there are five labelled transitions: $t1$, $t2$, $t3$, $t4$ and $t5$. Assume $t1$, $t2$, $t3$ and $t4$ are co-enabled transitions, and $t1$, $t3$ and $t5$ access the same shared variable enclosed in squares in Figure 2.6, and $t2$ and $t4$ access only local variables. Moreover, $t1$ and $t3$ have an access conflict with $t5$ that succeeds $t2$ in the future. The goal of POR is to determine which permutations among co-enabled transitions are necessary. The general rule is that any two transitions should be permuted if they are conflicting accesses or their permutations may lead to permutations of conflicting transitions. Two transitions conflict if they access the same shared variable, and at least one of them is a write. *Sleep-set* reduction reports the permutation between $t1$ and $t3$ are necessary because they access the same variable and have an access conflict. *Persistent-set* reduction adds $t2$ into the permutation list along with $t1$ and $t3$, even though $t2$ does not directly conflict with either $t1$ or $t3$. The addition of $t2$ is necessary because $t2$ is followed by transition $t5$ that is in conflict with $t1$ and $t3$. By yielding the execution of $t1$ and $t3$ for $t2$, and its succeeding transitions until $t5$ becomes co-enabled with $t1$ and $t3$ enables $t5$ to permute with $t1$ and $t3$ leading to new program states. Moreover, in practice, *sleep-set* reductions are generally used to complement *persistent-set* reduction. For example, after *persistent-set* enables permutations between $t1$ and $t2$, *sleep-set* ensures that $t1$ does not permute with any intermediate transitions between $t2$ and $t5$ because $t1$ operates on a different memory address from all of them. To summarize, two co-enabled transitions from two threads are permuted if either their permutations could produce a new value schedule locally or at some point in the future. Any permutation that does not satisfy *sleep-set* or *persistent-set* is considered as redundant and pruned out.

Sleep-set is easy to implement. It is straightforward to determine whether two co-enabled transitions access the same shared variable by comparing the target memory

addresses. *Persistent-set* is harder to implement because it requires knowledge about future execution. The knowledge of the future can only be approximated using various means. Hence, the effectiveness of *persistent-set* POR is largely determined by the accuracy of these approximations. Some approximation techniques are presented.

JPF simulates *persistent-set* reduction using on-the-fly heap-based reachability-analysis [72]. Two co-enabled transitions are permuted if one of them accesses a variable that is reachable from multiple threads. A variable is reachable from a thread if there is a reference path from the thread object to that variable on heap. It is important to note that the existence of a reference path to a variable does not necessarily mean that variable is actually accessed in the program. The assumption is that if a transition accesses a variable reachable by multiple threads, then there are probably other MHP accesses to the same variable following the co-enabled transitions. Therefore, the permutation of these co-enabled transitions might be allowed by the *persistent-set*. The advantage of this approach is that it is straightforward to implement. The reachability analysis based on the heap can be directly adapted from the garbage collection mechanisms implemented in various Java Virtual-Machines. The disadvantage of this technique is that it is coarse-grained. An object can be reachable from a thread object but never actually accessed by that thread. Thus, a permutation based on the heap reachability alone leads to testing of redundant interleavings if the expected accesses do not occur in the future execution.

Verisoft uses static analysis to assist the *persistent-set* reduction. Static alias-analysis is used to determine whether a future transition in another thread may conflict with a currently enabled transition. The *may-alias* analysis discussed in Section 2.2.1 is used to compute the conflicting access-set for each instruction. At model-checking time, two transitions are added to the *persistent-set* if one of them has a successor that may access the same shared variable as another transition. However, due to the coarse-grained nature of the static analysis, many instructions believed to access the same variable turn out to access non-aliased variables in the concrete execution. Thus, a significant number of superfluous permutations are carried out leading to many redundant interleavings.

Verisoft has another way to implement *persistent-set* reduction that is called Dynamic Partial Order Reduction (DPOR) [25]. Instead of relying on heap-based reachability analysis, runtime information is incrementally collected to determine the existence of an MHP access to a shared variable that could happen in the future. First, a program is allowed to run nondeterministically to completion. The program trace of the execution is saved. Each executed instruction is recorded with a Lamport-style time-stamp [39] to determine the *may-happen-before* relationship among them, and the memory address accessed by that instruction. Then, post-mortem processing determines which instructions may happen in parallel and access the same memory address. The program is instrumented with those instructions marked as backtracking points. After that, the program is rerun until the last backtracking point identified in the previous trace. That instruction is permuted

	Thread 1		Thread 2
1	b.val = 1;	1	if (b.val < 0){
2	s = a.val;	2	a.val = 0;
3	b.val = −1;	3	}

Figure 2.7: Example where DPOR is not Optimal

with all co-enabled transitions from other threads that have been determined to have MHP accesses to the same shared variable later in their execution. Moreover, new MHP accesses discovered in the execution of the permuted interleaving are added to the backtracking list. The advantage of DPOR is that *persistent-set* reduction only performs a permutation if a conflicting access has been shown to exist in the previous executions. If a variable is only reachable but never accessed by a thread, then transitions from that thread are never considered in a *persistent-set* involving accesses to that variable. Thus, DPOR is more precise than the heap reachability and static analysis reduction. In an extension of Java PathFinder(JPF) [10], a coarser but similar technique is implemented. A program statement is considered to be unsafe if it is shown to access a shared variable. A transition representing that program statement is always permuted regardless of the program context. This may produce a large number of superfluous permutations because different instances of a program statement, such as an array element access in a loop, may access different memory addresses.

However, DPOR still leaves room for further improvement. A conflicting access that exists for a transition in one interleaving might not exist in another interleaving. Some intermediate permutations among accesses to other variables might block the expected access. For example, in Figure 2.7, if the initial execution of the program follows the interleaving $t1.1 \rightarrow t1.2 \rightarrow t1.3 \rightarrow t2.1 \rightarrow t2.2$, DPOR identifies $t1.2$ as a conflicting access, with $t2.2$, in $Thread2$. Thus, DPOR permutes $t1.2$ with another co-enabled instruction, such as $t2.1$ from $Thread2$, to try to get to $t2.2$. However, the interleaving spawned from this permutation cannot enable the instruction $t2.2$, because if $t2.1$ happens before $t1.3$, the branch statement skips the write to a. Therefore, testing this interleaving is superfluous.

Heuristic Guided Testing

Besides POR, another set of important techniques to improve the efficiency of model checking for concurrent programs is to use heuristics to test interleavings that are suspected to contain errors earlier in the testing process. Testing suspicious interleavings first improves the chances that an error is exposed before computational resources run out.

The essence of heuristic-guided model-checking for concurrent programs is to have a function that evaluates the likelihood that further explorations along different threads reaches an error state. A transition from a thread that is believed to be more likely to produce an error is considered to be more relevant. From the list of next-to-explore transitions, the model checker visits transitions that have most relevant values first. This technique is generally referred to as *best-first* search [32]. There are variations of *best-first* search, such as *beam search*, that only explore a fixed number, k, of the next-to-explore states from a given state. In general, they all follow the same philosophy of prioritizing the computational resources to test more relevant interleavings first.

It is important to note that the error states sought by an explicit-state model-checker are unknown during testing. Moreover, the state-transition graph is computed incrementally. With the target unknown and transition graph only partially known, a precise evaluation of the relevance of a transition with respect to a potential error becomes difficult. Therefore, heuristic functions need to make various assumptions when evaluating the relevance of a transition. One approach plans future searches based on results from previous ones [32]. Heuristic functions that use information collected from explored states are: *branch*, *most-blocked*, and *interleaving*. The *branch* heuristic gives higher relevance to transitions that improve branch coverage. The assumption is that an interleaving that covers more branches is more likely to trigger an error. An extended version of the *branch* heuristic measures the percentage of branches that can be covered if a given state is visited next, using a simple syntactic scan of the program to compute the total number of branches in the program. The *most-blocked* heuristic assigns higher relevance to the states that produce more blocked threads. The assumption is that an interleaving with a higher number of blocked threads is more likely to produce concurrency errors such as deadlock. The *interleaving* heuristic assigns higher relevance to interleavings that produce more context switches. The assumption is that an interleaving that contains more context switches is more likely to trigger errors.

The effectiveness of heuristic-guided testing is determined by the accuracy of the heuristic function. Looking for an error with an incomplete state-transition graph is similar to the situation of looking for a house with an incomplete map. When the instructions that have already executed are used to predict future testing, heuristic functions can only offer imprecise guidance. Experimental results from [32] show that the proposed heuristic functions report mixed performance across test suites. To produce better guidance, heuristic functions need to have a better understanding of further exploration. For example, if an accurate transition graph cannot be obtained, then other constructs like a statically-generated CFG could provide a coarse-grained view on what might execute during further exploration.

Combining Static and Dynamic Testing

Due to their respective strengths and limitations, many existing techniques combine static and dynamic techniques to leverage the advantages of both.

The most common collaboration scheme is to have the static tool identify a set of relevant artefacts that help guide the dynamic tool during testing. For example, static analysis provides a set of instructions that access variables shared by multiple threads using alias and MHP analysis. In Verisoft, this statically generated information is used by the *persistent-set* POR to determine whether an enabled transition accesses a variable shared across threads. This information is also used by [55, 75] to select a set of instructions that are instrumented for permutation at runtime. For the atomicity checking in [1], MHP and alias analysis are applied to identify a set of instructions irrelevant for atomicity checking. Then, the runtime checking focuses on those instructions not pruned out by the static phase.

Static techniques can also be combined with dynamic techniques to provide better heuristic guidance on searches. Rungta et al. [63] introduce a search heuristic that gives higher priority to transitions with the shortest path to the next possible program error-checking point, such as an assertion or any user-specified program statement. The computation of a path to an error-checking program-point requires a statically computed CFG to be traversed. It is important to note in this case that the coarse-grained CFG is used to complement the imprecise transition graph to predict future testing. However, due to the coarse-grained nature of the CFG, such paths are always imprecise. For example, it is difficult for static analysis to determine which concrete method is actually dispatched from a polymorphic call-site. Moreover, it is difficult to determine the number of iterations a loop executes dynamically. Thus, the number of instructions in a path does not always reflect the distance between a transition and the error-checking point at runtime. Therefore, this problem raises an interesting possibility of using precise dynamic information to correct the imprecision of CFG and improve the overall precision of heuristic functions. In this arrangement, the collaboration requires information to be exchanged both ways between static analysis and the dynamic testing-tool.

2.3 Summary

In this chapter discusses different types of concurrency bugs, along with a review of existing techniques for detecting concurrency bugs are reviewed that attempt to deal with nondeterministic execution. Static-analysis testing-techniques bypass the nondeterminism issue. However, due to their coarse-grained nature, static-based techniques produce many false positives in a bug report. Dynamic testing-techniques try to execute as many different interleavings as possible to uncover hidden bugs. Depending on the techniques

used, dynamic testing can be either unsound or sound. However, soundness requires testing all possible interleavings, which is often intractable. Model checking stands out as a technique that, at least theoretically, could test all concurrent interleavings of a program. Explicit-state model-checking is easier to use than abstract model-checking because it works directly on the program instead of its high-level abstraction. The performance of dynamic testing-tools, especially model checkers, can be improved using POR and heuristic search. POR techniques reduce the total number of interleavings that need to be checked to cover all possible concurrent-value schedules. Heuristic search attempts to guide the dynamic testing-tool to the error-triggering interleavings first. Attempts have been made to combine static and dynamic techniques to improve the overall efficiency of the testing tool. In following chapters, a new testing technique that combines static analysis and a dynamic explicit-state model-checker is presented.

Chapter 3

Overview

This chapter gives a high-level view of value-schedule-based testing using a simple example. Then, an overview of the components that make up the new value-schedule testing-technique is presented. The existing technologies that these components are built upon are discussed. Finally, a suite of test programs used to evaluate the various techniques is presented.

3.1 A Motivating Example

In this section, a motivating example, shown in Figure 3.1, illustrates the workflow and some of the main features of value-schedule-based testing. Later chapters describes the implementation of these features in detail.

Value-schedule-based testing starts with a static analysis of the program to identify points of interests, such as concurrent accesses to shared variables. In Figure 3.1, there are four shared objects, $v1$, $v2$, $v3$ and $v4$, accessed by two threads $Thread1$ and $Thread2$. The static analysis reports four concurrent accesses to the fields of shared objects, as shown in Table 3.1. Of the four pairs of concurrent accesses, three of them have read-/write conflicts on the contents of shared variables, and one of them is a race on monitor acquisition. Moreover, it is important to note that accesses on $v3.val$ from both threads are not included in the table because static analysis can determine they are consistently protected by the same monitor object $v3$, so there is no write conflict.

Permuting the ordering of execution for accesses in each entry leads the program into two different states if the two accesses operate on the same memory address. However, due to the coarse-grained nature of static analysis, this assumption is not always true. Among these four pairs, only two are true positives. For example, the concurrent accesses on $v1.val$ is a false positive because the write to $v1.val$ from $Thread2$ cannot happen since the branch statement always skips the write. As well, the concurrent accesses on $v2[i]$

Shared Variables

```
1   class T{int val;}
2   T v1, v3, v4;
3   int[] v2 = new int[20];
```

Thread 1

```
1  run(){
2      int id = 1;
3      int local = 0;
4      int q = 0;
5
6      local = v1.val;
7
8      local += v2[id];
9      synchronized(v3){
10         v3.val = 1;
11     }
12     q = local / v4.val;
13 }
```

Thread 2

```
1  run(){
2      int id = 2;
3      int local = 0;
4
5      if (local > id) {
6          v1.val = 3;
7      }
8      v2[id] = 2;
9      synchronized(v3){
10         v3.val = 1;
11     }
12     v4.val *= local;
13 }
```

Figure 3.1: Motivating Example

Variable	Thread1	Thread2
$v1$	read of $v1.val$ at line 6	write of $v1.val$ at line 6
$v2$	read of $v2[i]$ at line 8	write of $v2[i]$ at line 8
$v3$	monitor acquisition of $v3$ at line 9	monitor acquisition of $v3$ at line 9
$v4$	read of $v4.val$ at line 12	write of $v4.val$ at line 12

Table 3.1: Concurrent Accesses to Shared Variables from Figure 3.1

turns out to be a false positive because $Thread1$ and $Thread2$ access different elements of $v2$.

For the true race-conditions, it is unknown if they are benign races [51] or can lead to real errors. For example, the general race on the monitor acquisition of the shared object $v3$ does not cause any error because of the common protection by the monitors on $v3$, while the race on the concurrent accesses to $v4.val$ may lead to a division by zero error because of the multiplication by zero.

The second part of value-schedule-based testing is to use static-analysis results to guide dynamic testing to check the program correctness under the different execution orderings implied by each pair of concurrent accesses. At the same time, concrete runtime information is collected and is used to filter out those race conditions that are either false positives or benign.

This guided testing consists of two subtasks. First, for a particular execution ordering implied by a pair of concurrent accesses, a static module computes a set of interleavings. The pair of concurrent accesses that triggers the interleavings is referred to as the permutation assumption of those interleavings. Second, the dynamic testing-tool is used to force the program to execute according to the computed interleavings. If at least one of these interleavings is successfully executed, then the permutation assumption is considered to be feasible under that ordering. For example, conditional branching can result in no successful interleavings.

The concept of guided testing is demonstrated using pairs of concurrent accesses from Table 3.1. Every instruction in the interleaving is denoted in the form of $ty.x$, which stands for the statement at line x of thread y. An interleaving is broken into multiple lines at the context-switch points. For example, for the concurrent accesses on $v1.val$ shown in Table 3.1, the interleaving:

$t1.2 \rightarrow t1.3 \rightarrow t1.4 \rightarrow$

$t2.2 \rightarrow t2.3 \rightarrow t2.5 \rightarrow t2.6 \rightarrow$

$t1.6$

enables the access of $v1.val$ written by $Thread2$ at line 6. In this case, the dynamic testing-tool is unable to fulfill this interleaving because the branch statement at line 5

takes the false branch and skips the write statement contained within. This interleaving is the only interleaving that generates the identified concurrent accesses on $v1.val$. Thus, the concurrent access partner on $v1.val$ from $Thread2$ is said to be inaccessible, and the race condition implied by this concurrent access-pair is a false positive. This interleaving is explored no further.

For the concurrent accesses on $v2$ in Table 3.1, an interleaving such as:

$t1.2 \rightarrow t1.3 \rightarrow t1.4 \rightarrow t1.6 \rightarrow$

$t2.2 \rightarrow t2.3 \rightarrow t2.5 \rightarrow t2.6 \rightarrow t2.8 \rightarrow$

$t1.8$

enables $Thread1$ to read in the value written by $Thread2$ at line 8. This interleaving can be successfully executed by the dynamic testing-tool. However, a runtime check reveals two different memory addresses are referenced by the two instructions. Therefore, the race condition implied by that pair of concurrent accesses is a false positive as well, and the interleaving is explored no further.

The branching and alias information obtained in the previous two examples is used for all further testing of the programs. In both cases, the dynamic testing-tool can suspend testing and feed this information back into the static module to refine its results, which then reduces the scope of the dynamic testing.

For the concurrent accesses on $v3$ in Table 3.1, an interleaving such as:

$t1.2 \rightarrow t1.3 \rightarrow t1.4 \rightarrow t1.6 \rightarrow t1.8 \rightarrow$

$t2.2 \rightarrow t2.3 \rightarrow t2.5 \rightarrow t2.8 \rightarrow t2.9 \rightarrow t2.10 \rightarrow t2.11 \rightarrow$

$t1.9$

fulfills the concurrent monitor acquisitions on $v3$. Note that $t2.11$ corresponds to the '}' representing the monitor exit instruction. This interleaving is also feasible and can be executed by the dynamic testing-tool. Because the permutation assumption of this interleaving is shown to be valid, this interleaving is explored for further testing. If the interleaving is allowed to be extended sequentially, first all remaining instructions from $Thread1$ are explored, followed by all remaining instructions from $Thread2$. For these tests, the program produces no errors at all. Thus, the implied race on the monitor entries is benign.

An interleaving such as:

$t1.2 \rightarrow t1.3 \rightarrow t1.4 \rightarrow t1.6 \rightarrow t1.8 \rightarrow t1.9 \rightarrow t1.10 \rightarrow t1.11 \rightarrow$

$t2.2 \rightarrow t2.3 \rightarrow t2.5 \rightarrow t2.8 \rightarrow t2.9 \rightarrow t2.10 \rightarrow t2.11 \rightarrow t2.12 \rightarrow$

$t1.12$

is computed to fulfill the concurrent accesses on $v4.val$ such that the read of $v4.val$

from $Thread1$ reads in the value written by $Thread2$. The controlled execution of this interleaving leads the program to a *DivsionByZero* error. In the end, the value-schedule-based testing-tool reports only one harmful race on the shared variable $v4.val$.

The testing process illustrates the main feature of value-schedule-based testing: using a collaboration between static and dynamic testing where the information flows bidirectionally between the two modules. The static module instructs the dynamic module on where to permute instructions and how the order of concurrent accesses implied by that permutation can be fulfilled. The dynamic module informs the static module on the validity of a permutation assumption, such as whether concurrent accesses are actually accessible at runtime or accesses are actually operating on the same memory address. Through this collaboration, value-schedule-based testing performs an aggressive POR such that 1) permutations only happen on an instruction that may have concurrent access partners, and 2) the testing of the interleaving is aborted as soon as the permutation assumption on which it is based is shown to be invalid. Moreover, the collaboration is an iterative process. For every pair of concurrent accesses, the static module computes the fulfilling interleaving followed by a validity check in the dynamic module. The same process is repeated for the next pair of concurrent accesses.

Because the static module produces a set of points of interest at the beginning of testing, it is possible for a value-schedule-based testing-tool to prioritize the checking of permutations on concurrent accesses. For example, general races on monitor entries are more likely to be the effect of desirable synchronization to protect critical sections than an error-triggering bug. On the other hand, a data race is generally considered to be bad programming practice that tends to lead to errors. Thus, the value-schedule-based testing can prioritize the checking of permutations to consider concurrent reads/writes to shared variables before concurrent acquisitions of monitors.

Using both aggressive POR and prioritized testing, value-schedule-based testing aims to locate error-triggering concurrency-bugs more effectively. The pairs of concurrent accesses extracted from the sample program are relatively simple, and do not represent all types of conflicting concurrent accesses that may exist in a program. Moreover, the example uses only two threads that have no dependencies between them. Also, only limited types of dynamic information are used for validity checking.

3.2 Overview of Components

This section briefly introduces the existing tools and techniques used to implement the value-schedule-based testing.

As shown in Figure 3.2, value-schedule-based testing-technique consists of two main modules: static and dynamic analysis. The static-analysis module consists of two sub-

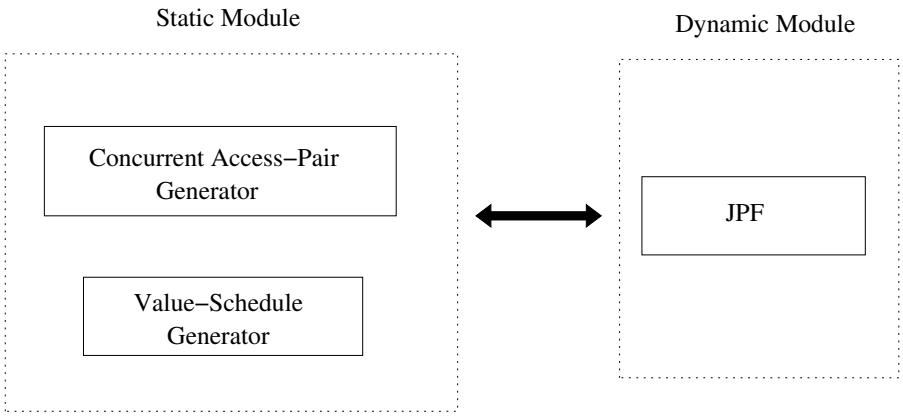


Figure 3.2: Overview of Components

modules: *Concurrent Access-Pair Generator* and *Value-Schedule Generator*. The *Concurrent Access-Pair Generator* produces pairs of concurrent accesses to shared variables in the program. The *Value-Schedule Generator* computes fulfilling interleavings for concurrent access-pairs. Both modules use the *Soot* analysis-package [71] for Java. *Soot* can extract useful information about a program, such as alias analysis [1], call graph analysis [69] and MHP analysis [52].

Unfortunately, the MHP analysis provided by *Soot* has several limitations. First, it cannot process call edges across polymorphic call-sites. Second, it cannot perform MHP analysis over monitor accesses that may operate on multiple monitor-objects. A monitor access may operate on different objects if the variable through which the monitor is referred points to multiple objects. Third, the MHP analysis reports the MHP relationships among all instructions. Many of the reported MHP instructions operate on local variables and are irrelevant to uncovering concurrent bugs. Some of my work extends the existing MHP analysis to handle various program constructs, along with a filtering scheme to extract MHP instructions that are relevant to concurrent testing.

The dynamic module is a customized version of the explicit-state model-checker, JPF [73]. The implementation is built upon JPF 3.1.2 because this work began before the release of JPF 4.1. The dynamic information obtained from the controlled execution of a program is fed back into the *Value Schedule Generator* to help further value-schedule discovery.

3.3 Test Suite

The test programs used to evaluate the various techniques presented in this thesis are listed in Table 3.2. These test programs are either categorized as *kernel (K)*, *real (R)* or *benchmark (B)*, as in [20, 18]. The *kernel* programs are relatively simple and designed

Kind	Program	# Threads	SLOC	Error(bugs)
K	**twostage**	2	52	Exception (Race)
	wronglock	2	4	Exception (Race)
	producerconsumer	7	87	Exception (Race)
	blockbarrier	4	120	Exception (Barging)
	reorder	2	44	Exception (Atomicity)
	deadlock.d1	2	24	Deadlock
	deadlock.d2	2	24	Deadlock
	diningphilosopher	5	25	Deadlock
	losenotify	2	41	Deadlock
	clean	2	51	Deadlock
	nestedmonitor	2	53	Deadlock
R	**alarmclock**	3	125	NullException (Race)
	raxextend	11	103	Exception (Race)
	daisy	2	744	Exception (Race)
	RW-deadlock	4	103	Deadlock (Race)
	RW-exception	4	103	Exception (Race)
	replicatedworker	5	304	Deadlock (Race)
	boundedbuffer	8	65	Deadlock
B	**linkedlist**	2	117	Exception (Race)
	piper	32	71	Deadlock (Barging)
	account-race	5	66	Exception (Race)
	account-deadlock	5	66	Deadlock
	account-subtype	5	91	Exception (Race)

Table 3.2: Test Cases

to illustrate different types of concurrency bugs. The *real* and *benchmark* programs are taken from real-world applications, e.g., *daisy* is a disk simulator. Source code for these test cases can be downloaded from http://sir.unl.edu. The *#Threads* column specifies the number of concurrent threads used in each program excluding the *main* thread. The SLOC column specifies the lines of code in the source program. The *Error(bugs)* column specifies the type of concurrency error in the program followed in parenthesis by the way the program fails as the result of the error. Some programs, such as *RW*, may produce more than one type of error, and are separated into multiple test cases, one for each possible error. For example, an unprotected access of a shared variable in test program *RW*, may cause the program to produce either a runtime exception or a deadlock under different interleavings.

3.4 Summary

This chapter gives a brief overview of the proposed new testing technique. This technique combines both static analysis and dynamic testing to identify concurrency bugs in concurrent programs. Finally, the test suite used to evaluate this new testing technique is presented.

Chapter 4

Concurrent Access-Pair Generation

A concurrent access-pair consists of two concurrent instructions that have conflicting accesses to the same shared variable. Two accesses conflict if the permutation of their execution orders brings the program into different states. In Java, a pair of conflicting accesses could either be concurrent reads/writes to fields of shared objects or contention on acquiring the same monitor object. The goal of concurrent access-pair generation is to extract all such pairs from the program. These pairs provide the dynamic testing-tool with two pieces of relevant information: 1) a set of instructions or accesses that should be considered for permutations when encountered during testing, 2) and for each access, which accesses in other threads are conflicts. The former piece of information is given by the distinct instructions that make up the concurrent access-pairs. The latter piece of information is given by accesses that form concurrent access-pairs with it. These two pieces of information forms the basis for a novel POR technique that is introduced in later chapters. This chapter introduces different types of concurrent access-pairs and various techniques used to extract these pairs.

4.1 Concurrent Access-Pairs

There are five types of concurrent access-pairs: Concurrent Read-Write Pairs (RW), Concurrent Monitor-Entry Pairs (MM), Concurrent Wait-Notify Pairs *(WN)*, Concurrent Wait/Monitor-Entry Pairs (WM) and Concurrent Wait/Wait Pair (WW). The contention caused by each type of concurrent access-pairs is discussed in detail.

Two concurrent accesses to a shared variable form a RW pair if at least one of the two accesses is a write access. In Java, a shared variable can be accessed as a field of an

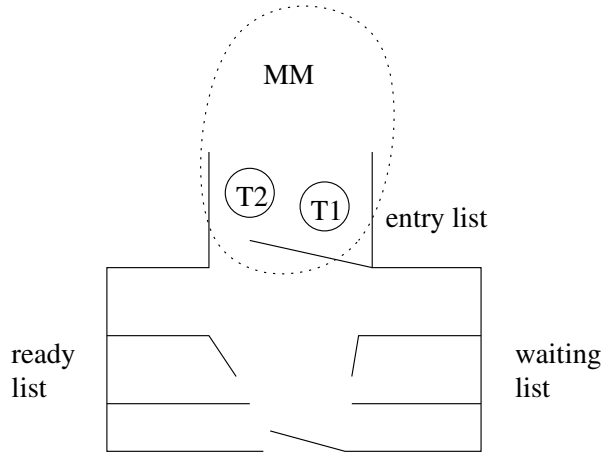


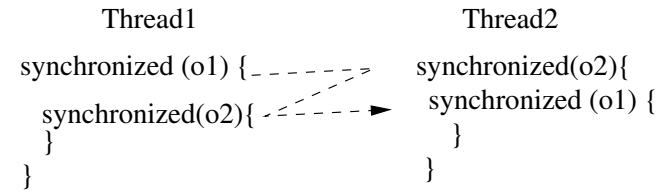
Figure 4.1: Contention on Monitor Entry

instance variable or a static field of a class. A *RW* pair indicates a potential data race in the program.

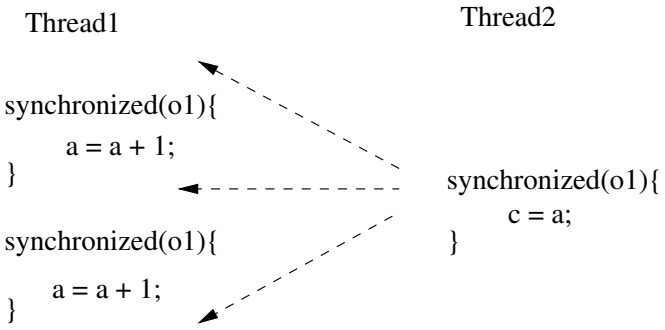
In Java, contention for monitor acquisition happens among threads that are waiting at the monitor entry-point outside of the monitor and those waiting on a *ready* or *urgent* [34] list inside the monitor. Two concurrent attempts to acquire a monitor from its entry point form a *MM* concurrent access-pair. In Java, an acquisition of a monitor from its entry occurs by either executing a *synchronized* method or a *synchronized* statement. In Figure 4.1, the threads $T1$ and $T2$ are attempting to simultaneously acquire the same monitor. Every pair of acquisitions among all tasks form a *MM* concurrent access-pair.

A single *MM* concurrent access-pair does not directly indicate a potential bug. However, the contentions on the monitor involving multiple *MM* concurrent access-pairs can lead to deadlocks or atomicity violations. The code in Figure 4.2(a) can produce a deadlock because the acquisition of monitors occurs in the opposite order. The code in Figure 4.2(b) shows an atomicity violation if the synchronized block from $Thread2$ is executed in between two synchronized blocks from $Thread1$.

Another type of concurrent access-pair is the *WN* pair. A *WN* pair forms when a thread waits in the monitor, and another thread enters the monitor and signals the waiting thread. In Java, a thread waits (blocks) by calling to the monitor's *wait*() method; a waiting thread is signalled by calling the monitor *notify()* or *notifyAll()* method by some other thread. Collectively, the two signalling methods are referred to as *notify/All()*. *WN* differs from *MM* concurrent access-pairs in that there is no contention of execution order in between the instructions because a thread is required to hold a monitor before a call is made on the *wait()* or *notify/All()* methods. In other words, the execution order of the *wait()* and the *notify/All()* element of a *WM* pair is determined by the contention of the monitor entry enclosing the *wait()* and *notify/All()*. Figure 4.3 shows an example of a *WN* concurrent access-pair. Figure 4.3(a) shows $T1$ entering the monitor and blocking



(a) Deadlock Error



(b) Atomicity Violation

Figure 4.2: Possible Concurrency Errors Caused by Contentions on Monitor Entry

by calling the *wait*() method. Figure 4.3(b) shows either $T2$ or $T3$ can enter and call to $notify()$, which unblocks $T1$. Therefore, a *WN* concurrent access-pair is made up of a *wait*() method call and a $notify()$ method call. In Figure 4.3(c), $T1$ is signalled and moved to the ready list by the $notify()$ method invoked by $T2$. It is important to note that these two choices can also be captured using a *MM* concurrent access-pair on the monitor acquisitions for $T2$ and $T3$. Nevertheless, the *WN* pair is treated as a special case because it is relevant to uncovering deadlock. For example, if a program has invocations of *wait*() but no *notify/All()* to form a *WN* pair with that *wait()* statement, then that program may have a potential deadlock. Moreover, combined with an *MM* pair, a *WN* pair can indicate a potential deadlock caused by a *LostNotify* [23] where the $notify()$ procedes the *wait*().

The contention for monitor acquisitions between a thread waiting at the monitor entry-point and a thread waiting on the ready list is captured by a *WM* concurrent access-pair. Therefore, a *WM* concurrent access-pair is made up a *wait()* method and a monitor entry statement. An example of a *WM* pair is shown in Figure 4.4, which continues from Figure 4.3 by assuming $T2$ exits the monitor, and so $T1$ is waiting on the ready list while $T3$ is waiting on the monitor entry list. The contention between $T1$ and $T3$ form the *WM* concurrent access-pair. A *WM* pair may indicate a potential barging problem in a program. In general, a thread blocks inside of a monitor if some condition is unsatisfied, and made ready again by another thread when the condition is satisfied. It is possible that before the newly-ready thread is able to reacquire the monitor, another thread at the monitor entry-point barges in and acquires the monitor. Thus, when the

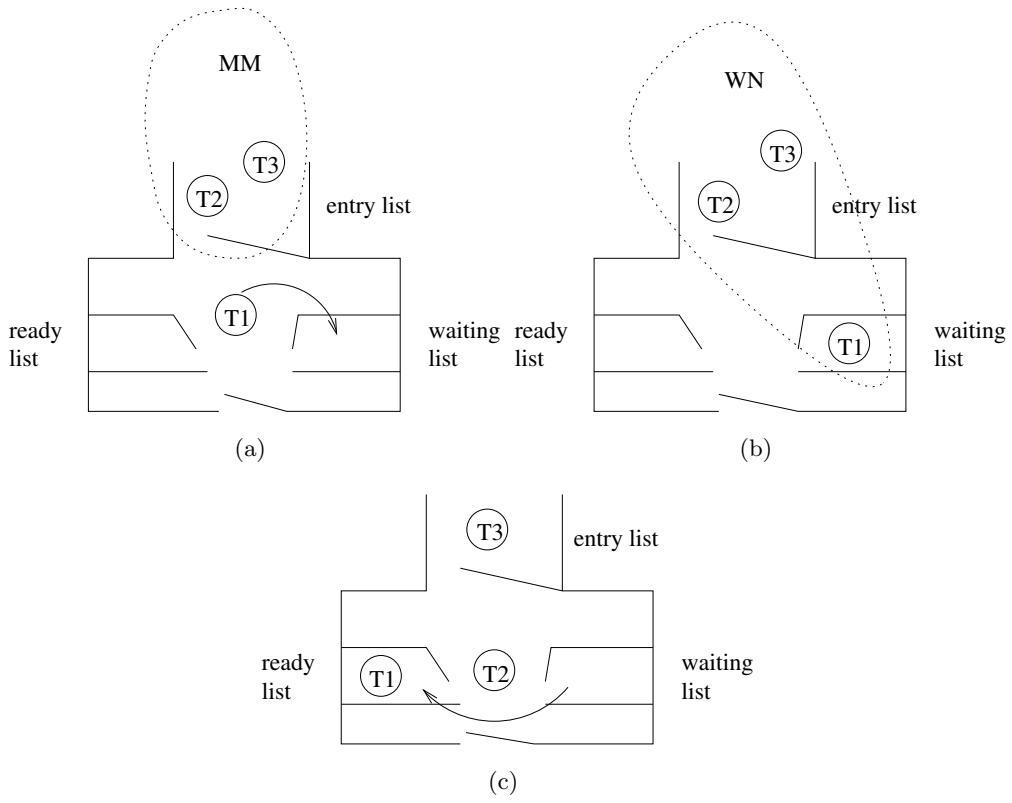


Figure 4.3: Contention on Wait-Notify

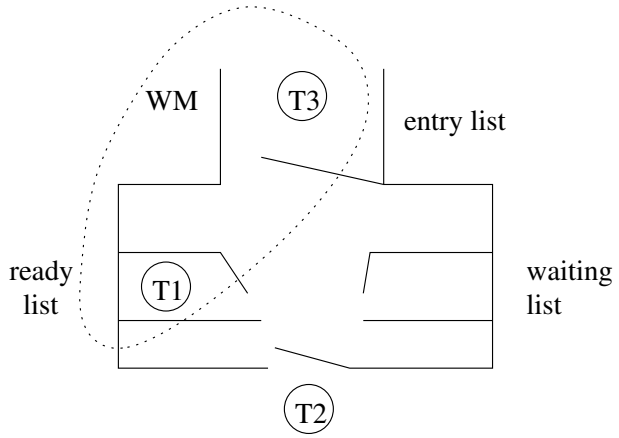


Figure 4.4: Contention on Monitor Entry and Ready List

thread from the ready list finally acquires the monitor, the condition may no longer be satisfied.

Finally, there is another type of contention on monitor acquisition that can happen to threads on the ready list, forming an additional kind of barging. This type of contention consists of two *wait()* method calls and is referred to as a WW pair. An example of WW pair is shown in Figure 4.5, which continued from Figure 4.4. As shown in Figure 4.5(a), $T3$ enters the monitor and blocks by calling $wait()$, and $T4$ arrives and blocks on the entry list while the monitor is occupied by $T3$. In Figure 4.5(b), $T4$ enters the monitor and notifies $T3$, which is moved to the ready list. In Figure 4.5(c), $T4$ exits the monitor, and the ready list has two threads $T1$ and $T3$ that can acquire the monitor next. The contention between $T1$ and $T3$ is captured by a WW pair. It is important to note that WW pair always depends on contentions captured by some other MM, WM and WN pairs. For example, the WW contention depicted in Figure 4.5 is a result of choices made on the MM contentions from Figure 4.1, WN contentions from Figure 4.3 and WM contentions from Figure 4.4. In other words, threads that form WW pairs in the ready list of a particular monitor can be determined by choices made on other types of contentions on the same monitor. Thus, it is impossible to compute WW pairs statically, so the analysis of WW contentions is delayed until runtime. At runtime, it is straightforward to extract threads from the ready list from the JVM to compute WW pairs, thus static computation of WW pairs is unnecessary.

