Hominy Grits:

Specification and Inference of Synchronization Disciplines for Concurrent Programs

by
Benjamin P. Wood

Stephen N. Freund, Advisor

A thesis submitted in partial fulfillment
of the requirements for the
Degree of Bachelor of Arts with Honors
in Computer Science

Williams College
Williamstown, Massachusetts

May 18, 2008

Abstract

Most previous work on race condition detection and inference of synchronization disciplines (methods of synchronizing access to shared data in concurrent programs) has focused on the lockset model, using both static and dynamic analyses. However, the lockset model is restrictive and is not sufficiently expressive to capture synchronization disciplines other than mutual exclusion locks. We present the Grits synchronization discipline specification language, an expressive model for ensuring race-freedom in concurrent programs. We also present Hominy, an accompanying synchronization discipline specification inference tool for the Grits specification language. Our experience shows that Grits and Hominy are sufficiently expressive to capture nearly all synchronization disciplines.

Acknowledgments

I would like to thank my advisor, Stephen Freund, for being an excellent mentor and teacher throughout the process of writing this thesis and during my four years in the computer science department. I am also grateful to Jeannie Albrecht, my second reader, who provided helpful feedback and a fresh perspective on the thesis. Thanks also to Tom Murtagh, who shared valuable comments during the final revisions. My decision to take on this project was greatly influenced by the supportive research and academic environment the computer science department provides. Thanks are due as well to everyone outside of the department for their patience as I devoted much of my senior year at Williams to this thesis. Finally, I would like to thank my family for their continued support and encouragement during this project and in all of my endeavors.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Concurrency offers improved performance in many software applications by exploiting multiple cores or processors to increase throughput. Given the slowing improvements of raw processor speed, concurrency and parallelism are the most likely candidates to provide the performance necessary for future computational needs. Programming with concurrency is not without its problems, however, because concurrent execution comes with no guarantees about the relative ordering of operations executed by different threads. This can cause subtle defects when multiple threads manipulate shared resources. The programmer must manually synchronize a program to ensure proper ordering, and synchronization mistakes are all too common.

Race conditions, where multiple threads access or modify shared data simultaneously without synchronization, and atomicity violations, where accesses to shared data may be free of race conditions but are interleaved in unexpected ways, are two types of concurrency errors that may arise from improper synchronization of a program. Because they are CPU scheduler-dependent, these errors have non-deterministic behavior and may not be easily reproducible. This makes concurrency errors extremely difficult to debug with standard methods.

Several approaches to checking programs for race conditions have been developed, but no single method is perfect. Static analyses (which examine program source code) are guaranteed to catch all errors, but the inherent undecidability of precise static analysis forces them to be conservative, and they often produce many false alarms. Dynamic analyses (which observe a program running) have been more successful in precisely identifying errors, but because they observe only a single scheduler-dependent program trace, they are not guaranteed to catch all possible errors.

Many checkers require specifications to be added to the code. For example, the RccJava [9] static race condition checker analyzes a program annotated with a description of its *synchronization discipline* (the rules that are followed to synchronize the program). An annotation could indicate that the variable x is only accessed by a thread when that thread holds the mutual exclusion lock m, for example. RccJava checks the annotations against the program source code to verify that they are correct and that they prevent race conditions. In addition to guaranteeing race freedom, such annotations serve as human-readable documentation of a program's synchronization discipline, and aids in describing what code does. However, there are two major factors limiting the usefulness of

RccJava-style annotations.

1. The annotations only cover the use of mutual exclusion locks. They do not take into account other methods of synchronization, such as thread control, *wait* and *notify*, and volatile memory accesses. They also do not allow the lock protecting a variable to change over time during execution. Many programs that are synchronized through these alternative methods cannot be checked effectively by tools like RccJava.

2. Existing programs must be annotated before they can be checked by RccJava. Manual annotation of programs as they are being written is not difficult, but annotating existing programs which the programmer may not understand fully is time consuming and difficult.

The problem of annotating existing programs has been addressed previously by static and dynamic tools that infer RccJava-style annotations for a program, but the annotations are still limited to the protecting lock model. Also, they do not appear to be scalable to large systems [9, 2].

1.1 Goals

The first goal of this thesis is to design a more expressive specification language for describing the synchronization disciplines of programs. This language describes how a program imposes ordering between accesses to a variable and is thus free of the constraints of any specific model of synchronization. It can be extended to support arbitrarily complex specifications.

The second goal is to develop a tool to dynamically infer synchronization disciplines for programs. The specifications will be correct for observed executions, but a tool to check that the inferred specifications are valid for all executions is also necessary. That aspect remains for future work.

This project builds on an implementation of the Goldilocks [8] race condition detection plugin for RoadRunner, a dynamic program analysis framework for concurrent programs. The nature of the Goldilocks algorithm is well-suited for showing the user exactly why a given race condition occurred in the context of a specific program trace. Goldilocks determines whether an access is race-free by analyzing the synchronization events between the previous and current access and determining whether the synchronization events impose an ordering of the two accesses. If they did not impose an ordering, a race condition occurred.

Applying Goldilocks to demonstrate why a race condition occurred is simple. The set of synchronization events performed between the two conflicting accesses demonstrates the lack of ordering between the accesses. Extending Goldilocks to describe why a race condition occurred raised a second, more complex, question:

> Can Goldilocks explain why a race condition did *not* occur? In other words, can we extend Goldilocks to infer a synchronization discipline for a program?

1.2 Contributions

This thesis develops a specification language to describe, and a tool to infer synchronization disciplines. Our tool, Hominy, infers synchronization disciplines for a program by observing which synchronization events impose ordering on accesses to each variable. The results of this analysis are expressed in the Grits specification language for describing synchronization disciplines for race-free programs. Our experience running Hominy on a number of benchmark programs shows that the analysis successfully captures a range of synchronization disciplines. Additionally, our results show that the large majority of shared data in concurrent programs is protected by a few simple synchronization disciplines.

1.3 Thesis Overview

Our work is presented as follows:

Chapter 2 is an overview of related work on checking for race conditions and inferring synchronization disciplines.

Chapter 3 describes the Grits specification language, a language we have developed to specify synchronization disciplines.

Chapter 4 describes the Hominy synchronization discipline specification analysis for the Grits specification language.

Chapter 5 describes and evaluates our implementation of the Hominy analysis and our experience applying it to programs.

Chapter 6 provides an overview of our accomplishments and directions for future work.

Appendix A describes extensions to the Grits specification language and the Hominy specification inference analysis to express and infer synchronization disciplines that include concurrent read access.

Chapter 2

Background

Concurrency in software offers improved performance by running multiple *threads of control* in a single program. A program with several threads can exploit multiple cores or processors to increase throughput in a number of applications. Given the recent slowing of the growth of raw processor speed, concurrency is the most likely candidate to provide the performance necessary for future computational needs. Programming with concurrency is not without its problems, however, because concurrent execution comes with no guarantees about the relative ordering of operations executed by different threads. This can cause subtle defects when multiple threads manipulate shared resources. The programmer must manually synchronize a program to ensure proper ordering, and synchronization mistakes are common. Subtle but potent concurrency errors, such as race conditions or atomicity violations, result from these programming mistakes.

2.1 Race Conditions

A *race condition* occurs when two or more threads access or modify shared data simultaneously without synchronization. Since the concurrent model does not enforce an ordering between actions executed by different threads, the components of the two accesses may occur in any order. The ordering is scheduler-dependent and may be different on each execution of the program, rendering traditional debugging techniques ineffective in tracking down race conditions.

Race conditions in mission critical applications have caused major losses of revenue, data, and even life. The Northeast Blackout of 2003 was in part the result of a race condition in power transmission control systems that caused a localized power instability to cascade into a regional failure [14]. Race conditions may not always cause such massive failures, but even on individual systems they may be disastrous. A race condition in software for the Therac-25 radiation therapy machine caused several deaths and injuries of patients by administering overdoses of radiation [12].

2.1.1 The Happens-before Relation

The *happens-before* relation described by Lamport in [11] formalizes the ordering of events in a concurrent system and is useful for defining and reasoning about race conditions in concurrent programs. Three basic rules drive the construction of a directed graph representing the partial ordering of the events in a program execution trace by the happens-before relation.

1. *Threads define an internal ordering.* If a single thread performs two actions a and b, and a occurs before b, then $a \rightarrow b$ (a happens before b).

2. *Messages induce ordering between events in different threads.* If a thread t_1 performs the action a of sending a message m to a second thread t_2, and t_2 performs the action b of receiving m, then $a \rightarrow b$. In a multithreaded program, synchronization actions such as releasing and acquiring a lock are analogous to sending and receiving messages, respectively. They enforce ordering and ensure that two threads have a consistent view of shared data by forcing processor caches to be flushed.

3. *The happens-before relation is transitive.* If $a \rightarrow b$ and $b \rightarrow c$, then $a \rightarrow c$.

Although Lamport does not treat the creation and termination of threads explicitly, it is easy to see that if thread t_1 created a new thread t_2, the creation of t_2 by t_1 happens before any event e executed by thread t_2. Similarly, if thread t_1 waited for thread t_2 to terminate, the termination of thread t_2 happens before any events executed by thread t_1 after waiting for t_2.