4.1.1 Concurrent Access-Pairs

A concurrent access-pair is defined by

$$(access_i,\ access_j) = <(T_i, I_i, C_i),\ (T_j, I_j, C_j)>$$

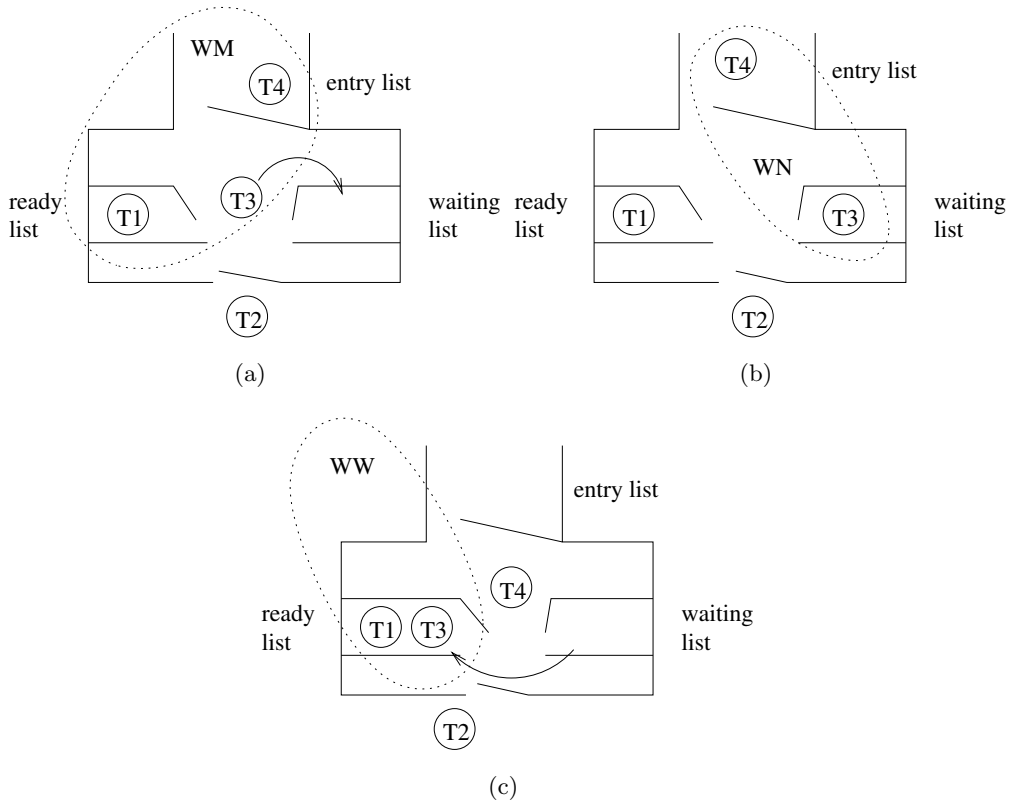


Figure 4.5: Contention on Ready List

$CAccess = FieldAccess \bigcup MonitorEntry \bigcup WaitNotifyInvc$
$FieldAccess$ = all accesses to fields, instance or static
$MonitorEntry$ = all accesses trying to acquire monitors
$WaitNotifyInvc = WaitMethodInvc \bigcup NotifyMethodInvc$
$WaitMethodInvc$ = all monitor accesses that contain invocations of the $wait()$ method
$NotifyMethodInvc$ = all monitor accesses that contain invocations of the $notify/All()$ method

Figure 4.6: Types of Accesses Forming Concurrent Access-Pairs

Thread 1

```
1   void f1(){
2       f2();
3   }

1   void f2(){
2       o.val = 1;
3       synchronized(o){
4           o.wait();
5       }
6   }
```

Thread 2

```
1   void fa(){
2       fb();
3   }

1   void fb(){
2       l = o.val;
3       synchronized(o){
4           o.notifyAll();
5       }
6   }
```

Figure 4.7: Sample Code Illustrating Concurrent Access-Pairs

Type	Concurrent Access-Pairs
RW	<(t1, f2.(w(o.val), line 2), f1.(f2(), line 2)), (t2, fb.(r(o.val), line 2), fa.(fb(), line 2)) >
MM	<(t1, f2.(monenter, line 3), f1.(f2(), line 2)), (t2, fb.(monenter, line 3), fa.(fb(), line 2)) >
MW	<(t1, f2.(o.wait(), line 4), f1.(f2(), line 2)), (t2, fb.(monenter, line 3), fa.(fb(), line 2)) >
WN	<(t1, f2.(o.wait(), line 4), f1.(f2(), line 2)), (t2, fb.(o.notifyAll(), line 4), fa.(fb(), line 2)) >

Table 4.1: Concurrent Access-Pairs from Program in Figure 4.7

where T is the thread id executing the instruction, I is the instruction that performs the access, and C is the call-graph context under which this instruction is executed. Different types of relevant accesses that might be used to form concurrent access-pairs are summarized in Figure 4.6. These accesses include: $FieldAccess$ representing accesses to fields of objects, $MonitorEntry$ representing monitor entries, and $WaitMethodInvc$ and $NotifyMethodInvc$ representing method calls on the monitors.

Some sample concurrent access-pairs extracted from Figure 4.7 are shown in Table 4.1. The first row of Table 4.1 reports a RW pair: <(t1, f2.(w(o.val), line 2), f1.(f2(), line 2)), (t2, fb.(r(o.val), line 2), fa.(fb, line 2)) >. It states the write access to $o.val$ at line 2 of function $f2()$ called from line 2 of function $f1()$ by $Thread1$ conflicts with the read access of $o.val$ at line 2 of function $fb()$ called from line 2 of function $fa()$ of $Thread2$. Note that the instructions in Table 4.1 are specified in a pseudo-code manner for the simplicity. A Java implementation uses bytecodes and their offsets into methods to represent instructions. For example, the instruction f1.(w(o.val), line 2) is internally stored as f1.(PUTFIELD #2, 2).

This new approach extends existing techniques, such as [50, 49], which only identify concurrent reads and writes to shared variables for detecting data races, to cover concurrent accesses among higher-level program constructs, such as monitor acquisitions. In

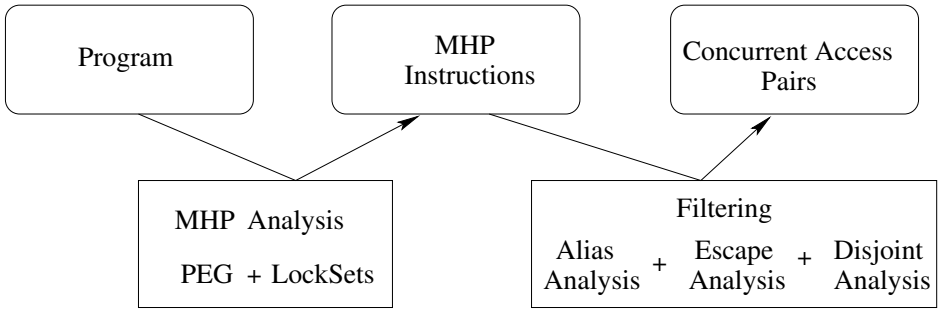


Figure 4.8: Stages in Generating Concurrent Access-Pairs

later chapters, these high-level concurrent access-pairs are shown to facilitate the uncovering of concurrency bugs involving subtle contention on monitor acquisitions.

4.2 Concurrent Access-Pairs Generation

In this section, the discussion centers on various techniques that are used to compute concurrent access-pairs. In essence, concurrent access-pairs are identified in two steps as shown in Figure 4.8: identifying pairs of MHP instructions using MHP analysis and filtering out pairs of MHP instructions that do not have conflicting accesses on shared variables. The MHP analysis is driven by the construction of a PEG for a concurrent program, then performing lockset analysis over it. The filtering process consists of alias, escape and disjoint analysis. The details of these steps are explained in the following sections.

4.2.1 May Happen in Parallel (MHP) Analysis

In this section, a brief overview of MHP analysis is given. The MHP analysis of a concurrent program starts with constructing a PEG of the program. The relationship between the MHP analysis and concurrent access-pairs is addressed.

The details of PEG construction and lockset analysis are given in [52]. The process starts by constructing the intra-thread CFG. Then, inter-thread edges are created between synchronization and mutual exclusion among threads. Figure 4.9 shows a PEG of the program in Figure 4.7. In the graph, an interprocedural CFG is generated for both threads with each statement represented as a node in the graph. Each node consists of a triple <monitor-object, type, thread>. The *monitor-object* represents the monitor object this node operates on. If a node does not operate on a monitor, the *monitor-object* field has a '$*$'. The *type* represents the operation this node performs. Some types relevant to synchronization and mutual exclusion are: *entry* for monitor entry, *exit* for monitor exit, *notify/All* for signaling methods, and *wait()* which is represented with three types

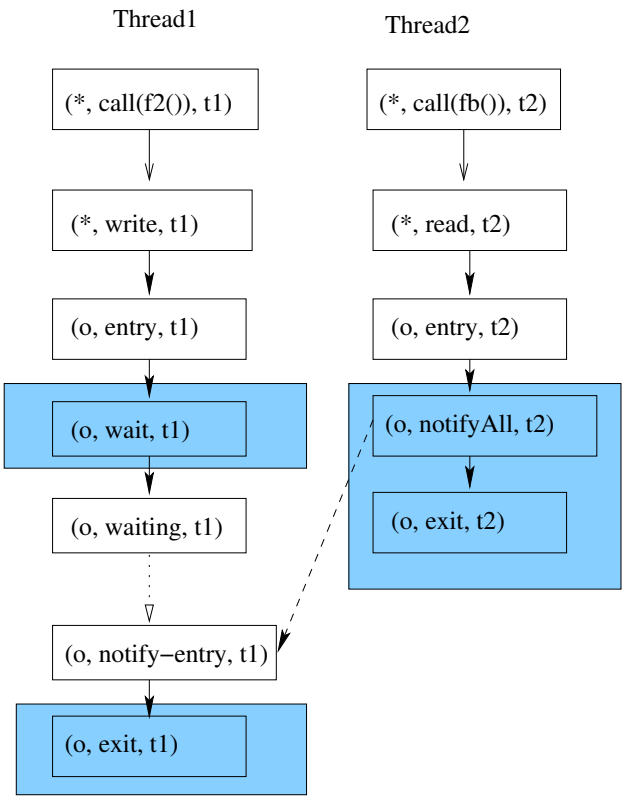


Figure 4.9: PEG for Sample Program in Figure 4.7

of nodes: *wait*, *waiting* and *notify-entry*. The *wait* stands for the program point where a *wait()* is called. *waiting* represents the program point when a thread is waiting for signalling. The *notify-entry* stands for the program point when a thread is moved to the ready list after signalling. Finally, the *thread* field represents which thread a node belongs to.

In Figure 4.9, a synchronization dependency edge is drawn as a dashed line, e.g., from (o, notifyAll, t2) to (o, notify-entry, t1), which implies $Thread1$ depends on the execution of (o, notifyAll, t2) to proceed beyond (o, waiting, t1). Mutually excluded nodes are shaded, and a control flow line is dotted when a task is blocked.

After the PEG is computed, a flow-sensitive lockset analysis is applied. Nodes from different threads that are not protected by the same set of monitors and not blocked by synchronization dependency edges are marked as MHP. Synchronization constructs such as *notifyAll* and *join* are used to indicate the *must-happen-before* relationships among nodes. Moreover, the MHP analysis keeps track of recursive entry of monitors when computing MHP relationships.

The different types of concurrent access-pairs discussed in Section 4.1 can be derived from a PEG. For example, the (o, entry, t1) and (o, entry, t2) are shown to have MHP relationships. Moreover, they may form a *MM* pair because they operate on the

Thread 1

```
1  f1(){
2    if (f.val > 1 ) {
3      //a subtype of F
4      f = new FSub();
5    } else {
6      f = new F();
7    }
8    f.f2();
9  }
```

Callees

```
1  class F{
2    f2(){
3    //as in Figure 4.7
4    }
5  }
6  class SubF extends F{
7    f2(){
8      synchronized(o){
9        o.val = 1;
10     }
11   }
12 }
```

Figure 4.10: Sample Code with Polymorphic Calls

same object o according to static analysis. The synchronization dependency edge can be transformed into a *WN* pair.

Finally, by splitting a *wait*() call into three nodes, PEG exposes the concurrency that happens between *wait()* and other statements accessing the same monitor. As shown in Figure 4.9, the (o, wait, t1) node representing the invocation of *wait()* is mutually exclusive with the *entry* and *exit* nodes in $Thread2$. On the other hand, the (o, waiting, t1) node and (o, notify-entry, t1) nodes are not shown to have any mutually exclusive relationships with nodes in $Thread2$. More importantly, the (o, notify-entry, t1) node is shown to have an MHP relationship with the (o, notifyAll, t2) node, so the *wait()* and monitor entry form a *WN* pair.

4.2.2 Customizing Existing MHP Analysis

Existing PEG generation and MHP analysis [44] have some limitations. PEG construction tends to make conservative approximations when extended over unresolved polymorphic call-sites or monitor entries. A call site is considered unresolved if the exact runtime type of a callee cannot be resolved by static analysis. A monitor entry becomes unresolved if static analysis believes it may point to more than one runtime object. Implementation in [3] requires the precise *type* and *points-to* information on nodes that are relevant to inter-procedural and inter-thread analysis. These limitations are intentional and are designed to reduce the complexity of the MHP analysis.

In the original MHP analysis given in [52], a cloning technique is used to create extra branches in the CFG to handle ambiguous type and points-to relationships. For example,

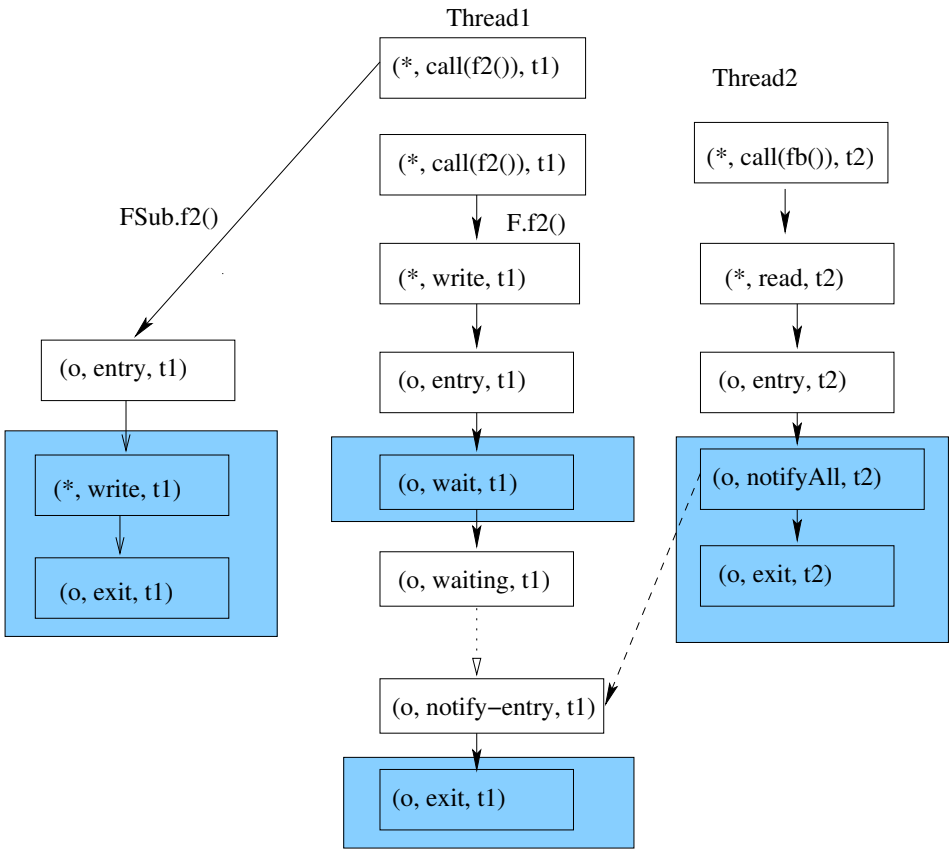


Figure 4.11: Polymorphic PEG for Code in Figure 4.7 and 4.10

if a callee of a method invocation can be resolved to two different types, then two method calls, one for each possible type of the callee, are added into the CFG. The same cloning approach is applied for entries on monitors that may point to more than one runtime object.

An example of cloning is shown using the sample code in Figure 4.10, which modifies the $f1()$ method in $Thread1$ of Figure 4.7 to initialize f to one of two possible types depending on the branch taken at runtime. The static type analysis resolves the type of the callee at line 8 to either F or $SubF$. A sound PEG construction should always take both cases into consideration.

The clone technique produces a separate node for each possible type of callee in the caller's CFG as in [52]. For example, the call to $f2()$ in Figure 4.10 is resolved to two sequential nodes: $F.f2()$ and $SubF.f2()$. Thus, each of the two method calls is extended separately. Moreover, the MHP analysis will be performed as if there are two consecutive calls in $Thread1$, one to $F.f2()$ and another to $SubF.f2()$. The resulting PEG for the code in Figure 4.10 is shown in Figure 4.11. The same cloning technique is applied to an unresolved monitor entry to clone all entry nodes for each possible object type.

Thread 1

```
1  f1(){
2    if (val > 1) {
3        o = o1;
4    } else {
5        o = o2;
6    }
7    f2();
8  }
```

Thread 2

```
1  fa(){
2    if (val > 1) {
3        o = o2;
4    } else {
5        o = o1;
6    }
7    fb();
8  }
```

Figure 4.12: Aliasing of Monitor Operations

As reported by [44], the cloning method leads to a significant increase in the size of the PEG, and limits the applicability and scalability of the MHP analysis. Thus, [44] proposes the use of Class Hierarchy Analysis to resolve as many polymorphic call-sites and monitor entries as possible. The analysis aborts if the program has unresolved call sites or monitor entries. The argument is that in the presence of many unresolved program facts, the cloning technique produces many superfluous nodes in the PEG. Thus, the MHP analysis is unable to finish or produces too many false positives. Either way, little useful information can be obtained using MHP analysis.

A MHP analysis that produces no report is useless to the proposed collaboration between static and dynamic analyses. Therefore, I propose a compromise between the approach taken by [52, 44]. In my approach, the cloning technique is applied only on unresolved polymorphic call-sites, but not the unresolved monitor-entries. My technique conservatively treats nodes guarded by unresolved monitor entries as if they are protected by no monitor entry at all, for the following reason. The cloning due to polymorphism is necessary to preserve the completeness of the PEG. If a possible target method of a polymorphic call is not included in the graph, relevant MHP relationships among nodes are missed, thus rendering the analysis incomplete. On the other hand, treating unresolved monitor entries as if they offer no protection produces false positives in the MHP pairs only if two unresolved monitor entries turn out to hold the same monitor at runtime. However, this conservative approximation does not produce any false negatives, which is safe. Moreover, different dynamic mechanisms can identify these false positives caused by unresolved monitor entries. Compared to [52], the new technique reduces the workload of MHP analysis but produces more false positives. Compared to [44], the new technique is more complete because it allows MHP analysis to produce useful information in the presence of unresolved call-sites, but might run more slowly if many cloned polymorphic calls exist in the program.

For example, Figure 4.12 shows a modified version of $f1()$ and $fa()$ from Figure 4.7

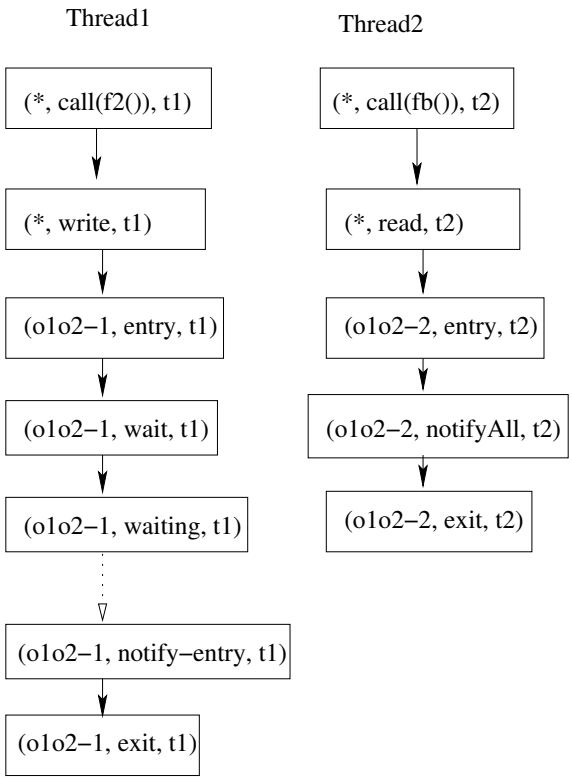


Figure 4.13: PEG for Code in Figure 4.7 and 4.12

which initializes the shared variable o to one of two variables if another shared variable val is greater than 1. Multiple assignments lead the static alias-analysis to report the monitor entries on o in methods $f2()$ and $fb()$ point-to multiple objects. The analysis in [52] clones the monitor entry for the two possible monitor objects, while analysis from [44] aborts when this case is encountered.

In my solution, when an unresolved monitor entry is encountered, a distinct *artificial* object is created for that monitor entry. For example, if a monitor entry is believed to operate on two possible objects, such as $o1$ and $o2$ for the monitor entry in Figure 4.7 called from the code in Figure 4.12, a distinct object with a name formed by concatenating the names of the possible points-to objects and a special *sequence id* is created. The sequence id is always incremented by one whenever an unresolved monitor entry node with the same set of aliasing objects is encountered. The aim of this naming convention is to make this artificial object unique throughout the program, so the names of may-alias objects forms a key for that particular alias relationship. Thus, every unresolved monitor entry is considered by the MHP analysis to operate on a unique object, so that all nodes enclosed by this unresolved monitor entry are considered to be free of mutual exclusion with all other nodes in the program.

Figure 4.13 shows the new PEG for the program constructed using the code given

in Figure 4.7 and Figure 4.12. The monitor entry nodes from both threads now have distinct object names $o1o2-1$ and $o1o2-3$. Note this special naming scheme is used instead of '*' because the underlying MHP analysis must distinguish between monitor and non-monitor objects. This approach enables the MHP analysis to be carried out across a synchronization clause on a may-alias monitor object without cloning the synchronized region. As a result, the shaded mutual exclusive regions in Figure 4.9 no longer exist, and no nodes are not considered to be mutual exclusive by the MHP analysis. However, this approach may result in superfluous MHP pairs if the may-alias monitor-entries actually access the same shared variable.

4.2.3 Extracting Relevant MHP Pairs

In the previous section, the MHP analysis was customized to uniquely identify MHP instructions in a program with unresolved monitor entries. In this section, the discussion examines techniques used to filter out irrelevant MHP instruction pairs, such as accesses to local variables, and produce a report that contains only the types of MHP pairs outlined in Section 4.1. All five types of concurrent access-pairs identified in Section 4.1 require concurrent instructions that operate on the same shared variables. The two key criteria here are "same" and "shared". Intuitively, if two concurrent instructions operate on the same variable, then the variable is "shared". In this case, an analysis, such as alias analysis, should be sufficient to identify relevant MHP pairs. However, in practice, due to the imprecision of alias analyses, these two criteria have to be validated with different static analyses.

The filtering process is arranged in a similar fashion as that in [50]. The first stage is to filter out MHP pairs that could never operate on the same variable using alias analysis. For example, accesses on primitive-type variables declared within methods are filtered out. Then, concurrent accesses that operate on *must-not-alias* variables are filtered out. The remaining pairs are those believed to at least operate on *may-alias* variables.

Because coarse-grained alias-analyses, including that implemented for the *Spark* package of *Soot*, do not offer context sensitivity during analysis, two *may-alias* variable-accesses from two threads might be local to the individual thread. In Figure 4.14, a context-insensitive analysis reports write accesses to the variable *o.val* from both threads through the calling sequence $f1() \rightarrow f2()$, because the o accessed in $f2()$ always resolves to the object instantiated in $f1()$. However, because $f1() \rightarrow f2()$ is invoked by different threads, each thread is actually accessing a different instance of o. One solution is to use context-sensitive alias-analyses, such as [5], but this requires more computational resources than context-insensitive analysis. Moreover, depending on the level of sensitivity, false positives may still be reported. Or, an escape analysis [14, 7] can be used to determine that the objects instantiated in function $f1()$ never escape the thread scope. Thus,

Thread 1

```
1  run(){
2      f1();
3  }
```

Thread 2

```
1  run(){
2      f1();
3  }
```

Functions

```
1  //assume globally defined
2  void f1(){
3      o = new O();
4      f2(o);
5  }
6  void f2(O o){
7      o.val = 2;
8  }
```

Figure 4.14: Combining Alias and Escape Analysis

no concurrent accesses to the object allocated in the function $f1()$ operate on the same object even though a coarse-grained alias-analysis may report it. Currently, the escape analysis from [17] is used in my implementation.

As discussed in Section 4.2.2, the monitor protection offered by an unresolved monitor-entry is deliberately removed from the PEG to compute a safe set of MHP relationships. This approach unavoidably introduces many false positives into the report if two unresolved monitor entries turn out to operate on the same monitor. Therefore, it makes sense to minimize the impact of this technique. In Figure 4.15, the synchronized function $f2()$ of class O is invoked in both $Thread1$ and $Thread2$. Moreover, the exact identity of O for the invocation of $f2()$ cannot be resolved using alias analysis. Thus, monitor entries on the synchronized $f2()$ are artificially hidden away during the MHP analysis, and $f2()$ is considered to be executed concurrently by the two threads. Hence, the variable accesses in $f2()$ are considered to form concurrent access-pairs if they operate on the same object. A closer inspection of the program reveals that accesses to the *val* fields in $f2()$ cannot form any relevant concurrent access-pairs, for the following reason. If $f2()$ is invoked on two different instances of o by two threads, then the access of *val* at line 3 operates on different objects. On the other hand, if callees of $f2()$ use the same object at runtime, then accesses at line 3 have mutual exclusion anyway. Thus, concurrent accesses on *val* at line 3 can always be safely excluded.

The *must-alias* relationship between an object access within a synchronized region and the monitor object that protects the synchronized region provides new opportunities for pruning out irrelevant MHP pairs. The *must-alias* relationship can always be established between *this* object-references in a synchronized method and the callee of that method. However, such a *must-alias* relationship is harder to establish between a protecting monitor object and objects accessed in the protected region if the monitor is acquired using a synchronized statement, such as "synchronized(o){}". For example, it

```
1  class O{
2      public synchronized void f2(){
3          val = val + 1;
4      }
5      int val;
```

Thread 1

```
1  f1(){
2    if (val > 1) {
3        o = o1;
4    } else {
5        o = o2;
6    }
7    o.f2();
8  }
```

Thread 2

```
1  fa(){
2    if (val > 1) {
3        o = o2;
4    } else {
5        o = o1;
6    }
7    o.f2();
8  }
```

Figure 4.15: Aliasing of Monitor Operations

is possible to change o after the monitor for o is acquired. Therefore, in the current implementation, only accesses protected by synchronized methods are considered for such filtering. The remaining false positives are identified using the collaboration between static and dynamic testing techniques discussed later. This filtering technique is similar to that introduced in [49], where MHP accesses to fields of array-elements do not contribute to data races if elements of the array are instantiated with different objects. My technique recognizes that the concurrent accesses to fields referenced by *this* are always conflict free if methods enclosing them are synchronized, called *disjoint* filtering.

The program artifacts and constructs used in the rules for computing concurrent access-pairs are given in Figure 4.16. The most relevant constructs are: $mhp(a1,\ a2)$, $mayAlias(ref1,\ ref2)$, $escape(ref)$ and $disjoint(a1,\ a2)$. The $mhp(a1,\ a2)$ determines MHP relationship between two program statements $stmt1$ and $stmt2$. The $mayAlias(ref1, ref2)$ determines whether two object references are to the same object. The $escape(ref)$ determines whether any object referenced by ref is thread global. The escape status of a reference to an object is conservatively estimated as the disjunction of the escape status of all objects that may be pointed to by that reference. The $disjoint(a1,\ a2)$ determines whether two accesses $a1$ and $a2$ can be filtered out using disjoint analysis.

Figure 4.17 summarizes the rules for computing different types of concurrent access-pairs. Rule 1 specifies steps for computing RW pairs from accesses to instance and static fields. The MHP analysis is used to determine whether any two accesses may happen in parallel. After that, filtering determines whether the references to base objects in the

$\mathcal{P}$ = program under analysis $\quad$ $\mathcal{O}$ = objects allocated in $\mathcal{P}$

$\mathcal{T}$ = all threads in $\mathcal{P}$ $\quad$ $CAccess$ = relevant accesses as defined in Figure 4.6

$\mathcal{SM}$ = all static/instance synchronized methods in $\mathcal{P}$

$Ref_{instance}$ = all references to instance variables in $\mathcal{P}$

Ref_{static} = all references to static variables in $\mathcal{P}$

$\mathcal{W}$ = all write accesses in $\mathcal{P}$

$decl(a) = \{m \mid m \in \mathcal{SM}, a \in CAccess \bigwedge a \text{ is defined in } m\}$

$ref(a)$ = base reference of a field access, where $a \in CAcess$

$field(a)$ = field referenced by an access, where $a \in CAcess$

$method(a)$ = method referenced by an access, where $a \in CAcess$

$pointsTo(ref)$ = objects that may be pointed to by ref where $ref \in Ref_{inst} \bigcup Ref_{static}$

$$mhp(a1,\ a2) = \begin{cases} true, & \text{if a1 may execute concurrently with a2} \\ false, & otherwise; \end{cases}$$

where $a1$, $a2 \in CAccess$

$$mayAlias(ref1,\ ref2) = \begin{cases} true, & \text{if } pointsTo(ref1) \bigcap pointsTo(ref2) \neq \emptyset \\ false, & \text{otherwise;} \end{cases}$$

where $ref1$, $ref2 \in (Ref_{inst} \bigcup Ref_{static})$

$$escape(o) = \begin{cases} true, & \text{if } \exists(a1 \in t1 \bigwedge a2 \in t2) where \\ & o \in pointsTo(ref(a1)) \bigwedge o \in pointsTo(ref(a2)) \\ false, & \text{otherwise;} \end{cases}$$

where $o \in \mathcal{O}$, $a1$, $a2 \in CAccess$, $t1$, $t2 \in \mathcal{T}$

$$escape(r) = \begin{cases} true, & \text{if } \exists o \in pointsTo(r) \text{ such that } escape(o) = true \\ & o \in pointsTo(ref(a1)) \bigwedge o \in pointsTo(ref(a2)) \\ false, & \text{otherwise;} \end{cases}$$

where $r \in (Ref_{inst} \bigcup Ref_{static})$

$$disjoint(a1,\ a2) = \begin{cases} true, & \text{if } ref(a1) = this \bigwedge ref(a2) = this \\ & \bigwedge decl(a1) \in \mathcal{SM} \bigwedge decl(a2) \in \mathcal{SM}; \\ false, & \text{otherwise;} \end{cases}$$

where $a1$, $a2 \in CAccess$

Figure 4.16: Constructs for Filtering

Rule 1 : (a1, a2) where a1, a2 $\in FieldAccess$
$\in$ RW IF
$mhp(a1,\ a2) \bigwedge$
$mayAlias(ref(a1),\ ref(a2)) \bigwedge field(a1) = field(a2) \bigwedge$
$escape(ref(a1)) \bigwedge escape(ref(a2)) \bigwedge$
$\neg\ disjoint(a1, a2) \bigwedge$
(a1 $\in \mathcal{W} \bigvee$ a2 $\in \mathcal{W}$)

Rule 2 : (a1, a2) where a1, a2 $\in MonitorEntry$
$\in$ MM IF
$mhp(a1,\ a2) \bigwedge mayAlias(ref(a1),\ ref(a2))$

Rule 3 : (a1, a2) where a1 $\in WaitMethodInvc \bigwedge$ a2 $\in MonitorEntry$
$\in$ WM IF
$mhp(a1,\ a2) \bigwedge mayAlias(ref(a1),\ ref(a2))$

Rule 4 : (a1, a2) where a1, a2 $\in WaitNotifyInvc$
$\in$ WN IF
((a1 $\in WaitMethodInvc \bigvee$ a2 $\in WaitMethodInvc$)
$\bigwedge$ (a1 $\in NotifyMethodInvc \bigvee$ a2 $\in NotifyMethodInvc$))
$\bigwedge mhp(a1,\ a2) \bigwedge mayAlias(ref(a1),\ ref(a2))$

Figure 4.17: Rules for Computing Concurrent Access-Pairs

accesses have a *may-alias* relationship and that the referenced field names are the same. It is important to note that an access to an element of an array is treated as an access to an instance field of that array object. Due to the coarse-grained nature of alias analysis, if the references to the base objects of two array element accesses may-alias, then elements accessed from those array objects are considered to have alias relationships as well. After the alias analysis, the escape analysis determines whether the base objects escape the thread scope. Last, the disjoint analysis is used to determine whether two accesses can be made irrelevant. For accesses to static fields, the names of classes and fields referenced are checked. A *RW* pair is formed if the access happens on the same static field of the same class. Notes that the escape filtering is unnecessary for static accesses because static fields are always visible from other classes as well.

Rule 2 specifies rules for computing *MM* pairs. The monitor entry can be attempted using a synchronized method invocation or a synchronized statement. The references to monitors are checked for alias relationships.

Rule 3 and 4 specify rules for computing *WM* and *WN* pairs. In Rule 3, the reference to the receiving object of the *wait()* method-call and the reference of the monitor object in the monitor-entry statement are checked for a possible alias relationship. A *WM* pair is created if such a *may-alias* relationship exists. In Rule 4, the receiver of the *wait()* and *notify/All()* method calls are checked for alias relationships. A *WN* pair is created if such a *may-alias* relationship is held. Note that there is no rule for computing *WW* pairs because they are intended to be computed at runtime.

To summarize, the MHP analysis is applied across all rules. Alias filtering and escape filtering are always applied for concurrent access-pairs that involve references to instance objects. The disjoint filtering is only applied for filtering *RW* pairs that make up accesses to field instances.

4.3 Evaluation

Table 4.2 reports the concurrent access-pairs extracted from the test programs listed in Table 3.2 of Chapter 3. All tests are performed using a Java 1.5 runtime on a Dual Core AMD Opteron(tm) Processor 885 with 1.0 GHZ CPU and 16GB of memory, and 2.0 GB of JVM heap. Column *Nodes* reports the number of nodes that make up the PEG. The cloning of polymorphic call-sites only occurred for the test program *account-subtype*. The disabling of unresolved monitor-entries only occurred for test programs: *deadlock.d2*, *diningphilosopher*, *account-deadlock*, *account-race* and *account-subtype*.

Column *MHP Pairs* reports the number of MHP pairs computed directly for the MHP analysis using [44]. The *RW*, *MM*, *WM* and *WN* columns report the number of pairs computed for each type of concurrent access-pair using the filtering process consisting

of alias, escape and disjoint pruning. Column *Time* reports the time taken to complete the extraction of concurrent access-pairs. In general, the time spent on each program is directly proportional to the number of nodes in that program's PEG. The algorithmic complexity of the MHP analysis is $O(N^3)$ [52] where N is the number of PEG nodes. Table 4.2 reports time out (TO) for three test programs: *piper*, *daisy* and *replicatedworkers*, when performing the MHP analysis, where the time out limit is 1 hour. It is important to note that these three test programs have the highest number of PEG nodes in the test suite. This result is consistent with the experimental results reported in [52] and [44]. In [52], all test cases have less than 500 PEG nodes. In [44], time outs are reported for test programs with more than 1000 nodes. For example, a program with 2173 PEG nodes takes 44553 seconds to complete on a Pentium 4 1.8 GHZ machine and 1.5 GB of JVM heap. While these previous numbers were collected under different environments, they do serve as an indication of the limitations of the existing MHP analysis techniques.

For the three test programs that time out, the MHP analysis is relaxed to reduce the complexity by ignoring the synchronization constructs, reducing the complexity to $O(N^2)$. Because this coarse-grained MHP analysis does not consider any synchronization and mutual exclusion in program, it is referred to as *unsync* MHP analysis. Moreover, because it does not require the program facts such as mutual exclusion and synchronization dependencies to be maintained and propagated through nodes in the PEG, it tends to complete much faster but generates a far less precise report. Table 4.3 shows the experiment output for those three programs that timed out.

Next, the effectiveness of the different filtering techniques is evaluated. The alias filtering achieves reduction in all 23 test programs. The escape and disjoint filtering achieves improvements in only the test programs listed in Table 4.4 and Table 4.5.

The two test programs where escape filtering achieves an improvement are listed in Table 4.4. The escape analysis reduces the number of *RW* pairs 95 to 41 for *Producer-Consumer* program and from 272 to 1 in *daisy*. The escape-analysis filtering has no effect on the number of monitor related pairs generated because attempts to acquire a monitor for an object that is not shared are rare in practice. The escape-analysis filtering does slightly increase overall computation time.

The four test programs where the disjoint analysis achieves reduction are listed in Table 4.5. In the *account-race* test program, the disjoint filtering reduces the number of *RW* pairs from 4320 to 1890, and in the *account-subtype* test program disjoint filtering reduces the number of *RW* pairs from 4590 to 2040. In both cases, two fold reductions are achieved. These two test programs contain code that attempts to acquire monitor entries on some elements of an array, then call synchronized methods on those objects to update their fields. The disjoint-analysis filtering is ideal for recognizing whether synchronized methods are invoked. The concurrent updating of the *this* referenced fields in the method is considered to be safe by the disjoint-analysis filtering. The *piper* and

Program	Nodes	MHP	Alias + Escape + Disjoint				Time
		Pairs	RW	MM	WM	WN	(sec.)
twostage	84	1255	0	2	0	0	4.54
wronglock	79	863	5	0	0	0	3.72
producerconsumer	380	26354	41	44	54	54	23.97
blockbarrier	226	6196	0	6	12	12	7.54
reorder	77	1082	20	0	0	0	3.62
deadlock.d1	63	672	0	2	0	0	4.09
deadlock.d2	45	435	0	4	0	0	3.37
diningphilosopher	108	3225	0	40	0	0	5.02
losenotify	77	713	0	1	1	1	3.53
clean	71	912	7	2	2	2	4.45
nestedmonitor	75	923	0	2	2	2	3.97
alarmclock	263	13948	16	6	4	2	16.61
raxextend	218	16159	10	35	0	0	15.83
daisy	957						TO
RW-deadlock	415	31063	51	52	36	12	27.27
RW-exception	415	31063	51	52	36	12	26.27
replicatedworker	1650						TO
boundedbuffer	349	41325	0	28	56	56	38.51
linkedlist	272	8432	144	0	0	0	9.97
piper	1133						TO
account-race	314	33825	1890	1210	0	0	86.48
account-deadlock	269	24150	0	1210	0	0	57.83
account-subtype	256	18440	2040	490	0	0	29.10

Table 4.2: Concurrent Access-Pairs Computed with Precise MHP Analysis

Program	Nodes	MHP	Alias + Escape + Disjoint				Time
		Pairs	RW	MM	WM	WN	(sec)
piper	1133	104281	0	496	992	992	11.84
replicatedworker	1650	28901	545	1100	640	295	18.90
daisy	957	202531	1	45	12	2	16.90

Table 4.3: Concurrent Access-Pairs Computed with Coarse MHP Analysis

Program	Alias + Disjoint				Time
	RW	MM	WM	WN	(sec.)
producerconsumer	95	44	54	54	23.82
daisy	272	45	12	2	16.24
	Alias + Escape + Disjoint				
producerconsumer	41	44	54	54	23.97
daisy	1	45	12	2	16.90

Table 4.4: Test Programs Improved by Escape Analysis

Program	Alias + Escape				Time
	RW	MM	WM	WN	(sec.)
account-race	4320	1210	0	0	83.00
account-subtype	4590	490	0	0	26.16
piper	2208	496	992	992	11.04
replicatedworker	1025	1390	640	295	18.37
	Alias + Escape + Disjoint				
account-race	1890	1210	0	0	86.48
account-subtype	2040	490	0	0	29.10
piper	0	496	992	992	11.84
replicatedworker	545	1100	640	295	18.90

Table 4.5: Test Programs Improved by Disjoint Analysis

replicatedworkers programs are computed with the coarse-grained MHP analysis without considering synchronization. However, by recognizing the synchronized method that an access to a shared variable is enclosed in and the callee objects of that synchronized method, the disjoint analysis can filter out concurrent access-pairs that are irrelevant if the enclosing synchronizated methods are invoked on different objects. Again, the disjoint analysis filtering leads to some small computational overhead.

4.4 Summary

This chapter introduces static analyses used in computing different types of concurrent access-pairs that may potentially contribute to different types of concurrency bugs. These concurrent access-pairs include: *RW*, *MM*, *WM*, *WW* and *WN*. Then, an extension to the existing PEG generation technique is introduced. The new extension aims to preserve the completeness of the PEG by cloning the polymorphic call-edges while avoiding generating too many spurious PEG nodes by treating unresolved monitor-entries as an acquisition of a unique monitor. After that, different filtering techniques are introduced to extract

relevant concurrent access-pairs from the program. These filtering techniques make use of static analyses such as alias, escape and disjoint analysis. Finally, the generation process for concurrent access-pairs is applied over the test programs presented in Chapter 3. The experimental results confirm that the various techniques discussed in this chapter do improve the effectiveness of the concurrent access-pair generation.

Chapter 5

Value-Schedule-Based Testing

This chapter examines how value-schedule-based testing makes use of statically computed information, such as concurrent access-pairs, to guide dynamic testing. The original definition of value schedule from [6] is extended to give a more precise specification of the ordering of concurrent accesses to shared variables, followed by a straightforward implementation of value-schedule-based testing. The value schedules and their fulfilling interleavings are computed and executed for testing. The testing process illustrates how the two main features of value-schedule-based testing, accessibility-based POR and prioritized testing, are achieved. Finally, value schedules and their fulfilling interleavings are derived for different program constructs in a concurrent Java program, along with proofs to show the completeness of the derivation with a distinct set of concurrent accesses.

5.1 Motivation

The sample program in Figure 5.1 is used to demonstrate several important concepts. The sample program has two threads that deposit and withdraw money from two accounts, $accX$ and $accY$. If the two threads execute sequentially, i.e., one after another, then both accounts have a value of zero at the end, and the assertion at line 11 of the sample program evaluates to true. However, when the program is run concurrently, the implementation of $Thread2$ leads to a data race because it mistakenly acquires monitor $accY$ instead of $accX$ when updating account $accX$. For example, it is possible that both threads read in the original values of $accX.val$ and write them out concurrently. In the end, $accX.val$ may have a non-zero value when the threads join. In this case, the assertion evaluates to false and fails. It is the task of the testing tool to uncover such an interleaving efficiently. As well, the if statement at line 4 of $Thread2$ can never be true, because $accY.val$ is always greater than or equal to 0. In this simple case, static analysis might discover this dead code; in general, static analysis cannot always detect dead code precisely. For this discussion, it is assumed the static analysis did not detect the dead code.

Main Thread

```
1  main(String[] args){
2    Account accX = new Account(1);
3    Account accY = new Account(2);
4    Thread1 t1 =
5      new Thread1(accX, accY);
6    Thread2 t2 =
7      new Thread2(accX, accY);
8    t1.start(); t2.start();
9    t1.join(); t2.join();
10
11   assert(accX.val == accY.val);
12  }
13
14 class Account{
15     Account(int id){
16         this.id = id;
17         this.val = 0;
18     }
19     int val;
20     int id;
21     public static int amt = 10;
22 }
```

Thread 1

```
1  run(){
2    synchronized(accX){
3      accX.val += Account.amt;
4    }
5    synchronized(accY){
6      accY.val += Account.amt;
7    }
8  }
```

Thread 2

```
1    run(){//should be accX & accY
2     synchronized(accY){
3      accX.val -= Account.amt;
4      if (accY.val < 0){
5          accX.val = 0;
6      }
7     }
8     synchronized(accY){
9        accY.val -= Account.amt;
10    }
11   }
```

Figure 5.1: Sample Program

As discussed in Chapter 2, a dynamic testing-tool, such as a model checker, may test all concurrent behaviours of a program by running at least one interleaving for every distinct partial ordering of accesses to shared variables. However, the task of generating an interleaving for each distinct partial ordering is nontrivial. In general, it requires the dynamic testing-tool to know whether a permutation of execution orders should be taken before executing an instruction, and which threads are runnable after the instruction is executed. The first issue is to determine whether an instruction accesses a shared variable that is concurrently accessed by other threads. The second issue tries to determine which threads execute the concurrent conflicting-accesses to the shared variable.

In many existing dynamic testing-tools, these two issues are evaluated using a coarse-grained estimate. Thus, it is possible that a context switch is taken based on some incorrect assumptions, and the resulting interleaving does not lead to the testing of any new partial ordering among concurrent accesses. For example, the sample program in Figure 5.1 is supposed to have two threads deposit and withdraw money from two accounts, $accX$ and $accY$. The partial interleaving 5.1 is a possible interleaving of the sample program.

$$\begin{aligned}
&\text{read of } accX.val \text{ at line 3 of } Thread1 \rightarrow \\
&\text{read of } accX.val \text{ at line 3 of } Thread2 \rightarrow \\
&\text{write of } accX.val \text{ at line 3 of } Thread2 \rightarrow \\
&\text{compare of } accY.val \text{ at line 4 of } Thread2 \rightarrow \\
&\text{write of } accX.val \text{ at line 5 of } Thread2 \rightarrow \\
&\text{write of } accX.val \text{ at line 3 of } Thread1
\end{aligned} \tag{5.1}$$

As shown in Figure 5.2, the context switch may be forced on the read of $accX.val$ at line 3 of $Thread1$ because the static analysis might report the existence of a conflicting write to $accX.val$ at line 5 of $Thread2$. The assumption of the context switch is that it leads to an interleaving where the write at line 5 of $Thread2$ happens before the read at line 3 of $Thread1$. However, because the write in $Thread2$ is not reachable at runtime, this context switch is not necessary. To identify such a superfluous interleaving, the dynamic testing-tool has to keep track of the assumption of the context switch from which the interleaving is spawned. Moreover, the dynamic testing-tool should be able to determine when the assumption of a context switch becomes infeasible an this interleaving, and abort further extension of that interleaving promptly. In this example, the dynamic testing-tool should remember that this context switch is intended to reach line 5 of $Thread2$. When the statement is bypassed by the condition at line 4 because $accY.val$ is initialized to zero, further extension of this interleaving can be abandoned.

Moreover, existing testing techniques do not offer mechanisms for prioritizing the testing of more relevant permutations on the execution orders between an access and

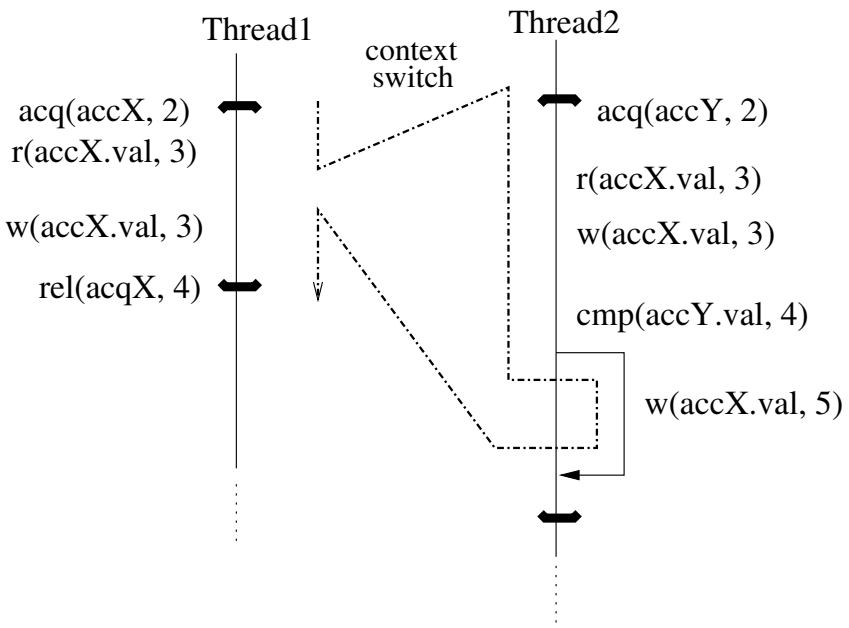


Figure 5.2: Illustration of Interleaving 5.1

its conflicting concurrent accesses, either by programmers or by some heuristic estimate. Prioritizing is important because exhaustive testing of all partial orderings of shared variables is often impossible due to constraints on computational resources as discussed in Chapter 2. Dynamic testing-tools, such as [72, 25], cannot achieve complete coverage because they require exhaustive testing to ensure all possible permutations between an access and its concurrent conflicting-accesses be covered. An essential requirement for such testing is to be aware of conflicting accesses when encountered, and explicitly guide further testing to fulfill the orderings between an access and its concurrent conflicting-accesses.

In value-schedule-based testing, the testing process is driven by deriving and fulfilling value schedules of a program. A value schedule defines a partial ordering of concurrent accesses to shared variables and is constructed from concurrent access-pairs discussed in Chapter 4. From the value schedule, the dynamic testing-tool knows two important pieces of information: 1) accesses where context switches should take place, and 2) concurrent conflicting-accesses of a given instruction. For an access i, all threads that contain accesses forming concurrent access-pairs with i are considered for context switches in value-schedule-based testing. Moreover, from the concurrent access-pairs associated with a concurrent access, it is possible to derive all possible orderings between that access and its concurrent conflicting-accesses. All such orderings are considered to be potentially fulfillable by performing a context switch before i. In other words, in the value-schedule-based testing, a context switch is always taken with the knowledge of the new orderings that might be fulfilled with this context switch.

Using this information, for each new ordering of concurrent accesses implied by a context switch, value-schedule-based testing tries to compute a feasible fulfilling-execution-path. An extension of an interleaving spawned from a context switch is aborted when value-schedule-based testing determines there is no feasible execution-path to fulfill the new order of accesses implied by the context switch. The process of determining the existence of a fulfilling execution-path for the access order implied by a context switch is a mechanism that helps to remove superfluous interleavings introduced by imprecise program information.

In interleaving 5.1, value-schedule-based testing might initially consider the read at line 3 of $Thread1$ for a context switch because it forms a concurrent access-pair with the other accesses to $accX.val$, such as the write to $accX.val$ at line 5 in $Thread2$. However, through the computation of fulfilling execution-paths, value-schedule-based testing determines that there is no feasible execution-path that leads to any more concurrent accesses to $accX.val$ in $Thread2$ because the branch statement at line 4 of $Thread2$ always evaluates to false. Thus, any interleavings derived from this context switch are aborted for further testing.

Because value-schedule-based testing is aware of all possible concurrent conflicting-accesses of a given access, it can guide the dynamic testing-tool to test some relevant orderings associated with a particular access by explicitly computing execution paths that fulfill these orderings before others. For example, programmers might want to evaluate the effects of permuting the execution order between the write to $accX.val$ from $Thread1$ with all of its concurrent accesses in $Thread2$. Value-schedule-based testing can derive and mark the fulfilling interleavings to be tested before others.

5.2 Value Schedules of a Concurrent Program

The value schedule defined in [6] is expanded to cover orderings of different types of concurrent accesses other than those between reads and writes to the same shared variable. The goal of the modification is to have the value schedule provide sufficiently detailed ordering constraints on concurrent accesses to guide the dynamic testing.

5.2.1 Extension

The original definition of value schedule is given as follows in [6]:

> A value schedule is the set (equivalence class) of all thread schedules that agree, for every critical read event r, on the critical write event that produced the value read by r.

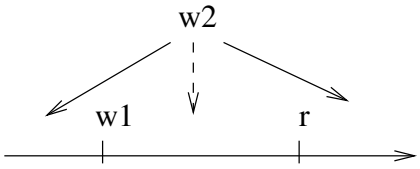
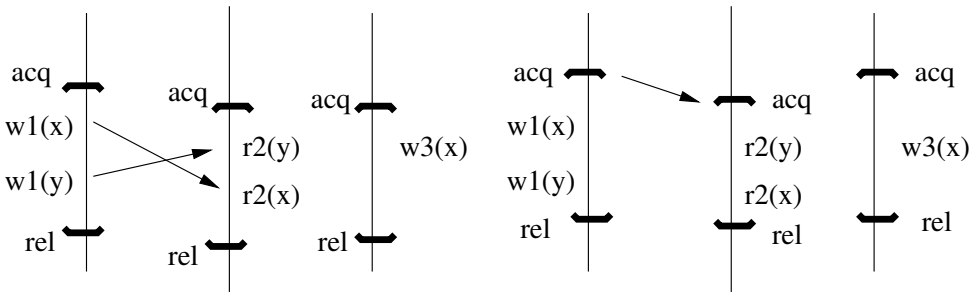


Figure 5.3: Ordering of Reads/Writes in Value Schedule

For my purpose, a write event in a value schedule that produces a value read by a read event is a visible write event. A visible write event forms a *def-use* pair with each read event that reads in the value in the value schedule. Thus, a value schedule consists of a sequence of *def-use* pairs. However, a program can also have invisible write events that are not read by any read event. An important property of the above definition is that a value schedule does not define the order of invisible writes as they do not affect the set of def-use pairs in a value schedule and thus cannot affect the state of the program. For example, consider the sequence of events shown in Figure 5.3. There are two concurrent writes ($w1$, $w2$) and one concurrent read (r) to a shared variable, where the read event could read either value. The value schedule only specifies one def-use pair indicating which of $w1$ or $w2$ produces the value read by r; the other write event does not introduce a second *def-use* pair and thus is irrelevant. One valid value schedule for Figure 5.3 is that r reads in the value of $w1$, making $w2$ invisible. The value schedule does not specify if $w2$ happens before $w1$ or after r.

However, to guide the testing-tool to test a def-use relationship, the tool has to know the desired ordering of the visible read/write accesses as well as orderings among invisible concurrent writes that may interfere with the def-use relationship. For example, the dynamic testing-tool has to guide the program to follow an interleaving such that $w1$ happens before r, and $w2$ does not happen in between $w1$ and r. However, the orderings of $w2$ with respect to $w1$ and r are not present in the original definition of a value-schedule. Therefore, the original definition of the value schedule is expanded to cover orderings among all concurrent read and write accesses such as those among concurrent writes.

The original definition of the value schedule is also expanded to cover the ordering of concurrent accesses on higher-level concurrent constructs such as monitor entry. The benefit of specifying the order among higher-level constructs is two-fold. First, although monitor operations, such as acquiring and releasing a monitor, could be expressed as a set of reads/writes to the monitor object's internal fields, this process breaks the abstraction provided by the programming language or the underlying virtual machine, and exposes accesses that were originally hidden away from programmers. Thus, I believe it is necessary for the value schedule to be specified at a level that can be understood by programmers. The second benefit to specifying orderings on high-level concurrent con-



(a) Ordering on Reads Writes (b) Ordering on Enclosing Monitors

Figure 5.4: Extending Value Schedules over High-Level Constructs

structs is that it can reduce the size of a value schedule. For example, in Figure 5.4(a), two orderings have to be specified for the reads/writes of variables x and y separately. If a value schedule specifies the order of monitor entries, then the read/write ordering of x and y is resolved to the order of the enclosing monitor entries as shown in Figure 5.4(b). Thus, the number of orderings that need to be specified for the same set of reads/writes is reduced from two to one.

As a result, a value schedule is extended to cover partial orderings among concurrent writes, among concurrent reads and writes, and among monitor acquisitions. The definition of the extended value schedule is given below.

Definition 5.1. *The new value schedule corresponds to a set (equivalent class) of interleavings that 1) agree on the same set of accesses, both reads/writes and higher-level monitor acquisitions, to shared variables and 2) where all conflicting concurrent-accesses to shared variables follow the same partial order.*

The first criterion defines the relevant concurrent accesses in the program. The second criterion defines the total ordering among conflicting concurrent accesses for any particular shared variable. The new value schedule augments the partial ordering of concurrent accesses from [6] to include read/write access and write/write accesses. This extension allows the dynamic testing-tool to perform more aggressive POR and prioritized testing as detailed in subsequent sections. In the rest of this dissertation, this extended definition for a value schedule is used unless otherwise noted.

For example, Figure 5.5 shows another possible interleaving that fulfills the same partial ordering as that from Figure 5.2. In both interleavings, the read of $accX.val$ from $Thread1$ happens before all its conflicting accesses from $Thread2$, and the write of $accX.val$ from line 3 of $Thread1$ happens after the write of accX.val from line 5 of $Thread2$. Hence, both interleavings fulfil the same value-schedule because they agree on the same set of accesses, and all conflicting concurrent-accesses to $accX.val$ follow the same partial order. Therefore, two interleavings correspond to one value schedule. This

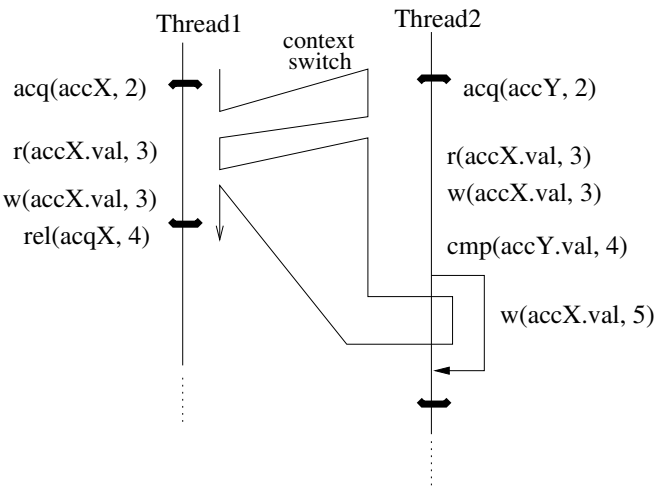


Figure 5.5: An Equivalent interleaving of Interleaving 5.1

equivalence class is subsequently used to reduce the number of interleavings that need to be tested.

5.2.2 Constructs

To specify a value schedule for a program, two pieces of relevant information are needed: 1) the conflicting accesses in the program, and 2) the ordering among conflicting accesses. The concurrent access-pairs defined in Chapter 4 provide the conflicting accesses in a program. The representation of a concurrent access-pair consists of a tuple, $(a_i,\ a_j)$, where a_i and a_j are conflicting accesses. The information missing from a concurrent access-pair is the ordering between these two conflicting accesses.

To represent the ordering relationship between two conflicting accesses, the ordering symbol $\rightarrow$ is used to represent the happens-before relationship as defined by Lamport [39], called an *ordered concurrent-access-pair*. For example, to represent that access a_j happens before a_i, the ordered concurrent-access-pair is written as $(a_j \rightarrow a_i)$. To ensure the value written by a_j reaches a_i, the execution order of other concurrent accesses that conflict with a_j and a_i need to be defined. For example, if a_j has another concurrent conflicting write a_k, an ordering such as $a_k \rightarrow a_j$ or $a_i \rightarrow a_k$ must also exist in the same value schedule to ensure the value given by a_j is read by a_i. When defining an ordering that directly results in an assignment relationship, a special happens-before symbol, $\xrightarrow{def}$, is used. Thus, $a_j \xrightarrow{def} a_i$ implies that a_j happens before a_i and its value is read by a_i. Similarly, for concurrent access-pairs associated with monitor acquisitions, in the cases that the immediate happens-before ordering is not required for conflicting monitor acquisitions, the symbol $\rightarrow$ is used. The symbol $\xrightarrow{acq}$ is used to define the immediate happens-before ordering of monitor acquisitions such that no other conflicting monitor acquisition may

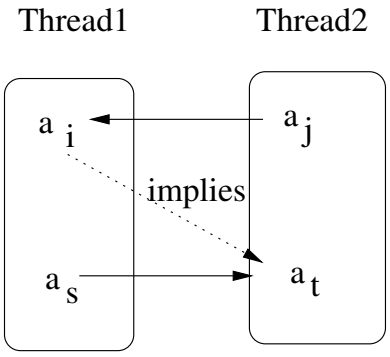


Figure 5.6: Ordering implied by Sequential Dependencies

happen in between.

As discussed, the ordered concurrent-access-pairs act as the basic blocks for a value schedule. The ordering among accesses to a shared variable is given by the happens-before relationships directly or transitively specified in the ordered concurrent-accesses involving this shared variable and the dependencies of accesses from the threads forming those ordered concurrent-access-pairs. These orderings are always transitive; therefore, two ordered concurrent-access-pairs $(a_i \rightarrow a_j)$ and $(a_j \rightarrow a_k)$ imply $(a_i \rightarrow a_k)$. The dependencies among accesses for a thread become relevant if two ordered concurrent-access-pairs involve at least two accesses from the same thread. In that case, the dependencies among accesses from a thread imply orderings in addition to those defined in the two ordered concurrent-access-pairs. For example, assume there are four concurrent accesses, a_i, a_j, a_s and a_t, to the same shared variable. a_i and a_s are from $Thread1$, while a_j and a_t are from $Thread2$. Assume $(a_j \rightarrow a_i)$ and $(a_s \rightarrow a_t)$ for $Thread1$. These concurrent access-pairs imply an additional ordering $(a_i \rightarrow a_t)$ following the sequential dependency between a_i and a_s. As shown in Figure 5.6, the orderings specified by the ordered concurrent-access-pairs are given in solid lines while the orderings implied by sequential dependencies are given in the dashed line.

Table 5.1 shows that the static concurrent access-pair generation discussed in Chapter 4 produces five concurrent access-pairs of type RW, and two of type MM for the program in Figure 5.1. Because all accesses to shared variables happen in the $run()$ method of the threads, the contexts of the accesses are considered to be empty because there is no direct caller, and thus they are omitted from all future references to the concurrent access-pairs for this sample program.

The possible value schedule in Figure 5.7 illustrates the use of ordered concurrent-access-pairs to define the orderings among concurrent accesses. This value schedule consists of three concurrent access-pairs. The first ordered concurrent-access-pair states that the read from line 3 in $Thread1$ happens before the first write to $accX.val$ at line 3 in $Thread2$. This ordering ensures that $Thread1$ reads in the default value of $accX.val$.

Pair Type	Concurrent Access-Pairs
RW	((t1, run.(r(accX.val), line 3)), (t2, run.(w(accX.val), line 3)))
RW	((t1, run.(r(accX.val), line 3)), (t2, run.(w(accX.val), line 5)))
RW	((t1, run.(w(accX.val), line 3)), (t2, run.(r(accX.val), line 3)))
RW	((t1, run.(w(accX.val), line 3)), (t2, run.(w(accX.val), line 3)))
RW	((t1, run.(w(accX.val), line 3)), (t2, run.(w(accX.val), line 5)))
MM	((t1, run.(acq(accY), line 5)), (t2, run.(acq(accY), line 2)))
MM	((t1, run.(acq(accY), line 5)), (t2, run.(acq(accY), line 8)))

Table 5.1: Concurrent Access-Pairs for Program in Figure 5.1

(t1, run.(r(accX.val), line 3))$\longrightarrow$ (t2, run.(w(accX.val), line 3))
(t1, run.(w(accX.val), line 3)) $\longrightarrow$ (t2, run.(w(accX.val), line 5))
(t1, run.(acq(accY)), line 5) $\stackrel{acq}{\longrightarrow}$ (t2, run.(acq(accY), line 2))

Figure 5.7: Sample Concurrent Access-Pairs and Value Schedules from Figure 5.1

The second ordered concurrent-access-pair states that the write to $accX.val$ from line 3 of $Thread1$ happens before the write to $accX.val$ from line 5 of $Thread2$. Thus, the write to $accX.val$ from line 5 of $Thread2$ is the last write to $accX.val$ before the threads join. The third ordered concurrent-access-pair states that $Thread1$ acquires the monitor of $accY$ at line 5 before $Thread2$ acquires the same monitor at line 2. This ordering on monitor entry implies the reads enclosed in the synchronization clauses of $Thread2$ read the value given by the writes enclosed in the synchronized clauses of $Thread1$. The deduction drawn from this value schedule might be used to refine both static and dynamic analysis.

5.2.3 Implications

A value schedule is suitable for guiding dynamic testing because it captures two important pieces of information about concurrent execution before the program is run: 1) the accesses where permutations or context switches should take place, and 2) the orderings intended by each permutation. For example, for a concurrent access-pair, $w \stackrel{def}{\longrightarrow} r$, the first piece of information given by this pair is that w and r should be considered for context switching when encountered. If r is encountered during the execution and r forms concurrent access-pairs with w, the execution of r should yield to let w execute first. After w is executed, the execution should resume with the execution of r. Moreover, the concurrent access-pair makes the intended ordering triggered by the context switches visible to the testing tool. This information is particularly important because it gives the dynamic testing-tool the necessary information to drive the interleavings spawned off by a context switch. When the value-schedule-based testing determines an interleaving cannot fulfill

the ordering of concurrent accesses implied by the context switch that had originally spawned it, value-schedule-based testing aborts further execution along that interleaving and saves computational resources. In the above example, if value-schedule-based testing determines that control flow prevents w from being executed, the concurrent access-pair cannot be fulfilled and the execution of the interleaving can be stopped. In contrast, existing dynamic testing-tools do not associate a particular ordering with a context switch. Therefore, they cannot detect when a desired access is no longer reachable and continue to explore the program, unaware that the test case cannot be fulfilled.

Moreover, with the knowledge of the exact identities of concurrent conflicting-accesses, it is possible to guide the dynamic testing-tool to check permutations of execution orders between a particular access and its concurrent accesses before checking other orderings. For example, a programmer might want to know what happens if a particular read access retrieves different values given by its concurrent writes.

In following sections, the discussion centers on showing how the derivation of value schedules of a concurrent program could be used to drive dynamic testing. In this process, the interleavings that could fulfill value schedules are computed, and the dynamic testing-tool is instructed to execute those computed interleavings to test the correctness of the program execution under corresponding value schedules. Because the testing process is driven by the derivation of value schedules, it is called value-schedule-based testing.

5.3 Generating Value Schedules and Fulfilling Interleavings

This section introduces a simple implementation of value-schedule-based testing. The sample program in Figure 5.1 is used to illustrate the general workflow of the value-schedule-based testing. It is important to note that the implementation presented in this section is not efficient; a practical implementation of value-schedule-based testing is given in Chapter 6. However, this simple implementation is sufficient to illustrate the core concept of value-schedule-based testing. Moreover, by applying this simple implementation to the sample program in Figure 5.1, the improvement offered by the value-schedule-based testing over a traditional dynamic testing-tool is illustrated. The dynamic testing-tool used for the implementation of value-schedule-based testing is JPF, an explicit model checker. In the remaining part of this chapter, the dynamic testing-tool is always referred to as JPF.

The main idea of value-schedule-based testing is to have the dynamic testing-tool execute one interleaving for each value schedule of the program. Thus, the value-schedule-based testing has three parts: 1) deriving the value schedules of a program, 2) deriving the fulfilling interleavings of the value schedules, and 3) executing fulfilling interleavings of value schedules to uncover bugs. Note that a fulfilling interleaving of a value schedule

is an interleaving that executes the accesses to shared variables in the order given by the value schedule.

The first two tasks are accomplished by traversing the CFG of the program. An ordered concurrent-access-pair is added to a value schedule when one of its member accesses is encountered during the traversal. As an ordered concurrent-access-pair is added, its fulfilling interleaving is also computed from the CFG. The third task is fulfilled by executing the generated fulfilling interleavings in JPF.

5.3.1 Two-Stage Testing

Figure 5.8 shows the two-staged implementation of value-schedule-based testing. First, the value schedules and their fulfilling interleavings are statically computed. Second, each fulfilling interleaving is run by the dynamic testing-tool. The algorithm is first illustrated using the sample program in Figure 5.1. Then, the sample program is extended with three threads to show the general applicability of this approach.

CFG Traversal

The algorithm for computing value schedules and their fulfilling interleavings is implemented using a worklist algorithm that repeatly calls the *search*() procedure. The testing process starts by selecting a random thread to start the CFG traversal. The algorithm for computing value schedules and their fulfilling interleavings is implemented using a working algorithm that repeatedly calls the *search*() procedure. The *search*() function adds new search paths to the work list, where these new paths are formed by extending its partial input path with a successor CFG node. A CFG node that may form a concurrent access-pair with other nodes contains a value-schedule-relevant access. A stack in the search-path object is used to track the fulfilling of concurrent access-pairs involving each value-schedule-relevant access as it is encountered during the CFG traversal. This processing of value-schedule-relevant accesses produces value schedules and their fulfilling interleavings that cover all possible orderings between each access and its concurrent conflicting-accesses.

For example, for the sample program in Figure 5.1, assume the exploration starts with $Thread1$. Then, the read access of $accX.val$ at line 3 is the first value-schedule-relevant access encountered during the traversal. Recall that in Table 5.1, this read access forms two concurrent access-pairs with two other writes from $Thread2$. These three accesses of $accX.val$ produce three distinct orderings as shown in Figure 5.9. The three orderings represent the cases where 1) the read happens before the two writes in $Thread2$, 2) the read happens in between the two writes in $Thread2$, 3) the read happens after the two writes in $Thread2$. The first ordering is captured by the ordered concurrent access-pair

```
1  main()
2      Select random thread t and create a SearchPath p
3      Initialize p with the entry node of t
4      Push p onto worklist
5      while worklist has more SearchPath p
6          Set p as the SearchPath removed from the front of worklist
7          search (p, worklist)
8      verify();
9  search (SearchPath p, List worklist)
10     Let c be CFG node at end of path
11     Set S to a set of successors of c
12     if (S == empty && p.stack == empty && hasOnlyOneThreadLeft(p)){
13         fulfilling.add(p);
14     }
15     Foreach s in S
16       Append s to p
17       If s is search target (top of the fulfillment stack of p)
18          Pop head of the fulfillment stack of p
19       Add extended p to the front of worklist
20
21       If s is in any concurrent access-pairs
22          Foreach instruction i in such a pair
23            Set newpath as a clone of p
24            Push i onto the fulfillment stack of newpath
25            Add newpath as the 2nd element of worklist
26 verify()
27    Foreach p in fulfilling
28       status = Dynamic.execute(p);
29       if (status == infeasible)
30         pruning();
31       else if (status == error)
32         report();
```

Figure 5.8: Overview of Search Algorithm

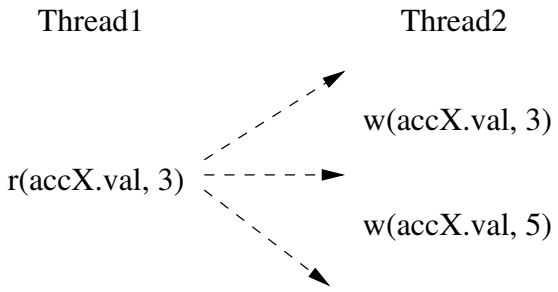


Figure 5.9: Possible Orderings of the Read at Line 3

(t1, run.(r(accX.val), line 3))$\longrightarrow$ (t2, run.(w(accX.val), line 3)). The second ordering is captured by the ordered concurrent-access-pair (t2, run.(w(accX.val), line 3)) $\xrightarrow{def}$ (t1, run.(r(accX.val), line 3)). The third ordering is captured by the ordered concurrent-access-pair (t2, run.(w(accX.val), line 5)) $\xrightarrow{def}$ (t1, run.(r(accX.val), line 3)). Thus, the processing of the current value-schedule-relevant node derives three different partial value-schedules of length 1, which are added to the search path.

The fulfilling interleavings for these ordered concurrent-access-pairs are computed as follows. For the first case, the read access is appended to the existing interleaving, ensuring the read is executed before both writes from $Thread1$. The computed interleaving is added to the end of the worklist for future expansion. By explicitly adding the read access to the interleaving, this interleaving ensures that this read access happen before all of its concurrent accesses.

To derive interleavings for the second and third partial-value schedules, the original interleaving is cloned without the read access appended at line 23 of Figure 5.8. Moreover, each cloned interleaving takes one of the two writes from $Thread2$ as the search target that is pushed onto a stack in the path. A search target is a concurrent conflicting-access intended to happen before the current access, in this case the read of $accX.val$ at line 3 of $Thread1$. Each cloned interleaving is expanded to derive a path that leads to the execution to the specified search target. Then, these two interleavings are added to the worklist, at line 25 of Figure 5.8, as the second and third items, while the first item on the worklist is the fulfilling interleaving of the first partial value-schedule. This approach ensures a depth-first search of the CFG.

When an interleaving with a search target attached is removed from the worklist, it is processed with the goal of traversing the CFG to reach the search target. Assume the interleaving for the second case, with the first write to $accX.val$ as the search target removed from the worklist. Then, the CFG traversal starts with $Thread2$ to compute a possible execution path that reaches the first write to $accX.val$ in $Thread2$. During the traversal, every node is added to the interleaving until the node corresponding to the search target is encountered (lines 16 to 18 of Figure 5.8). Then, the original node that

Rule-1 A default value-schedule is derived and fulfilled by directly appending the value-schedule-relevant instruction to the interleaving.

Rule-2 For each concurrent access, a_t, which forms a concurrent access-pair with this value-schedule-relevant instruction, a value schedule is derived and fulfilled such that a_t happens before the value-schedule-relevant instruction being processed.

Figure 5.10: General Derivation Rules

triggered this search (the read of $accX.val$ at line 3 of $Thread1$) is appended to the path after the search target. In the end, the partial value-schedule and its fulfilling interleaving is computed as given in value schedule 5.2.

Value Schedule: (5.2)
(t2, run.(w(accX.val), line 3)) $\xrightarrow{def}$ (t1, run.(r(accX.val), line 3))

Interleaving :
t1.(acq(accX, 2)) →
t2.(acq(accY, 2)) →
t2.(r(accX.val, 3))→ t2.(w(accX.val, 3)) →
t1.(r(accX.val, 3));

If this interleaving is executed, it fulfills the intended ordered concurrent-access-pair. Then, this generated path is added to the front of the worklist for further extension as shown at line 19 of Figure 5.8.

A similar target search and interleaving computation is carried out for the third case with the second write to $accX.val$ by $Thread2$ as the search target. The fulfillment of this ordered concurrent-access-pair results in one interleaving. So, when the processing of the value-schedule-relevant node of the read access to $accX.val$ from line 3 of $Thread1$ is done, three partial value-schedules and their fulfilling interleavings are generated.

In general, when a value-schedule-relevant access is encountered during the traversal, the derivation rules in Figure 5.10 are applied to derive and fulfill new value schedules. Similar to dynamic testing-techniques such as model checking, these two rules capture possible context-switch decisions that might be made by a dynamic testing-tool when an access to a shared variable is encountered during testing. Rule 1 corresponds to the dynamic testing-tool's decision to not yield for other threads, while Rule 2 corresponds to the decision to yield for other threads that may contain conflicting accesses. However, the main difference is that a traditional dynamic-testing-tool yields for threads while the value-schedule-based testing yields for specific accesses. This difference enables the

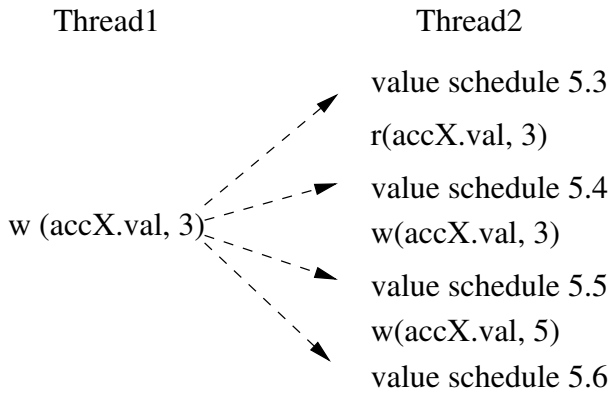


Figure 5.11: Processing of the Write Access at Line 3 of Thread1

accessibility-based POR and prioritized testing. In Figure 5.8, the code at lines 16 to 19 handles Rule 1, while the code at lines 21 to 25 handles Rule 2.

After the processing of a value-schedule-relevant access is completed, the interleaving is added back to the worklist. Interleavings are repeatedly removed from the front of the worklist for processing until the worklist is exhausted. Any further processing of an interleaving starts from the previous value-schedule-relevant access in the CFG and is extended to reach the next value-schedule-relevant access. Following this process, additional ordered concurrent-access-pairs are fulfilled and added to the partial value-schedule. When the algorithm determines no further extension is possible, the interleaving has fulfilled a complete value-schedule of the program. Then, the fulfilling interleavings of this value schedule are set aside for feasibility and correctness checking using the dynamic testing-tool.

Illustration of Derivation Process

To illustrate the task of deriving a complete value-schedule, assume an ordered concurrent-access-pair, (t1, run.(r(accX.val), line 3)) $\longrightarrow$ (t2, run.(w(accX.val), line 3)), has been added into a value schedule by applying Rule 1 on the read to $accX.val$ at line 3 of $Thread1$. After that, the exploration continues on $Thread1$ from the program point after t1.(r(accX.val, 3)). Then, the next value-schedule-relevant access encountered during the traversal is the write access to $accX.val$ from line 3 of $Thread1$, t1.(w(accX.val, 3)). From Table 5.1, this node forms concurrent access-pairs with three other accesses from $Thread2$.

These concurrent access-pairs imply four possible extensions to the partial-value schedule, in which the write to $accX.val$ from $Thread1$ and its concurrent accesses might happen in any of four orders shown in Figure 5.11. Like the previously discussed cases, the first ordering has the write from $Thread1$ happen before all of its possible concurrent

partners, which is easily fulfilled by directly appending the current CFG node to the existing interleaving. The second and third orderings have the read access of $accX.val$ from line 3, and the write accesses to $accX.val$ from line 3 by $Thread2$ happen before the write to $accX.val$ from $Thread1$, respectively. The fourth ordering specifies that case where the write to $accX.val$ happens after all conflicting accesses to $accX.val$ from $Thread2$. Fulfillment of these orderings is similar to that of value schedule 5.2. The four partial value-schedules and their fulfilling interleavings computed from processing the write access to $accX.val$ from line 3 of $Thread1$ are represented by partial value-schedules 5.3, 5.4, 5.5 and 5.6 below.