Two events a and b are *concurrent* if neither of the happens-before relations $a \rightarrow b$ or $b \rightarrow a$ exists. Applied to data access, the happens-before relation can be used to define a race condition. A race condition exists on x if two access to x occur concurrently, since they may occur in any order and x will be left in an unknown state. Thus, all accesses to a shared data variable x must be ordered to guarantee the absence of races on x. Figure 2.1 shows a program with race-prone and safe accesses, where `account` is a lock, `setupBarrier` is a volatile variable whose initial value is `false`, `tr1` and `tr2` are thread-local temporary variables, and `transactions`, `savings`, and `checking` are shared variables.

- The accesses in lines 1-4 are all racy because there is no happens-before ordering imposed between the accesses by Thread 1 and Thread 2. For example, when Thread 2 reads `transactions` in line 2, it may see either the initial value of `transactions` or the value written by Thread 1 in line 3, since these two accesses may happen in either order.

- The access to the variable `checking` in lines 6 and 8 are ordered by the volatile barrier condition `setupBarrier`. Thread 2 does not proceed to line 8 until `setupBarrier` becomes true. Thread 1 accesses `checking` in line 6 before writing the value `true` to `setupBarrier`. Since volatile memory access flushes the processor's cache, Thread 2 is guaranteed to see this value on its next read of `setupBarrier`, so there is a happens-before ordering created.

- The accesses to the variable `savings` in lines 10 and 13 are ordered by the lock `account`. Since only one thread can hold the lock at a time and both Threads 1 and 2 acquire `account` before

	Thread 1	Thread 2	Status
1:	`tr1 = transactions;`		`// race`
2:		`tr2 = transactions;`	`// race`
3:	`transactions = tr1 + 1;`		`// race`
4:		`transactions = tr2 + 1;`	`// race`
5:		`while (!setupBarrier)`	
6:	`checking = 300;`		`// safe`
7:	`setupBarrier = true;`		
8:		`checking = checking - 100;`	`// safe`
9:	`acquire(account);`		
10:	`  savings = savings - 200;`		`// safe`
11:	`release(account);`		
12:		`acquire(account);`	
13:		`  savings = savings + 100;`	`// safe`
14:		`release(account);`	

Figure 2.1: Example program execution with racy and safe accesses. (Time flows down.)

accessing `savings` and release it afterwards, there is a happens-before ordering of these two accesses. Additionally, the release and acquire each force caches to be flushed, so the value written to `savings` by Thread 1 is that seen by Thread 2 when it reads `savings`.

2.2 Race Condition Checkers

A variety of tools to check programs for race conditions have been developed using both dynamic and static analyses. The following three characteristics are important in considering the effectiveness of a race condition and to what applications it is well suited.

Soundness A race condition checker is *sound* if it reports all race conditions that can occur in a program.

Precision A race condition checker is *precise* if it does not report any false positives. All errors reported by a precise checker are real errors.

Performance Dynamic race condition checkers typically introduce overhead into the execution of the target program. While this is primarily a speed penalty, memory usage must also be considered. Static checkers may be evaluated similarly, though their speed is of less urgent importance because it does not affect the execution speed of the target program.

Typically, there is a tradeoff between soundness, precision, and performance of race condition checkers. Most checkers sacrifice good precision to achieve soundness with reasonable performance. We outline below a number of existing analysis techniques.

2.2.1 Dynamic Checkers

Dynamic analyses monitor programs as they execute, rather than examining program source code, and can report errors precisely with modest computational overhead. However, full program coverage

is difficult to achieve with a dynamic checker, since the checker monitors a single program trace and may miss errors that occur on other traces. As a result, dynamic analyses are not sound. We describe three dynamic race condition checkers below.

Vector Clocks: Happens-Before

Vector clocks [11, 16] provide a mechanism to compute the happens-before relation on a concurrent trace. We attach a vector of logical clocks or timestamps vc to each thread, lock, and data variable in the program. The vector stores one logical clock for each thread i, representing the timestamp of the last step taken by thread t_i which happened before the last action involving the vector. Thus $vc_{t_i}[j]$ represents the timestamp of the last step taken by thread t_j of which thread t_i is aware. The vector clock $vc_m[j]$ for lock m represents the timestamp of the last event by thread t_j that has effected m. The vector clock $vc_x[j]$ for the variable x represents the last step of thread t_j that has affected x. Vector clocks are updated after each action according to the following rules:

- On all actions by thread t_i, increment the clock for t_i in t_i's vector:

$$vc_{t_i}[i] \coloneqq vc_{t_i}[i] + 1$$

- When thread t acquires lock m, update the vector clock vc_t by taking the maximum of vc_t and vc_m:

$$\forall h.\ vc_t[h] \coloneqq \max(vc_t[h], vc_m[h])$$

- When thread t releases lock m, update the vector clock vc_m by taking the maximum of vc_t and vc_m:

$$\forall h.\ vc_m[h] \coloneqq \max(vc_t[h], vc_m[h])$$

- When thread t accesses variable x:

  1. Check that vc_t is at least as up-to-date as vc_x for all threads, otherwise report an error:

$$\forall h.\ \text{report an error if } vc_t[h] < vc_x[h]$$

  2. Update vc_x with the clocks from vc_t:

$$\forall h.\ vc_x[h] \coloneqq \max(vc_t[h], vc_x[h])$$

The access step is used to detect race conditions. Intuitively, if the vector clock of the variable x shows newer actions than the vector clock of the accessing thread t, then other accesses to x of which thread t was not aware have occurred. Since t_j was not aware of these accesses, they did not happen before or happen after this access by t, and a race condition exists. Formal definitions of a race condition in terms of the clock vector system are presented in [13, 3].

Clock vector based race condition checkers, such as TRaDE [7] and RecPlay [17], are precise and raise no false alarms. However, as it is a dynamic analysis, the clock vector-driven approach is

limited to reasoning about the specific trace it observes and does not generalize to other traces. The cost of updating several clock vectors imposes severe performance penalties. While some tools such as RecPlay [17] take the approach of logging a program trace and then replaying it for happens-before analysis to avoid heavy run-time costs, the analysis is still computationally expensive and requires enormous resources to handle large programs.

Eraser: Protecting Locks and Locksets

The protecting lock model is a synchronization discipline that guarantees a happens-before ordering between every access to a shared data variable with the following rule:

> Every shared data variable v_i has an associated protecting lock l_i that must be acquired prior to accessing it.

The protecting locks l_i and l_j for two distinct data variables x_i and x_j need not be distinct. With the proper use of locks, the lock release by one thread and subsequent lock acquire by the next imposes a happens-before ordering on every access.

Eraser [19] is a fast dynamic race condition checker that uses a lockset algorithm to ensure that every shared data variable in a program has a protecting lock. The *lockset* of a shared data variable x at an access a_n is defined as the set of locks that have been held by the accessing thread at every access a_i to x performed thus far in the execution of the program. The thread $thread(a)$ is the thread performing a. The set $locksheld(a)$ is the set of locks held by $thread(a)$ during a. Initially $ls(x) = ls_0$, the set of all locks in the program. At access a_n,

$$ls(x) = \bigcap_{i=0}^{n} locksheld(a_i)$$

If the lockset of a data variable is ever empty, this means that there is no lock protecting that variable at all points in the program and a race condition may occur. To compute locksets incrementally and check for errors, Eraser performs the steps in Algorithm 2.1. An example of the lockset algorithm in action can be found in Figure 2.2.

Algorithm 2.1 Eraser lockset updates on access to a.

$ls(x) \coloneqq ls(x) \cap locksheld(a)$
if $ls(x) = \varnothing$ **then**
    report an error on a
end if

Eraser introduces relatively little overhead during a program execution and can generalize an execution to reason about traces other than the one observed, achieving a more sound (though not fully sound, since it is dynamic) analysis than most dynamic race condition checkers can. However, the lockset algorithm only accounts for mutual exclusion locks and does not capture all valid synchronization disciplines. As a result, Eraser raises several false alarms, which lessens its value as a debugging tool.