Value Schedule: (5.3)
(t1, run.(r(accX.val), line 3)) ⟶ (t2, run.(w(accX.val), line 3))
(t1, run.(w(accX.val), line 3)) ⟶ (t2, run.(r(accX.val), line 3));

Interleaving:
t1.(acq(accX, 2)) →
    t1.(r(accX.val, 3)) → t1.(w(accX.val, 3));

Value Schedule: (5.4)
(t1, run.(r(accX.val), line 3)) ⟶ (t2, run.(w(accX.val), line 3))
(t2, run.(r(accX.val), line 3)) ⟶ (t1, run.(w(accX.val), line 3));

Interleaving:
t1.(acq(accX, 2)) →
    t1.(r(accX.val, 3)) →
t2.(acq(accY, 2)) →
        t2.(r(accX.val, 3)) →
    t1.(w(accX.val, 3));

Value Schedule: (5.5)
(t1, run.(r(accX.val), line 3)) ⟶ (t2, run.(w(accX.val), line 3))
(t2, run.(w(accX.val), line 3) ⟶ (t1, run.(w(accX.val), line 3));

Interleaving:
t1.(acq(accX, 2)) →
    t1.(r(accX.val, 3)) →
t2.(acq(accY, 2)) →
        t2.(r(accX.val, 3)) → t2.(w(accX.val, 3)) →
    t1.(w(accX.val, 3));

Value Schedule: (5.6)
(t1, run.(r(accX.val), line 3)) ⟶ (t2, run.(w(accX.val), line 3))
(t2, run.(w(accX.val), line 5) ⟶ (t1, run.(w(accX.val), line 3));

Interleaving:
t1.(acq(accX, 2)) →
    t1.(r(accX.val, 3)) →
t2.(acq(accY, 2)) →
        t2.(r(accX.val, 3)) → t2.(w(accX.val, 3)) →
        t2.(cmp(accY.val, 4)) → t2.(w(accX.val, 5))→
    t1.(w(accX.val, 3));

Continuing the exploration of value schedule 5.3 further, the next value-schedule-relevant access encountered in the traversal of the CFG of $Thread1$ is the monitor entry of $accY$ at line 5. A direct insertion of this monitor entry into the value schedule and fulfilling interleaving ensures $Thread1$ always acquires the monitor of $accY$ before concurrent conflicting-monitor-entries from $Thread2$, and produces partial value-schedule 5.7, with a graphical representation of the fulfilling interleaving shown in Figure 5.12.

Value Schedule: (5.7)
(t1, run.(r(accX.val), line 3)) ⟶ (t2, run.(w(accX.val), line 3))
(t1, run.(w(accX.val), line 3) ⟶ (t2, run.(r(accX.val), line 3)))
(t1, run.(monenter, line 5)) ⟶ (t2, run.(monenter, line 2));

Interleaving:
t1.(acq(accX, 2)) →
    t1.(r(accX.val, 3)) → t1.(w(accX.val, 3)) →
t1.(rel(accX, 4))→
t1.(acq(accY, 5));

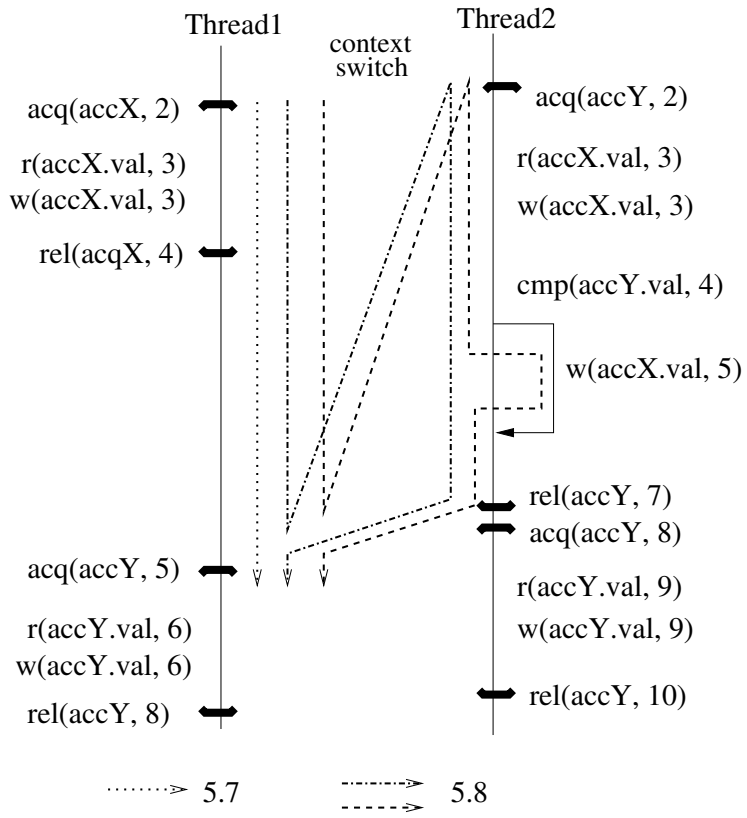


Figure 5.12: Value Schedules 5.7 and 5.8

The monitor acquisition at line 5 of $Thread1$ also forms concurrent access-pairs with monitor entries on the same object at line 2 and line 8 of $Thread2$ as shown in Table 5.1. For example, the partial value-schedule 5.8 extends the partial value-schedule 5.3 so that the monitor entry of *accY* at line 2 of $Thread2$ happens before that from $Thread1$ by applying Rule 2 from Figure 5.10 on the monitor entry at line 5 of $Thread1$. Using static analysis alone, it is difficult to determine which branch is actually taken at runtime by $Thread2$, so two execution paths are conservatively considered to fulfill the partial value-schedule. As shown at line 15 of Figure 5.8, the exploration of the CFG is carried over all possible successors of a node. Hence, the ordered concurrent-access-pair, (t2, run.(monenter, line 2)) $\xrightarrow{acq}$ (t1, run.(monenter, line 5)) can be decomposed into two interleavings for value schedule 5.8. A graphical illustration of the fulfilling interleavings of partial value-schedule 5.8 are shown in Figure 5.12.

Value Schedule: (5.8)
(t1, run.(r(accX.val), line 3)) $\longrightarrow$ (t2, run.(w(accX.val), line 3))
(t1, run.(w(accX.val), line 3)) $\longrightarrow$ (t2, run.(r(accX.val), line 3))
(t2, run.(monenter, line 2)) $\xrightarrow{acq}$ (t1, run.(monenter, line 5));

Interleaving 1:
t1.(acq(accX, 2)) →
    t1.(r(accX.val, 3)) → t1.(w(accX.val, 3)) →
t1.(rel(accX, 4)) →
t2.(acq(accY, 2)) →
        t2.(r(accX.val, 3)) → t2.(w(accX.val, 3)) →
        t2.(cmp(accY.val, 4)) → t2.(w(accX.val, 5)) →
t2.(rel(accY, 7))→
t1.(acq(accY, 5));

Interleaving 2:
t1.(acq(accX, 2)) →
    t1.(r(accX.val, 3)) → t1.(w(accX.val, 3)) →
t1.(rel(accX, 4)) →
t2.(acq(accY, 2)) →
        t2.(r(accX.val, 3)) → t2.(w(accX.val, 3)) → t2.(cmp(accY.val, 4)) →
t2.(rel(accY, 7))→
t1.(acq(accY, 5));

Now, consider a further extension of the partial value-schedule 5.4. The next value-schedule-relevant access encountered in the traversal of the CFG of $Thread1$ is also the monitor entry of *accY* at line 5 following the fulfilling interleaving given in value schedule 5.4. Again, as shown in Table 5.1, this monitor entry forms concurrent access-pairs

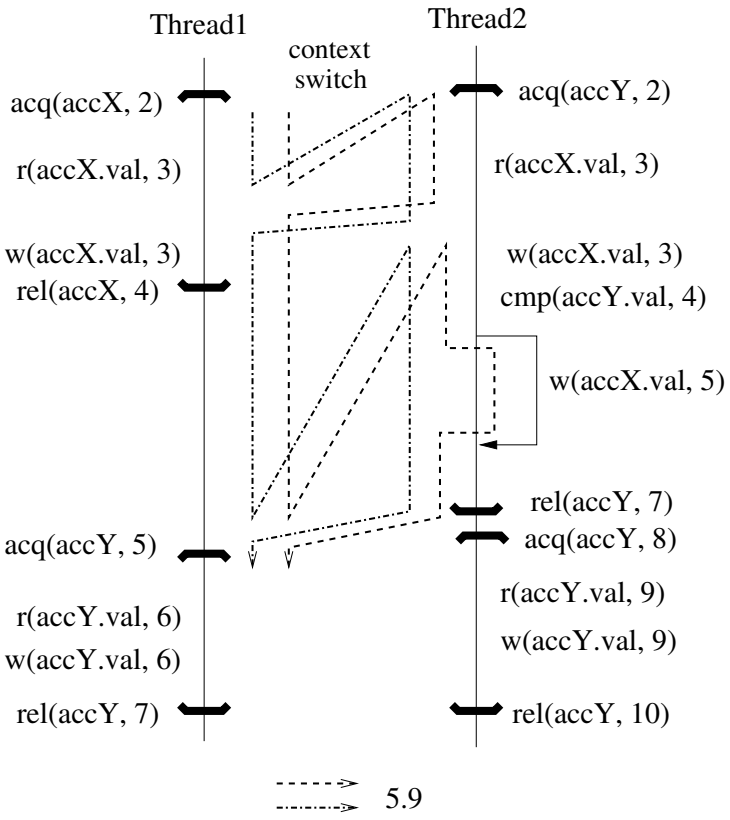


Figure 5.13: Value Schedule 5.9

with two other accesses from $Thread2$. The processing of this monitor entry node shows how different dependencies among ordered concurrent-access-pairs are resolved by the CFG traversal.

The direct insertion of the monitor acquisition at line 5 of $Thread1$ into the already-generated fulfilling interleaving of value schedule 5.4 extends the original partial value-schedule with another ordered concurrent-access-pair in which the monitor entry on *accY* at line 5 of $Thread1$ happens before the monitor entry to the same object at line 8 of $Thread2$. A closer inspection of the fulfilling interleaving of value schedule 5.4 shows that appending the monitor entry statement on *accY* from $Thread1$ to the existing fulfilling interleaving does not work because $Thread2$ still holds the monitor at line 3. Thus, an execution path that releases the monitor on *accY* must be computed first. However, the computation for the monitor exit-path also produces two fulfilling interleavings, because of the split in the control flow caused by the branch statement at line 4 of $Thread2$. Thus, adding the concurrent access-pair (t1, run.(monenter, line 5)) $\xrightarrow{acq}$ (t2, run.(monenter, line 8)) to value schedule produces the value schedule 5.9 with two possible fulfilling interleavings shown in Figure 5.13.

Value Schedule: (5.9)
(t1, run.(r(accX.val), line 3)) ⟶ (t2, run.(w(accX.val), line 3))
(t2, run.(r(accX.val), line 3)) ⟶ (t1, run.(w(accX.val), line 3))
(t1, run.(monenter, line 5)) ⟶ (t2, run.(monenter, line 8));

Interleaving 1:
t1.(acq(accX, 2)) →
t1.(r(accX.val, 3)) →
t2.(acq(accY, 2)) →
t2.(r(accX.val, 3))
t1.(w(accX.val, 3))→
t1.(rel(accX, 4))→
t2.(w(accX.val, 3)) → t2.(cmp(accY.val, 4)) → t2.(w(accX.val, 5)) →
t2.(rel(accY, 7)) →
t1.(acq(accY, 5));

Interleaving 2:
t1.(acq(accX, 2)) →
t1.(r(accX.val, 3)) →
t2.(acq(accY, 2)) →
t2.(r(accX.val, 3)) →
t1.(w(accX.val, 3))→
t1.(rel(accX, 4))→
t2.(w(accX.val, 3)) → t2.(cmp(accY.val, 4)) →
t2.(rel(accY, 7))→
t1.(acq(accY, 5));

Moreover, because the monitor acquisition from line 5 of $Thread1$ forms concurrent access-pairs with two monitor acquisitions on $accY$ from $Thread2$, the derivation algorithm applies Rule 2 from Figure 5.10 to fulfill additional ordered concurrent-access-pairs so that both the monitor acquisitions from $Thread2$ happen before that of $Thread1$. These two ordered concurrent-access-pairs are: (t2, run.(monenter, line 2)) $\xrightarrow{acq}$ (t1, run.(monenter, line 5)) and (t2, run.(monenter, line 8)) $\xrightarrow{acq}$ (t1, run.(monenter, line 5)). However, only the latter of these two ordered concurrent-access-pairs needs to be explicitly fulfilled at this point because the former has already been implicitly fulfilled by the previous fulfillment of (t2, run.(r(accX.val), line 3)) → (t1, run.(w(accX.val), line 3)) in partial value-schedule 5.4. In this fulfillment, the monitor acquisition of $accY$ from line 2 of $Thread2$ has been added to the fulfilling interleaving already. At this point, its execution order with respect to its concurrent conflicting-accesses is determined and happens before the conflicting monitor entry from $Thread1$. Thus, as the monitor acquisition on $accY$ is encountered following the partial value-schedule 5.4, it is unnecessary to fulfill the ordered concurrent-access-pair (t2, run.(monenter, line 2)) $\xrightarrow{acq}$ (t1, run.(monenter, line 5)). On the other hand, the ordered concurrent-access-pair (t2, run.(monenter, line 8)) $\xrightarrow{acq}$(t1, run.(monenter, line 5)), can be added to the partial value-schedule 5.4 because the monitor acquisition at line 8 of $Thread2$ has not yet been added to the fulfilling interleaving. Fulfillment of this ordered concurrent-access-pair forms the new value schedule 5.10 with two fulfilling interleavings due to the split in control flow from the branch as shown in Figure 5.14.

Value Schedule: (5.10)
(t1, run.(r(accX.val), line 3)) ⟶ (t2, run.(w(accX.val), line 3))
(t2, run.(r(accX.val), line 3)) ⟶ (t1, run.(w(accX.val), line 3))
(t2, run.(monenter, line 8)) $\xrightarrow{acq}$(t1, run.(monenter, line 5));

Interleaving 1:
t1.(acq(accX, 2)) →
t1.(r(accX.val, 3)) →
t2.(acq(accY, 2)) →
t2.(r(accX.val, 3)) →
t1.(w(accX.val, 3))→
t1.(rel(accX, 4)) →
t2.(w(accX.val, 3)) → t2.(cmp(accY.val, 4)) → t2.(w(accX.val, 5)) →
t2.(rel(accY, 7))→
t2.(acq(accY, 8)) →
t2.(r(accX.val, 9)) → t2.(w(accX.val, 9)) →
t2.(rel(accY, 10)) →
t1.(acq(accY, 5));

Interleaving 2:
t1.(acq(accX, 2)) →
t1.(r(accX.val, 3)) →
t2.(acq(accY, 2)) →
t2.(r(accX.val, 3)) →
t1.(w(accX.val, 3))→
t1.(rel(accX, 4)) →
t2.(w(accX.val, 3)) → t2.(cmp(accY.val, 4)) →
t2.(rel(accY, 7)) →
t2.(acq(accY, 8)) →
t2.(r(accX.val, 9)) → t2.(w(accX.val, 9)) →
t2.(rel(accY, 10))→
t1.(acq(accY, 5));

Further processing of the partial value-schedules 5.5 and 5.6 is done in a similar fashion as that of value schedule 5.4. For example, the value schedule and fulfilling interleavings derived from the direct insertion of the monitor entry for *accY* in $Thread1$ to partial value schedule 5.6 is shown as value schedule 5.11, and the value schedule derived from deferring this monitor entry for that at line 8 in $Thread2$ is shown as value schedule 5.12. The fulfilling interleavings for these two value schedules are computed in the same way as those presented so far and are not written out, but they are shown graphically in Figure 5.15.

Value Schedule: (5.11)
(t1, run.(r(accX.val), line 3)) ⟶ (t2, run.(w(accX.val), line 3))
(t2, run.(w(accX.val), line 5)) ⟶ (t1, run.(w(accX.val), line 3))
(t1, run.(monenter, line 5)) $\xrightarrow{acq}$ (t2, run.(monenter, line 8));

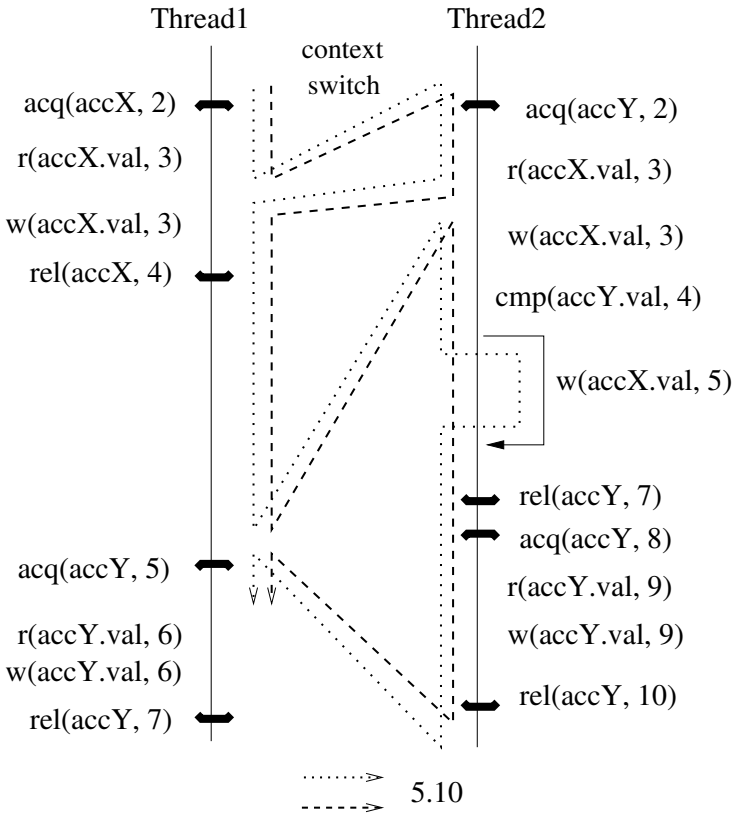


Figure 5.14: Value Schedule 5.10

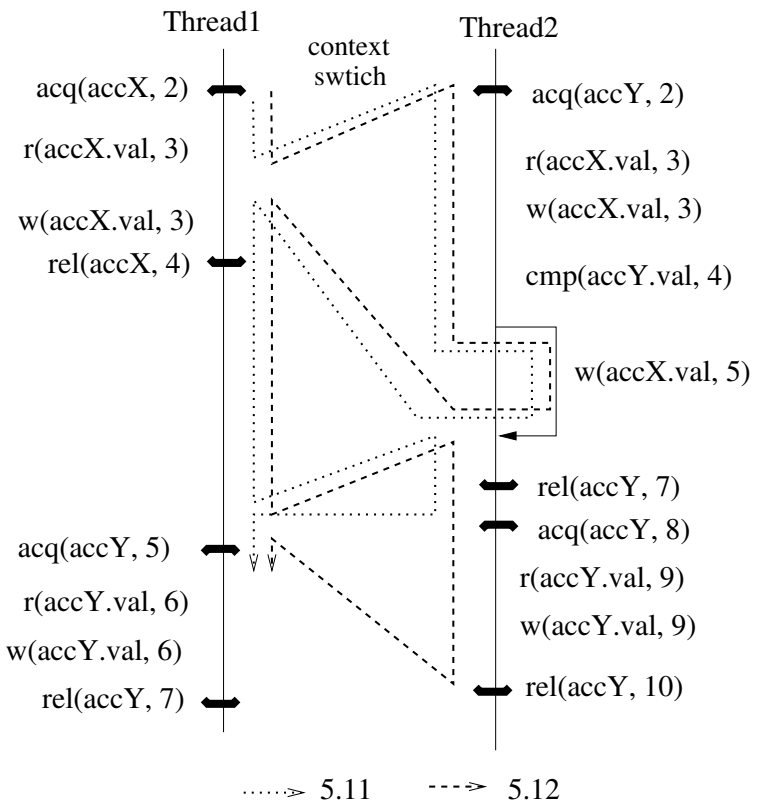


Figure 5.15: Value Schedule 5.11 and 5.12

Value Schedule: (5.12)
(t1, run.(r(accX.val), line 3)) $\longrightarrow$ (t2, run.(w(accX.val), line 3))
(t2, run.(w(accX.val), line 5)) $\longrightarrow$ (t1, run.(w(accX.val), line 3))
(t2, run.(monenter, line 8)) $\stackrel{acq}{\longrightarrow}$ (t1, run.(monenter, line 5));

Further exploration of the CFG of $Thread1$ following the fulfilling interleavings for value schedules 5.7, 5.8, 5.9, 5.10, 5.11 and 5.12 lead the traversal to the end of the $run()$ method without encountering any new value-schedule-relevant CFG accesses. When the CFG of $Thread1$ is exhausted, the remaining nodes from the CFG of $Thread2$ do not trigger any additional concurrent accesses. This termination is because the concurrent access partners are all from $Thread1$ and have been incorporated into the fulfilling interleavings of the already-generated value schedules. At this point, the fulfilling interleaving is not added back to the worklist, but stored for dynamic testing. The generation of value schedules and their fulfilling interleavings continues until the worklist becomes empty. The conditional statement at line 12 of Figure 5.8 corresponds to detecting the end of exploration condition for a value schedule.

Once the worklist becomes exhausted, a set of potential fulfilling interleavings of the possible value schedules of the program is generated. A fulfilling interleaving of a value

schedule follows the ordering of concurrent accesses dictated by the ordered concurrent-access-pairs of the value schedule and the sequential dependencies among accesses dictated by the CFGs of concurrently running threads. Despite observing these constraints, a fulfilling interleaving of a value schedule may still be an infeasible path, because the static analysis cannot determine dynamic dependencies, like the exact branch a program takes at runtime. Then, the derivation algorithm conservatively assumes all nodes along every branch are accessible. Thus, it is possible that a derived value-schedule may contain an ordered concurrent-access-pair that involves inaccessible accesses from a branch that is not executed at runtime. In the case that all fulfilling interleavings of a value schedule do not reflect the control flow taken at runtime, that value schedule is infeasible. Finally, note that the feasibility property of a computed value schedule is defined in the context of a fixed set of program inputs. A value schedule that is infeasible under one set of program input might be feasible in another.

Feasibility and Correctness Checking

After value schedules and their fulfilling interleavings are computed, testing enters the dynamic stage. This stage has two main goals. First, it tries to determine whether a computed value schedule is feasible in a concrete execution. Second, if a value schedule is feasible, then it is important to know whether a program execution that satisfies this value schedule may expose a harmful concurrency bug. Through the feasibility checking of value schedules, dynamic testing can identify superfluous interleavings spawned to fulfill an infeasible ordered concurrent-access-pair. The process of checking the feasibility and correctness properties is illustrated using the value schedules 5.7-5.12 derived in the previous section. The code for the dynamic testing corresponds to line 28 in Figure 5.8.

It is important to note that the interleavings presented in this section only show instructions that operate on shared variables. The instructions that operate on local variables and the program stack are removed for clarity. In practice, every bytecode of a Java class is represented by *Soot* as a CFG node in Baf intermediate representation. Thus, the fulfilling interleaving computed from the CFG traversal includes all of the bytecodes encountered and is ready for execution by a dynamic testing-tool.

During the dynamic testing, the computed fulfilling interleavings of each value schedule are fed into the dynamic testing-tool for execution. From Chapter 2, the dynamic testing-tool used in the implementation of value-schedule-based testing is based on JPF. It explores the program state-space of a concurrent program by treating the execution of a bytecode in a Java program as a transition in the program state-space, and an execution of an instruction is yielded for other co-enabled instructions if it operates on an object that is reachable from the heap by more than one thread. Because the dynamic testing-tool actually executes each bytecode in the Java program, it is possible

to take advantage of this mechanism to instruct the dynamic testing-tool to execute the program in the order specified by a given interleaving. For example, among co-enabled transitions, the dynamic testing-tool always selects the transition that corresponds to the next-to-execute instruction in the fulfilling interleaving. If the next-to-execute instruction specified by the fulfilling interleaving does not appear as a co-enabled transition in the dynamic testing-tool, then this fulfilling interleaving cannot be used to fulfill the expected value schedule.

In the case of testing the fulfilling interleaving for value schedule 5.7, the dynamic testing-tool always selects the next-to-explore transition from $Thread1$ until all instructions in the fulfilling interleaving are executed. Because this interleaving is generated by traversing the CFG of $Thread1$, which contains no branch statement, every instruction specified by the fulfilling interleaving appears as a co-enabled transition when needed. After the fulfilling interleaving is executed, the dynamic testing-tool is allowed to run the rest of program freely to the end. The execution of the interleaving for value schedule 5.7 shows that it is feasible and produces no error or warning message.

Value schedule 5.8 has two fulfilling interleavings, each following a different branch for the branch statement at line 4 of $Thread2$. Assume the fulfilling interleaving that follows the true branch is checked first. The controlled-execution starts from $Thread1$ until the monitor entry $accY$ appears as the next-to-execute transition of $Thread1$. Then, execution starts executing instructions from $Thread2$ as specified by the fulfilling interleaving. When the instruction corresponding to the branch statement is executed, the next-to-execute instruction specified by the fulfilling interleaving is the write to $accX.val$ at line 5 of $Thread2$. However, this expected instruction does not appear as the next-to-execute instruction in $Thread2$. Therefore, this fulfilling interleaving is infeasible. Then, the other fulfilling interleaving of this value schedule that takes the false branch is tested. The execution of the second interleaving is successful, such that every instruction specified by the fulfilling interleaving appears as expected. Therefore, value schedule 5.8 is feasible, and the execution produces no error.

Value schedule 5.9 also has two fulfilling interleavings because of the branch statement at line 4 of $Thread2$. The execution of the fulfilling interleaving that follows the true branch is infeasible for the same reason as for value schedule 5.8. So only the value schedule for the false branch is feasible. After the fulfilling interleaving is completely executed, the execution of the rest of the program causes the assertion at line 11 to evaluate to false because, for in this interleaving $accX.val$ has the value 10 and $accY.val$ has the value 0. The successful execution of this fulfilling interleaving presents a feasible value schedule that brings the program execution into an undesirable state. The code at lines 31 and 32 of Figure 5.8 capture the handling of such a situation.

Both value schedules 5.11 and 5.12 have only one fulfilling interleaving. Both interleavings require the write to $accX.val$ enclosed by the branch statement of $Thread2$ to

be executed, and the value written by the enclosed write retrieved by the read access from $Thread1$. However, as the dynamic testing-tool executes the fulfilling interleavings up to the instruction corresponding to the branch statement, it determines that this interleaving is infeasible because the concrete execution takes a different branch. Because both value schedules have only one fulfilling interleaving, these two value schedules are infeasible.

Once an infeasible value schedule is identified, all other untested value schedules which share the same infeasible partial-value schedule are pruned out. For example, assume the value schedule 5.11 is tested first and shown to be infeasible. Because value schedule 5.12 shares the same infeasible partial value-schedule with value schedule 5.11, value schedule 5.12 can be immediately considered as infeasible without additional testing. The code at lines 29 and 30 of Figure 5.8 captures the infeasible value schedules and performs pruning.

In summary, value-schedule-based testing identifies infeasible value-schedules by deriving and testing their fulfilling interleavings. An infeasible value-schedule contains at least one infeasible ordering among concurrent conflicting-accesses. In other words, the fulfilling interleaving of an infeasible value-schedule contains a permutation that does not lead to the testing of a new ordering among concurrent-accesses. Value-schedule-based testing is able to identify such an interleaving and abort any further testing along this interleaving.

5.3.2 Features of Value-Schedule-based Testing

In previous sections, the value schedules are computed and tested using the value-schedule-based testing. In this section, the two main features of value-schedule-based testing, accessibility-based POR and prioritized testing, are incorporated into the two-stage testing process.

Accessibility-based POR

As discussed in Chapter 2, the model checker yields the execution of an access if it believes there are conflicting concurrent accesses from other threads. However, these conflicting accesses may not always occur at runtime. Accessibility-based POR is designed to recognize and prune out interleavings that are derived because of the incorrect assumptions about the availability of conflicting accesses to shared variables at runtime. The goal is to reduce the computational cost of testing.

Consider the following partial interleaving, derived by a model-checker-based dynamic testing-tool, to test the ordering such that the write to $accX.val$ at line 3 of $Thread2$ happens before the read of $accX.val$ from line 3 of $Thread1$:

t1.(acq(accX, 2)) →
t2.(acq(accY, 2)) →
t2.(r(accX.val, 3)) → t2.(w(accX.val, 3)) →
t1.(r(accX.val, 3));

After following this partial interleaving, the dynamic testing-tool splits the exploration of the current interleaving into two interleavings when the write access, w(accX.val, 3) in $Thread1$ is encountered. Figure 5.16 displays the subsequent interleavings starting at t1.(r(accX.val, 3)) above. One of the two interleavings continues executing instructions from $Thread1$, while the other interleaving yields the execution of $Thread1$ before the write for $Thread2$. In the former case, the dynamic testing-tool triggers another two new interleavings at the monitor acquisition, acq(accY, 5) for $Thread1$ to test the race on the monitor acquisition of $accY$. These two interleavings are denoted as $d1$ and $d2$. In the latter case, the dynamic testing-tool yields, t1.w(accX.val, 3), because the remaining execution of $Thread2$ contains conflicting concurrent accesses to $accX.val$, such as the write at line 5 of $Thread2$. However, as shown from the testing of fulfilling interleaving 5.11 in the previous section, the continued execution of the interleaving along $Thread2$ does not lead to any access to $accX.val$ in $Thread2$. Thus, this yielding produces a superfluous interleaving annotated with $p1$ in Figure 5.16. Moreover, this superfluous interleaving leads the program into the same state as $d2$.

With the accessibility-based-POR, the value-schedule-based testing tool is able to recognize the interleaving spawned from yielding at t1.w(accX.val, 3) cannot lead to any access on $accX.val$ in $Thread2$ as soon as the branch statement at line 4 of $Thread2$ is executed. Then, the execution of the rest of this superfluous interleaving is abandoned. Moreover, by pruning out all value schedules that are spawned based on the same assumption, the accessibility-based-POR can achieve better savings in computational resource. This saving is illustrated by introducing a new thread, $Thread3$, into the sample program. $Thread3$ contains a simple synchronized clause on $accY$:

Thread3

```
1  run(){
2      synchronized(accY){
3      ...
4      }
5  }
```

As before, the dynamic testing-tool spawns an interleaving to look for a conflicting access to $accX.val$ in $Thread2$, even though the access is never reached by $Thread2$.

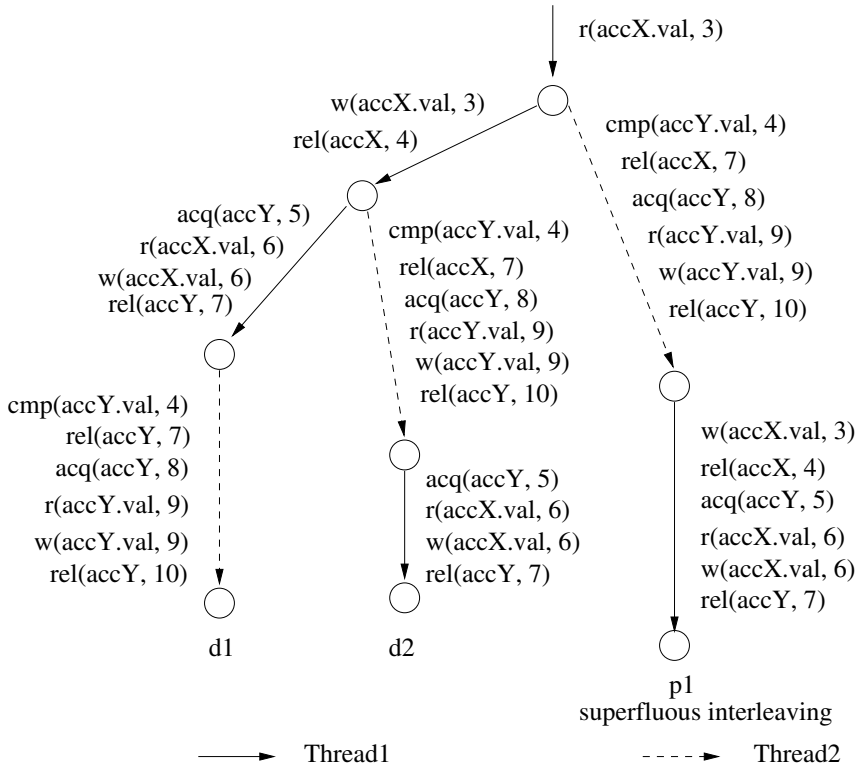


Figure 5.16: Testing without Accessability POR

When the tool reaches the monitor entry to *accY* at line 8 of $Thread2$, it tries to spawn a new interleaving to permute the monitor acquisitions on *accY* between $Thread2$ and $Thread3$ as shown in the dashed cloud of Figure 5.17. Thus, the original superfluous assumption on the availability of *accX.val* on $Thread2$ produces another new superfluous interleaving. The number of superfluous interleaving increases further as more accesses to shared variables are encountered during the execution of those superfluous interleavings. This problem is especially serious for model checkers that record various program states during execution for backtracking. As the number of superfluous interleavings increases, the amount of testing time and the memory costs spent on irrelevant interleavings increases. As shown in Figure 5.17, extensions along the interleavings in the cloud are always superfluous because the original interleaving spawning them is already superfluous. However, with the support of accessibility-based-POR, the value-schedule-based testing stops the execution as the original superfluous interleaving and prunes out all value schedules whose fulfilling interleavings share the same prefix of the identified superfluous interleaving; hence, no additional superfluous interleavings are executed.

To summarize, the value-schedule-based testing provides accessibility-based POR through the dynamic execution of the fulfilling interleaving. It ensures that the permutations based on imprecise static information, such as the control flow, are actively identified. As a result, derived interleavings that fulfill an ordered concurrent-access-pair

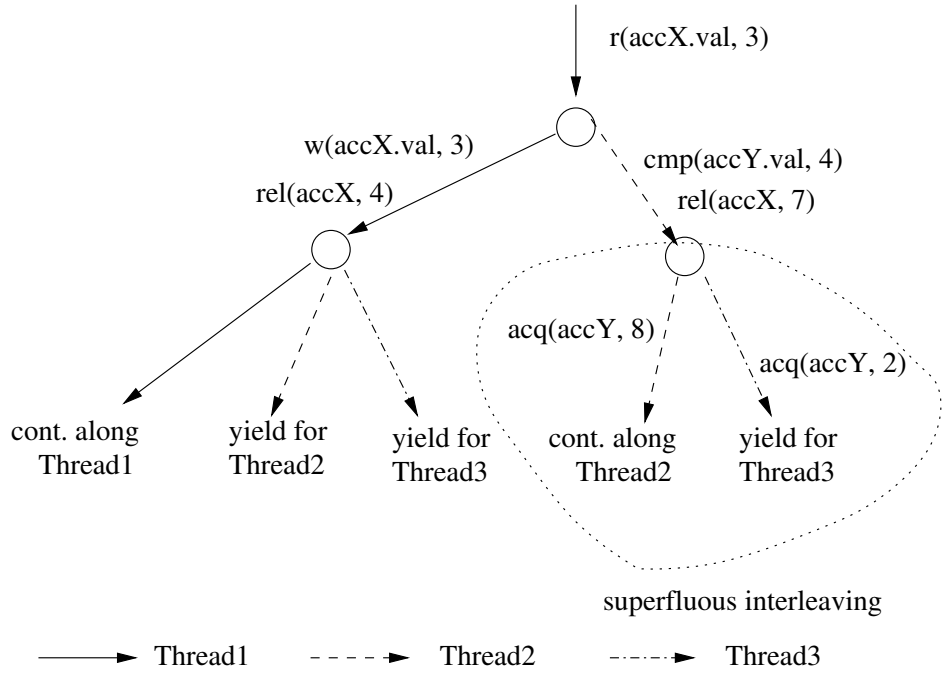


Figure 5.17: Testing Without Accessability POR

that contains an inaccessible access are not run and tested.

Prioritized Testing

It is now possible to illustrate the implementation of prioritized testing of permutations concerning a particular value-schedule-relevant access. In practice, due to the limitations on computational resources and time, exhaustive testing of all value schedules is infeasible. The goal of the prioritized testing is to prioritize the evaluation of a selected set of ordered concurrent access-pairs before others. For example, if a developer suspects a particular read access might read in a value given by multiple concurrent writes, the value-schedule-based testing offers a mechanism to prioritize the testing of the concurrent accesses for that variable during the testing process.

Given an access of interest, the first step of the value-schedule-based testing is to identify a set of value schedules in which the effect of different ordered concurrent-access-pairs related to the access of interest is evaluated. An ordered concurrent-access-pair is related to an access of interest if 1) the access of interest is a member of the pair or 2) this ordered concurrent-access-pair is formed from the direct insertion of an access that may form some concurrent access-pairs with the access of interest. The first criteria deals with the situation when the execution order of the access of interest is explicitly fulfilled. The second criteria deals with the situation when the execution order between the access of interest and one of its conflicting accesses is implied by the dependencies among existing

ordered concurrent-accesses in the partial value-schedule.

The second step is to select a set of value schedules that is sufficient to evaluate these concurrent access-pairs related to the access of interest for prioritized testing. The derivation of value schedules are carried out using the same approach introduced in previous sections. As the access of interest is inserted into the fulfilling interleaving of a partial value-schedule, the value-schedule-based testing applies a marker to the value schedule extended from this partial value-schedule such that only Rule 1 (direct insertion) from Figure 5.10 is applied on all subsequently-encountered value-schedule-relevant accesses. It is straightforward to see that once an access of interest is inserted into a fulfilling interleaving, the runtime execution orders between this access and its conflicting accesses have been determined. Execution of such value schedules are sufficient to show the feasibility and correctness properties of different ordered concurrent-access-pairs related to an access of interest. Moreover, value schedules with additional ordered concurrent-access-pairs whose fulfillment require context switches after the access of interest are excluded. Therefore, the number of value schedules that need to be prioritized in testing is significantly narrowed down.

For the sample program in Figure 5.1, if the access of interest is the write to $accX.val$ at line 3 for $Thread1$, then, value schedules 5.7 and 5.11 are marked and tested first because their fulfilling interleaving includes no additional fulfillment of other ordered concurrent-access-pairs requiring the direct insertion of encountered value-schedule-relevant accesses after the access of interest is added to the fulfilling interleaving. The execution of the fulfilling interleavings of these value schedules triggers the assertion and exposes a concurrency bug.

5.3.3 Recursive Derivation

Previous discussion focused on the computation of value schedules and their fulfilling interleavings for programs where the accesses to shared variables happen in two threads. The implication is that when a fulfilling interleaving of an ordered concurrent-access-pair is being computed, the CFG traversal does not encounter a value-schedule-relevant access that may trigger the fulfillment of an ordered concurrent-access-pair where this new ordered concurrent-access-pair requires the traversal of the CFG of a different thread.

The following shows how a value schedule involving concurrent accesses from more than two threads can be derived and fulfilled. Moreover, it shows how the recursive derivation and fulfillment of ordered concurrent-access-pairs may help to identify feasible fulfilling-interleavings for value schedules. The sample program in Figure 5.1 is augmented with a new thread, $Thread3$, having an unprotected write-access to $accY.val$ as shown in Figure 5.18. This unprotected access to $accY.val$ contributes additional concurrent access-pairs to the program. For example, the read access on $accY.val$ at line 4 of $Thread2$ forms

Main Thread

```
1  main(String[] args){
2    Account accX = new Account(1);
3    Account accY = new Account(2);
4    Thread1 t1 =
5      new Thread1(accX, accY);
6    Thread2 t2 =
7      new Thread2(accX, accY);
8    Thread2 t3 =
9      new Thread3(accX, accY);
10   t1.start(); t2.start(); t3.start();
11   t1.join(); t2.join(); t3.join();
12
13   assert(accX.val == accY.val);
14  }
15
16 class Account{
17     Account(int id){
18        this.id = id;
19        this.value = 0;
20     }
21     int val;
22     int id;
23     public static int amt = 10;
24 }
```

Thread 1

```
1  run(){
2    synchronized(accX){
3      accX.val += amt;
4    }
5    synchronized(accY){
6      accY.val += amt;
7    }
8  }
```

Thread 2

```
1   run(){
2    synchronized(accY){
3     accX.val -= amt;
4     if (accY.val < 0){
5          accX.val = 0;
6     }
7    }
8    synchronized(accY){
9       accY.val -= amt;
10   }
11  }
```

Thread3

```
1  run(){
2      accY.val = -999;
3  }
```

Figure 5.18: Extended Sample Program

a concurrent access-pair with the write access from $Thread3$.

As shown in the previous sections, the partial value-schedule 5.6 and its extended value-schedule 5.12, are infeasible for the sample program in Figure 5.1 because the write access to *accX.val* at line 5 of $Thread2$ is always inaccessible following its fulfilling interleavings. Moreover, because the branch at line 4 never takes the true branch, all derived value schedules including the ordered concurrent-access-pair, (t2, run.(w(accX.val), line 5)) $\longrightarrow$ (t1, run.(w(accX.val), line 3)) are infeasible. However, the introduction of the new write access to *accY.val* in $Thread3$ produces a new value schedule in which the ordered concurrent-access-pair (t3, run.(w(accY.val), line 2)) $\stackrel{def}{\longrightarrow}$ (t2, run.(r(accY.val), line 4)) is fulfilled. Moreover, the fulfillment of this new ordered concurrent-access-pair leads to the derivation of a new value schedule in which the ordered concurrent access-pair (t2, run.(w(accX.val), line 5)) $\longrightarrow$ (t1, run.(w(accX.val), line 3)) is feasible. The derivation of such a value schedule is used to illustrate the recursive expansion of value schedules that involve concurrent accesses to shared variables from more than two threads.

Assume the CFG traversal has already generated a fulfilling interleaving that fulfills the ordered concurrent access-pair.

(t2, run.(w(accX.val), line 3)) $\stackrel{def}{\longrightarrow}$ (t1, run.(r(accX.val), line 3))
with the interleaving,

t1.(acq(accX, 2)) $\rightarrow$
t2.(acq(accY, 2)) $\rightarrow$
t2.(r(accX.val, 3)) $\rightarrow$ t2.(w(accX.val, 3)) $\rightarrow$
t1.(r(accX.val, 3))

As the traversal encounters the write to *accX.val* at line 3 of $Thread1$, the fulfillment process for the ordered concurrent-access-pair ((t2, run.(w(accX.val), line 5)) $\longrightarrow$ (t1, run.(w(accX.val), line 3)) is triggered. Then, the traversal switches to $Thread2$ for the write access on *accX.val*. As the read of *accY.val* is encountered at line 4 of $Thread2$, the derivation process recognizes this read access is value-schedule-relevant because it is part of the concurrent access-pair (t2, run.(r(accY.val), line 4))$\longrightarrow$ (t3, run.(w(accY.val), line 2)). This concurrent access-pair implies that an ordered current-access-pair on *accY.val* might take place while $Thread1$ waits to write to *accX.val* at line 3 and before the write of *accX.val* at line 5 of $Thread2$ is executed.

The processing of this read access to *accY.val* at line 4 of $Thread2$ produces two partial value-schedules. In the first case, the ordered concurrent-access-pair (t2, run.(r(accY.val), line 4)) $\rightarrow$ (t3, run.(w(accY.val), line 2)) is fulfilled by adding the read access directly to the fulfilling interleaving and continuing the search for the write to *accX.val* along the CFG of $Thread2$. In the second case, the partial value-schedule is extended with the ordered concurrent-access-pair (t3, run.(w(accY.val), line 2)) $\stackrel{def}{\longrightarrow}$ (t2, run.(r(accY.val),

line 4)). If the partial value schedule is extended with the first case, the value of *accY.val* retrieved at the branch statement leads to the false branch at runtime. Therefore, no fulfilling interleaving extended from this interleaving is feasible. If the partial value-schedule is extended with the second case, the traversal of the CFG of *Thread*3 first computes a path to the write of *accY.val* from *Thread*3. After that, the traversal restarts *Thread*2 to complete the fulfillment of the original ordered concurrent-access-pair. The value given by the write to *accY.val* from *Thread*3 means the program takes the true branch for the conditional branch at line 4 of *Thread*2. The second partial value-schedule and its fulfilling interleaving for the second case are shown in value schedule 5.13.

Value Schedules: (5.13)
(t2, run.(w(accX.val), line 3)) $\xrightarrow{def}$ (t1, run.(r(accX.val), line 3))
(t3, run.(w(accY.val), line 2)) $\xrightarrow{def}$ (t2, run.(r(accY.val), line 4))
(t2, run.(w(accX.val), line 5)) $\longrightarrow$ (t1, run.(w(accX.val), line 3));

Interleaving:
t1.(acq(accX, 2)) →
t2.(acq(accY, 2)) →
t2.(r(accX.val, 3)) → t2.(w(accX.val, 3)) →
t1.(r(accX.val, 3)) →
t3.(w(accY.val, 2)) →
t2.(cmp(accY.val, 4))→
t2.(w(accX.val, 5))→
t1.(w(accX.val, 3));

Figure 5.19 illustrates the implementation of the recursive value schedule generation using interleaving 5.13. The implementation keeps track of the ordered concurrent accesses triggered during the fulfillment process using a stack. As an ordered concurrent-access-pair is set to be fulfilled, the desired target access is pushed onto a stack referred as the fulfilling stack. For example, as shown in Figure 5.19, the fulfillment of (t2, run.(w(accX.val), line 3)) $\xrightarrow{def}$ (t1, run.(r(accX.val), line 3)), pushes w(accX.val, 3) from *Thread*2 onto the fulfilling stack (box 1). Line 24 of the algorithm in Figure 5.8 captures such an update of the stack for each new search target. When the search target is found, the stack is popped (box 2). This operation is shown at line 18 of Figure 5.8. As the traversal on *Thread*1 continues, the fulfillment process for the ordered concurrent-access-pair (t2, run.(w(accX.val), line 5)) $\longrightarrow$ (t1, run.(w(accX.val), line 3)) pushes another search target, w(accX.val, 5) from *Thread*2, onto the fulfilling stack (box 3). As the read of *accY.val* at line 4 is encountered, the fulfillment of the ordered concurrent-access-pair (t3, run.(w(accY.val), line 2)) $\xrightarrow{def}$ (t2, run.(r(accY.val), line 4)) is recursively triggered, while the previous ordered concurrent-access-pair remains unfulfilled. This new fulfill-

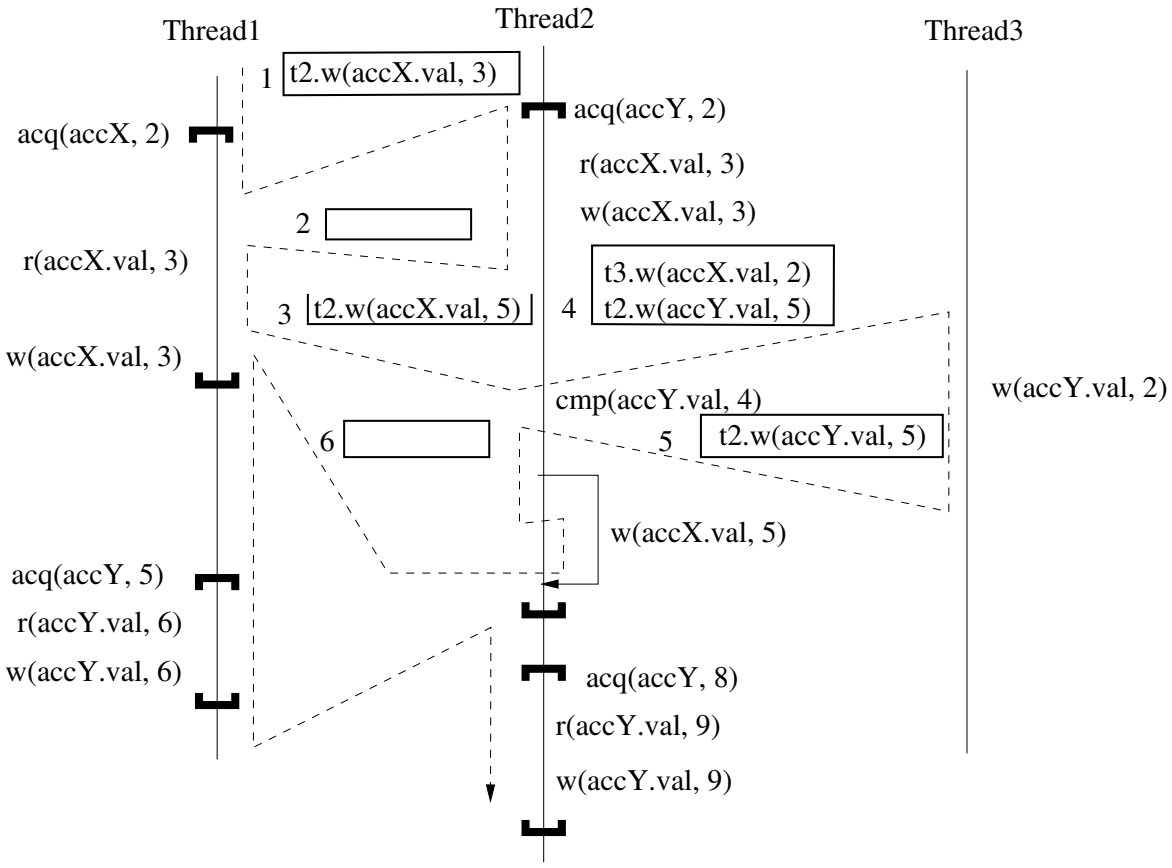


Figure 5.19: Recursive Expansion of Value Schedules

ment process pushes the search target, w(accY.val, 2) from *Thread*3, onto the stack (box 4) and switches to *Thread*3. Now, the fulfilling stack has two elements. After the write to *accY.val* by *Thread*3, the stack is popped (box 5). Then, the algorithm starts searching for the target access stored at the top of the stack, which is w(accX.val, 5) from *Thread*2. As the write access is located, the stack is popped again and becomes empty(box 6). After that, the CFG traversal switches to *Thread*1.

Continuing the traversal of the CFG of *Thread*1 generates value schedules that cover the permutation of monitor entries on *accY* from *Thread*1 and *Thread*2 as discussed in the previous section. The only difference is that each fulfilling interleaving is extended until both *Thread*1 and *Thread*2 exhaust all nodes in their CFGs.

5.3.4 Detailed Derivation and Fulfilling Rules

Previous discussion concerned the general rules for deriving and fulfilling different ordered concurrent-access-pairs. This section presents detailed rules concerning the derivation and fulfillment of different types of ordered concurrent-access-pairs in the context of the Java programming-language.

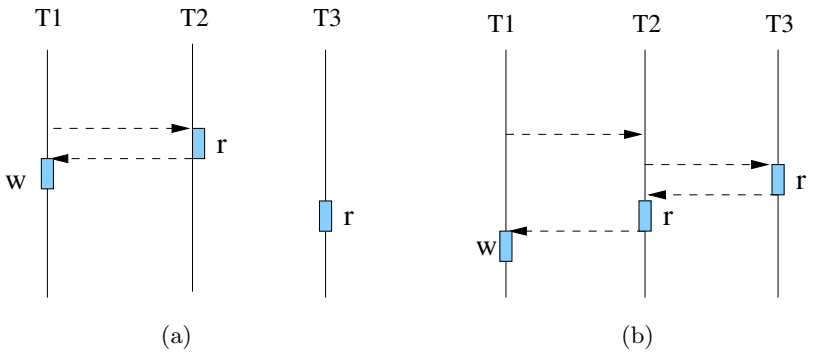


Figure 5.20: Deriving Value Schedules over Concurrent Reads

RW Pairs

The derivation of ordered concurrent-access-pairs for value-schedule-relevant read or write accesses is largely the same as those defined by the general derivation-rules. The exception happens when processing a value-schedule-relevant write access that has more than one concurrent read from more than one other thread.

As a value-schedule-relevant read or write access is encountered, the two derivation rules are always applied to compute interleavings that capture different orderings among the current access and each of its concurrent conflicting-accesses. As discussed before, the potential concurrent conflicting-accesses of an access are those with which it can form a concurrent access-pair. Because a read access does not form a concurrent access-pair with another read access, no new ordered concurrent access-pair is derived for two reads of the same shared variable. This is equivalent to stating that permuting the execution orders of two concurrent read accesses of the same shared variable does not lead the program into a new state, and therefore, all such accesses are ignored.

The following discusses how the recursive-derivation algorithm handles the permutations among write accesses with different sets of its concurrent conflicting-read-accesses. Without this, the derivation algorithm misses many feasible value schedules based on the discussed recursive derivation algorithm. For instance, as shown in Figure 5.20, assume a write by $T1$ triggers the traversal of $T2$ to have a read of $T2$ happen before the write from $T1$. As the read from $T2$ is encountered, the derivation and fulfillment algorithm determines it forms no concurrent access-pair with other accesses besides that from $T1$, as shown in Figure 5.20(a). However, without the recursive derivation on the read from $T1$ for the read from $T3$, the derivation rules miss the ordering where both reads from $T2$ and $T3$ happen before the write from $T1$ as shown in Figure 5.20(b). Hence, $T3$ could either read the value before or after the write by $T1$.

A simple observation shows that it is not the permutations among concurrent reads that lead to new value schedules, but rather it is the distinct sets of concurrent reads that

happen before and after a write access that leads to the derivation of new value schedules. For n concurrent reads, the number of distinct subsets is given by $\sum_{k=0}^{n} \binom{n}{k} = 2^n$. For example, if both the reads from $T2$ and $T3$ happen before the write from $T1$, permuting the execution order of those two reads does not produce any new value schedule. The number of distinct subsets of concurrent reads is $\sum_{k=0}^{2} \binom{n}{k} = 2^2 = 4$. This result correctly gives the number of relevant orderings among these three accesses. Those relevant ordering are: only the read from $T2$ misses the value given by the write; only the read from $T3$ misses the value given by the write; both reads miss the value given by the write; and both reads retrieve the value given by the write.

Thus, a write access has to permute not only with each individual concurrent conflicting write but also different subsets of its concurrent conflicting reads. This task consists of two steps. First, the derivation algorithm has to dynamically compute all concurrent reads for an encountered write access because concurrent read-pairs are not collected during the computation of concurrent access-pairs. This computation is done as follows. Because an encountered read is value-schedule-relevant, it has to form concurrent access-pairs with some concurrent write. By combining all reads that form concurrent access-pairs with the same write into a set, the derivation algorithm can compute concurrent reads for a read access on the fly. Second, all possible subsets of concurrent conflicting-reads are collected using the *combinadics* technique from [38]. Basically, each concurrent read is tagged with a unique lexical identifier. During the derivation, if a value-schedule-relevant read access is needed for the recursive derivation because of a conflicting write already in the fulfilling stack, then that read only yields for concurrent reads from threads with higher lexical-order. This ensures that every distinct subset is derived exactly once. In the end, all partial value-schedules, where a write access yields for k concurrent reads, are collected by applying the direct insertion after the k^{th} recursive expansion on the concurrent conflicting-reads from the write access.

MM Pairs

The derivation of ordered concurrent-access-pairs for a value-schedule-relevant monitor entry is exactly the same as the general derivation-rules. Two new ordered concurrent-accesses are derived: one acquires the monitor immediately, while the other yields for the other concurrent monitor-entry.

The fulfillment of an ordered concurrent-access-pair of *MM* type requires extra operations over a *RW* pair. Assume a concurrent monitor-entry is located in target thread T_j and has been added to the fulfilling interleaving. The CFG traversal cannot resume immediately on the thread T_i that yielded to T_j because thread T_j is still holding the monitor and T_i tries to acquire the monitor in its next instruction. The traversal can continue on T_i only after T_j has released the monitor. Therefore, after the target concurrent monitor

entry is found by T_j, the traversal needs to continue on T_j until a monitor release on the same monitor object is reached. A monitor release can either be a *monexit* bytecode or an invocation of *wait*() the monitor. As shown by the fulfilling interleaving for value schedule 5.10 for the sample program in Figure 5.1, the ordered concurrent-access-pair, (t2, run.(monenter, line 8)) $\xrightarrow{acq}$ (t1, run.(monenter, line 5)), is fulfilled by instructions leading to the monitor entry at line 5 of $Thread1$, the instructions in the synchronized block starting at line 8 of $Thread2$, and the release of the monitor by $Thread2$ at line 10.

In Java, a thread releases a monitor after the same number of monitor entry and exit instructions have been encountered for a given monitor object or when the *wait*() method on the same monitor is invoked. The former case deals with the reentrant locking mechanism in Java. That is, a thread is allowed to repeatedly acquire a monitor that it already holds, and the monitor is released after the same number of monitor exits occurs. Thus, the fulfilling interleaving always keeps track of the number of times a thread acquires and releases a monitor. An ordered concurrent-access-pair of type *MM* is fulfilled when the number of times a monitor is released equals the number of times the same monitor is acquired in the fulfilling interleaving. For the latter case, the fulfilling interleaving considers the monitor is released by its owner when the *wait*() method on the same monitor is encountered.

It is important to note that searching for a monitor exit-path does not push a specific access on the fulfilling stack. Instead, an artificial search target representing the monitor object is pushed onto the stack. All monitor release operations on that monitor object encountered during the traversal are processed to determine whether a monitor exit-path has been computed.

WN Pairs

As discussed in Chapter 4, the execution order for member accesses of a *WN* pair cannot be permuted because a notify on an empty condition is lost. The only partial ordering implied by an ordered concurrent-access-pair of type *WN*, such as $(w_i,\ n_i)$, is $w_i \longrightarrow n_i$. However, its fulfillment is necessary for further extensions of a fulfilling interleaving that is blocked by a *wait*() invocation.

There are three scenarios in which a *wait*() call may be encountered within a particular monitor. In the first case, a *wait*() is encountered when the CFG is traversing for the next value-schedule-relevant access, such as r, as shown in Figure 5.21(a). In this case, the derivation of a value schedule continues on $T1$ until a *wait*() is encountered. Then, the interleaving is instructed to explore $T2$ to unblock $T1$. In the second case, a *wait*() is encountered when the traversal is trying to fulfill some ordered concurrent-access-pairs, such as that between r and w, as shown in Figure 5.21(b). In this case, the interleaving is context switched from $T1$ to $T2$ to reach w. When the *wait*() is encountered by $T2$,

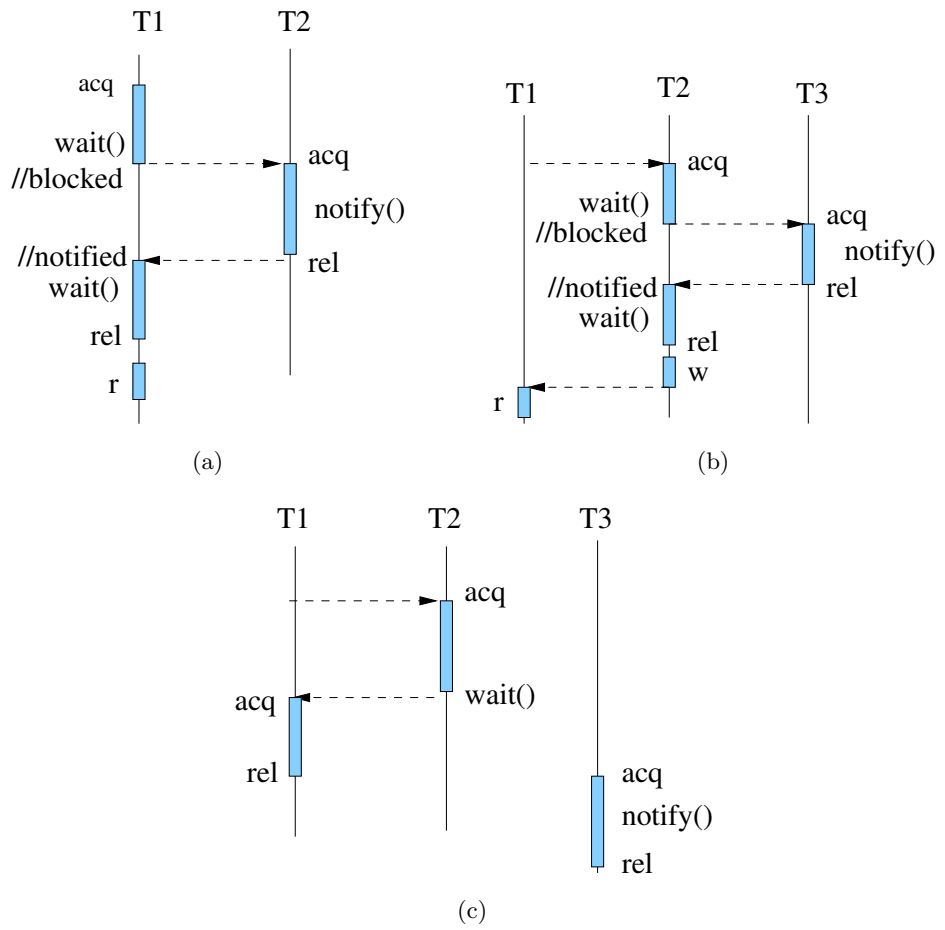


Figure 5.21: Three Scenarios in which *wait*() is Encountered

the interleaving is instructed to execute $T3$ to unblock $T2$. In the third case, a *wait*() is encountered while a thread is completing a monitor exit-path and context switches back to the original yielding thread. As shown in Figure 5.21(c), $T1$ attempts to enter the critical section, but it yields the execution for $T2$ because the monitor entries in both threads forms a MM pair. The path from the monitor entry to the *wait*() in $T2$ is a monitor exiting path. Hence, when *wait*() is encountered by $T2$, the execution is switched back to $T1$. $T2$ remains blocked until the $notify()$ from $T3$ is executed. The difference between these three scenarios is that in the first and the second case the *wait*() form a WN pair because the notify is required to fulfill the ordered concurrent-access-pair. However, the third case only forms a MM pair because the *wait*() only contributes to the monitor-exit pair without using the notify. Therefore, the fulfillment of a WN pair can only be triggered for the first and second cases.

The computation of a fulfilling interleaving for a WN pair happens after a *wait*() invocation is added to the fulfilling interleaving. The traversal of the CFG of the owner thread performs a wait and computes a path leading to a target notify. Java supports two

types of notify method. The $notify()$ method makes ready a single thread blocked on a monitor's waiting list, while the $notifyAll()$ method makes ready all threads blocked on the waiting list. The implementation always keeps track of which threads are waiting in which monitors in the current fulfilling-interleaving.

After the path leading to the *notify/All()* node is added to the fulfilling interleaving, the thread containing the target *notify/All()* still holds ownership of the monitor. Thus, the CFG traversal continues on the signalling thread to compute a path that releases the monitor. Only after that can the traversal be resumed for a blocked thread.

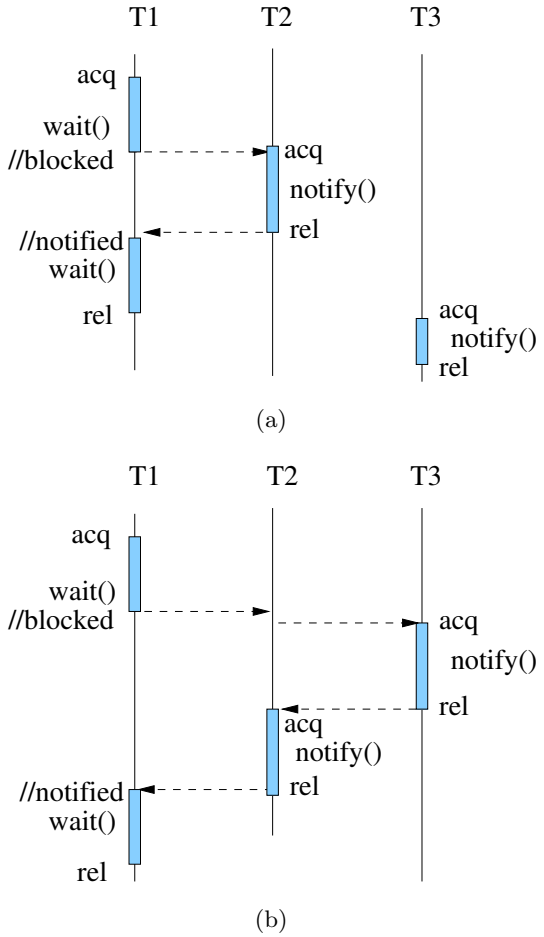


Figure 5.22: $wait()$ with Multiple Notifiers

As discussed before, fulfilling an ordered concurrent-access-pair of type *WN* does not directly produce any new value schedule. However, in the case when a $wait()$ forms *WN* pairs with multiple *notify/All()* statements, different value schedules can be triggered by having different *notify/All()* statements unblock the $wait()$. The process of deriving and fulfilling different value schedules for each possible *WN* pair is implemented as a process of deriving and fulfilling ordered concurrent accesses for the enclosing monitor entries of each *notify/All()* statement. An example of such a derivation process is shown in

Figure 5.22. $T1$ is woken by one of two $notify()$ statements in $T2$ and $T3$. Assuming the $notify()$ in $T2$ is randomly selected to unblock $T1$, then the CFG traversal on $T2$ finds a path to reach that statement in $T2$. At the monitor entry of $T2$, two value schedules are produced. In the first value schedule shown in Figure 5.22(a), $T2$ enters the monitor first and runs its $notify()$ statement to unblock $T1$. In the second value schedule shown in Figure 5.22(b), a recursive derivation on the monitor entry from $T2$ selects $T3$ to enter the monitor before $T2$. During the computation of the monitor exit-path for $T3$, a $notify()$ is encountered. This $notify()$ wakes $T1$ as a side-effect of computing a monitor exit-path from $T3$. The traversal returns to $T2$ after $T3$ exits the critical section. Then, $T2$ executes its critical section before the execution finally returns to $T1$. In this case, the $notify()$ from $T2$ is still executed but with no effect, which may lead to a concurrency error.

Recall from Chapter 4, the analysis splits a *wait*() invocation into a blocked *waiting* node and an unblocked *notify-entry* node, which the *notify-entry* node represents a *notified wait*() in the program. The derivation and fulfillment of *WN* pairs of a *wait*() invocation covers the processing of the *waiting* node in the CFG. During the fulfilling of a *WN* pair, recursive derivation is carried out whenever a value-schedule-relevant access is encountered. However, the complication comes after the *notify/All()* wakes the blocked thread. This step introduces a new ready thread that contends for the monitor. The contention between a *notified wait*() and monitor entries is captured next using *WM* concurrent access-pairs.

WM Pairs

After a fulfilling interleaving is computed to unblock a thread T_i, T_i is in the ready list after the fulfilling interleaving is executed. At the end of the interleaving, T_i is ready to reacquire the monitor and proceed. However, if the currently-notified *wait*() being processed forms *WM* pairs with other threads awaiting entry on the same monitor object, these *WM* pairs imply additional contention for monitor acquisitions between T_i and these other threads.

As shown in Figure 5.23, thread $T1$ is blocked by a *wait*() and may be unblocked by the $notifyAll()$ in $T2$. Because of the recursive derivation on the monitor entry by $T2$, two fulfilling interleavings are generated from the fulfillment of a *WN* pair. In first interleaving, shown in Figure 5.23(a), the critical section in $T1$ is executed without yielding to $T3$. This example shows the fulfillment of an ordered concurrent-access-pair, (T1.wait()) $\rightarrow$ (T3.acq()), of type *WM* by default insertion (Rule 1). Moreover, the next value-schedule-relevant access encountered in the traversal of $T1$ is the *notified wait*() node. Hence, an ordered concurrent-access-pair of (T2, acq()) $\rightarrow$ (T3, acq()) is implicitly fulfilled by not yielding on the monitor entry for $T2$.

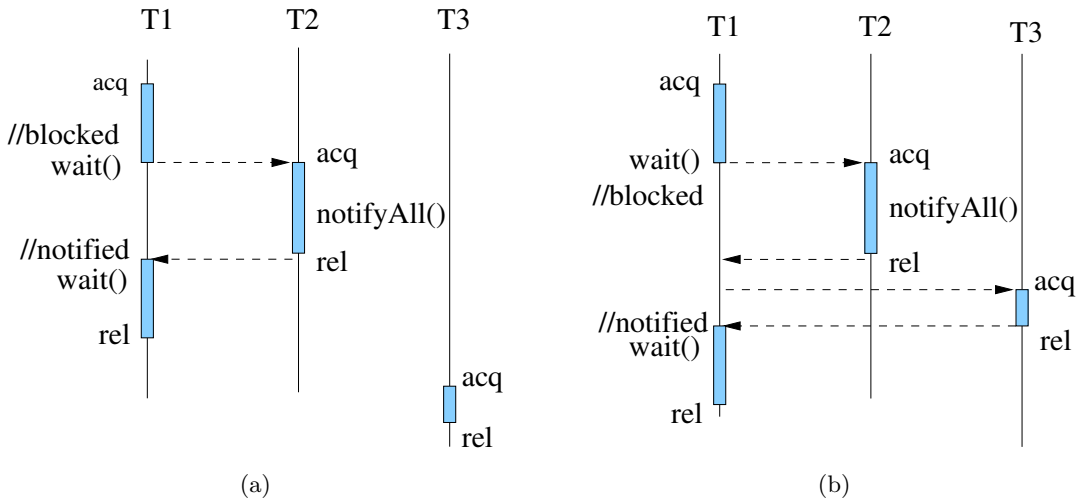


Figure 5.23: Derivation and Fulfillment over a *WM* Pair

In the second interleaving, shown in Figure 5.23(b), two ordered concurrent-access-pairs could be fulfilled between the *notified wait*() on $T1$ and the monitor entry from $T3$. The direct insertion of the *notified wait*() node into the fulfilling interleaving has $T1$ acquire the monitor first. Yielding before $T1$, by inserting the acquisition of the monitor in $T3$ into the interleaving, causes $T3$ to acquire the monitor first. As a result, an ordered concurrent-access-pair, (T3.acq()) $\rightarrow$ (T1.wait()), is fulfilled by yielding (Rule 2). The derivation and fulfillment of the ordered concurrent-access-pair on the *notified wait*() node captures the contention between threads on the ready list and entry list of a monitor, showing an example of barging, which may be a concurrency error.

It is possible for the fulfillment of an ordered concurrent-access-pair of type *WM* to be triggered at monitor entry. This case is possible when a *wait*() method is added to the fulfilling interleaving to compute a monitor exit-path to fulfill some ordered concurrent-access-pair of type *MM*. If during further traversal, a $notifyAll()$ on the same monitor is added to the interleaving either in the process of searching for the next value-schedule-relevant access or in the process of fulfilling some other ordered concurrent-access-pair, then all threads blocked on that monitor are unblocked. Then, the next monitor entry encountered on the same monitor object triggers the fulfillment of an ordered concurrent-access-pair of type *WM*. However, as discussed in Section 5.4, the derivation process always computes the same set of value schedules from a given set of concurrent accesses regardless of which access starts the derivation process.

For example, in Figure 5.24, if a monitor entry for $T1$ yields to $T3$, a monitor exit-path is computed for $T3$. Then, the execution returns to $T1$, and triggers a $notifyAll()$. This $notifyAll()$ unblocks $T3$. After the critical section on $T1$ is completed, exploration starts in $T2$. Upon encountering the acquisition of the monitor by $T2$, the derivation algorithm knows there is another previously blocked thread, $T3$, that is on the ready list, and may

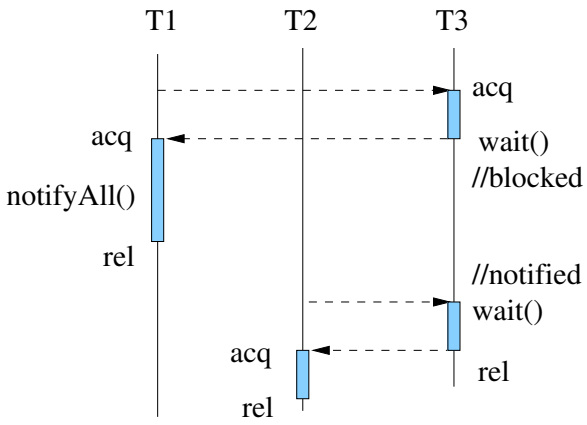


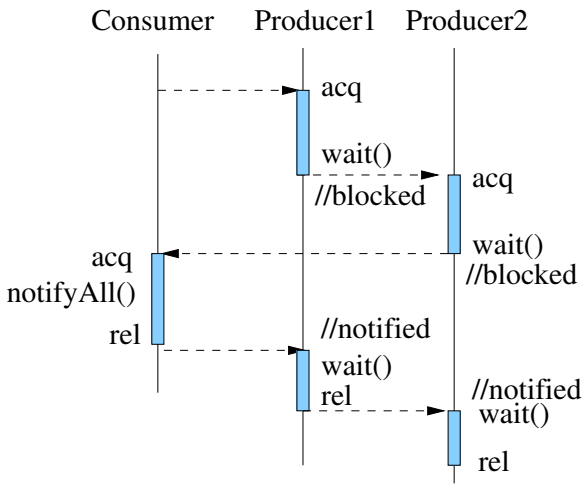
Figure 5.24: Concurrently Unblocked *wait*()s

acquire the monitor before $T2$. Hence, the derivation process proceeds to compute an interleaving to fulfill an ordered concurrent-access-pair of WM type that allows $T3$ to acquire the monitor before $T2$.

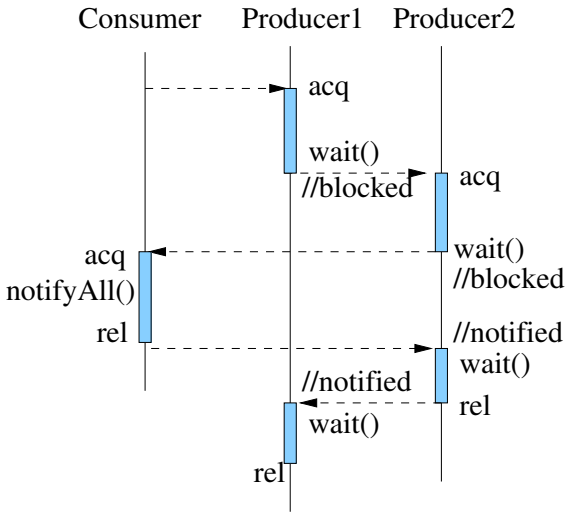
WW Pairs

The fulfillment of an ordered concurrent-access-pair of type WW is triggered when a *notified wait*() is the next value-schedule-relevant access encountered during CFG traversal and there are other threads that are also ready to acquire the same monitor from their respective *notified wait*(). It is important to note that during the concurrent access-pair generation, concurrent access-pairs that consist of two *notified wait*()s are not computed. Instead, the derivation algorithm is responsible for identifying competing *notified wait*()s at derivation time. The derivation algorithm keeps track of which threads are blocked when *wait*()s are encountered during the traversal, and transitions a blocked thread to a ready thread when a *notify/All()* is encountered. Thus, when a *notified wait*() is encountered during the traversal, the derivation module knows the other threads on the ready list for the same monitor.

A common scenario in which multiple ready threads contend for a monitor is captured by a producer/consumer problem. Multiple producers and consumers compete to add and remove items in a shared buffer. When the buffer limit is reached, all the producer threads attempting to insert more items are blocked by *wait*(). The producers are unblocked when a consumer thread takes an item out of the buffer and calls the $notifyAll()$ method. Then, the unblocked producers compete for monitor ownership from the ready list. This scenario is shown in Figure 5.25, where two producer threads are made ready by a consumer thread. Assume the buffer is already full when the monitor entry on $Consumer$ is encountered. Then, the derivation algorithm recursively computes ordered concurrent-access-pairs on concurrent monitor entries from $Producer1$ and $Producer2$.



(a)



(b)

Figure 5.25: Derivation and Fulfillment over a WW Pair

Supposing $Consumer$ yields for $Producer1$, and $Producer1$ yields for $Producer2$, then both $Producer1$ and $Producer2$ become blocked by the $wait()$ invocations due to the full buffer. When the traversal returns to $Consumer$, it acquires the monitor and performs the $notifyAll()$ in the critical section to make both $T2$ and $T3$ ready. Now, suppose the continued exploration of the consumer thread reaches its end, and that the traversal continues from the *notified* $wait()$ node in $Producer1$. Because the derivation algorithm knows that $Producer2$ is ready to acquire the same monitor, it computes two ordered concurrent-access-pairs of type WW. In one partial value-schedule, the *notified* $wait()$ from $T2$ is immediately added to the fulfilling interleaving as shown in Figure 5.25(a), while in the other interleaving the remaining instructions in the critical section of $T3$ are added to the fulfilling interleaving before those from $T2$ as shown in Figure 5.25(b).