Program event stream	Eraser		Goldilocks		
Initial state	$ls(\texttt{transactions}) = \texttt{ls}_0$ $ls(\texttt{checking}) = \texttt{ls}_0$ $ls(\texttt{savings}) = \texttt{ls}_0$ $locksheld(t_0) = \varnothing$ $locksheld(t_1) = \varnothing$		$G(\texttt{transactions}) = \varnothing$ $G(\texttt{checking}) = \varnothing$ $G(\texttt{savings}) = \varnothing$		
$t_0 : read(\texttt{transactions})$	transactions is thread-local	OK	$G(\texttt{transactions}) = \varnothing$	OK	$G(\texttt{transactions}) \coloneqq \{t_0\}$
$t_1 : read(\texttt{transactions})$	$ls(\texttt{transactions}) \coloneqq \varnothing$	*race!*	$t_1 \notin G(\texttt{transactions})$	*race!*	$G(\texttt{transactions}) \coloneqq \{t_1\}$
$t_0 : write(\texttt{transactions})$	$ls(\texttt{transactions}) \coloneqq \varnothing$	*race!*	$t_0 \notin G(\texttt{transactions})$	*race!*	$G(\texttt{transactions}) \coloneqq \{t_0\}$
$t_1 : write(\texttt{transactions})$	$ls(\texttt{transactions}) \coloneqq \varnothing$	*race!*	$t_1 \notin G(\texttt{transactions})$	*race!*	$G(\texttt{transactions}) \coloneqq \{t_1\}$
$t_1 : read(\texttt{setupBarrier})$					
$t_0 : write(\texttt{checking})$	checking is thread-local	OK	$G(\texttt{checking}) = \varnothing$	OK	$G(\texttt{checking}) \coloneqq \{t_0\}$
$t_0 : write(\texttt{setupBarrier})$					$G(\texttt{checking}) \coloneqq \{t_0, \texttt{setupBarrier}\}$
$t_1 : read(\texttt{setupBarrier})$					$G(\texttt{checking}) \coloneqq \{t_0, \texttt{setupBarrier}, t_1\}$
$t_1 : read(\texttt{checking})$	$ls(\texttt{checking}) \coloneqq \varnothing$	*race!*	$t_1 \in G(\texttt{checking})$	OK	$G(\texttt{checking}) \coloneqq \{t_1\}$
$t_0 : acquire(\texttt{account})$	$locksheld(t_0) \coloneqq \{\texttt{account}\}$				
$t_0 : read(\texttt{savings})$	$ls(\texttt{savings}) \coloneqq \{\texttt{account}\}$	OK	$G(\texttt{savings}) = \varnothing$	OK	$G(\texttt{savings}) \coloneqq \{t_0\}$
$t_1 : write(\texttt{checking})$	$ls(\texttt{checking}) \coloneqq \varnothing$	*race!*	$t_1 \in G(\texttt{checking})$	OK	$G(\texttt{checking}) \coloneqq \{t_1\}$
$t_0 : write(\texttt{savings})$	$ls(\texttt{savings}) \coloneqq \{\texttt{account}\}$	OK	$t_0 \in G(\texttt{savings})$	OK	$G(\texttt{savings}) \coloneqq \{t_0\}$
$t_0 : release(\texttt{account})$	$locksheld(t_0) \coloneqq \varnothing$				$G(\texttt{savings}) \coloneqq \{t_0, \texttt{account}\}$
$t_1 : acquire(\texttt{account})$	$locksheld(t_1) \coloneqq \{\texttt{account}\}$				$G(\texttt{savings}) \coloneqq \{t_0, \texttt{account}, t_1\}$
$t_1 : read(\texttt{savings})$	$ls(\texttt{savings}) \coloneqq \{\texttt{account}\}$	OK	$t_1 \in G(\texttt{savings})$	OK	$G(\texttt{savings}) \coloneqq \{t_1\}$
$t_1 : write(\texttt{savings})$	$ls(\texttt{savings}) \coloneqq \{\texttt{account}\}$	OK	$t_1 \in G(\texttt{savings})$	OK	$G(\texttt{savings}) \coloneqq \{t_1\}$
$t_1 : release(\texttt{account})$	$locksheld(t_1) \coloneqq \varnothing$				...

Figure 2.2: A possible execution of the example program in Figure 2.1, monitored by Eraser and Goldilocks.

Goldilocks: Efficient Happens-Before

Goldilocks [8] is a dynamic race condition detection algorithm that checks for happens-before ordering between accesses much more efficiently than the vector clock approach without loss of precision.

For each non-volatile data variable x, Goldilocks maintains a set $G(x)$ of threads that can safely access x and locks and volatile variables that a thread can acquire or read, respectively, to gain safe access to x. At a given point in the program, a thread can safely access the variable x if the last access to x happened before all subsequent actions of that thread. On each program action a, the set $G(x)$ is updated for each data variable x according to the cases presented in Algorithm 2.2 and described below.

Case 1 handles read and write access to data variables. If a thread $t \in G(x)$, then the last access to x happens before the next action of t, so thread t can safely access x. This happens-before ordering may exist because the same thread is executing multiple accesses in sequence or because of synchronization between threads via locks (Cases 2 and 3), volatile variables (Cases 4 and 5), or creation and termination of threads (Cases 6 and 7).

Cases 2 and 3 handle synchronization performed with locks. If a thread $t \in G(x)$, then the last access to x happens before the next action of t. Therefore, if thread t then releases the lock m and another thread u acquires m, then the last access to x also happens before the next action of u. Thread u thereby gains safe access to x.

Cases 4 and 5 handle synchronization performed with read and write operations for volatile variables. If a thread $t \in G(x)$, then the last access to x happens before the next action of t. Therefore, if thread t then writes a value v to a volatile variable $\hat{y}$ and another thread u subsequently reads from $\hat{y}$, then the last access to x also happens before the next action of u. Thread u thereby gains safe access to x.

Case 6 handles the creation (forking) of threads in the program. If a thread $t \in G(x)$, then the last access to x happened before the next action of t. If thread t then creates a new thread u, the creation of u by t happens before all actions of u, so the last access to x also happens before all actions of u. Thread u thereby gains safe access to x.

Case 7 handles termination (joining) of threads in the program. If a thread $u \in G(x)$, then the last access to x happened before the next action of u. If thread t waits for thread u to terminate, the termination of thread u happens before all subsequent actions of thread t, so the last access to x happens before all subsequent actions of t. Thread t thereby gains safe access to x.

An example of Goldilocks in action can be found in Figure 2.2.

2.2.2 Static Checkers

Static checkers examine the source code of a program to find race conditions. This process is computationally expensive and must be conservative in order to be sound, rejecting some programs that work correctly in practice, due to the undecidability of precise static analyses.

RccJava [9] is a type-checker for a type system for race-free programs. It requires that each non-final field in a class be protected by a lock (*i.e.* that lock is held whenever the field is accessed). Two

Algorithm 2.2 Goldilocks set updates for program event e executed by thread t.

1. **Access:** Thread t reads or writes variable x.
   if $G(x) \neq \varnothing$ and $t \notin G(x)$ **then**
   a race condition exists on x
   end if
   $G(x) \coloneqq \{t\}$

2. **Lock release:** Thread t releases lock m.
   for all data variables x **do**
   if $t \in G(x)$ **then**
   $G(x) \coloneqq G(x) \cup \{m\}$
   end if
   end for

3. **Lock acquire:** Thread u acquires lock m.
   for all data variables x **do**
   if $m \in G(x)$ **then**
   $G(x) \coloneqq G(x) \cup \{u\}$
   end if
   end for

4. **Volatile write:** Thread t writes volatile variable $\hat{y}$.
   for all data variables x **do**
   if $t \in G(x)$ **then**
   $G(x) \coloneqq G(x) \cup \{\hat{y}\}$
   end if
   end for

5. **Volatile read:** Thread u reads volatile variable $\hat{y}$.
   for all data variables x **do**
   if $\hat{y} \in G(x)$ **then**
   $G(x) \coloneqq G(x) \cup \{u\}$
   end if
   end for

6. **Thread creation:** Thread t forks thread u.
   for all data variables x **do**
   if $t \in G(x)$ **then**
   $G(x) \coloneqq G(x) \cup \{u\}$
   end if
   end for

7. **Thread termination:** Thread t joins thread u.
   for all data variables x **do**
   if $u \in G(x)$ **then**
   $G(x) \coloneqq G(x) \cup \{t\}$
   end if
   end for

```
class Account<ghost Person p> {

  int balance = 0 guarded_by p;

  void deposit(int x) requires p {
    this.balance = this.balance + x;
  }

  void withdraw(int x) requires p {
    this.balance = this.balance - x;
  }

}


class Person {

   Account savings  = new Account<this>() guarded_by this;

   Account checking = new Account<this>() guarded_by this;

   public synchronized void save(int x) {
     savings.deposit(checking.withdraw(x));
   }

}
```

Figure 2.3: `Account` and `Person` classes with RccJava annotations.

type annotations specify these relationships. The `guarded_by` annotation on a field declaration lists the set of locks that must be held on an access to that field. The `requires` annotation on a method declaration lists the set of locks that must be held when that method is entered. Additionally, RccJava allows parameterization of classes with `ghost` variables. These ghost variables can be used as arguments to `guarded_by` and `requires` annotations in the parameterized class. Figure 2.3 shows a program with `guarded_by`, `requires`, and `ghost` variable annotations. RccJava uses a static lockset-tracking algorithm to verify that the `guarded_by` and `requires` annotations are not violated.

Boyapati and Rinard extended RccJava's type system with ownership types to make the type system more expressive [4], but this extension, like RccJava, is limited to consideration of mutual exclusion locks and to the protecting lock model.

2.2.3 Hybrid Checkers

Several hybrid checkers, combining different analyses have also been developed. O'Callahan and Choi combine happens-before and lockset dynamic race detection methods to improve performance

and error-reporting quality in [13]. Specifically, lockset race detection is run on the program in one stage. In a second stage, potential race conditions discovered by the lockset algorithm are examined with a happens-before algorithm to determine whether they are actual races. Additionally, a set of heuristics is introduced into the lockset algorithm to avoid storing redundant information for two accesses which act the same way. This approach reduces the number of false positives reported by a lockset-based approach without paying the full performance penalties of a happens-before approach.