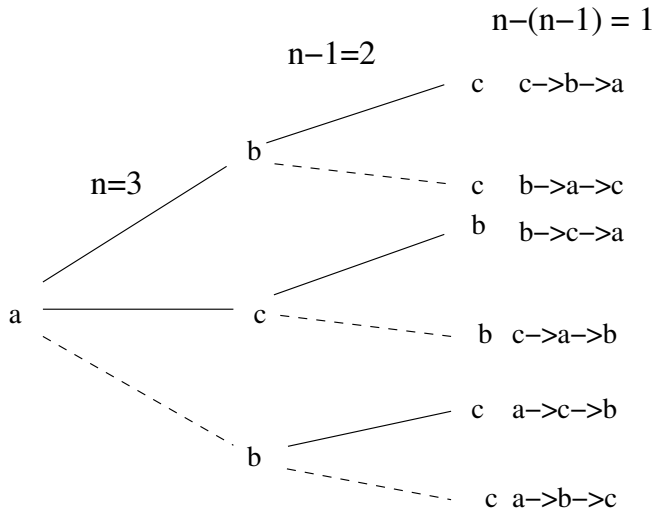


Figure 5.26: An Example of Deriving Rules with $n = 3$

5.4 Completeness

In this section, the derivation rules are shown to derive all value schedules in a program. That is, for a set of independent conflicting-concurrent-accesses, the derivation rules compute all value schedules.

Figure 5.26 illustrates the general derivation rules. A default insertion of a node derived from *Rule 1* is illustrated by a dashed line, while yielding the insertion of a node into a value schedule derived from *Rule 2* is shown as a solid line. The proof of the correctness of the derivation rules follows.

Proposition 1. *Given a set of independent conflicting accesses of size n, the derivation rules compute all possible orderings of those accesses.*

Proof. By counting.

For a set of independent conflicting accesses, the number of different orderings is $n!$ because $n!$ distinct permutations exists for any set of size n. A derivation can start with an arbitrary access. Processing of each node computes $n - k$ different value schedules where n is the total number of conflicting accesses in the set, and k is the number of nodes that are considered during the derivation. Among these value schedules, $n-k-1$ of them represent possible ordered concurrent-access-pairs that can be formed with accesses from other threads, and one of them represents the ordered concurrent-access-pair implied by the default insertion of the access. Thus, the whole derivation process computes $\prod_{k=0}^{n-1} n - k = n!$ different value schedules.

The presented derivation rules are correct if they also compute $n!$ distinct orderings. As shown in the previous discussion, the derivation rules always compute $n!$ orderings.

Moreover, because each yielding and default insertion from any value-schedule-relevant access always fulfills a different ordering among conflicting accesses, every value schedule derived from the derivation rules on each access is distinct. Thus, the derivation rules derive all orderings among independent conflicting accesses, and it does not matter from which thread the derivation starts.

□

Figure 5.26 gives an example of deriving value schedules over a set of concurrent conflicting-accesses of size 3, such as *a*, *b and c*. For example, the line between *a* and *b* implies a permutation such that *a* yields for *b*. Moreover, the dashed line between *b* and *c* represents the direct insertion of *b*, then *c* is added afterward. In the end, the derivation rules produces six different orderings, which is 3!.

Now, in the presence of the dependencies among accesses, the derivation rules still derive all value schedules.

Proposition 2. *Given*

1. *a set of distinct concurrent accesses to shared variables,*
2. *all possible concurrent access-pairs that can be constructed from these accesses, and*
3. *CFGs corresponding to the participating threads of the program,*

the recursive value-schedule derivation derives all possible value schedules and their fulfilling interleavings of these accesses. Note, the program being tested is assumed to be finite; thus, all value schedules referred in the proof are also finite.

Proof. By contradiction.

Assume there is a feasible value schedule vs_1 such that vs_1 is not derived by the derivation algorithm, and a feasible value schedule vs_2 derived by the derivation algorithm. Because both value schedules are feasible, each has at least one fulfilling interleaving. Let int_1 and int_2 be the fulfilling interleavings of vs_1 and vs_2 respectively. Because vs_1 and vs_2 are different, there is at least one ordered concurrent-access-pair that is fulfilled in vs_1 but not in vs_2. This proof shows these assumption always lead to contradictions.

Stepping through int_1, for each access to a shared variable, its partial order with its concurrent conflicting-accesses can be determined by examining its relative position with its conflicting accesses in int_1. As discussed earlier, every pair of orderings between two conflicting accesses is an ordered concurrent-access-pair. Hence, a set of ordered concurrent-access-pairs for each access to shared variables fulfilled by vs_1 can be determined. Similarly, stepping through int_2, it can be determined if the same ordered

concurrent-access-pairs are fulfilled in vs_2. Let this mapping process continue until an ordered concurrent-access-pair is determined to be fulfilled in vs_1, but not in vs_2. Let those ordered concurrent-access-pairs that have been mapped until this point be referred to as the *common-prefix* of vs_1 and vs_2, and the identified ordered concurrent-access-pair that is fulfilled in vs_1 but not in vs_2 as the *diverging pair* of vs_1 and vs_2. Now, let vs_j be a value schedule derived by the derivation algorithm that has the longest common-prefix with vs_1, let $a_m \rightarrow a_n$ be the diverging pair, and int_j define the fulfilling interleavings of vs_j. There are three possible reasons that this ordered concurrent-access-pair is not fulfilled in vs_j,

1. a_m does not exist in int_j
2. $a_n \rightarrow a_m$ is fulfilled vs_j
3. a_m exist in int_j but a_n does not

Case 1: In the case when a_m does not exist in int_j, this implies there is some ordered concurrent-access-pair in vs_j that is fulfilled before a_m that made a_m unreachable. In other words, there is a branch, br, in the owner thread of a_m that reads in a value given by some concurrent writes and skips a_m in the following execution. The read access in br is referred to as the *dominating access* of a_m because it determines the accessibility of a_m. This scenario is the only way a_m can appear in int_1 but not in int_j.

Now, because a_m exists in int_1, then the read access in br in int_1 should have read in a value that leads to a_m. The read access in br must read in a different value in int_1 and int_j. This implies that int_1 has fulfilled a different ordered concurrent-access-pair for the read in br than the one fulfilled for the same read access in int_j. Furthermore, because br always happens before a_m, this implies the diverging pair of vs_1 and vs_j is an ordered concurrent-access-pair related the read access in br. This contradicts the claim that $a_m \rightarrow a_n$ is the diverging pair.

Thus, a_m should always exist in int_j. If a_n also exists in int_j and happens after a_m, then vs_j cannot be the derived value-schedule that shares the longest common-prefix with vs_1, and leads to a contradiction. If a_n happens before a_m in int_j, then the proof for Case 2) is needed. If a_n does not exist in int_j, the proof for Case 3) is needed.

Case 2: In the case where $a_n \longrightarrow a_m$ is fulfilled in vs_j, consider the different orders in which a_n and a_m can be encountered during derivation of vs_j. If a_m is encountered first during the CFG traversal of the derivation process, then a direct insertion (Rule 1) always adds a_m before a_n. Thus, $a_m \longrightarrow a_n$ is automatically implied if a_n also exists in int_j. Clearly, this contradicts to the claim that $a_n \longrightarrow a_m$ is the diverging pair. Otherwise, the proof for Case 3) is needed.

If a_n is encountered first during the derivation process, then a search of the CFG for the owner thread of a_m produces an interleaving where a_m is inserted into vs_j first. As

proven in Case 1), a_m should always be reachable given the fact that $a_m \rightarrow a_n$ is the diverging pair. Thus, the derivation algorithm can always generate a value schedule vs_k that not only fulfills the common prefix between vs_1 and vs_j but also contains $a_m \longrightarrow a_n$. Thus, vs_k shares a longer common prefix with vs_1 than vs_j. This contradicts the claim that vs_j is the longest such value schedule.

Case 3: In the case that a_n does not exist in int_j, a_n is made unreachable by some other ordered concurrent-access-pairs. For example, a read access, r_i, in a branch clause enclosing a_n may read in the value from a concurrent write and force the execution to skip a_n altogether. Moreover, this ordered concurrent-access-pair involving r_i cannot be an ordered concurrent-access-pair fulfilled before a_m is added in int_1. Otherwise, $a_m \longrightarrow a_n$ is not the diverging pair between vs_1 and vs_j.

On the other hand, the ordered concurrent-access-pair involving r_i can happen after a_m in int_1, and then a_n becomes unreachable. Let w_i be a concurrent write to r_i that allows access to a_n. Assume w_i exists in int_1, too. If w_i is encountered first during the traversal, it recursively yields to other concurrent writes based on Rule 2 in Figure 5.10, so these writes occurs before w_i. Continuing after w_i, r_i reads in its value and executes access to a_n. If r_i is encountered first, Rule 2 ensures w_i happens before r_i. Thus, r_i reads in the value of w_i and executes a_n.

If either w_i or r_i is missed from int_j, then it is disabled by some other ordered concurrent-access-pair. It is always possible to backtrack along int_j to locate accesses that dominate the accessibility of w_i and r_i respectively, and to have the dominating access form the same ordered concurrent-access-pairs with its conflicting accesses as in vs_1. This process can be applied recursively backwards along int_i until a_n becomes accessible. It is important to note that such a backward reconciliation process cannot disable accesses in the common prefix of vs_1 and vs_j, because a_n is accessible in vs_1. Thus, the execution ordering implied by the common prefix is not responsible for disabling of a_n. This further implies that the derivation algorithm should produce a value schedule vs_t that fulfills all ordered concurrent-access-pairs in the common prefix of vs_1 and vs_j and $a_m \rightarrow a_n$. Thus, vs_t has the longest common prefix with vs_1 instead vs_j, which leads to a contradiction.

Thus, vs_j cannot be the derived value-schedule that shares the longest common-prefix with vs_1. Moreover, it is possible to reconcile the diverging pair between vs_1 and a derived value-schedule to make vs_j the longest common-prefix by backtracking to the RW pair that disabled the ordered concurrent-access-pair. As a result, vs_1 should have been derived and fulfilled by the derivation algorithm, which contradicts the original premise. Therefore, all possible value schedules and their fulfilling-interleavings can be generated.

□

5.5 Limitations

The discussion on two-staged value-schedule-based testing demonstrates how value schedules of a program are tested by deriving their fulfilling interleavings from the CFG of the program. More importantly, through this value-schedule derivation and testing process, the accessibility-based POR and prioritized testing of certain permutations can be applied to improve efficiency.

However, there are some limitations that the two-stage approach needs to overcome to be applicable for real programs. These limitations are mainly due to the coarse-grained nature of the static analysis used in value-schedule-based testing. First, this simple implementation of value-schedule-based testing assumes that all distinct concurrent conflicting-accesses are given. However, the set of complete distinct accesses to shared variables might not be statically computable. An access defined in a concurrent access-pair may correspond to multiple instances of that access at runtime because of iteration. Therefore, this simple implementation of value-schedule-based testing is unsound. Second, two interleavings are always generated to cover both possible branches in a conditional statement. Assuming a thread has 20 conditional statements in its CFG, the traversal of the CFG of that thread produces at least 2^{20} execution paths to cover all possible branches while only one out of 2^{20} paths correctly captures a specific path through the branches taken by that thread at runtime. Moreover, all generated interleavings need to be stored for testing. Third, when a program contains polymorphic call sites, it is difficult to predict which concrete method executes at runtime. A conservative estimate computes fulfilling interleavings along all possible implementation methods. Thus, many infeasible paths are generated. Fourth, besides the reachability analysis on the CFG, the statically computed concurrent access-pairs might turn out to operate on different objects due to aliasing, which are irrelevant for testing purposes. All interleavings computed based on this false concurrent access-pair are superfluous interleavings since they do not correspond to distinct value schedules.

To resolve these issues, the next chapter introduces an improved version of value-schedule-based testing. In the improved approach, value schedules are derived and tested incrementally as the CFG is traversed. More importantly, intermediate dynamic information is combined with the static information to improve the testing process.

5.6 Summary

This chapter presents the main concepts of value-schedule-based testing. A value schedule defines the partial ordering of conflicting accesses to shared variables in an execution. Thus, only one interleaving needs to be tested for each distinct value schedule. Value-schedule-based testing is used to guide dynamic testing by deriving value schedules

of a program and executing their fulfilling interleavings. More importantly, the value schedule explicitly uses the conflicting relationships among accesses to compute ordered concurrent-access-pairs. A permutation and context switch of transitions is taken in the fulfilling interleaving only if static analysis can compute a path to fulfill a new ordering of two conflicting accesses. Thus, value-schedule-based testing performs POR based on the accessibility of conflicting accesses instead of less-precise flow-insensitive availability of conflicting accesses in concurrent threads. As a result, the value-schedule-based testing can deliver more aggressive POR than existing techniques. Moreover, because value-schedule-based testing is aware of all concurrent conflicting-accesses of a particular access and is capable of fulfilling them individually, value-schedule-based testing can prioritize testing of permutations of concurrent accesses to evaluate their effects first. The general implementation concepts of value-schedule-based techniques are applied over various program constructs. Despite its promising improvements, the static two-stage derivation algorithm presented in this section cannot be applied to real programs due to its conservative approximations to variable access and control-flow branches.

Chapter 6

Practical Value-Schedule-Based Testing

The previous chapter showed how statically-computed information can help dynamic testing-tools avoid executing irrelevant interleavings. This chapter presents a practical implementation of the value-schedule-based technique composed of two main features. First, this implementation derives and tests value schedules incrementally. The feasibility and correctness of a value schedule is evaluated one ordered concurrent-access-pair at a time. This incremental fulfillment allows value-schedule-based testing to identify infeasible value schedules and fulfilling interleavings early in the derivation stage to reduce superfluous traversal of the CFG. Thus, the accessibility-based reduction is more efficient. Moreover, the incremental testing allows value-schedule-based testing to prioritize the testing of some partial value-schedules and extend from ordered concurrent-access-pairs that are closely identified with harmful concurrency bugs. Thus, harmful bugs can be detected earlier in testing. Second, the incremental testing opens up various opportunities for value-schedule-based testing to make use of the dynamic information collected during testing to refine and improve the original guidance given by the static module. For example, the branch-decision on a CFG node is resolved using dynamic information. Hence, only one fulfilling interleaving is computed for each ordered concurrent-access-pair. Moreover, for more complicated situations such as the ambiguity of the callee type at a polymorphic callsite, the callee type can be dynamically resolved using runtime information to reduce the search space.

6.1 Incrementally Deriving and Testing Value Schedules

The algorithm for incremental testing has three features. First, it makes uses of saved execution states, called backtracking points, that serve as restore points. Different ordered

concurrent-access-pairs of a value-schedule-relevant access can be attempted without re-executing the common interleaving leading to that access. Second, the feasibility of an ordered concurrent-access-pair is checked before being added to a partial value-schedule. Thus, the accessibility-based-POR can immediately prune out a superfluous value schedule. Third, the incremental approach automates the prioritized testing by assigning heuristic relevance to the ordered concurrent-access-pairs.

6.1.1 Incremental Derivation with Backtracking

The incremental deriving and testing of value schedules using backtracking during the derivation process is shown in Figure 6.1. The initial explanation of the algorithm is presented using a simpler version of the sample program in Figure 5.1 from page 60 by removing two synchronized statements on *accY* (see Figure 6.2).

The derivation again starts from a random thread as shown at the line 2 of Figure 6.1. Assume $Thread1$ is again selected. A backtracking point is created, when a value-schedule-relevant access is encountered along the CFG of $Thread1$. For example, the processing of the read of *accX.val* at line 3 produces a backtracking point, *bpt*1, as shown in Figure 6.3(a). A backtracking point contains the execution state computed from executing the interleaving leading to that value-schedule-relevant access, not including that access. A cached backtracking-point can be later reloaded to reset the program execution. More importantly, unique to value-schedule-based testing, each backtracking point includes the set of accesses that are in conflict with the value-schedule-relevant access, one of which triggered the creation of this backtracking point. An access that triggers the creation of a backtracking point is called a *triggering access*. Note that the conflicting accesses associated with the backtracking point are accesses forming concurrent access-pairs with the triggering access. From the conflicting accesses associated with each backtracking point, the value-schedule-based testing can determine possible ordered concurrent-access-pairs that can be derived and tested from this backtracking point without re-executing the interleaving leading to it. For example, deriving and testing the ordered concurrent-access-pairs, (t2, run.(w(accX.val), line 3)) → (t1, run.(r(accX.val), line 3)) and (t2, run.(w(accX.val), line 5)) → (t1, run.(r(accX.val), line 3)), does not require re-executing the interleaving leading to the read of *accX.val* at line 3. Instead, a simple reloading of *bpt*1 is sufficient.

The creation of backtracking points are shown at line 23 and 34 of Figure 6.1. After a backtracking point is created, the triggering access is always explored first, while other possible ordered concurrent-access-pairs are cached for later processing. This approach ensures a depth-first search of the program CFG. In Figure 6.1, this operation is shown at line 38. As the path is extended, each triggering access is added to the front of the worklist and is processed first in the next iteration. Note that the worklist is always

```
1  main()
2      Select aribitrary thread t and create a SearchPath p
3      Initialize p with the entry node of t
4      Add p into worklist
5      while worklist is not empty
6          item = prioritize(worklist);
7          p = dispatch(worklist, item);
8          search (p)
9
10 search (SearchPath p)
11     Let c be CFG node at end of path
12     if (c.hasMultipleSuccessors())
13         if (c is BranchStmt) s = resolveBranch(c, p);
14         else if (c is Callsite) s = resolveCallSite(c, p);
15         r = reachability(s, p.target);
16         if (!r) return; else s = successor(c);
17
18     if (s == null && p.stack == empty
19         && hasOnlyOneThreadLeft(p))
20         complete(p);
21
22     If (s.isRelevant() && p.mode != PRIORITY)
23         bp = createBacktrackingPoint(p, s);
24         Add bp to the front of worklist
25
26     Append s to p
27     If s is the search target (top of the fulfillment stack of p)
28         Pop the fulfillment stack of p
29         fulfill(p);
30         valid = aliasChecking();
31         if (!valid && !reachability(s, p.target))  return;
32         resume(p);
33         if (reachability(s, p.target))
34             bp = createBacktrackingPoint(p, p.trigger, p.target)
35             Add bp to the front of worklist
36         Append p.trigger to p;
37         if (p.mode == PRIORITY) complete(p);
38     Add p to the front of worklist //DFS
```

Figure 6.1: Overview of Incremental Search Algorithm

Shared Variables

```
1    accX.val = 0;
2    Account.amt = 10;
3    accY.val = 0;
```

Thread 1

```
1  run(){
2    synchronized(accX){
3      accX.val += Account.amt;
4    }
5  }
```

Thread 2

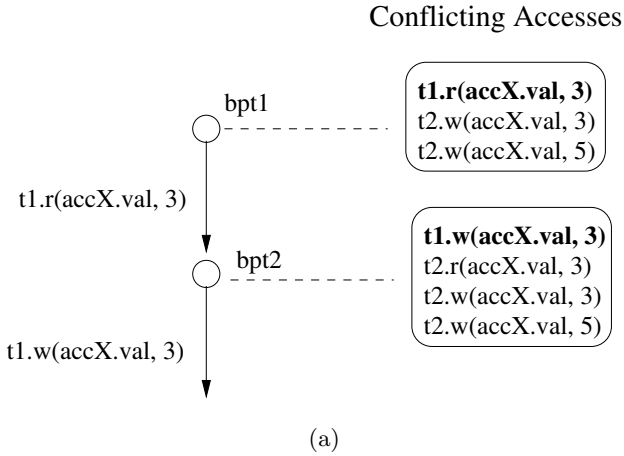
```
1    run(){
2      synchronized(accY){
3        accX.val -= Account.amt;
4        if (accY.val < 0){
5            accX.val = 0;
6        }
7      }
8    }
```

Figure 6.2: Simpler Sample-Program

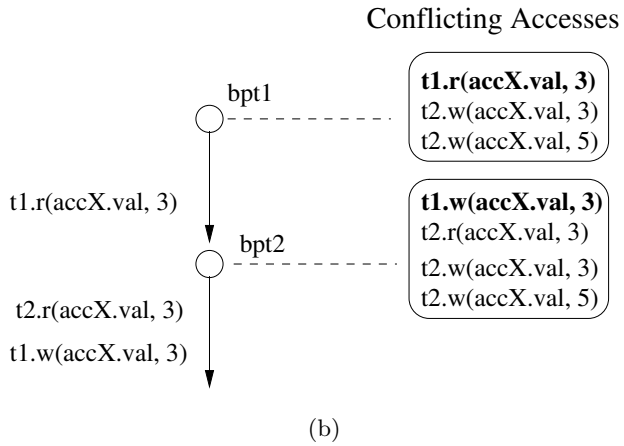
headed by a path and followed by a sequence of backtracking points. The *dispatch*() method at line 7 of Figure 6.1 is responsible for disambiguating different types of objects on the worklist and returns an interleaving to be processed further.

In the example, when the traversal of $Thread1$ reaches the end of the *run*() method, the first fulfilled complete-value-schedule consists of all accesses to shared variables from $Thread1$ happening before all of their concurrent conflicting-accesses from $Thread2$. Moreover, the derivation process of the first value-schedule produces two backtracking points corresponding to the value-schedule-relevant accesses for $Thread1$ as shown in Figure 6.3(a). The conflicting accesses associated with each backtracking point are shown in the boxes, and the triggering access is shown in bold.

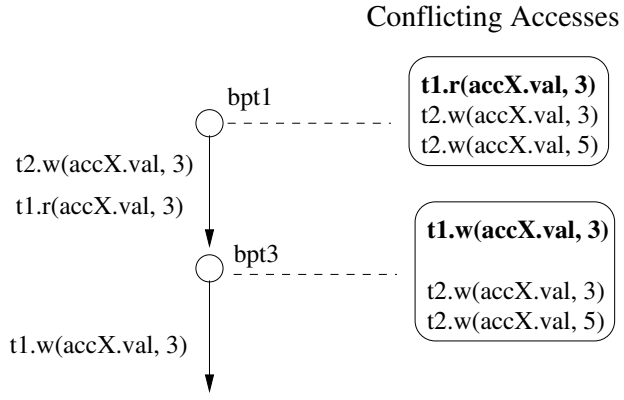
A cached backtracking-point is processed in the reverse order of its creation. The first backtracking point to be processed corresponds to the write to *accX.val* at line 3 of $Thread1$. It is shown as backtracking point *bpt*2 in Figure 6.3(a). This access has three concurrent conflicting-accesses from $Thread2$. The derivation algorithm tries to fulfill these three ordered concurrent-access-pairs using the derivation technique discussed for *RW* concurrent-access-pairs discussed in Chapter 5. Assume the ordered concurrent-access-pair, (t2, run.(r(accX.val), line 3)) → (t1, run.(w(accX.val), line 3)) is chosen to be fulfilled first. The program is restored to the execution state saved in *bpt*2. The restoration resets $Thread1$ to the program point right before the write to *accX.val* at line 3 is executed, and sets $Thread2$ to the program point at entry of its *run*() method. Then, the CFG of $Thread2$ is traversed to compute an interleaving to the read of *accX.val* at line 3 as discussed in Chapter 5. As the interleaving is computed, it is executed directly



(a)



(b)



(c)

Figure 6.3: Backtracking Points from Simple Sample-Program

from the current program state, *bpt*2, to verify the feasibility of this interleaving as shown in Figure 6.3(b). In this case, the interleaving to the read of $accX.val$ at line 3 of $Thread2$ is feasible. Finally, execution along $Thread1$ reaches the end of its $run()$ method again. Then, the next backtracking point is still *bpt*2, which contains two unfulfilled ordered concurrent-access-pairs.

The next ordered concurrent-access-pair to be processed from *bpt*2 is (t2, run.(w(accX.val), line 3)) → (t1, run.(w(accX.val), line 3)) and it is fulfilled and tested in the same way as that of (t2, run.(r(accX.val), line 3)) → (t1, run.(w(accX.val), line 3)). This ordered concurrent-access-pair is also feasible. The last pair for *bpt*2 is (t2, run.(w(accX.val), line 5)) → (t1, run.(w(accX.val), line 3)). After the fulfilling interleaving is computed for this ordered concurrent-access-pair, the feasibility testing determines its only fulfilling interleaving is infeasible because the branch statement in $Thread2$ does not take the true branch. Thus, no traversal is carried out along this infeasible partial value-schedule to include the permutation on the monitor entries of $accY$. This example shows how the accessibility-based-POR incrementally prunes out superfluous partial value-schedules.

As all possible ordered concurrent-access-pairs from *bpt*2 have been derived, the backtracking returns to *bpt*1, which has two conflicting accesses from $Thread2$. The processing of the ordered concurrent-access-pair, (t2, run.(w(accX.val), line 3)) $\xrightarrow{def}$ (t1, run.(r(accX.val), line 3)), produces a new backtracking point *bpt*3 as shown in Figure 6.3(c). Th is processed identically to *bpt*2, and does not lead to any error.

As the backtracking point returns to *bpt*1, the ordered concurrent-access-pair, (t2, run.(w(accX.val), line 5)) $\xrightarrow{def}$ (t1, run.(r(accX.val), line 3)), is dispatched to be fulfilled next. Similarly, the fulfilling interleaving computed for this ordered concurrent-access-pair is infeasible because the true branch at line 4 of $Thread2$ is never taken. The incremental feasibility testing of this interleaving instructs the value-schedule-based testing not to carry out any further derivation from the superfluous partial value-schedule (t2, run.(w(accX.val), line 5)) $\xrightarrow{def}$ (t1, run.(r(accX.val), line 3)). Therefore, superfluous value schedules such as,

(t2, run.(w(accX.val), line 5)) $\xrightarrow{def}$ (t1, run.(r(accX.val), line 3))
(t2, run.(w(accX.val), line 5)) $\longrightarrow$ (t1, run.(w(accX.val), line 3))

are never derived because the first ordered concurrent-access-pair in the partial value-schedule is infeasible. This example shows how the incremental testing helps to identify superfluous ordered concurrent-access-pairs before the superfluous ordered concurrent-access-pairs are appended to partial value-schedules for further derivations. As all possible permutations from *bpt*1 have been attempted, there is no backtracking point left in the worklist, and the testing process terminates.

Now, returning to the original sample program in Figure 5.1, after the testing process finishes deriving and testing the first value schedule, there are three backtracking points

that need to be processed instead of two, as shown in Figure 6.4. Two of them, $bpt1$ and $bpt2$, contain concurrent-access-pairs of type RW, while the third one, $bpt3$, contains concurrent-access-pairs of MM type. As discussed in Chapter 4, a RW pair always implies the possible existence of an unprotected access to a shared variable if it is shown to be feasible, while a MM pair may imply a possible general race. Therefore, it makes more sense to process the backtracking points that contain RW pairs first. A straightforward depth-first testing processes the backtracking point $bpt3$ first before it proceeds to the backtracking points that may contain error-generating concurrent-access-pairs. For example, if the backtracking point $bpt2$ is processed out of order, the testing of the fulfillment of any one of its two feasible ordered concurrent-access-pairs compute an interleaving that triggers an assertion failure. Thus, prioritizing the testing of backtracking points containing concurrent-access-pairs that are more correlated with potential concurrency errors can lead the testing process to check potential incorrect interleavings more quickly.

However, such out-of-order testing runs the risk of turning a depth-first testing process into something similar to breadth-first testing. For example, the out-of-order processing of the backtracking point corresponding to the write to $accX.val$ at line 3 of $Thread1$ creates two new feasible value schedules, and each of these value schedules contributes new backtracking points on the monitor acquisition on $accY$. Moreover, the processing does not return to the original three backtracking points until the new backtracking points computed in the out-of-order testing are completely processed. As a result, the dynamic testing-tool has to store backtracking points as well as the fulfilling interleavings from more than one value schedule at the same time. Out-of-order testing might thus negatively affect both the computational and memory use if the digression does not quickly lead to the discovery of a concurrency bug. Therefore, if out-of-order testing cannot uncover a bug within a reasonable time, the testing tool should abandon the out-of-order testing to reduce the negative effects. A practical implementation of prioritized testing is given in the following section.

6.1.2 Prioritizing the Testing

This section shows how the prioritized testing introduced in Chapter 5 can be implemented within incremental value-schedule-based testing. First, instead of relying on a specification from programmers, the prioritized testing introduced in this section estimates the relevance of an ordered concurrent-access-pair based on a heuristic value. Moreover, by leveraging some existing features of value-schedule-based testing, the out-of-order testing of cached ordered concurrent-access-pairs can be performed without turning the depth-first processing into a breadth-first one. This practical implementation of prioritized testing is illustrated by continuing the testing process from the point when the first value schedule from the original sample-program is fulfilled and tested as shown in Figure 6.4.

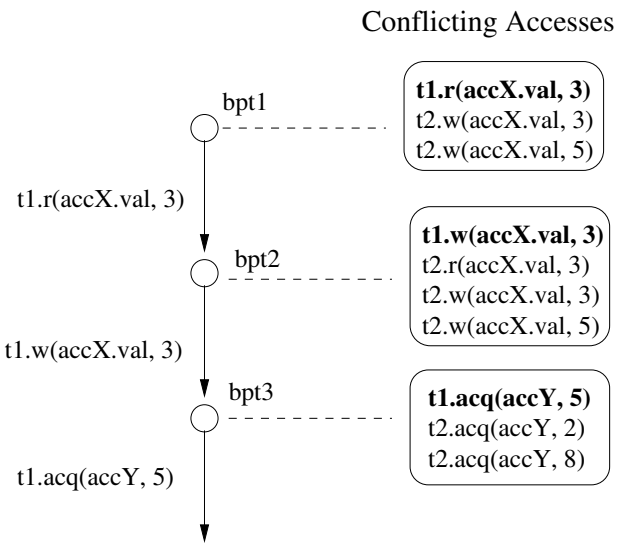


Figure 6.4: Backtracking Points from Original Sample-Program (From Figure 5.1)

First, the heuristic scheme for evaluating the relevance of a backtracking point is presented. In general, the backtracking points that contain the types of concurrent access-pairs that are more commonly related with concurrency bugs are processed first. As discussed in Chapter 4, different types of concurrent access-pairs correspond to different types of concurrency bugs. An ordered concurrent-access-pair derived from a pair of type *RW* is a clear indication of a data race if it can be fulfilled. Moreover, if no *WN* pairs associated with a backtracking point can be fulfilled, it is a good indication that there is a deadlock in the program. On the other hand, a concurrent-access-pair of type *MM*, *WM* or *WW* may or may not contribute to a harmful general race. In most cases, it may just serve to provide the desired synchronization for the program. Among these three types of concurrent access-pairs, types *WM* and *WW* may indicate possible barging behaviours. Thus, concurrent access-pairs of type *WM* and *WW* are considered more likely to lead to errors than those of type *MM*. In the end, the different types of concurrent access-pairs are heuristically assigned the following priorities:

$$RW > WN > WM \geq WW > MM \tag{6.1}$$

When backtracking, backtracking points that contain concurrent access-pairs with higher priority are candidates for prioritized processing (see Figure 6.5). The prioritized selection of backtracking points access by the *prioritize*(*worklist*) procedure is called at line 6 of Figure 6.1. The code between lines 6 and line 13 of Figure 6.5 iterates through the worklist to find the backtracking point with the most relevant access type.

After a backtracking point is selected, the next step is to monitor the interleavings derived from the out-of-order processing. The goal is to fulfill the ordered concurrent-access-pairs associated with each selected backtracking point with the fewest side-effects.

```
1  prioritize(worklist)
2    item = null;
3    if (worklist.front() is Path)
4      return worklist.front();
5
6    foreach BacktrackingPoint bpt in worklist
7      if (item == null)
8          if (!bpt.OutofOrderOnce)
9              item = bpt;
10
11     if (bpt.relevance > item.relevance)  // find highest priority
12         if (!bpt.OutofOrderOnce)       //item
13             item = bpt;
14
15   if (item != null)
16      if (item.numOfPairsLeft() == 1)  // the last pair
17         bpt.OutofOrderOnce = true;
18      return item;
19   else
20      return worklist.front();
```

Figure 6.5: Prioritizing Procedure

The value-schedule-based testing achieves this by disabling the generation of backtracking points during the out-of-order processing, which also prevents the fulfillment process from performing the recursive expansion of value schedules. This restriction is shown by the branch statement at line 22 of Figure 6.1. The goal is to limit the out-of-order processing on each ordered concurrent access-pair to exactly one fulfillment, so no additional backtracking point is introduced. After the ordered concurrent access-pair is fulfilled, the value-schedule-based testing checks for an error by completing the remaining instructions in each active thread. The evaluation is done by running the program "sequentially" to completion, following the interleaving for the just-fulfilled ordered concurrent-access-pair. For example, all instructions in $Thread1$ are completed first, then those from $Thread2$, etc. This execution determines whether an out-of-order fulfillment of an ordered concurrent-access-pair is sufficient to lead the program to a hidden concurrency bug in a single fulfillment.

Because the value-schedule-based testing only commits one context-switch for an out-of-order processing of an ordered concurrent-access-pair, an ordered concurrent-access-pair that requires more than one fulfillment through recursive expansion is not tested. Thus, the out-of-order processing does not remove the backtracking point from the worklist. Instead, such a backtracking point is marked so it is not processed out-of-order again. The setting of flag *OutofOrderOnce* is shown at line 16 of Figure 6.5 as the last ordered concurrent-access-pair associated with a backtracking point is issued for testing. The checking of this flag is carried out for each backtracking point on the worklist as shown at lines 8 and 12 in Figure 6.5. A marked backtracking-point is processed without any limitation on the number of fulfillments when it is encountered again.

The controlled derivation and evaluation of an selected ordered concurrent-access-pair is possible because the value-schedule-based testing has full control over the derivation and testing of value schedules. In particular, it controls the order in which value-schedule-relevant accesses are executed and can monitor the execution to detect when an interleaving can fulfill its intended order. Using these mechanisms, the value-schedule-based testing offers the necessary control to exploit the benefit of the out-of-order testing while reducing the side effects.

For example, instead of processing the ordered concurrent-access-pairs corresponding to the monitor acquisition on *accY* from the backtracking point *bpt*3 in Figure 6.4 due to the depth-first search, the out-of-order processing starts fulfilling the ordered concurrent-access-pairs corresponding to the write of *accX.val* at line 3 by $Thread1$ at *bpt*2 as shown in Figure 6.4. The fulfillment process for the ordered concurrent-access-pair (t2, run.(r(accX.val), line 3)) → (t1, run.(w(accX.val), line 3)) starts by computing its fulfilling interleavings over the CFG of $Thread2$ as discussed in Chapter 5. Clearly, this fulfilling interleaving is feasible. Moreover, the execution of this fulfilling interleaving advances the execution to a new program state. To evaluate the effects of this partial

value-schedule, the program is directed to finish sequentially from this new program state. This execution triggers the assertion because the read of $accX.val$ in $Thread2$ misses the value updated by $Thread1$.

At this point, a user can choose to stop further testing and fix the bug or continue testing. If testing continues, the out-of-order processing continues by selecting the ordered concurrent-access-pair (t2, run.(w(accX.val), line 3)) → (t1, run.(w(accX.val), line 3)) next from backtracking point *bpt*2. The fulfilling interleaving of this ordered concurrent-access-pair is computed and dynamically checked. The "sequential" run from the resulting program state also triggers the assertion. After all ordered concurrent-access-pairs associated with the current backtracking point are evaluated, the ordered concurrent-access-pairs corresponding to the read of $accX.val$ at the line 3 of $Thread1$ as shown by *bpt*1 is processed out-of-order. Note that out-of-order processing from this backtracking point does not trigger the assertion. After all of the selected prioritized backtracking points are processed out-of-order, testing resumes using normal backtracking along the depth-first path from backtracking point *bpt*3 in Figure 6.4.

Discussion

Backtracking of accesses to shared variables encountered during testing is commonly used by many existing testing techniques [72, 25, 30] to reduce memory overhead during testing. The contribution of value-schedule-based testing is its ability to make use of the incremental accessibility-based-POR to verify the validity of ordered concurrent-access-pairs incrementally and perform prioritized testing. As discussed in Chapter 5, a pure dynamic-based testing tool, such as [72, 25], cannot easily perform such testing because it does not know the conflicting accesses of a particular access to a shared variable when attempting an interleaving. Therefore, it is difficult to evaluate concurrent behaviours between a given access and its conflicting accesses as thoroughly as the value-schedule-based approach. Even for tools that use static alias-information to locate a permutation [30], out-of-order testing can turn depth-first testing into breadth-first testing if it cannot determine when an interleaving's goal is achieved to allow the tool to return the normal flow of testing. Moreover, if a statically-located concurrent-conflicting-access is not reachable at runtime, then out-of-order testing may increase testing time by running irrelevant interleavings. The ability to know all conflicting accesses and keep track of the fulfillment process allows value-schedule-based testing to perform the out-of-order testing of some concurrent behaviours while preserving the benefit of low memory consumption offered by depth-first testing.

To summarize, the incremental value-schedule-based testing can dynamically perform accessibility-based POR to prune out superfluous interleavings that cannot be detected in other dynamic testing-tools. The prioritized testing makes use of the accessibility-based

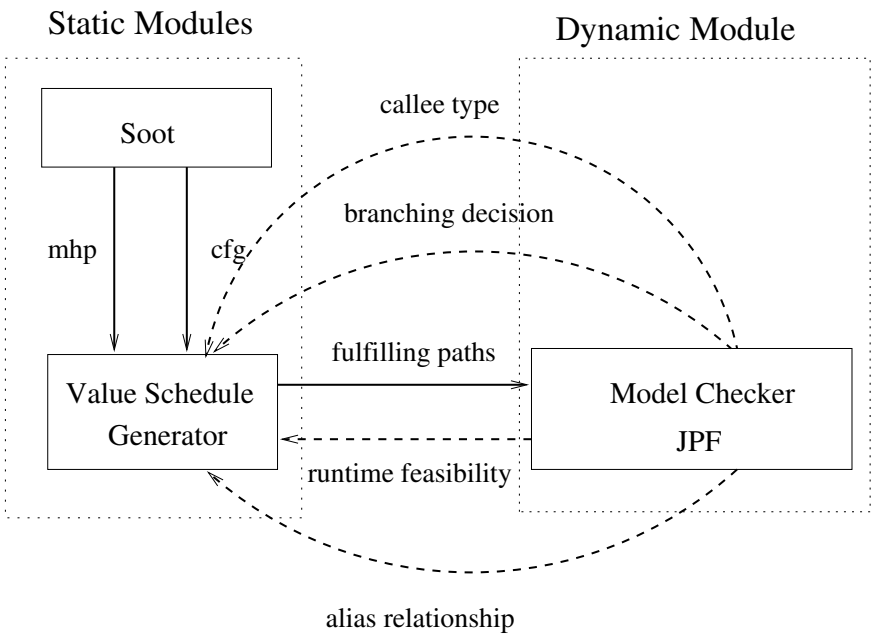


Figure 6.6: Combining Static and Dynamic Information

POR to facilitate the detection of the most likely concurrency bugs, while minimizing the possible negative side-effects of out-of-order testing. In the previous section, the emphasis was placed on the incremental derivation and testing of a value schedule. However, the value-schedule-based approach still suffers from imprecision caused by limitations in static analysis. For example, multiple fulfilling interleavings are computed for an ordered concurrent-access-pair but only one of them is feasible. Moreover, concurrent accesses formed by superfluous alias-relationships may still cause the value-schedule-based approach to fulfill superfluous value-schedules. In the next section, dynamic information collected during testing is used to refine the coarse-grained statically-derived information.

6.2 Combining Static and Dynamic Information

In the naive implementation of value-schedule-based testings presented in Chapter 5, only static information is used to guide the dynamic testing. However, due to the coarse-grained nature of static analysis, the guidance provided by the static derivation is imprecise. For example, a partial value-schedule may be further extended even if it includes infeasible ordered concurrent-access-pairs. Moreover, more than one fulfilling interleaving is computed for an ordered concurrent-access-pair but at most one of them is feasible at runtime. Essentially, the dynamic information only serves to differentiate the correctness and feasibility of value schedules after all potentially-relevant fulfilling-interleavings are statically computed.

On the other hand, in the incremental technique, dynamic information such as the feasibility of a partial-fulfilling interleaving for an ordered concurrent-access-pair can be used by the static derivation-module to determine which fulfilling interleaving of which a partial value-schedule should be further derived to guide testing. In this collaboration scheme, the static and dynamic-testing techniques are tightly coupled to each other. The static technique incrementally determines the next interleavings to test by using statically-computed information, while the dynamic testing determines the correctness of the static derivations to improve its precision in further derivation.

In addition, the incremental nature of the testing opens up various opportunities for the static module to make use of different kinds of dynamic information collected during the fulfillment of ordered concurrent-access-pairs (see Figure 6.6). This section introduces three types of dynamic information which improve static derivation: branching decisions, callee type from a polymorphic callsite, and the validity of aliasing relationships captured in an ordered concurrent-access-pair. The first two types of dynamic information help to reduce the number of superfluous interleavings computed, and the third type of dynamic information helps the static derivation-module to identify the concurrent access-pairs constructed based on imprecise aliasing relationships. The sample program in Figure 5.1 and its variations are used to show how each type of dynamic information improves the precision of the static-derivation tool.

6.2.1 Dynamic Branching Resolution

In the previous analysis, the processing of the backtracking point corresponding to the monitor entry at line 2 of $Thread2$ triggers the computation of a fulfilling-interleaving for the ordered concurrent-access-pair (t2, run.(monenter, line 8)) $\xrightarrow{acq}$ (t1, run.(monenter, line 5). Because of the branch statement at line 4 of $Thread2$, two fulfilling interleavings were computed. Thus, the feasibility testing, in the worst case, runs two interleavings to determine the feasible fulfilling-interleaving for the ordered concurrent-access-pair. Now, if the static derivation-module knows which branch the program takes, then it knows which CFG edge to follow and can compute only the feasible fulfilling interleaving for that ordered concurrent-access-pair. However, the precise branch taken by a branch statement cannot always be known until the statement is executed at runtime. When a fulfilling interleaving leading to a branch statement is dynamically executed, its branching target is the next-to-run instruction for that thread. The dynamic testing-tool can then feed the dynamic branching-decision back to the static derivation-module. The static derivation-module maps the next instruction to a successor node in the CFG and the search continues for the target node following that successor node.

The dynamic testing-tool used in the implementation of value-schedule-based testing is JPF, which provides a mechanism for exporting various types of runtime information

Branch Resolving

```
1  resolveBranch(CFGNode pred, INTERLEAVING interleaving)
2      exec(interleaving);
3      programCounter = extractProgramCounter();
4      foreach succ of successors(pred)
5          if (succ.offset == programCounter)
6              interleaving.append(succ);
```

Figure 6.7: Overview of Branch Resolving

from the underlying JVM. When the CFG traversal continues from a node that has more than one successor as shown at line 13 of Figure 6.1, *resolveBranch*() is called, (see Figure 6.7). The partial fulfilling-interleaving and the CFG node corresponding to the branch statement are passed into *resolveBranch*(). The existing partial fulfilling interleaving is executed by the dynamic-testing module. Then, the program counter is extracted in the form of its bytecode offset. Because the *Soot* package keeps track of the corresponding bytecode offset of each CFG node in the original class-file, the mapping of the next instruction after the branch and its CFG node is accomplished by comparing their bytecode offsets. The CFG node for the successor instruction is appended to the partial fulfilling-interleavings.

Reachability Refinement

During the computation of a fulfilling interleaving for an ordered concurrent-access-pair, every CFG node encountered during the search is added to the fulfilling interleaving. An interleaving is only abandoned if an access cannot be reached after the CFG is exhaustively traversed. However, such an interleaving may grow very long before it is abandoned. Moreover, if the traversal encounters many branch statements, the *resolveBranch*() function is called frequently and slows down the overall testing process. Thus, it is important to precisely determine the reachability of the target access and abandon the traversal as early as possible. The solution I implemented is to use the dynamic branching-decision obtained during the traversal to incrementally refine the reachability status of the target access as captured by the call to the *reachability*() procedure at line 15 and 33 of Figure 6.1.

Before the fulfilling process for an ordered concurrent-access-pair is started after a branch is resolved, a flow-sensitive reachability-analysis is performed on the owner thread of that target access to determine the reachability of the target access from the next node to run. The flow-sensitive analysis summarizes the reachability result along different control flows to produce a safe estimate. If the target node is not reachable, then the

```
1 if (condition) // outer if
2     if (condition) //inner if
3         //target
```

Figure 6.8: Branching over Nested Loops

fulfilling traversal does not start. The result of the reachability analysis is saved in a hashtable that acts as a cache for future queries. The key of a HashMap entry is a pair of CFG nodes (n_s, n_t) from the owner thread of the target access. n_s is the starting CFG node of the analysis, and n_t is the target access-node used in the reachability analysis. The value of the entry is the result of the analysis.

In this static-reachability analysis, it is possible the target access is determined to be reachable but not accessible at runtime. For example, a target access might be enclosed by two *if* clauses as shown in Figure 6.8. Assume the *true* branch is taken for the first outer *if* clause, and the false branch is taken for the inner *if* clause. A reachability analysis from the outer *if* reports the target access to be reachable without the branch decision for the inner *if* statement. However, the target access node is only determined to be inaccessible after the reachability analysis is performed on the immediately enclosing *if* statement. This process shows how a reachability analysis is incrementally turned into an accessibility analysis by performing branch resolving on each branch node encountered during CFG traversal.

Concurrent Access-Pairs with Multiple Instances

Until now, an assumption made during the derivation process is that every value-schedule-relevant access computed by the initial concurrent access-pair generation is distinct. However, this is generally not the case for a real program. For example, the same access may be enclosed by a loop. Thus, an instance of an ordered concurrent-access-pairs can be formed for the different iterations of the loop. Moreover, because the concurrent-access-pair computation discussed in Chapter 4 only uses one level of context-sensitivity, it cannot distinguish accesses with more than one level of call sensitivity. Therefore, a statically-computed concurrent access-pair may correspond to multiple instances of a concurrent access-pair at runtime.

The processing of an ordered concurrent-access-pair that may have multiple runtime instances is presented using the branch resolution and reachability refinement processes for the new sample programs shown in Figures 6.9 and 6.10. In both examples, the accesses to $accX.val$ in $Thread1$ are enclosed in a loop that iterates twice; in the second example, the accesses to $accX.val$ in $Thread2$ are enclosed in a loop. Thus, an ordered concurrent-access-pair that involves conflicting accesses on $accX.val$ from $Thread1$ or

Thread 1

```
1 run(){
2  // two instances of trigger
3  for(int i=0; i<2; i++){
4    accX.val -= Account.amt;
5  }
6 }
```

Thread 2

```
1 run(){
2  // one conflicting access
3  accX.val += Account.amt;
4 }
```

Figure 6.9: Multiple Instances of Target Access

Thread 1

```
1 run(){
2  // two instances of trigger
3  for (int i=0; i<2; i++){ {
4   accX.val -= Account.amt;
5  }
6 }
```

Thread 2

```
1 run(){
2  // two conflicting accesses
3  for (int i=0; i<2; i++){
4   accX.val += Account.amt;
5  }
6 }
```

Figure 6.10: Multiple Instances of Triggering Access

*Thread*2 may have more than one instance. Now, consider the case of the sample program in Figure 6.9, in which only *Thread*1 contains accesses with multiple instances. This example illustrates how the value-schedule-based technique incrementally derives different instances of an ordered concurrent-access-pair caused by the existence of multiple instances of the triggering access. When the write to *accX.val* at line 4 of *Thread*1 is encountered in the traversal of its CFG, a backtracking point, *bpt*1, is created with the encountered write access to *accX.val* as its triggering access as shown in Figure 6.11. A direct extension of this backtracking point for the write to *accX.val* at line 3 of *Thread*1 completes the fulfillment of the ordered concurrent-access-pair (t1, run.(w(accX.val), line 4)) → (t2, run.(r(accX.val), line 3)). This ordered concurrent-access-pair implies that the first write to *accX.val* at line 4 of *Thread*1 happens before the read of *accX.val* at line 3 in *Thread*2.

By continuing the extension of the partial value-schedule resulting from the direct insertion, the traversal of the CFG of *Thread*1 reaches the end of the loop. The branch resolution technique is then used to compute the next-to-run instruction following the end of the first loop iteration. The branch statement at the end of the loop brings the CFG traversal back to the second iteration of the loop and reaches the (t1, run.(w(accX.val),

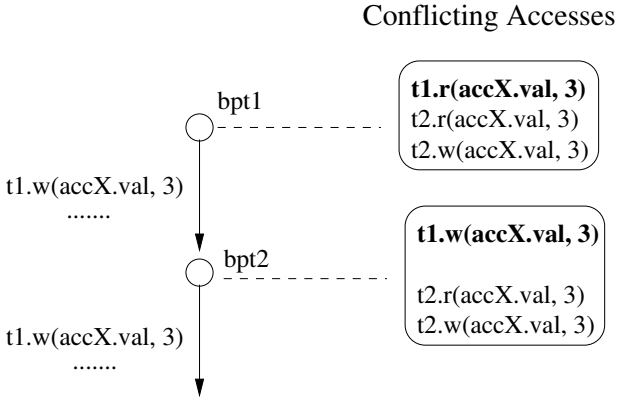


Figure 6.11: Backtracking Points for Sample Code in Figure 6.9

line 4)), and generates backtracking point *bpt*2 in Figure 6.11. The processing of *bpt*2 fulfills the ordered concurrent-access-pair (t2, run.(r(accX.val), line 3) → (t1, run.(w(accX.val), line 4)) in which the second instance of (t1, run.(w(accX.val), line 4)) is the triggering access. This instance of (t2, run.(r(accX.val), line 3)) → (t1, run.(w(accX.val), line 4)) ensures the read of $accX.val$ from $Thread2$ happen after the first instance and before the second instance of the write to $accX.val$ in $Thread1$. As the backtracking point *bpt*2 is processed, a fulfillment process for another ordered concurrent-access-pair ((t2, run.(r(accX.val), line 3)) → (t1, run.(w(accX.val), line 4)) is carried out again. This instance of the ordered concurrent-access-pair ensures both instances of the read to $accX.val$ from $Thread1$ happens after the write to $accX.val$ in $Thread2$.

This simple example shows how multiple instances of a triggering access can lead to multiple instances of ordered concurrent-access-pairs that are derived incrementally by traversing the CFG using the branch-resolution techniques. It is also necessary to show how the incremental value-schedule-based technique handles ordered concurrent-access-pairs caused by multiple instances of its target access using the sample program in Figure 6.10.

The second version of the example program has both members access the ordered concurrent-access-pair (t2, run.(r(accX.val), line 4)) → (t1, run.(w(accX.val), line 4)) in loops. Therefore, this ordered concurrent-access-pair has five instances. When the write to $accX.val$ is first encountered during the traversal of the CFG of $Thread1$, the backtracking point, *bpt*1, is created while the DFS traversal continues on (see Figure 6.12(a)). Further traversal of the CFG of $Thread1$ leads to the second iteration of the loop after dynamically resolving the branching instruction at line 3 of $Thread1$. When the write to $accX.val$ in $Thread1$ is encountered again, another backtracking point is created, *bpt*1.1. For clarity, if a backtracking point A is derived from the processing of another backtracking point B, the name of A is extended from the label of B. For example, the name *bpt*1.1 implies this backtracking point is derived from *bpt*1. Continuing the traversal of

the CFG for $Thread1$ causes the loop to exit, so the write at line 4 of $Thread1$ is no longer reachable. This deals with the case where both writes from $Thread1$ happen before those from $Thread2$.

Both the backtracking points trigger the fulfillment of the ordered concurrent-access-pair, (t2, run.(r(accX.val), line 4)) → (t1, run.(w(accX.val), line 4)), because the write to $accX.val$ from line 4 of $Thread1$ is their triggering access. Processing of these two backtracking points triggers the fulfillment of two instances of (t2, run.(r(accX.val), line 4)) → (t1, run.(w(accX.val), line 4)). The partial value-schedule where the first read of $accX.val$ in $Thread2$ happens before the second write to $accX.val$ in $Thread1$ is marked as $vs1$, and the partial value-schedule where both the first and second read of $accX.val$ in $Thread2$ happens before the second write to $accX.val$ in $Thread1$ are marked as $vs2$.

$vs1$ is generated from processing the backtracking point $bpt1.1$. At $bpt1.1$, the next instruction to run from $Thread1$ is the second instance of (t1, run.(w(accX.val), line 4)). Thus, the fulfillment of (t2, run.(r(accX.val), line 4)) → (t1, run.(w(accX.val), line 4)) is triggered. Fulfillment of this ordered concurrent-access-pair includes all value-schedule-relevant accesses from $Thread1$ in the fulfilling interleaving. During the processing of the ordered concurrent-access-pair, another backtracking point $bpt1.1.2$ is produced. The processing of $bpt1.1.2$ starts another search in $Thread2$ for (t2, run.(r(accX.val), line 4)), which enables the second instance of (t1, run.(w(accX.val), line 4)) to happen after the second instance of (t2, run.(w(accX.val), line 4)), and produce partial value-schedule $vs2$. The fulfillment of this ordered concurrent-access-pair produces another backtracking point $bpt1.1.2.2$. However, the attempt to fulfill (t2, run.(w(accX.val), line 4)) → (t1, run.(w(accX.val), line 4)) from $bpt1.1.2.2$ fails because the branch-resolution mechanism indicates $Thread2$ exits the loop after its second iteration. Then, the reachability refinement mechanism reports the target access is no longer reachable and abandons further traversal.

During the derivation, backtracking point $bpt1.1.2$ is a special type of backtracking points constructed to tackle ordered concurrent-access-pairs caused by a target access with multiple instances. This type of backtracking point has two characteristics. First, it is constructed after an ordered concurrent-access-pair is fulfilled when another instance of a target access is still considered to be reachable from the CFG. Second, the triggering access of such a backtracking point is not the target access of recursive derivation, but rather the triggering access that originally started the fulfillment process. Thus, this backtracking point is dedicated to handle another instance of a just-fulfilled ordered concurrent-access-pair.

In the case of the sample program, $bpt1.1.2$ is created because the reachability analysis determines (t2, run.(r(accX.val), line 4)) is still reachable from the CFG of $Thread2$ after the fulfillment of the previous ordered concurrent-access-pair. Moreover, this backtracking point is explicitly dedicated to fulfill an ordered concurrent-access-pair between the

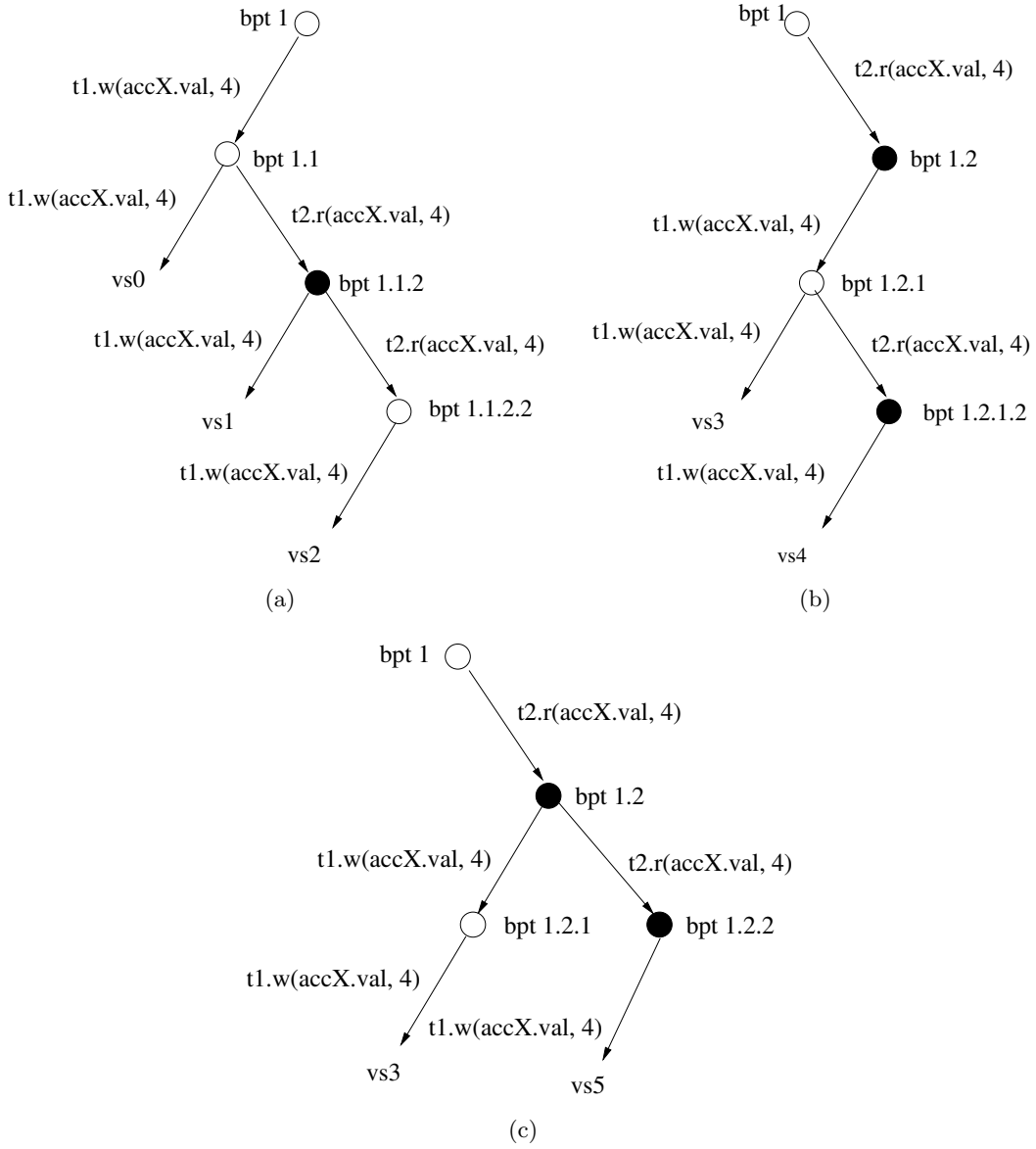


Figure 6.12: Fulfillment of Multiple Instances of an Ordered Concurrent-Access-Pair

instance of (t1, run.(w(accX.val), line 4)) that triggered the current fulfillment process and the next instance of (t2, run.(r(accX.val), line 4)). This operation is captured by the code between line 33 and line 35 in Figure 6.1. In Figure 6.12, a backtracking point created to handle multiple instances of a search target is shown as a solid circle.

The backtracking processing returns to $bpt1$ to enable the both instances of (t1, run.(w(accX.val), line 4)) to happen after the first instance of (t2, run.(r(accX.val), line 4)). The partial value-schedule resulting from this ordered concurrent-access-pair is marked as $vs3$ in Figure 6.12(b). The new backtracking points constructed in fulfilling $vs3$ are marked as $bpt1.2$ and $bpt1.2.1$. $bpt1.2$ is specially generated for the first instance of (t1, run.(w(accX.val), line 4)) because the reachability analysis determines (t2, run.(r(accX.val), line 4)) is still reachable after the first instance of (t2, run.(r(accX.val), line 4)) has been added to the interleaving leading to $bpt1.2$. $bpt1.2.1$ is generated when the second instance of (t1, run.(w(accX.val), line 4)) is encountered in the second iteration of loop in $Thread1$. The backtracked processing of $bpt1.2.1$ fulfills another instance of (t2, run.(r(accX.val), line 4)) $\rightarrow$ (t1, run.(w(accX.val), line 4)) and produces a new partial value-schedule $vs4$. The special backtracking point $bpt1.2.1.2$ in Figure 6.12(b) does not lead to any additional instances of its dedicated ordered concurrent-access-pair because (t2, run.(r(accX.val), line 4)) is no longer accessible.

Finally, the backtracked processing of $bpt1.2$ fulfills another instance of (t2, run.(r(accX.val), line 4)) $\rightarrow$ (t1, run.(w(accX.val), line 4)). This instance enables the second instance of (t1, run.(w(accX.val), line 4)) to happen after the second instance of (t2, run.(r(accX.val), line 4)). This instance is marked as $vs5$ in Figure 6.12(c). The derivation of $vs5$ produces another special backtracking point $bpt1.2.2$. Similar to $bpt1.1.2.2$, the branch resolution and reachability refinement mechanism show that extension of $bpt1.2.2$ does not fulfill a new instance of (t2, run.(r(accX.val), line 4)) $\rightarrow$ (t1, run.(w(accX.val), line 4)). At the end, all five instances of (t2, run.(r(accX.val), line 4)) $\rightarrow$ (t1, run.(w(accX.val), line 4)) are fulfilled.

6.2.2 Dynamic Polymorphism Call Site Resolution

In an object-oriented language, a CFG may also contain branches caused by polymorphic calls. This section discusses how branches caused by polymorphic call-sites can be resolved dynamically using the sample program from Figure 6.13, modified from the original sample program in Figure 5.1. This program introduces polymorphism by encapsulating the updates to $accX.val$ in $Thread2$ into a polymorphic method of the two types, $Manager$ and its subtype $ManagerB$. At runtime, the static $choose()$ method in the $Manager$ class uses the val field of an account to create a different concrete type, which is assigned to the variable m for updating $accX.val$ in $Thread2$. Therefore, the $m.withdraw()$ method invoked at lines 3 and 5 of $Thread2$ may invoke $withdraw()$

Manager Classes

```
1  class Manager{
2    public void withdraw(Account x){
3        x.val -= Amount.amt;
4     }
5
6    public static Manager choose(Account x){
7      if (x.val >= 0){
8         return new Manager();
9      }else{
10        return new ManagerB();
11     }
12   }
13 }
14 class ManagerB extends Manager{
15   public void withdraw(Account x){
16       println(``AccountFrozen");
17 ␣␣}
18 }
```

Thread 1

```
1 run(){
2   synchronized(accX){
3     accX.val += Amount.amt;
4   }
5   synchronized(accY){
6     accY.val -= Amount.amt;
7   }
8 }
```

Thread 2

```
1   run(){
2    Manager m = Manager.choose(accX);
3    m.withdraw(accX);
4    m = Manager.choose(accX);
5    m.withdraw(accX);
6
7    synchronized(accY){
8     accY.val -= Amount.amt;
9    }
10 }
```

Figure 6.13: Polymorphic Call Example

from either $Manager$ or $ManagerB$. Due to the coarse-grained nature of static analysis, two out-edges are produced for each call to $withdraw()$ in $Thread2$. As discussed in Chapters 2 and 5, imprecise static analysis results lead to the testing of superfluous interleavings in a model checker such as JPF [72]. In value-schedule-based testing, the superfluous call edges lead to the unnecessary traversal of CFG edges to fulfill infeasible ordered concurrent-access-pairs.

For example, consider a backtracking point corresponding to the read of $accX.val$ at line 3 of $Thread1$ with the interleaving:

t1.(acq(accX, 2)) →

The concurrent access-pair generation-module determines that this read access may form concurrent access-pairs with the write accesses to $accX.val$ reachable from both invocations of $withdraw()$ in $Thread2$ because it always considers $Manager.withdraw()$ as a possible target. Examples of such concurrent access-pairs are shown in Table 6.1. Note, the member accesses from $Thread2$ in these concurrent access-pairs are annotated with the context information that represents the different callsites of the $withdraw()$ method.

Assume the processing of this backtracking point triggers the fulfillment process for the ordered concurrent-access-pair (t2, withdraw.(w(accX.val), line 3), run:3) → (t1, run.(r(accX.val), line 3)). The fulfillment process traverses the CFG of $Thread2$ to the node corresponding to the invocation of $m.withdraw()$ at line 3. From this CFG node, there are two call edges leading to different implementations of $withdraw()$. If the fulfillment process follows both out-edges to search for the target access, then feasibility testing on the interleavings filters out the one that follows the superfluous call-edge. However, a more efficient way is to have the dynamic testing-tool execute the already-computed partial interleaving until the instruction corresponding to the invocation is loaded into the program counter. At this point, the concrete target of the call is resolved by the underlying virtual machine, and the method jump is computed as well. Thus, by accessing the dynamic information computed by the dynamic testing-tool, the static derivation-module is able to determine which implementation of $withdraw()$ is called at runtime and only traverse the CFG nodes for that implementation. Moreover, the reachability analysis is performed again after the polymorphic method-call is resolved to refine the accessibility prediction of the search target. These operations are shown in lines 14 and 15 in Figure 6.1. When $Manager.withdraw()$ is called, the reachability analysis reconfirms the reachability of the target access.

The fulfillment of this ordered concurrent-access-pair advances the testing process to the next value-schedule-relevant access, which is the write to $accX.val$ at line 3 of $Thread1$. Again, this access forms concurrent access-pairs with accesses to $accX.val$ from $Thread2$. For example, the processing of this value-schedule-relevant access eventually triggers the fulfillment of the ordered concurrent-access-pair (t2, withdraw.(w(accX.val),

Pair Type	Concurrent Access-Pairs
RW	<(t1, run.(r(accX.val), line 3)), (t2, withdraw.(w(accX.val), line 3), run:3)>
RW	<(t1, run.(r(accX.val), line 3)), (t2, withdraw.(w(accX.val), line 3), run:5)>

Table 6.1: Concurrent Access-Pairs for the Code in Figure 6.13

line 3), run:5) → (t1, run.(w(accX.val), line 3)). However, the dynamic method-resolution for the call to *withdraw*() at line 5 reveals the concrete runtime-type of the callee is $ManagerB$, which contains no access to $accX.val$. The reachability analysis after the branch resolution reports the target access is no longer reachable. Thus, no further traversal is carried out along $Thread2$. Moreover, the dynamic method-resolution for *withdraw*() shows there is no ordered concurrent-access-pair formed with the write to $accX.val$ from $Thread1$ and all conflicting accesses from $Thread2$ are actually fulfilled under this partial value-schedule.

6.2.3 Dynamic Alias-Refinement

Previous discussion assumed the alias relationships that form concurrent access-pairs remain valid at runtime. However, in practice, a concurrent access-pair can also become superfluous if the may-alias relationship that originally established the pair turns out to be false at runtime. Using a technique similar to the one for polymorphic call-site resolution, it is possible to check if two object references in a concurrent access-pair are still aliases at runtime, and abandon testing if they are not.

The dynamic alias refinement process is demonstrated using the sample program in Figure 6.14. $Thread2$ has a branch statement at line 4 to adjust which account instance is updated at line 9. A coarse-grained static analysis, which conservatively estimates the data flows of both branches, reports the object referenced by s at line 9 can be either $accX$ or $accY$. Thus, the value-schedule-based testing-tool concludes the accesses on $s.val$ in $Thread2$ form concurrent access-pairs with those accesses on $accY.val$ at line 6 in $Thread1$.

Consider processing a backtracking point corresponding to the write access to $accY.val$ at line 6 from $Thread1$, where that backtracking point contains the interleaving,

t1.(acq(accX, 2)) →
  t1.(r(accX.val, 3)) → t1.(w(accX.val, 3)) →
t1.(rel(accX, 4)) →
t1.(acq(accY, 5)) →
  t1.(r(accY.val, 6))

The derivation process now tries to fulfill the ordered concurrent-access-pair (t2, run.(r(s.val),

Thread 1

```
1  run(){
2    synchronized(accX){
3      accX.val += Account.amt;
4    }
5    synchronized(accY){
6      accY.val -= Account.amt;
7    }
8  }
```

Thread 2

```
1   run(){
2     Account s = null;
3     synchronized(accX){
4       if (accX.val >= 0){
5           s = accX;
6       }else{
7           s = accY;
8       }
9       s.val -= Account.amt;
10    }
11    synchronized(accY){
12      accY.val += Account.amt;
13    }
14  }
```

Figure 6.14: Exploiting Precise Alias Relationships - I

line 9)) → (t1, run.(w(accX.val), line 6)). Following the same fulfilling technique discussed in this chapter, a fulfilling interleaving that leads to the execution of the instruction corresponding to the read of *s.val* is computed. After this fulfilling interleaving is statically computed, it is executed and generates a new backtracking point. Ideally, an execution along *Thread*1 from this new backtracking point should complete the fulfillment in the ordered concurrent-access-pair. Moreover, this new backtracking point serves as a starting point to derive and test new value schedules. However, in the presence of superfluous alias-relationships, the validity of the alias relationship between accesses in an ordered concurrent-access-pair must be verified dynamically. At runtime, the partial value-schedules derived from a newly computed backtracking point are superfluous if the ordered concurrent-access-pair that produces this backtracking point is based on an invalid alias relationship. The validity checking is done by executing the fulfilling interleaving and comparing the precise memory-references of the instructions corresponding to the conflicting accesses defined in the concurrent access-pair. The call to the alias checking is shown at line 30 of Figure 6.1.

Obtaining the precise runtime memory-reference of the two instructions is carried out as follows. When all but the last instruction in the fulfilling interleaving are dynamically executed, the instruction in the program counter of the owner thread is the instruction corresponding to the target access, and the instruction in the program counter of the triggering thread is the access that triggered the fulfillment process. The dynamic testing-tool can export the memory-address referenced by the instruction in the program counter.

Main Thread

```
1  main(String[] args){
2     .....
3     Account[] accs = {accX, accY};
4     Thread1 t1 = new Thread1(accs[0]);
5     Thread1 t2 = new Thread2(accs[1]);
6     .....
7  }
```

Thread 1

```
1  Thread1(Account acc){
2     this.alocal = acc;
3  }
4  run(){
5     synchronized(this.alocal){
6        alocal.val += Account.amt;
7     }
8     Account alocal;
9  }
```

Thread 2

```
1  Thread2(Account acc){
2     this.alocal = acc;
3  }
4  run(){
5     synchronized(this.alocal){
6        alocal.val -= Account.amt;
7     }
8     Account alocal;
9  }
```

Figure 6.15: Exploiting Precise Alias Relationship - II

Therefore, the precise memory address referenced by the instructions corresponding to the accesses defined in an ordered concurrent-access-pair can be retrieved and compared. If they do not match, the newly computed backtracking point is abandoned.

For example, the execution of the fulfilling interleaving of the ordered concurrent-access-pair (t2, run.(w(s.val), line 9)) → (t1, run.(r(accX.val), line 3)) indicates that these two accesses are actually operating on *accX* and *accY* at runtime because *accX.val* is 10. Therefore, the new backtracking point computed from the fulfillment process of this ordered concurrent-access-pair is abandoned. In both [72, 30], the execution of the instruction corresponding to the write of *accY.val* at line 3 by *Thread*1 causes a yield to *Thread*2 because of the superfluous conflict. This yielding leads to the testing of superfluous interleavings because they provide no mechanism to check the validity of the permutation assumption.

So far, the precise alias-information reported by the dynamic testing-tool can be used to determine the superfluous instances of an ordered concurrent-access-pair. Moreover, it can be used to invalidate some concurrent access-pairs altogether, if the member accesses of that pair can be shown to reference different objects throughout the lifetime of their owner thread. Now consider the sample program in Figure 6.15, where *accX* and *accY* are

passed into $Thread1$ and $Thread2$ as an array element and assigned to field *alocal* of each thread. Because static analysis has difficulty in tracking object references across arrays, it reports that the *alocal* field in each thread may refer to the same object. Thus, concurrent accesses to the objects referenced by the *alocal* fields from both threads form concurrent access-pairs. However, any fulfillment of an ordered concurrent access-pair consisting of accesses to objects referenced by $Thread1.alocal$ and $Thread2.alocal$ invalidates the assumed alias-relationships. A flow-insensitive constant-analysis can easily determine the objects referenced by the fields $Thread1.alocal$ and $Thread2.alocal$ remain constant after the thread initialization. Therefore, it is safe to remove all statically-computed concurrent access-pairs that are formed with conflicting accesses to the objects pointed by $Thread1.alocal$ and $Thread2.alocal$. In the end, all concurrent access-pairs computed for the sample program in Figure 6.15 can be safely removed after attempting to fulfill a single value-schedule.

6.3 Completeness

In this section, the incremental algorithm presented in Figure 6.1 is shown to remove the incompleteness in the algorithm shown in Figure 5.8. Namely, the algorithm in Figure 5.8 is incomplete because it cannot handle concurrent access-pairs with multiple instances. Moreover, the incremental-derivation algorithm presented in Figure 6.1 is shown to always derive all value schedules of the program. Thus, given a sufficient amount of memory, the value-schedule-based testing implemented in the incremental derivation algorithm discovers all feasible deadlocks and race conditions in the program under a given set of inputs.

Lemma 6.3.1. *A partial value-schedule is always extended with all feasible instances of an ordered concurrent-access-pair.*

Proof. Let $a_n \rightarrow a_m$ be the next ordered concurrent-access-pair to be added to a partial value-schedule. Without loss of generality, there are three cases where an ordered concurrent-access-pair may have multiple instances:
1) a_n has k instances where $k \geq 1$; a_m has 1 instance,
2) a_m has k instances where $k \geq 1$; a_n has 1 instance,
3) both a_n and a_m have k instances where $k \geq 1$
where a_n and a_m have a single instance, the proof in Section 5.4 is sufficient.

Case 1) If a_n is encountered as the triggering access, then Rule 1 (see Section 5.3.1, page 73) is applied to every k instances of a_n. For example, applying Rule 1 on the i^{th} instance of a_n derives a partial value-schedule in which i instances of a_n happen before a_m. In the end, k instances of $a_n \rightarrow a_m$ are fulfilled.

If a_m is encountered first, Rule 2 derives a partial value-schedule where the first a_n happens before a_m by fulfilling $a_n \rightarrow a_m$ with a_m as the triggering access. If the Rule 1 is applied afterward, then one a_n is selected to happen before a_m. Then, because the reachability analysis happens after the fulfillment of the first $a_n \rightarrow a_m$ determines that a_n is still reachable, then the incremental derivation tries to fulfill the second instance of $a_n \rightarrow a_m$. . This process continues until the reachability analysis determines a_n is no longer reachable, i.e., when all k instances of a_n have been inserted before a_m. Thus, the partial value-schedule is extended with all k possible instances of $a_n \rightarrow a_m$.

Case 2) If a_n is encountered as the triggering access, then directly applying Rule 1 extends the partial value-schedule to have a_n happen before all instances of a_m. The derivation for the ordered concurrent-access-pair, in which a_n only happens before the k^{th} instance of a_m, is shown.

When a_n is encountered, if an ordered concurrent-access-pair, $a_m \rightarrow a_n$, is derived and added to the partial value-schedule first by applying Rule 2, then directly applying Rule 1 after this fulfillment derives a partial value-schedule in which a_n happens before all but the first instance of a_m. Thus, the k^{th} instance of $a_n \rightarrow a_m$ can be fulfilled by applying Rule 2 on a_n to derive $a_m \rightarrow a_n$ k times, followed by applying Rule 1 on a_n. Every fulfillment of $a_m \rightarrow a_n$ produces a partial value-schedule in which the number of a_m happens after a_n is reduced by one. As the incremental derivation fulfills $a_m \rightarrow a_n$, where a_m is the second last instance, the direct insertion of a_n afterward derives a partial schedule in which a_n happens before only the last instance of a_m.

On the other hand, if a_m is encountered as the triggering access, every direct insertion of a_m by Rule 1 reduces an instance of a_m that happens after a_n. Assuming there are k instances of a_m, applying Rule 2 on the i^{th} instance of a_m derives a partial value-schedule in which a_n happens before $(k - i + 1)$ instances of a_n.

In the end, the partial value-schedule can be extended with k instances of $a_n \rightarrow a_m$.

Case 3) Proof by contradiction.

Assume the incremental derivation process fails to extend a partial value-schedule with a feasible instance of $a_{ni} \rightarrow a_{mj}$, where a_{ni} and a_{mj} are the i^{th} and j^{th} instance of a_n and a_m respectively. This assumption is shown to lead to a contradiction.

Assume a_{ni} is the triggering access. As shown in Case 1), for a given value-schedule relevant-access, a_{ni}, the incremental derivation fulfills all possible instances of $a_{ni} \rightarrow a_m$, which includes $a_{ni} \rightarrow a_{mj}$, if a_{mj} is still reachable following the current partial value-schedule. Thus, a contradiction arises.

Assume a_{mj} is the triggering access. As shown in Case 2), for a given value-schedule relevant access, a_{mj}, the incremental derivation fulfills all possible instances of $a_n \rightarrow a_{mj}$, which includes $a_{ni} \rightarrow a_{mj}$, if a_{ni} is still reachable following current partial value-schedule. Thus, a contradiction arises.

Thus, a partial value-schedule is always extended with all feasible instances of an ordered concurrent-access-pair. □

Lemma 6.3.2. *Dynamic branch-resolution, polymorphic callsite-resolution and alias pruning do not prevent a feasible ordered concurrent-access-pair from being added to a partial value-schedule.*

Proof. The dynamic branch and polymorphic callsite resolution only prunes out infeasible paths. Thus, no feasible ordered concurrent-access-pair can be fulfilled along an infeasible path.

The alias-pruning only abandons the extension of superfluous partial value-schedules that try to fulfill ordered concurrent-access-pairs based on a false alias relationship. Thus, alias-pruning does not prevent the extension of a feasible ordered concurrent-access-pair. □

Theorem 6.3.3. *Given a program P, the incremental derivation-algorithm derives and tests all feasible value schedules.*

Proof. The proof in Section 5.4 shows that the derivation algorithm in Figure 5.8 always derives all value-schedules given a distinct set of concurrent access-pairs of the program. In other words, the derivation algorithm, namely Rule 1 and 2, ensures that all value-schedules of the program are derived if all distinct concurrent access-pairs associated with each value-schedule-relevant access are present during derivation. As shown in Lemma 6.3.1, the incremental derivation always derives all feasible instances of an ordered concurrent-access-pair. Thus, when processing a value-schedule relevant access, all distinct instances of concurrent access-pairs are available for further derivation. Rules 1 and 2 derive and fulfill all ordered concurrent access-pairs from those concurrent access-pairs according to the proof in Section 5.4.