Choi *et al* present a hybrid of static and dynamic methods in [6]. Dynamic analysis monitors a program's run-time and records data access events, eliminating redundant access information and evaluating the *weaker-than* relation, which imposes a partial ordering on accesses by the "strength" of their protection against races. Statically, the tool applies points-to analysis and control flow analysis extended with thread interactions using the program trace information recorded by the run-time monitor. The lightweight dynamic analysis adds little overhead at run time and the combined dynamic and static techniques report very few false positives.

2.3 Lock Inference

Although RccJava [9] and other static race condition checking tools may use intelligent defaults and other features such as thread-local classes and escape mechanisms to reduce the number of annotations the programmer must provide, annotation is still necessary to ensure meaningful results. Annotating large existing code bases can be tedious and time-consuming. Additionally, libraries must be handled specially or assumed to be race-free to provide any useful results on other programs. Several algorithms, both static and dynamic, have been developed to infer these annotations.

2.3.1 Static Inference

Static type inference for RccJava is NP-complete [10]. A variety of approaches have been explored, including heuristic- and constraint-based generation algorithms [10].

Houdini/rcc [1] generates a number of sets of RccJava annotations of the program and removes invalid ones repeatedly to find a fixed point of valid RccJava annotations, remembering those annotations which were correct.

Locksmith [15] uses an approach similar to locksets to infer locking relationships in C programs by building a set of constraints based on relationships between memory locations and locks. A memory location ρ is *correlated* with a lock l if and only if any thread holds l while accessing ρ at least once. If ρ is correlated with l for all accesses to ρ, then ρ is *consistently correlated* with l. Locksmith also handles many characteristics of the C language which are not present in Java.

2.3.2 Dynamic and Hybrid Inference

Agarwal et al implemented type discovery for the Parameterized Race-Free Java type system [4] in [2]. Their tool uses static and dynamic analyses to discover ownership information in addition to `guarded_by` and `requires` relations. By using ownership information, the tool can better reason about what data is shared, what data is thread-local, and when data escapes from a thread. Their

tool is effective enough to discover 98% of the requires annotations, meaning that an average of less than 2 annotations per thousand lines of code must be supplied by a programmer. However, the tool still reports some false alarms, particularly on fields which are not declared to be final, but which are treated as such in the program. The algorithm also does not infer multiple lock parameters for classes or the use of these parameters in the `guarded_by` and `requires` declarations of members of such classes.

FindLocks [18] uses an Eraser-style dynamic lockset tracking algorithm to track the set of locks that are held during all accesses to a field over the course of a program execution trace. FindLocks stores additional information about the trace to reconstruct `guarded_by` relationships effectively. After dynamically inferring RccJava types for the program, FindLocks calls on RccJava to verify the inferred types statically.

Unlike [2], FindLocks does not handle the extended ownership types of Parameterized Race-Free Java, the relations extracted from the additional information FindLocks stores are sufficient to infer all lock types for classes that must be parameterized by multiple locks. FindLocks faces two main problems. Full program coverage is hard to achieve, as in any dynamic analysis, and some properties inferred by the dynamic analysis are not expressible in RccJava's type system.

Chapter 3

Grits: Specifying and Verifying Race Freedom

In this chapter we present a model for race free concurrent programs (Section 3.1), and a small core language JG to describe the execution semantics of concurrent programs (Section 3.2). Next, we present the Grits specification language for describing how programs follow the race freedom model (Section 3.3) and the application of Grits to JG programs (Section 3.4). Finally, we propose some useful extensions to the Grits specification language (Section 3.5).

3.1 Model for Race Freedom

Our model for race freedom is based on the following observation:

> If the accesses to a variable x in a program are totally ordered, then race conditions cannot occur on x because the total ordering guarantees that all accesses to x are performed sequentially.

In other words, if every pair of distinct accesses to x is ordered by the happens-before relation (e.g. $a \rightarrow b$), then it is obvious that no two accesses to x can happen concurrently in the program. (See also [8].)

To show that a sequence of actions $a_0; a_1; a_2; ...; a_n$ is totally ordered, we show that there exists a happens-before relation $a_i \rightarrow a_{i+1}$ for all i such that $0 \leq i < n$. The transitivity of the happens-before relation dictates that if $a_i \rightarrow a_{i+1}$, then $a_i \rightarrow a_j$ for all j such that $i < j \leq n$ and $a_h \rightarrow a_{i+1}$ for all h such that $0 \leq h < i$. Therefore, the sequence of actions is totally ordered.

This happens-before model for ensuring race freedom is more flexible than the lockset model, since it can express any type of synchronization discipline. (Many synchronization primitives, not just mutual exclusion locks, can induce ordering of events in a program.) However, effective static checking of this model is not possible, since it requires temporal reasoning that cannot be precisely performed with current techniques. It is possible to check whether a program prevents races according to this model using dynamic race condition checkers such as Goldilocks or a vector clock

$$
\begin{aligned}
x, y &\in \mathit{Variable} \\
v &\in \mathit{Value} \\
\hat{x}, \hat{y} &\in \mathit{VolatileVar} \\
l, m &\in \mathit{Lock} \\
t, u, w &\in \mathit{Tid} \\
a, b &\in \mathit{Action} \qquad \text{(See Table 3.1.)}
\end{aligned}
$$

Figure 3.1: JG domains

Action	Description
$rd_t(x, v)$	Thread t reads value v from variable x.
$wr_t(x, v)$	Thread t writes value v to variable x.
$acq_t(m)$	Thread t acquires lock m.
$rel_t(m)$	Thread t releases lock m.
$rd_t(\hat{x}, v)$	Thread t reads value v from volatile variable $\hat{x}$.
$wr_t(\hat{x}, v)$	Thread t writes value v to volatile variable $\hat{x}$.
$fork_t(u)$	Thread t forks a new thread u and u starts executing.
$join_t(u)$	Thread t waits for thread u to terminate and joins u.

Table 3.1: JG actions

happens-before race condition checker, but we know of no previous work on specifying *how* a program prevents race conditions under this model.

3.2 JG Syntax and Execution Semantics

To simplify the formal development we describe our analysis for JG, a small language capturing the basic ideas of concurrent execution. This section introduces the syntax and informal semantics of a JG program and the Grits specification language for it.

A JG program's syntax (see Figure 3.1) consists of *Variable*s, *VolatileVar*s, *Value*s, *Lock*s (mutual exclusion locks), and *Tid*s (thread identifiers). *Variable*s and *VolatileVar*s are memory storage locations for *Value*s. *VolatileVar*s are variables whose accesses force a processor's cache to be flushed to main memory and therefore impose ordering between reads and writes from different threads. Threads execute *Action*s by applying the operations read (rd), write (wr), lock acquire (acq), lock release (rel), volatile read (rdv), volatile write (wrv), fork, and join to *Variable*s, *VolatileVar*, *Value*s, *Lock*s, or *Tid*s.

A JG thread is a series of actions $a_1; \ldots; a_n$ where all a_i are executed by a single thread t. The semantics of a JG program's execution is then described by a program trace $a_1; \ldots; a_m$ where steps from different threads are interleaved. (We assume two accesses cannot occur at exactly the same time.)

Some actions, such as lock acquire and release, are dependent on the state of the lock that they target. Thus $acq_t(m); rel_t(m); rel_t(m)$ is not a valid trace because a thread cannot release a lock that it does not hold. We do not expressly forbid these possibilities, but we do assume that all observed program traces are free of race conditions and obey the standard semantics for basic memory and synchronization operations:

- A thread cannot acquire a lock that is held by another thread.
- A thread cannot release a lock that it does not hold.

Wait and Notify We do not define wait and notify actions for JG because from the perspective of race detection, the "wait on m" operation is equivalent to "release m, sleep, acquire m." The sleep operation does not induce any ordering in a happens-before graph, so we can safely ignore it, reducing "wait" to a simple release/acquire pair. The notify operation does not impose any ordering in a program. It simply perturbs the scheduler to wake a waiting thread from sleep earlier. The same thread can wake and continue regardless of whether m is notified or not, as long as m is available.

3.3 Grits: Specifying Synchronization Disciplines

In this section, we present Grits, an extensible human-readable specification language for describing how a Java program prevents race conditions on each of its variables. A Grits specification for a variable x describes the *synchronization discipline* used to ensure race freedom for x. A synchronization discipline is in essence an abstraction of the sequences of events in a program that impose ordering on accesses to x. The Grits specification language captures synchronization disciplines that are commonly used in concurrent programs. All possible synchronization specifications for a variable x are described by expansions of *Discipline* in the following grammar:

$$\textit{SimpleDiscipline} ::= \texttt{thread-local} \mid \texttt{fork} \mid \texttt{join} \mid \texttt{guarded-by}\ \textit{Lock} \mid \texttt{vol}\ \textit{VolatileVar}$$
$$\textit{Discipline} ::= \textit{Discipline};\ \textit{SimpleDiscipline} \mid \textit{SimpleDiscipline}$$

The definition *SimpleDiscipline* lists the simple synchronization disciplines that are commonly used to protect a variable form race conditions. These are described below. The definition *Discipline* describes how simple disciplines may be combined to form *compound* disciplines. A compound discipline describes synchronization for traces that do not use a single simple discipline for the entirety of the program's execution. Although Grits can be extended to express arbitrarily complex access patterns, this small set of disciplines and rules for combining them is sufficient for describing most programs.

3.3.1 Simple Synchronization Disciplines

Thread-local

If a variable x is `thread-local` then it is only accessed by one thread. The variable x is obviously free of race conditions in this case, because each access must happen after the last access, as shown in the example in Figure 3.2(a). The total ordering of the actions executed by one thread serves as a total ordering of all accesses to x. For example, the following trace is consistent with the synchronization discipline `thread-local` for x because thread t is the only thread that accesses x in the trace. (The actions by each thread are shown on separate lines. Time flows left to right.)

Thread t :	$wr_t(x,1);$	$rd_t(x,1);$	$acq_t(m); rd_t(x,1)$	
Thread u :		$rd_u(y,1);$	$rel_u(m);$	

Guarded by lock

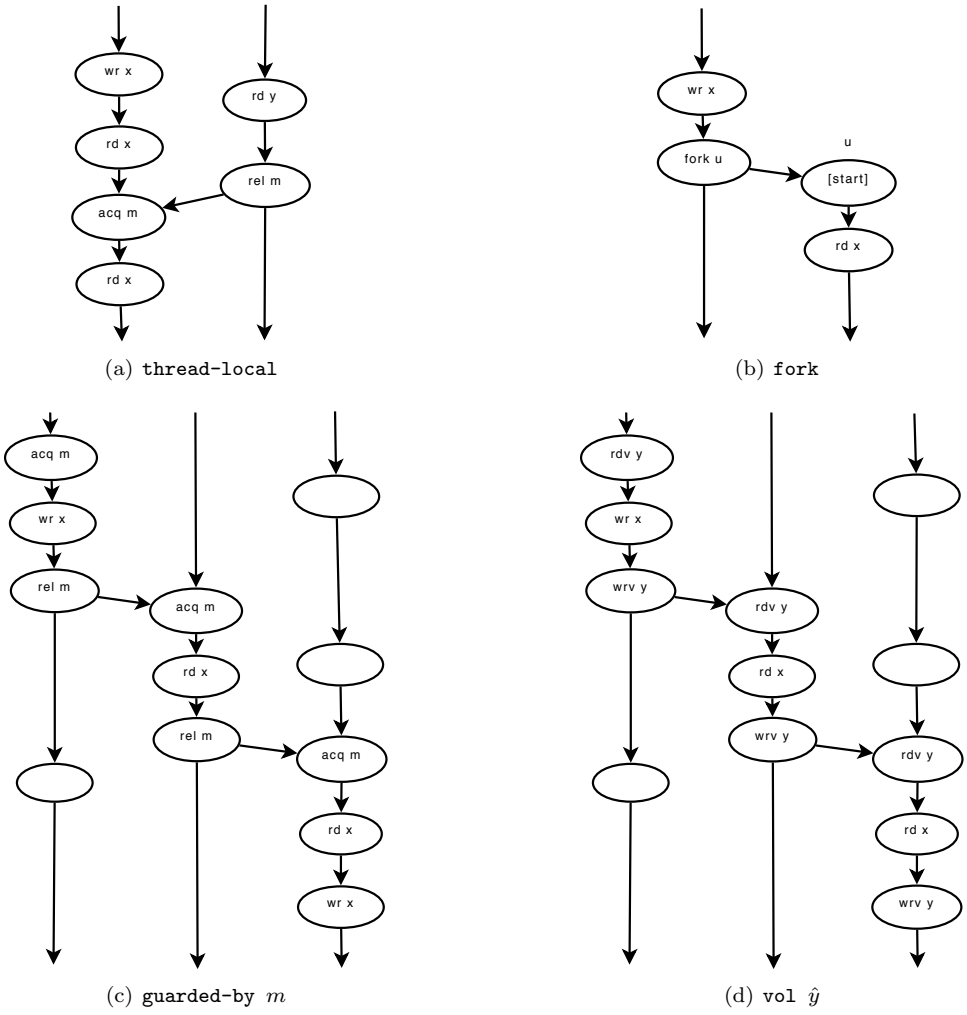
The use of mutual exclusion locks to protect variables is one of the most common synchronization disciplines used in concurrent code. A variable x that is *guarded by* a lock m is only accessed by threads while they hold the lock m. No threads can race to access x, since only one thread can hold a lock at time. Under the happens-before model, each time thread t acquires lock m, it must release m before another thread u can acquire m, creating a happens-before relation between the actions that each thread executes while holding the lock m, as shown in Figure 3.2(c). Enforcing the use of m to guard x thereby guarantees race freedom for x. For example, the following trace is consistent with the synchronization discipline `guarded-by` m:

Thread t :	$acq_t(m); rd_t(x,1); wr_t(x,2); rel_t(m);$		
Thread u :		$acq_u(m); rd_u(x,2); rel_u(m);$	
Thread w :			$acq_w(m); rd_w(x,2); rel_w(m)$

Barriers Barriers are points in a program where all threads must wait until all threads have arrived. Barriers are often implemented using monitors (though volatile variables may also serve as barriers). Since monitors protect data using locks, and the acquire and release actions on these locks induce ordering at barriers, Grits represents synchronization for a barrier that uses the monitor lock `b` in terms of lock actions as `guarded-by` b. Volatile-based barriers are described below.

Volatile condition variables

If a variable x is protected by access to a volatile variable $\hat{y}$, then $\hat{y}$ is a condition variable used to coordinate control of x. Since all caches are flushed on volatile memory operations, threads do not suffer from stale cache data when reading or writing volatile variables. In essence, volatile accesses impose ordering between reads and writes by different threads. A value v written by thread t to a volatile condition variable $\hat{y}$ is guaranteed to be observed by any thread u the next time u reads $\hat{y}$



(a) `thread-local` (b) `fork`

(c) `guarded-by` m (d) `vol` $\hat{y}$

Figure 3.2: Happens-before orderings of accesses under the simple synchronization disciplines.

(as long as a new value has not been written to $\hat{y}$ since). This property induces a happens-before ordering such that $wr_t(\hat{y}, v)$ happens before $rd_u(\hat{y}, v)$, where thread u reads the value v written to $\hat{y}$ by thread t if t executes its write to $\hat{y}$ before u reads from $\hat{u}$. This is illustrated in Figure 3.2(d) and in the following trace, which is consistent with the synchronization discipline `vol` $\hat{y}$:

$$\begin{array}{rlll} \text{Thread } t: & wr_t(x,1); wr_t(\hat{y},2); & & \\ \text{Thread } u: & & rd_u(\hat{y},2); rd_u(x,1); wr_u(\hat{x},3); & \\ \text{Thread } w: & & & rd_w(\hat{y},3); rd_w(x,1) \end{array}$$

Fork and Join

Thread management through the fork and join actions is commonly used to synchronize a switch between the `thread-local` discipline and other disciplines. The disciplines `fork` and `join` cannot stand alone. They only appear in compound synchronization disciplines, discussed in Section 3.3.2.

The fork and join actions induce happens-before orderings between access to a variable x. If a thread t accesses the variable x and then forks another thread u that proceeds to access x, the access by t happens before the access by u, as shown in Figure 3.2(b). Join works similarly. If a thread u accesses x and subsequently terminates and thread t joins u, proceeding to access x, then the access by u happens before the access by t. Together, a number of fork or join operations by the same thread can have the same effect as a barrier. In the case of a thread t consecutively joining several threads u_0 through u_n, no thread u_i can execute further actions once it has terminated and been joined by t. Thread t, on the other hand, must wait for all of the threads u_0 through u_n to terminate before it can continue. For example, the access by thread u in the trace below is consistent with the `fork` discipline:

$$\begin{array}{rll} \text{Thread } t: & wr_t(x,1); fork_t(u); & \\ \text{Thread } u: & & rd_u(x,1) \end{array}$$

3.3.2 Compound Synchronization Disciplines

A *compound synchronization discipline* is a concatenation of simple synchronization disciplines that describes a variable that changes synchronization disciplines during program execution. A compound synchronization discipline is valid if it represents a total ordering of accesses. For example, the following compound discipline is valid:

`thread-local; fork; guarded-by` m`; join; thread-local`

It describes program traces such as the one in Figure 3.3. In this trace, the first access is executed by thread t. Since it is the first access, it is not ordered from any previous access and is therefore `thread-local`. The second access ($rd_u(x, v)$) is separated from the first by a fork operation and an acquire operation on lock m. Since no thread has released m since the first access to x, this acquire operation does not have any effect on ordering the second access from the first. Thus the

$$
\begin{array}{rl}
\texttt{thread-local}; & \{\ wr_t(x,v); \\
\texttt{fork}; & \left\{ \begin{array}{l} fork_t(u); \\ acq_u(m); \\ rd_u(x,v); \end{array} \right. \\
\texttt{guarded-by}\ m; & \left\{ \begin{array}{l} rel_u(m); \\ acq_t(m); \\ wr_t(x,v); \\ rel_t(m); \\ acq_u(m); \\ rd_u(x,v); \end{array} \right. \\
\texttt{join}; & \left\{ \begin{array}{l} rel_t(m); \\ join_t(u); \\ rd_t(x,v); \end{array} \right. \\
\texttt{thread-local} & \{\ wr_t(x,v)
\end{array}
$$

Figure 3.3: A program trace consistent with a compound synchronization discipline.

second access is ordered by the operation $fork_t(u)$. The next series of accesses is `guarded-by` m. Before each of these accesses, the thread executing the last access releases the lock m and the thread executing the next access acquires m, so each access is ordered by synchronization on the lock m. The second-to-last access ($rd_t(x,v)$) is executed while thread t does not hold the lock m. This access is therefore ordered only by a join operation.