Moreover, the incremental algorithm differs from that in Figure 5.8 by introducing runtime pruning of superfluous interleavings and dynamic derivation of feasible instances of an ordered concurrent-access-pair. The same derivation rules are still applied to each value-schedule relevant access. As shown in Lemma 6.3.2, the pruning process does not disable the extension of a feasible partial value-schedule.

Thus, incremental derivation algorithm always derives all possible value schedules of the program. □

6.4 Evaluation

An evaluation of the new practical implementation for value-schedule-based testing is presented by testing the programs listed in Table 3.2 of Chapter 3. Moreover, these

results are compared with the results obtained from running the same test suites on JPF, and analyzed to demonstrate strengthens and limitations of the value-schedule-based testing technique.

6.4.1 Setup and Test Suite

JPF 4.1 is compared with my implementation built using JPF 3.1.2 because this work began before JPF 4.1 was released. JPF 4.1 incorporates better state-matching mechanisms, such as canonical heap symmetry and faster state-hashing. During the test, JPF 4.1 was run with all default properties enabled, which includes DFS and on-the-fly POR. For both implementations, the Java heap-size is set to 2GB. All tests are performed using a Java 1.5 runtime on a Dual Core AMD Opteron(tm) Processor 885 with a 1.0 GHz CPU and 16GB of memory.

The test programs used are listed in Table 3.2 of Chapter 3. Most of these test programs contain complicated control flows. Branch statements and loops are present in all of the test cases. Some test cases, such as *account-subtype*, make use of polymorphic calls. The prioritized testing only had an effect for the 14 test cases annotated with *race* or *barging* as the source of error in Table 3.2. Finally, aliasing among shared variables is present in most of the programs.

6.4.2 Results and Analysis

The results are summarized in four tables capturing the relevant data points: *New States* (Table 6.2), *Instructions* (Table 6.3), *Overall time* (Table 6.4), and *MC time* (Table 6.5). *New States* and *Instructions* measure the number of program states generated and the number of bytecodes executed by the model checker during the exploration before encountering an error. Note that *New States* are stored in JPF for the purpose of stating matching. A technique that generates fewer program states and executes fewer bytecodes before finding an error is considered better in tackling the program state-space explosion problem. The *Overall time* for JPF 4.1 measures the time taken by the model checker to uncover an error, while the *Overall time* for the value-schedule-based testing includes the time spent on performing initial MHP pair generation and the time spent exploring the program state-space using collaborating static and dynamic analysis. The time spent by value-schedule-based testing excluding the initial static-MHP-pair computation is presented as the *MC time*. For each test case, data points collected from value-schedule-based testing are headed by *value schedule*, while results from JPF 4.1 are headed by *jpf 4.1*. The test cases from *Kernel*, *Real*, and *Benchmark* groups are separated with double-lines in each table. For each data point, the improvement is computed as follows:

$$Improvement\ Ratio = \frac{JPF4.1}{value\text{-}schedule\text{-}based\ testing}$$

Program States and Instructions

As shown in Table 6.2, the value-schedule-based testing creates fewer new states than JPF 4.1 for all test programs. On average, the improvement ratio for the program state-space is 33.89 for the *Kernel* group, 118.89 for the *Real* group, 59.46 for the *Benchmark* group. These savings are expected because the value-schedule-based testing uses a program-state based on more precise information than JPF. Recall that the value-schedule-based testing produces a cached program state for backtracking if that access has statically-reachable conflicting-accesses in other threads from the current program point. JPF produces program states for all accesses to variables that can be reached by multiple threads from the heap. As discussed earlier, reachability from the heap by multiple threads does not imply an object is accessed by multiple threads. The accessibility-based POR provided by value-schedule-based testing handles such cases. For example, in the *diningphilospher* test cases, every $fork$ is accessed by at least two threads. Therefore, every access to a $fork$ always triggers $n-1$ permutations where n is the number of philosophers. However, through the computation of ordered concurrent-access-pairs, the value-schedule-based testing can easily reduce the number of permutations for each fork access to 2 because every fork is only accessed by two adjacent philosophers. Moreover, after a philosopher exits, all the remaining accesses to its fork, no longer produce any new backtracking points because these forks can be exclusively held by adjacent philosophers from this point on. The accessibility-based-POR can determine such a safe state by examining accessibility of the conflicting access through the static-reachability analysis.

Several test cases in each group showed exceptional improvements. These test cases are *raxextend* and *RW-deadlock* cases in the *Real* group, and *account-subtype* in the *Benchmark* group, and their average improvement are not included in the above averages. These test cases are marked with '*' in Tables 6.2, 6.3, 6.4 and 6.5.

For the *raxextend* test case, JPF 4.1 times out after an hour. This test case poses difficulties for JPF4.1 because it dispatches a large number of threads and only two of them contain conflicting *RW* pairs that can produce errors. JPF4.1 spends a large amount of time trying out permutations on monitor acquisitions that do not lead to any fruitful results. Prioritized testing tests the conflicting *RW* pairs first. A similar situation happens in *RW-deadlock* and *account-subtype*. In the *RW-deadlock*, there are four threads: two readers and two writers. The data race is caused by conflicting *RW* accesses on a reader and a writer, while the permutations caused by *MM* pairs among writers and readers are benign. In the *account-subtype* test case, there is a large number of different types of conflicting accesses as indicated in Table 4.4, both *RW* and *MM*. Only threads that access some types of accounts cause data races. In all three cases, the value-schedule-based testing outperforms JPF4.1 by its ability to narrow down the testing of relevant permutations among relevant threads and prioritize their testing.

Fewer program states implies fewer interleavings are actually explored. As seen in Table 6.3, the improvement in instructions bytecode executed by the model checker is strongly correlated to that of program states. On average, the improvement ratio for instruction execution is 8.90 for *Kernel* group,of 50.23 for *Real* group, and 58.48 for *Benchmark* group. Note that all these averages have excluded the exceptional test-cases.

Running Time

The improvements in program-state and instruction execution come at the cost of doing initial static analysis, and runtime value-schedule derivation. The latter includes the computation of fulfilling interleaving to perform accessibility-base POR. As shown in Table 6.4, many test cases report improvement ratio with value less than 1, which indicates a slow down in overall testing time.

In total, only 6 out of 23 test cases achieve speedup. On average, the improvement ratio for the overall (real time) speedup is 0.4 for the *Kernel* group, 1.33 for the *Real* group, and 3.11 for the *Benchmark* group, with the exceptional test-cases excluded. A closer look reveals that many programs in the test suite require less than 5 seconds of testing time by JPF4.1. For these programs, the cost of the initial static analysis cannot be made up by a decrease in dynamic analysis-time. However, Table 6.5 shows that when the initial time spent on static analysis is excluded, the number of test cases achieving speedup increases to 10 in the *Kernel* group, 5 in the *Real* group, and 4 in the *Benchmark* group. Overall, 19 out of 23 test cases report speedup if the initial static analysis time is excluded. The improvement ratio on the running time for the *Kernel*, *Real* and *Benchmark* group rises to 4.44, 6.24 and 4.19, respectively. Despite the overhead, the value-schedule-based testing reduces the dynamic analysis time for several programs even with the initial static-analysis included. These programs are significantly larger and more complicated, and demonstrates that initial static-analysis does pay off as the accessibility-based POR prunes out more superfluous interleavings in these programs.

Another observation is that the quality of the initial static-analysis does have a significant effect on the testing. Some of the test cases had a slowdown largely because of the imprecision of the initial static-analysis. A specific example is the *daisy* program from the *Real* group, which runs 5 times slower using the value-schedule-based technique. Excluding the initial static-analysis, the dynamic analysis still takes about 3 times longer than JPF 4.1. This slowdown is mainly due to a pair of superfluous concurrent conflicting read/write accesses that are enclosed in a loop that write to an array. The concurrent read/writes actually happen on different elements of the array. Because the array escapes the thread scope, so does all of its elements. Thus, every time an array-element access is encountered in one thread, value-schedule-based testing tries to fulfill a new instance of a superfluous ordered concurrent access-pair. Although the dynamic alias-checking

	value-schedule	**jpf4.1**	**Improvement**
twostage	7	82	11.71
wronglock	5	42	8.40
producerconsumer	30	121	4.03
blockbarrier	24	896	37.33
reorder	16	722	45.13
deadlock.d1	4	70	17.50
deadlock.d2	8	24	3.00
diningphilosopher	367	1927	5.25
losenotify	7	13	1.86
clean	3	614	204.67
nestedmonitor	19	33	1.74
Average			33.89
alarmclock	48	987	20.56
$*$ raxextend	11	TO	
daisy	6477	50222	7.75
$*$ RW-deadlock	28	65681652	2345773.29
RW-exception	6090	439092	72.10
replicatedworker	1206	614842	509.82
boundedbuffer	71876	344720	4.80
Average			118.89
linkedlist	197	42137	213.89
piper	15	224	14.93
account-race	124	344	2.77
account-deadlock	49991	314118	6.28
$*$ account-subtype	332	2132349	6422.74
Average			59.46

Table 6.2: Comparison on States Generated

	value-schedule	**jpf4.1**	**Improvement**
twostage	695	3541	5.09
wronglock	682	2545	3.73
producerconsumer	3568	5341	1.50
blockbarrier	1760	17463	9.92
reorder	1581	14767	9.34
deadlock.d1	532	3172	5.96
deadlock.d2	528	2664	5.05
diningphilosopher	2057	56749	27.59
losenotify	556	2519	4.53
clean	575	9383	16.32
nestedmonitor	677	2690	3.97
Average			8.90
alarmclock	2207	18329	8.30
$*$ raxextend	3843756	TO	
daisy	411742	11792715	28.64
$*$ RW-deadlock	125847	1390833819	11051.78
RW-exception	1555447	43843944	28.19
replicatedworker	184464	34381953	186.39
boundedbuffer	3130349	24847425	7.94
Average			50.23
linkedlist	299293	67193304	224.51
piper	7539	8741	1.16
account-race	12388	10310	0.83
account-deadlock	1119633	8308060	7.42
$*$ account-subtype	17699	68951390	3895.78
Average			58.48

Table 6.3: Comparison on number of Instructions Executed

	value-schedule	**jpf4.1**	**Improvement**
twostage	5.50	1.00	0.18
wronglock	5.01	1.00	0.20
producerconsumer	24.11	3.00	0.12
blockbarrier	8.50	4.00	0.47
reorder	5.22	3.00	0.57
deadlock.d1	4.95	1.00	0.20
deadlock.d2	4.33	1.00	0.23
diningphilosopher	6.52	7.80	1.20
losenotify	4.40	1.00	0.23
clean	4.90	3.00	0.61
nestedmonitor	4.72	1.00	0.21
Average			0.4
alarmclock	17.184	3.74	0.22
∗ raxextend	85.034	TO	
daisy	89.639	21.88	0.24
∗ RW-deadlock	27.01	416538.66	15421.65
RW-exception	218.383	439.09	2.01
replicatedworker	34.63	437.15	12.62
boundedbuffer	243.665	166.82	0.68
Average			1.33
linkedlist	20.53	91.00	4.43
piper	11.80	4.41	0.41
account-race	88.83	4.09	0.05
account-deadlock	333.127	160.73	0.48
∗ account-subtype	29.575	1125.44	38.05
Average			3.11

Table 6.4: Comparison on Overall Time Taken (in seconds)

	Value-Schedule		jpf4.1	Improvement
	MHP	**MC**		MC vs. jpf 4.1
twostage	4.54	0.96	1.00	1.04
wronglock	3.72	1.29	1.00	0.78
producerconsumer	23.97	0.14	3.00	21.28
blockbarrier	7.54	0.96	4.00	4.18
reorder	3.62	1.60	3.00	1.87
deadlock.d1	4.09	0.86	1.00	1.16
deadlock.d2	3.37	0.96	1.00	1.04
diningphilosopher	5.02	1.50	7.80	5.20
losenotify	3.53	0.88	1.00	1.14
clean	4.45	0.45	3.00	6.67
nestedmonitor	3.97	0.93	1.00	1.08
Average				4.44
alarmclock	16.61	0.57	3.74	6.50
∗ raxextend	15.83	69.21	TO	
daisy	16.90	70.74	21.88	0.31
∗ RW-deadlock	26.27	0.74	416538.66	565949.27
RW-exception	27.27	191.11	439.09	2.30
replicatedworker	18.90	15.73	437.15	27.79
boundedbuffer	38.51	205.16	166.82	0.81
Average				6.24
linkedlist	9.97	10.56	91.00	8.62
piper	11.04	0.76	4.41	5.80
account-race	86.48	2.35	4.09	1.74
account-deadlock	57.83	275.30	160.73	0.58
∗ account-subtype	29.10	0.47	1125.44	2389.46
Average				4.19

Table 6.5: Comparison on Overall Time Taken on Model Checking (in seconds)

discussed earlier can determine such a pair is superfluous, it cannot safely conclude the next instance of this same pair, which consists of array accesses from the next iteration of the loop, is also superfluous. A similar situation happens in *account-deadlock* as well. As shown in Table 4.2 from Chapter 4, *account-deadlock* has 1210 concurrent access-pairs of *MM* type. However, most of these concurrent access-pairs are superfluous because these monitor acquisitions are performed on the elements of a shared array, and MHP analysis conservatively treats them as accessing the same monitor.

There are a few solutions to this problem. One is to use better static analysis that tracks aliasing across an array as in [49]. Or, the value-schedule-based testing can prioritize the testing of concurrent access-pairs, i.e., a concurrent access-pair of *must-alias* relationship is given higher priority than one based on a *may-alias* relationship.

6.4.3 Effectiveness of Techniques

Overall, the experimental results show the value-schedule-based technique does deliver improvements over the explicit model checker in many aspects, as promised in Chapter 1. The numbers of program states generated is consistently reduced in all test cases. This implies the value-schedule-based testing reduces the program state-space required for the model checking. As discussed in Chapter 2, the reduction in the program state-space implies the reduction on the memory consumption for model checkers. Moreover, the numbers of instructions (bytecodes) executed from the test programs are also consistently reduced. These savings imply the accessibility-based POR does prune out a large number of superfluous interleavings during the testing process.

Improvement on the overall running time is less significant, which is mainly due to two reasons: the initial static analysis to compute concurrent access-pairs, and accessibility-based POR at model-checking time. The initial static analysis has significant impact on smaller programs. Excluding initial static analysis, speedups are achieved in most of the test cases. This benefit shows the accessibility-based POR and prioritized testing do cut down the time spent on uncovering an error by the model checker. Even with the time spent on the initial static analysis included, several of the test cases achieved speedup. This benefit also supports the claim that the value-schedule-based testing does deliver savings in both program state and testing time, even with the extra overhead of performing accessibility-based POR.

A major limitation revealed through the experiments is that the cost of computing fulfilling interleavings to carry out the accessibility-based POR incurs significant overheads. In general, a 4 or 5 fold saving in program states and instructions only translates into a 1 or 2 fold saving in overall testing time. Moreover, if the initial static analysis produces a large number of superfluous concurrent access-pairs which cannot be pruned out by

dynamic information, the overhead incurred by accessibility-based POR can out-weight the benefits, slowing down the model-checking process.

6.5 Summary

A practical implementation of value-schedule-based testing is presented. This implementation derives and tests value schedules incrementally in a depth-first fashion. At any given time, only one value schedule is derived, and each value schedule extends and tests only one ordered concurrent-access-pair. Moreover, the prioritized testing within value-schedule-based testing is implemented by the controlled out-of-order testing of backtracking points that contain concurrent access-pairs more likely to expose concurrency bugs. A heuristic is proposed to evaluate the relevance of a concurrent access-pair based on its type. The main contribution is the different techniques to leverage the precise dynamic-information gathered during the incremental testing to improve the applicability and efficiency of the value-schedule-based testing in uncovering the first concurrency-related bug in the program. Namely, the dynamic-branch resolution and reachability analysis helped value-schedule-based testing disambiguate the different instances of a value-schedule-relevant access that is statically-computed with limited context-sensitivity. The dynamic resolution of polymorphic calls refined the statically-computed call-graph. The dynamic alias checking for a fulfilled ordered concurrent-access-pair pruned out superfluous value-schedules resulting from imprecise static-alias analysis. Finally, this practical implementation of value-schedule-based testing, when applied to a suite of concurrent programs, shows it is possible to achieve significant improvement over JPF in both testing time and memory consumption resulting from a reduction in the number of program states generated.

Chapter 7

Conclusions and Future Work

In this chapter, the contributions made in this thesis are presented, along with the benefits and limitations of my approach, and suggestions for future research.

7.1 Contribution

In this thesis, a new value-schedule-based testing technique is presented for uncovering concurrency-related bugs. This technique guides the testing to uncover the first concurrency-related bug in the program faster and with less memory by combining the strengths of both static and dynamic based testing-techniques when compared to the state-of-the-art dynamic testing-tool JPF.

The value-schedule-based technique computes a set of concurrent conflicting read-write pairs for shared variables to expose possible orderings that could lead to concurrency errors. Moreover, these pairs are categorized into different types based on their relationships to certain kinds of concurrency bugs. The concept of a value schedule is introduced to represent a distinct partial-ordering of accesses to shared variables for a program, consisting of the statically-computed ordered concurrent-access-pairs. The derivation and fulfillment of a value schedule is carried out by traversing a program's CFG and processing different types of concurrent access-pairs. The fulfilling interleavings of value schedules are dynamically tested to determine the feasibility and correctness properties of a computed value-schedule. A value schedule is feasible if at least one of its fulfilling interleavings runs to completion. Through feasibility testing, value-schedule-based testing aims to identify superfluous interleavings and abort further execution of them on-the-fly. Moreover, running a feasible fulfilling-interleaving checks the correctness of the program under the partial ordering corresponding to that value schedule.

A practical implementation of value-schedule-based testing is given. In this implementation, a value schedule is derived and checked incrementally, while making use of

dynamic information collected during testing. The CFG is traversed in a depth-first fashion. Moreover, the value schedule is derived and fulfilled one ordered concurrent access-pair at a time. Thus, infeasible ordered concurrent-access-pairs are identified, and no infeasible partial value-schedules are extended. Moreover, a partial value-schedule is always extended along its feasible interleaving. During backtracking, a controlled out-of-order processing is applied to backtracking points that contain ordered concurrent access-pairs that are more likely to expose concurrency bugs. This improves the efficiency of uncovering the first concurrency-related bug in the program, while preserving the memory efficiency offered by the depth-first traversal.

Moreover, the practical implementation of the value-schedule-based testing discussed makes uses of the dynamic information collected during the testing to help the static derivation-module derive and fulfill value schedules. The dynamic branch-resolution technique is used to resolve control-flow branches encountered during the CFG traversal. Then, the static reachability-analysis, with the support of the dynamic branch-resolution, helps the static derivation-module distinguish multiple instances of the same ordered concurrent-access-pair. Finally, the dynamic callsite-resolution helps the static derivation-module follow polymorphic call-edges at runtime, and the dynamic alias-refinement helps to identify concurrent access-pairs that are constructed based on imprecise alias information.

The value-schedule-based testing technique offers the following benefits. First, it only derives and tests interleavings that might cover a new partial-ordering of accesses to shared variables. Second, the accessibility-based POR enables the testing tool to identify superfluous interleavings that result from imprecise static or dynamic analysis, such as the analyses in [30] and [72]. By not executing the superfluous interleavings, the search space is reduced. Third, through incremental testing, prioritized testing is supported to first test concurrent conflicting-accesses that are more closely related to concurrency bugs. Fourth, the collaboration scheme between the static and dynamic testing opens a new way to make use of the concrete information offered by the dynamic testing-tool to refine the guidance offered by static analysis-tools. This collaboration scheme could lead to many different ways of combining static and dynamic-testing techniques. Last, experimental results confirm that value-schedule-based testing consistently reduces the number of program states needed for dynamic analysis, and often reduces the computational costs needed to uncover concurrency bugs.

In summary, the value-schedule-based testing-technique is an attempt to combine static and dynamic methods for uncovering concurrency bugs faster with less memory. It strives to use static methods to guide dynamic testing by providing value-schedule derivation, fulfillment and priority-based testing. As well, the dynamic testing verifies and improves the statically-computed guidance using feedback from the accessibility-based POR.

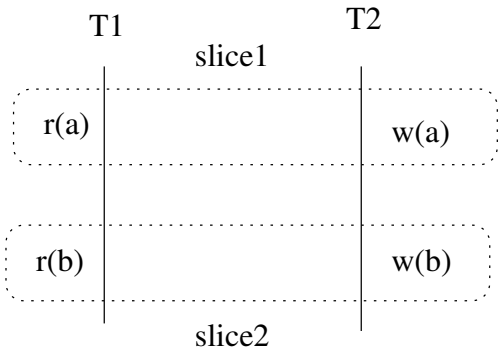


Figure 7.1: Simple Slicing-Example

7.2 Limitations

The newly developed implementation of value-schedule-based testing has the following limitations. First, it does not support the java.util.concurrent concurrent libraries supported by Java 1.5. Second, the current MHP analysis cannot handle dynamically spawned threads. For example, it cannot distinguish different instances of a thread spawned from statements enclosed in a loop. Thus, I have manually unrolled the loops in this work to expose different instances of threads.

7.3 Future Work

The first task in the future is to tackle the limitations of my implementation for value-schedule-based testing. Derivation and fulfillment rules could be introduced to deal with high-level constructs like *semaphores* and *barriers* introduced in java.util.concurrent in the Java 1.5 library. As well, the dynamic profiling could be employed to determine the number of different instances of a thread spawned from a static statement. Moreover, further research on value-schedule-based testing could proceed in two directions: 1) improving static and dynamic methods used in testing, and 2) applying the value-schedule-based testing to debugging.

First, one possible research direction for improving static analysis is to introduce more sophisticated analysis into the value-schedule-based testing. For example, better context-sensitive analysis [5] could be introduced to improve the MHP analysis and alias analysis. This information would help to improve the precision of the CFG and alias relationships among concurrent accesses. Moreover, it could reduce the ambiguity between multiple instances of the same concurrent access-pair. As well, static program-slicing [79] [37] or dependency analysis [13] could be introduced into value-schedule-based testing. By recognizing that context switches between concurrent accesses to variables in one program slice do not affect those in another slice, value-schedule-based testing could derive and test

value schedules for one program slice at a time. In the end, fewer value schedules would need to be tested. For example, consider a program with two slices spanning two threads where each slice contains accesses to a distinct shared variable as shown in Figure 7.1. A traditional model-checker tests all six possible value schedules according to Equation 1.1. However, by recognizing that permuting the execution order of concurrent accesses to a has no effect on accesses of b and vice versa, permutations of accesses on a and b could be tested independently. In the end, only four value schedules need to be derived and tested.

Second, one possible research direction is to construct a debugger for concurrent programs using the value-schedule-based technique. For example, after a concurrent program produces a runtime error during execution, a programmer may hypothesize the cause of this error to be data races on a shared variable, sv. In order to confirm this hypothesis, the programmer uses the value-schedule-based testing to generate concurrent access-pairs for accesses to the shared variable sv. Among all possible pairs, a set of "suspicious" concurrent access-pairs may be identified. Then, the value-schedule-based testing can be used to derive and test all value schedules that fulfill those suspicious concurrent access-pairs. The result of the controlled execution reveals the feasibility of those suspicious concurrent access-pairs and their effects on program correctness.

References

[1] R. Agarwal, A. Sasturkar, L. Wang, and S. D. Stoller. Optimized run-time race detection and atomicity checking using partial discovered types. In *Proc. 20th IEEE/ACM International Conference on Automated Software Engineering*, pages 233–242, 2005. 26, 33

[2] Lars Ole Andersen. Binding-time analysis and the taming of c pointers. In *Partial Evaluation and Semantics-Based Program Manipulation*, pages 47–58, 1993. 12

[3] Cyrille Artho and Armin Biere. Applying static analysis to large-scale, multi-threaded Java programs. In *ASWEC '01: Proceedings of the 13th Australian Conference on Software Engineering*, page 68, Washington, DC, USA, 2001. IEEE Computer Society. 14, 45

[4] David F. Bacon, Robert E. Strom, and Ashis Tarafdar. Guava: a dialect of Java without data races. In *Proc. 15th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 382–400, 2000. 11

[5] Marc Berndl, Ondrej Lhotak, Feng Qian, Laurie Hendren, and Navindra Umanee. Points-to analysis using bdds. In *Proc. ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, pages 103–114, 2003. 12, 49, 150

[6] Marina Biberstein, Eitan Farchi, and Shmuel Ur. Choosing among alternative pasts. In *Proc. 17th International Symposium on Parallel and Distributed Processing*, page 289, 2003. 17, 18, 59, 63, 65

[7] Bruno Blanchet. Escape analysis for object-oriented languages: application to Java. In *Proc. 14th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 20–34, 1999. 49

[8] Eric Bodden and Klaus Havelund. Racer: effective race detection using aspectj. In *ISSTA '08: Proceedings of the 2008 international symposium on Software testing and analysis*, pages 155–166, New York, NY, USA, 2008. ACM. 16

[9] Jeff Bogda and Urs Holzle. Removing unnecessary synchronization in Java. In *Proc. 14th ACM SIGPLAN conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 35–46, 1999. 12

[10] G. Brat and W. Visser. Combining static analysis and model checking for software analysis. In *Proc. 16th IEEE International Conference on Automated Software Engineering*, page 262. IEEE Computer Society, 2001. 24

[11] Peter Buhr. *Course Notes for CS343 Concurrent and Parallel Programming*. School of Computer Science, University of Waterloo, Waterloo, Ontario, Canada, Fall 2004. 1, 7

[12] R. H. Carver and Y. Lei. A general model for reachability testing of concurrent programs. In *Formal Methods and Software Engineering, Proc. 6th International Conference on Formal Engineering Methods*, page 76, 2004. 18

[13] Craig Chambers, Igor Pechtchanski, Vivek Sarkar, Mauricio J. Serrano, and Harini Srinivasan. Dependence analysis for Java. In *Languages and Compilers for Parallel Computing*, page 52. 150

[14] Jong-Deok Choi, Manish Gupta, Mauricio Serrano, Vugranam C. Sreedhar, and Sam Midkiff. Escape analysis for Java. In *Proc. 14th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, pages 1–19, 1999. 12, 49

[15] A. Cimatti, E. M. Clarke, F. Giunchiglia, and M. Roveri. Nusmv: A new symbolic model checker. *International Journal on Software Tools for Technology Transfer*, 2(4):410–425, 2000. 19

[16] S. Copty and S. Ur. Multi-threaded testing with aop is easy, and it finds bugs!. In *Proc. Euro-Par 2005*, pages 740–749, 2005. 16

[17] James C. Corbett, Matthew B. Dwyer, John Hatcliff, Shawn Laubach, Corina S. Robby Pasareanu, and Hongjun Zheng. Bandera: extracting finite-state models from Java source code. In *Proc. 22nd international conference on Software engineering*, pages 439–448, 2000. 20, 50

[18] Hyunsook Do, Sebastian G. Elbaum, and Gregg Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering: An International Journal*, 10(4):405–435, 2005. 33

[19] Matthew B. Dwyer, Sebastian Elbaum, Suzette Person, and Rahul Purandare. Parallel randomized state-space search. In *Proc. 29th International Conference on Software Engineering*, pages 3–12, Washington, DC, USA, 2007. IEEE Computer Society. 21

[20] Matthew B. Dwyer, Suzette Person, and Sebastian Elbaum. Controlling factors in evaluating path-sensitive error detection techniques. In *Proc. 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 92–104, 2006. 33

[21] Orit Edelstein, Eitan Farchi, Evgeny Goldin, Yarden Nir, Gil Ratsaby, and Shmuel Ur. Framework for testing multi-threaded Java programs. *Concurrency and Computation: Practice and Experience*, 15(3-5):485–499, 2003. 16

[22] D. Engler and K. Ashcraft. Racerx: effective, static detection of race conditions and deadlocks. In *Proc. 9th ACM Symposium on Operating Systems Principles*, pages 237–252, 2003. 14

[23] Yaniv Eytani, Klaus Havelund, Scott D. Stoller, and Shmuel Ur. Towards a framework and a benchmark for testing tools for multi-threaded programs: Research articles. *Concurr.Comput.: Pract.Exper.*, 19(3):267–279, 2007. 38

[24] C. Flanagan and S. N. Freund. Atomizer: a dynamic atomicity checker for multithreaded programs. In *Proc. 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 256–267, 2004. 16

[25] C. Flanagan and P. Godefroid. Dynamic partial-order reduction for model checking software. In *Proc. 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 110–121, 2005. 23, 62, 121

[26] Cormac Flanagan and Stephen N. Freund. Type-based race detection for Java. In *Proc. ACM SIGPLAN 2000 conference on Programming language design and implementation*, pages 219–232, 2000. 14

[27] Cormac Flanagan and Stephen N. Freund. Detecting race conditions in large programs. In *Proc. 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, pages 90–96, 2001. 14

[28] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for Java. In *Proc. ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, pages 234–245, 2002. 14

[29] J. Gait. A probe effect in concurrent programs. *Software: Practice and Experience*, 16(3):225–233, 1986. 1

[30] P. Godefroid. Model checking for programming languages using verisoft. In *Proc. 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 174–186, 1997. 20, 121, 135, 149

[31] Patrice Godefroid and Pierre Wolper. Using partial orders for the efficient verification of deadlock freedom and safety properties. In *Proc. 3rd International Workshop on Computer Aided Verification*, pages 332–342. Springer-Verlag, 1992. 21

[32] Alex Groce and Willem Visser. Model checking Java programs using structural heuristics. In *Proc. 2002 ACM SIGSOFT international symposium on Software testing and analysis*, pages 12–21, 2002. 25

[33] D. P. Helmbold and C. E. McDowell. A taxonomy of race conditions. *Journal of Parallel and Distributed Computing*, 33(2):159–164, 1996. 8

[34] C. A. R. Hoare. Monitors: An operating system structuring concept. *Communications of the ACM*, 17(10):549–557, October 1974. 37

[35] G. J. Holzmann. The model checker spin. *IEEE Transaction on Software Engineering*, 23(5):279–295, 1997. 19

[36] Gwan-Hwan Hwang, Kuo-Chung Tai, and Tin-Lu Huang. Reachability testing: An approach to testing concurrent software. In *Proceeding of the First Asia-Pacific Software Engineering Conference*, pages 246–255, Dec. 1994. 18

[37] Ganeshan Jayaraman, Venkatesh Prasad Ranganath, and John Hatcliff. Kaveri: Delivering the indus Java program slicer to eclipse. In *Proceeding of the 8th International Conference Fundamental Approaches to Software Engineering*, pages 269–272, 2005. 20, 150

[38] Donald E. Knuth. *The Art of Computer Programming, Volume 4, Fascicle 2: Generating All Tuples and Permutations (Art of Computer Programming)*. Addison-Wesley Professional, 2005. 97

[39] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978. 23, 66

[40] William Landi. Undecidability of static analysis. *ACM Letter on Programming Languages and Systems*, 1(4):323–337, 1992. 14

[41] Jaejin Lee. Compilation techniques for explicitly parallel programs, 1999. Ph. D. Thesis, Department of Computer Science, University of Illinois at Urbana-Champaign. 13

[42] Y. Lei and R. H. Carver. A new algorithm for reachability testing of concurrent programs. In *Proceeding of the 16th International Symposium on Software Reliability Engineering*, pages 346–355, 2005. 18

[43] Yu Lei and Richard H. Carver. Reachability testing of concurent programs. *IEEE Transaction on Software Engineering*, 6(32):382–403, 2006. 18

[44] Lin Li and Clark Verbrugge. A practical mhp information analysis for concurrent Java programs. *Languages and Compilers for High Performance Computing*, pages 194–208, 2005. 45, 47, 48, 54, 55

[45] Richard J. Lipton. Reduction: a method of proving properties of parallel programs. *Communications of the ACM*, 18(12):717–721, 1975. 16

[46] Kenneth Lauchlin McMillan. Symbolic model checking: an approach to the state explosion problem, 1992. Ph.D Thesis, Carnegie Mellon University. 19

[47] Ana Milanova, Atanas Rountev, and Barbara G. Ryder. Parameterized object sensitivity for points-to analysis for Java. *ACM Transactions on Software Engineering Methodology*, 14(1):1–41, 2005. 13

[48] Steven S. Muchnick. *Advanced compiler design and implementation*. Morgan Kaufmann Publishers Inc, 1997. 11

[49] M. Naik and A. Aiken. Conditional must not aliasing for static race detection. In *Proc. 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 327–338, 2007. 13, 42, 51, 146

[50] M. Naik, A. Aiken, and J. Whaley. Effective static race detection for Java. In *Proc. ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation*, pages 308–319, 2006. x, 9, 13, 42, 49

[51] Satish Narayanasamy, Zhenghao Wang, Jordan Tigani, Andrew Edwards, and Brad Calder. Automatically classifying benign and harmful data races using replay analysis. In *Proc. 2007 ACM SIGPLAN conference on Programming language design and implementation*, pages 22–31, New York, NY, USA, 2007. ACM. 15, 30

[52] G. Naumovich, G. S. Avrunin, and L. A. Clarke. An efficient algorithm for computing mhp information for concurrent Java programs. *Proc. 7th Symposium on the Foundations of Software Engineering*, pages 338–354, 1999. 33, 43, 45, 46, 47, 48, 55

[53] R. H. B. Netzer and B. P. Miller. What are race conditions?: Some issues and formalizations. *ACM Letter on Programming Languages and Systems*, 1(1):74–88, 1992. 8

[54] Diego Novillo. Analysis and optimization of explicitly parallel programs, 2000. Ph. D Thesis, Department of Computer Science, University of Alberta. 13

[55] Robert O'Callahan and Jong-Deok Choi. Hybrid dynamic data race detection. In *Proc. 9th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 167–178, 2003. 26

[56] Eli Poznianski and Assaf Schuster. Efficient on-the-fly data race detection in multithreaded c++ programs. In *Proc. 17th International Symposium on Parallel and Distributed Processing*, page 287.2. IEEE Computer Society, 2003. 16

[57] S. Qadeer and D. Wu. Kiss: keep it simple and sequential. In *Proc. ACM SIGPLAN 2004 conference on Programming Language Design and Implementation*, pages 14–24, 2004. 14

[58] G. Ramalingam. Context-sensitive synchronization-sensitive analysis is undecidable. *ACM Transactions on Programming Languages and Systems*, 22(2):416–430, 2000. 14

[59] Martin C. Rinard. Analysis of multithreaded programs. In *Proc. 8th International Symposium on Static Analysis*, pages 1–19. Springer-Verlag, 2001. 12

[60] M. Robby and J. Dwyer. Bogor: an extensible and highly-modular software model checking framework. In *Proc. 11th ACM SIGSOFT Symposium on Foundations of Software Engineering 2003*, pages 267–276, 2003 2003. 19

[61] Erik Ruf. Effective synchronization removal for Java. In *Proc. of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, pages 208–218, 2000. 12

[62] Radu Rugina and Martin Rinard. Pointer analysis for multithreaded programs. In *Proc. ACM SIGPLAN 1999 conference on Programming language design and implementation*, pages 77–90, 1999. 12

[63] Neha Rungta and Eric G. Mercer. A context-sensitive structural heuristic for guided search model checking. In *Proc. 20th IEEE/ACM international Conference on Automated Software Engineering*, pages 410–413, 2005. 26

[64] Vivek Sarkar and Barbara Simons. Parallel program graphs and their classification. In *Proc. 6th International Workshop on Languages and Compilers for Parallel Computing*, Lecture Notes in Computer Science, pages 633–655. Springer-Verlag, 1994. 12

[65] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: a dynamic data race detector for multi-threaded programs. In *Proc. 16th ACM Symposium on Operating Systems Principles*, pages 27–37, 1997. 15

[66] Koushik Sen. Effective random testing of concurrent programs. In *Proc. 22nd IEEE/ACM international conference on Automated Software Engineering*, pages 323–332, New York, NY, USA, 2007. ACM. 21

[67] Koushik Sen and Gul Agha. Cute and jcute: Concolic unit testing and explicit path model-checking tools. *Computer Aided Verification*, pages 419–423, 2006. 2

[68] Bjarne Steensgaard. Points-to analysis in almost linear time. In *Proc. 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 32–41, 1996. 2, 12

[69] Frank Tip and Jens Palsberg. Scalable propagation-based call graph construction algorithms. In *Proc. 15th ACM SIGPLAN conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 281–293, New York, NY, USA, 2000. ACM. 33

[70] Sriram K. Rajamani Jakob Rehof Yichen Xie Tony Andrews, Shaz Qadeer. Zing: A model checker for concurrent software. In *Computer Aided Verification 2004*, pages 484–487, 2004. publisher:. 20

[71] R. ValleRai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. Soot - a Java bytecode optimization framework. In *Proc. 1999 Conference of the Centre for Advanced Studies on Collaborative Research*, pages 125–135, 1999. 33

[72] W. Visser, K. Havelund, G. Brat, and S. Park. Java pathfinder - second generation of a Java model checker. In *Proceeding of Post-CAV Workshop on Advances in Verification*, July 2000. 20, 23, 62, 121, 132, 135, 149

[73] Willem Visser, Klaus Havelund, Guillaume Brat, SeungJoon Park, and Flavio Lerda. Model checking programs. *Automated Software Engineering*, 10(2):203–232, 2003. 33

[74] Willem Visser, Corina S. Pasareanu, and Sarfraz Khurshid. Test input generation with Java pathfinder. In *ISSTA '04: Proc. of the 2004 ACM SIGSOFT international symposium on Software testing and analysis*, pages 97–107, 2004. 6

[75] Christoph von Praun and Thomas R. Gross. Object race detection. In *Proc. 16th ACM SIGPLAN conference on Object Oriented Programming, Systems, Languages, and Applications*, pages 70–82, 2001. 11, 26

[76] John Whaley and Martin Rinard. Compositional pointer and escape analysis for Java programs. In *Proc. 14th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 187–206, 1999. 12

[77] Amy Williams, William Thies, and Michael D. Ernst. Static deadlock detection for Java libraries. In *ECOOP*, pages 602–629, 2005. 14

[78] Y. Yu, T. Rodeheffer, and W. Chen. Racetrack: efficient detection of data race conditions via adaptive tracking. In *Proc. 20th ACM Symposium on Operating Systems Principles*, pages 221–234, 2005. 16

[79] Xiangyu Zhang, Rajiv Gupta, and Youtao Zhang. Efficient forward computation of dynamic slices using reduced ordered binary decision diagrams. In *Proc. 26th*

International Conference on Software Engineering, pages 502–511. IEEE Computer Society, 2004. 150