The concatenation of two synchronization disciplines to form a compound discipline is straightforward. For all disciplines c and d, where neither c nor d is `thread-local`, the discipline "$c;d$" for the variable x indicates that accesses to x were first ordered by the discipline c and then ordered by the discipline d. For `thread-local`, two additional requirements are imposed.

1. The discipline "`thread-local`; d" for the variable x, where $d \neq$ `thread-local`, describes a sequence of actions $a_1; \ldots; a_i; \ldots a_j; \ldots; a_k; \ldots; a_n$. Action a_i is the last `thread-local` access to x, where a_i is executed by thread t, and action a_k is the first non-`thread-local` access ordered by the discipline d, where $1 \leq i < k \leq n$. To ensure that accesses to x are properly ordered, there must exist an action a_j executed by thread t that caused ordering of access a_k according to the synchronization discipline d.

   For example, in the trace above, the initial `thread-local` access $wr_t(x,v)$ by thread t is followed by an $fork_t(u)$ action, also executed by thread t, that orders the next access ($rd_u(x,v)$) under the synchronization discipline `fork`.

2. The discipline "d; `thread-local`" for the variable x, where $d \neq$ `thread-local`, describes a sequence of actions $a_1; \ldots; a_i; \ldots; a_j; \ldots; a_n$, where action a_i is the last access to x ordered by the discipline d and action a_j is the first access not ordered by the discipline d. To ensure that

$$
\begin{array}{rll}
r, s \in \mathit{SyncDevice} & = \mathit{Lock} \cup \mathit{VolatileVar} \cup \mathit{ForkingTid} \cup \mathit{JoinedTid} \\
t^{\mathrm{F}} \in \mathit{ForkingTid} & \\
t^{\mathrm{J}} \in \mathit{JoinedTid} & \\
M \in \mathit{OrderSet} & = 2^{\mathit{SyncDevice}} \\
\mathit{AccessOp} & = \{rd, wr, acc\} \\
\mathit{OrderedAccess} & = \mathit{OrderSet} \rhd \mathit{AccessOp}_{\mathit{Tid}}
\end{array}
$$

Figure 3.4: Ordered access trace domains

all accesses to x are properly ordered, the access a_i must be executed by the same thread as the access a_j.

For example, in the trace above, the access $rd_t(x, v)$ ordered by the discipline `join` is executed by thread t, the same thread that proceeds to execute the `thread-local` access $wr_t(x, v)$.

3.4 Access Patterns

Since our model for race freedom requires a total ordering of accesses to a variable x to ensure that no race conditions occur on x, Grits specifications describe the form of traces exhibiting total orderings as patterns of accesses and synchronization operations. A synchronization discipline for a variable x is actually a pattern of accesses that describes how x can be accessed in a program execution that is free of race conditions on x. In this section we formally define *ordered access traces* and the patterns of accesses described by the Grits specifications. Figure 3.4 summarizes the notations introduced in this section.

An *ordered access* represents one or more accesses to x, ordered from the previous access in the trace by an *OrderSet* of synchronization devices (locks, volatile variables, threads). An ordered access K is defined as follows:

$$
\begin{array}{rll}
K ::= & M \rhd rd_t & \text{A read access by thread } t\text{, ordered by synchronization using the devices in the } \mathit{OrderSet}\ M. \\
\mid & M \rhd wr_t & \text{A write access by thread } t\text{, ordered by synchronization using the devices in the } \mathit{OrderSet}\ M. \\
\mid & M \rhd acc_t & \text{A read or write access by thread } t\text{, ordered by synchronization using the devices in the } \mathit{OrderSet}\ M.
\end{array}
$$

where K consists of the following parts:

- An *OrderSet* M of synchronization devices which ordered the access from previous accesses. Synchronization devices are locks, volatile variables, and thread identifiers for threads that fork or join other threads (*ForkingTid*s and *JoinedTid*s).

Specification	Ordered Access Trace	Program Trace
`thread-local`;	$\varnothing \rhd wr_t$;	$\{\ wr_t(x,v)$;
`fork`;	$\{t^{\mathrm{F}}\} \rhd rd_u$;	$\left\{\begin{array}{l} fork_t(u); \\ acq_u(m); \\ rd_u(x,v); \end{array}\right.$
`guarded-by` m;	$\{m\} \rhd wr_t$;	$\left\{\begin{array}{l} rel_u(m); \\ acq_t(m); \\ wr_t(x,v); \end{array}\right.$
	$\{m\} \rhd rd_u$;	$\left\{\begin{array}{l} rel_t(m); \\ acq_u(m); \\ rd_u(x,v); \end{array}\right.$
`join`;	$\{u^{\mathrm{J}}\} \rhd rd_t$;	$\left\{\begin{array}{l} rel_t(m); \\ join_t(u); \\ rd_t(x,v); \end{array}\right.$
`thread-local`	$\varnothing \rhd wr_t$	$\{\ wr_t(x,v)$

Figure 3.5: A program trace and the corresponding ordered access trace and synchronization discipline for the variable x.

- The type of access. This may be a read access (rd), a write access (wr), or acc, representing a read or write access.
- The thread t that executed the access.

For example, the ordered access

$$\{m, n\} \rhd wr_t$$

represents one or more write accesses to x by thread t, each ordered from the previous access by synchronization with locks m and n. The *AccessOp*s rd and wr indicate read and write access, respectively. Accesses that may be either reads or writes are denoted by acc.

An *ordered access trace* O is a sequence of *ordered accesses* that describes the relevant synchronization and access actions in a trace for a certain variable. The form of an ordered access trace is described by O as follows:

$O ::=$	ε	The empty trace.
$\mid$	$O; K$	The concatenation of a sequence O of ordered accesses with an ordered access K.
$\mid$	$(O)^+$	A repeated pattern of ordered accesses.

For example, Figure 3.5 shows the ordered access trace for the program trace shown in Figure 3.3.

Discipline	Pattern	Description
`thread-local`	$\left(? \rhd acc_t\right)^+$	All accesses to x are executed by a single thread.
`guarded-by` m	$\left((\{m\} \cup ?) \rhd acc_?\right)^+$	All accesses to x are guarded by the lock m.
`vol` $\hat{x}$	$\left((\{\hat{x}\} \cup ?) \rhd acc_?\right)^+$	All accesses to x are ordered by volatile access to the volatile variable $\hat{x}$.
`fork`	$\left((\{t^{\mathrm{F}}\} \cup ?) \rhd acc_?\right)^+$	All accesses to x are ordered by a fork action executed by thread t.
`join`	$\left(\left(\{u_0^{\mathrm{J}}, ..., u_n^{\mathrm{J}}\} \cup ?\right) \rhd acc_t\right)$	An access to x is ordered by a one or more join actions on one or more threads $u_0, ..., u_n$.

Table 3.2: Access patterns for a variable x captured by Grits synchronization disciplines.

The first access is not ordered from any previous accesses since there are no previous accesses. The second access ($rd_u(x, v)$) is ordered from the first by thread t forking thread u. The $acq_u(m)$ action executed by thread u does not order this access from the previous one because thread t did not release lock m after the first access. The third and fourth accesses are ordered by the lock m because they are preceded by the release of m by the last thread to access x and the acquire of m by the thread that performs the access. The second-to-last access ($rd_t(x, v)$) is ordered by the thread t joining the thread u. The $rel_u(m)$ action does not order this access because there is no corresponding acquire of lock m by thread t. The final access is ordered from the second-to-last access only by thread-internal program ordering, so there are no synchronization devices that order it from the previous access.

The patterns for each synchronization discipline are described in Table 3.2. The question mark ("?") acts as a wildcard character that matches anything in the proper contextual domain. For example, the ordered access

$$? \rhd acc_u$$

matches any read or write access by thread u ordered by any set of synchronization devices. The ordered read

$$(M \cup ?) \rhd rd_?$$

matches a read access by any thread ordered by the synchronization devices in the set M and any other synchronization devices as well. Patterns for compound disciplines are defined by the concatenation of their simple parts.

3.5 Extensions to The Grits Specification Language

Since synchronization disciplines are defined in terms of access patterns, Grits can be extended with arbitrarily complex synchronization disciplines. For example, we can specify the common access paradigm `read-shared`, where multiple threads read a single variable without synchronization or ordering between the read accesses, by distinguishing accesses by type. (We extend Grits to handle unordered concurrent reads in Appendix A.) The `read-shared` discipline is defined by the following pattern:

$$\left(? \rhd rd_?\right)^+$$

A second extension to improve the expressiveness of Grits is the addition of conjunctions and disjunctions as operators for combining disciplines. Concatenations and repetitions allow Grits to specify linearly complex synchronization disciplines, but they do not capture the possibility that a given variable may follow two disciplines simultaneously. For example, the variable x may be `guarded-by` m in addition to being `thread-local`. Grits could specify a discipline that is compatible with this (either `thread-local` or `guarded-by` m, depending on context), but it cannot require that both be used. In most cases, it seems unnecessary to force two simultaneous methods of synchronization to be used to protect a variable at every access through the conjunction operator, but the options introduced by the disjunction operator would be useful. For example, we could model two major branches of execution in a program where one synchronization discipline is used in one branch but a different discipline is used in the other branch. However, we have not observed a need for such flexibility.

Grits can also be extended to support specifications of data invariants. For example, we might create a specification such as the following:

```
int b;
int x; // spec:  accessible-by t if (b == t)
```

This specification requires that all accesses to x be performed while the variable b holds the thread identifier of the thread performing the access.

3.6 Summary

The Grits specification language uses a flexible happens-before-based model to specify synchronization disciplines. A synchronization discipline for a variable x describes how accesses to x are ordered through a program trace. A synchronization discipline for x defines the pattern of accesses to x that may appear in a program trace. Compound synchronization disciplines allow the specification of synchronization disciplines that change over time. Grits can express most real world synchronization disciplines and can be extended to specify arbitrarily complex disciplines.

Chapter 4

Hominy: Dynamic Specification Inference

The previous chapter developed a specification language to describe the synchronization disciplines of concurrent programs. To check specifications or even take advantage of their documentation value, we must address the problem of generating specifications for existing programs that lack them. Code may not include specifications for several reasons:

- It is difficult to convince programmers to annotate their code.
- Many large legacy code bases were not written with specifications.
- It may be difficult for a programmer to figure out what code does.

To address this problem, we have developed a tool to accompany Grits that automatically infers specifications based on observed program executions. Hominy, this specification inference tool, is presented in this chapter.

4.1 The Inference Process

Hominy is a dynamic analysis that observes the execution of a single program trace. Hominy infers a Grits specification for each variable x in a program via a series of reductions on an observed program trace. In short, Hominy transforms a program trace into an ordered access trace and applies pattern matching against the specification patterns from Section 3.4. The major steps of this analysis are outlined below and illustrated in an example in Figure 4.1.

1. As the program runs, Hominy processes all variable accesses and synchronization actions, keeping the necessary information for the following steps in the instrumentation store φ. (See Section 4.2.)

0. Observed program trace:

$wr_t(x,1);$ $fork_t(u);$ $acq_u(m);$ $rd_u(x,1);$ $wr_u(x,2);$ $rel_u(m);$ $acq_t(m);$ $rd_t(x,2);$ $wr_t(x,3);$

1. Instrumentation trace:

$$\varphi_0 \xRightarrow{wr_t(x,1)} \varphi_1 \xRightarrow{fork_t(u)} \varphi_2 \xRightarrow{acq_u(m)} \varphi_3 \xRightarrow{rd_u(x,1)} \varphi_4 \xRightarrow{wr_u(x,2)} \varphi_5 \xRightarrow{rel_u(m)} \varphi_6 \xRightarrow{acq_t(m)} \varphi_7 \xRightarrow{rd_t(x,2)} \varphi_8 \xRightarrow{wr_t(x,3)} \varphi_9$$

2. Projection of x:

$$\varnothing \xRightarrow{wr_t(x,1)} \langle t \succcurlyeq_{acc} t\rangle \xRightarrow{fork_t(u)} \begin{matrix}\langle t \succcurlyeq_{acc} t\rangle \\ \langle t \succcurlyeq_{fork} u\rangle\end{matrix} \xRightarrow{acq_u(m)} \begin{matrix}\langle t \succcurlyeq_{acc} t\rangle \\ \langle t \succcurlyeq_{fork} u\rangle\end{matrix} \xRightarrow{rd_u(x,1)} \langle u \succcurlyeq_{acc} u\rangle \xRightarrow{wr_u(x,2)} \langle u \succcurlyeq_{acc} u\rangle \xRightarrow{rel_u(m)} \begin{matrix}\langle u \succcurlyeq_{acc} u\rangle \\ \langle u \succcurlyeq_{rel} m\rangle\end{matrix} \xRightarrow{acq_t(m)} \begin{matrix}\langle u \succcurlyeq_{acc} u\rangle \\ \langle u \succcurlyeq_{rel} m\rangle \\ \langle m \succcurlyeq_{acq} t\rangle\end{matrix} \xRightarrow{rd_t(x,2)} \langle t \succcurlyeq_{acc} t\rangle \xRightarrow{wr_t(x,3)} \langle t \succcurlyeq_{acc} t\rangle$$

3. Ordered access trace:

$\varnothing \rhd wr_t;$ $\{t^{\mathrm{F}}\} \rhd rd_u;$ $\varnothing \rhd wr_u;$ $\{m\} \rhd rd_t;$ $\varnothing \rhd wr_t$

4. Simplified ordered access trace:

$\varnothing \rhd wr_t;$ $\{t^{\mathrm{F}}\} \rhd acc_u;$ $\{m\} \rhd acc_t$

5. Synchronization discipline:

`thread-local;` `fork;` `guarded-by` m

Figure 4.1: Steps in Hominy's analysis for inferring the synchronization discipline for the variable x.

2. To determine a synchronization discipline for a specific variable x, Hominy extracts information about accesses and synchronization actions from the instrumentation store over the course of a full trace. We assume that the variable x is free of race conditions and that there exists a total ordering of all accesses to x. (See Section 4.3.)

3. Hominy then determines how each access was ordered from the previous access by examining the instrumentation store for x just prior to each access to x. This yields an ordered access trace, a sequence of all accesses to x performed in the program. Each access in the trace is annotated with a set of synchronization devices that ordered it from the last access. For example, the ordered write $\{m\} \rhd wr_t$ denotes a write action that was ordered by the release and acquisition of a lock m. (See Section 4.4.)

4. Hominy simplifies an ordered access trace by merging consecutive accesses that share ordering properties. This abstracts certain details to make the next step simpler. (See Section 4.5.)

5. Finally, Hominy determines the variable's synchronization discipline by matching the simplified access trace against patterns that correspond to constructs in the Grits specification language. (See Section 4.7.)

4.2 Recording Program Traces

We first describe the observations Hominy makes on programs. Given a program's execution trace in the form of an observed sequence of *Action*s $a_1; \ldots; a_n$, Hominy builds a corresponding *instrumentation trace* Σ.

$$\Sigma \;=\; (\varphi_0 \xrightarrow{a_1} \varphi_1 \xrightarrow{a_2} \ldots \xrightarrow{a_n} \varphi_{n+1})$$

The instrumentation trace encodes information about the ordering of synchronization events and variable access events in an instrumentation store. For each variable x, each instrumentation store records an *EventSet*:

$$\varphi : \mathit{Variable} \rightarrow \mathit{EventSet}$$

An *EventSet* for x stores a set of *Event*s that represent synchronization *Action*s that have happened since the last access to x. As such, it encodes an approximation of the section of the happens-

before graph rooted at the last access. *Event*s and *EventSet*s are defined as follows:

$$\begin{aligned} e \in \mathit{Event} = {} & \langle \mathit{Tid} \gtrdot_{\mathit{rel}} \mathit{Lock} \rangle \\ & \cup \ \langle \mathit{Lock} \gtrdot_{\mathit{acq}} \mathit{Tid} \rangle \\ & \cup \ \langle \mathit{Tid} \gtrdot_{\mathit{wrv}} \mathit{VolatileVar} \rangle \\ & \cup \ \langle \mathit{VolatileVar} \gtrdot_{\mathit{rdv}} \mathit{Tid} \rangle \\ & \cup \ \langle \mathit{Tid} \gtrdot_{\mathit{fork}} \mathit{Tid} \rangle \\ & \cup \ \langle \mathit{Tid} \gtrdot_{\mathit{join}} \mathit{Tid} \rangle \\ & \cup \ \langle \mathit{Tid} \gtrdot_{\mathit{acc}} \mathit{Tid} \rangle \\ S \in \mathit{EventSet} = {} & 2^{\mathit{Event}} \end{aligned}$$

An *Event* $\langle r \gtrdot_{op} s \rangle$ consists of the following parts:

- An operation $op \in \{rd, wr, acq, rel, rdv, wrv, fork, join\}$
- A *Lock*, *VolatileVar*, or *Tid* r called the *parent* of $\langle r \gtrdot_{op} s \rangle$. The term "parent" reflects the role of r in directed graphs built by linking events together.
- A *Lock*, *VolatileVar*, or *Tid* s called the *child* of $\langle r \gtrdot_{op} s \rangle$. The name "child" reflects the position of s in directed graphs constructed of linked events.

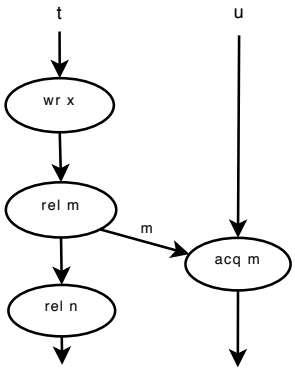
At least one of r and s must be a thread identifier. Each *Event* represents a program action. The types of events are described in Table 4.1. For example, the event $e = \langle m \gtrdot_{acq} t \rangle$ represents the thread t acquiring the lock m, where m is the parent of e and t is the child of e. The event $\langle t \gtrdot_{fork} u \rangle$ represents thread t forking thread u, where t is the parent of the event and u is the child of the event.

The question mark symbol ("?") functions as a wildcard when used in an event. For example, the event $\langle ? \gtrdot_{?} t \rangle$ matches any event that has the thread t as its child. Additionally, the special event $\langle t \gtrdot_{acc} t \rangle$, which uses the access wildcard *acc* operation, represents an event that is either a read event $\langle t \gtrdot_{rd} t \rangle$ or a write event $\langle t \gtrdot_{wr} t \rangle$.

An *EventSet* is used to approximate a happens-before graph rooted at a given access to a variable x. Pairs of release and acquire events or volatile write and volatile read events represent happens-before edges between threads, showing how threads safely exchange control of x by inducing ordering between accesses. For example, individual fork and join events, which involve two threads, represent full happens-before edges. For example, if thread t forks thread u, thread t's call to *fork* must happen before thread u can start executing, so the entire fork event creates a happens-before edge. A thread t that holds a lock m must release m before another thread u can acquire m, so there is a happens-before edge between these two events. Figure 4.2 shows the happens-before graph and corresponding *EventSet* for such an ordering. Table 4.2 contains a full list of the events that create happens-before edges between threads.

$$\underline{wr_t(x,v); rel_t(m); rel_t(n); acq_u(m)}; rd_u(x,v)$$

(a) Trace



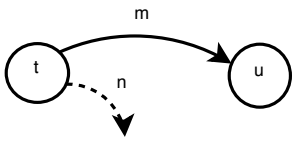
(b) Happens-before

$$\{t, m, n, u\}$$

(c) Goldilocks

$$\{\langle t \succcurlyeq_{acc} t\rangle, \langle t \succcurlyeq_{rel} m\rangle, \langle t \succcurlyeq_{rel} n\rangle, \langle m \succcurlyeq_{acq} u\rangle\}$$

(d) Hominy *EventSet*



(e) Graph encoded by the *EventSet*

Figure 4.2: Happens-before, Goldilocks, and Hominy representations of the underlined section of a program trace for the variable x until just before the next access to x ($rd_u(x,v)$).

Event	Action	Meaning
$\langle t \succcurlyeq_{acc} t\rangle$	$rd_t(x,v)$, $wr_t(x,v)$	Thread t accesses x.
$\langle t \succcurlyeq_{rel} m\rangle$	$rel_t(m)$	Thread t releases lock m.
$\langle m \succcurlyeq_{acq} t\rangle$	$acq_t(m)$	Thread t acquires lock m.
$\langle t \succcurlyeq_{wrv} \hat{y}\rangle$	$wr_t(\hat{y}, v)$	Thread t writes volatile variable $\hat{y}$.
$\langle \hat{y} \succcurlyeq_{rdv} t\rangle$	$rd_t(\hat{y}, v)$	Thread t reads volatile variable $\hat{y}$.
$\langle t \succcurlyeq_{fork} u\rangle$	$fork_t(u)$	Thread t forks thread u.
$\langle u \succcurlyeq_{join} t\rangle$	$join_u(t)$	Thread t waits for thread u to terminate.

Table 4.1: Event types for $\varphi(x)$.

Event(s)	Description
$\langle t \succcurlyeq_{rel} m\rangle\langle m \succcurlyeq_{acq} u\rangle$	Lock release and acquire
$\langle t \succcurlyeq_{wrv} \hat{y}\rangle\langle \hat{y} \succcurlyeq_{rdv} u\rangle$	Volatile write and read
$\langle t \succcurlyeq_{fork} u\rangle$	Thread creation
$\langle t \succcurlyeq_{join} u\rangle$	Thread termination

Table 4.2: *Event* pairs and single *Event*s representing happens-before edges that induce ordering from any previous actions of thread t to any subsequent actions of thread u.

4.2.1 Observing Program Executions

To construct an instrumentation trace Σ for a sequence of actions, Hominy sets the initial instrumentation store to be the empty *EventSet* at the start of the execution:

$$\forall x.\ \varphi_0(x) = \varnothing$$

At each program step, Hominy computes the new *EventSet* for each variable x from the existing *EventSet* for x. We use the notation

$$\varphi \overset{a}{\Longrightarrow} \varphi'$$

to describe this step for action a.

1. On an access to x by thread t, reset $\varphi(x)$ to $\{\langle t \succcurlyeq_{acc} t\rangle\}$, representing the access by t.

2. On a synchronization action b, for all x, add an event $\langle r \succcurlyeq_b s\rangle$ representing b to $\varphi(x)$ if $\varphi(x)$ contains an event $\langle ? \succcurlyeq_? r\rangle$. In other words, we add event e representing the action b to the set if e happened after any event in the set, as indicated by the fact that the parent of e is the child of an event that is already in the set.

Figure 4.3 contains the formal rules defining the judgment $\varphi \overset{a}{\Longrightarrow} \varphi$. These rules construct an *EventSet* to approximate the happens-before graph between two accesses to a variable x. The instrumentation rules for lock operations, volatile accesses, and thread control, add an event to the *EventSet* if that event happened after some other event already in the set. Since the set starts out containing only the access event, this access happens before all other events subsequently added to the set.

- The rule [INS ACCESS] replaces the contents of $\varphi(x)$ with an access event for t on any access to x executed by thread t if $\varphi(x)$ contains an event $\langle ? \succcurlyeq_? t\rangle$ (in other words, where t is a child of an event present in $\varphi(x)$ before the access). For example, on an access to x by thread t, if

  $$\varphi(x) = \{\langle u \succcurlyeq_{acc} u\rangle, \langle u \succcurlyeq_{rel} m\rangle, \langle m \succcurlyeq_{acq} t\rangle\}$$

  the event $\langle m \succcurlyeq_{acq} t\rangle$ has t as its child. (It matches $\langle ? \succcurlyeq_? t\rangle$.) We therefore apply the rule and set $\varphi(x)$ to $\langle t \succcurlyeq_{acc} t\rangle$.

- The rule [INS FIRST ACCESS] updates the *EventSet* in the instrumentation store $\varphi(x)$ for x on the first access to x. It recognizes that $\varphi(x) = \varnothing$ (which indicates that no previous accesses have occurred) and sets the store for x to an *EventSet* containing a single access event attributed to the thread that executed the access. (Note that an access that cannot be described by either of these rules indicates a race condition.)

- On a lock release by thread t, the rule [INS RELEASE] adds an event representing the release to each *EventSet* containing an event that matches $\langle ? \succcurlyeq_? t\rangle$. In other words, the event $\langle t \succcurlyeq_{rel} m\rangle$ is added to the set if the set contains an event whose child is t. For example, consider the

[INS ACCESS]

$$\frac{a \in \{rd_t(x,v), wr_t(x,v)\} \quad \langle ? \succcurlyeq_? t\rangle \in \varphi(x)}{\varphi \overset{a}{\Longrightarrow} \varphi[x := \{\langle t \succcurlyeq_{acc} t\rangle\}]}$$

[INS FIRST ACCESS]

$$\frac{a \in \{rd_t(x,v), wr_t(x,v)\} \quad \varphi(x) = \varnothing}{\varphi \overset{a}{\Longrightarrow} \varphi[x := \{\langle t \succcurlyeq_{acc} t\rangle\}]}$$

[INS RELEASE]

$$\frac{\forall x.\ \varphi'(x) = \begin{cases} \varphi(x) \ \cup\ \{\langle t \succcurlyeq_{rel} m\rangle\} & \text{if } \langle ? \succcurlyeq_? t\rangle \in \varphi(x) \\ \varphi(x) & \text{otherwise} \end{cases}}{\varphi \overset{rel_t(m)}{\Longrightarrow} \varphi'}$$

[INS ACQUIRE]

$$\frac{\forall x.\ \varphi'(x) = \begin{cases} \varphi(x) \ \cup\ \{\langle m \succcurlyeq_{acq} t\rangle\} & \text{if } \langle ? \succcurlyeq_? m\rangle \in \varphi(x) \\ \varphi(x) & \text{otherwise} \end{cases}}{\varphi \overset{acq_t(m)}{\Longrightarrow} \varphi'}$$

[INS VOLATILE WRITE]

$$\frac{\forall x.\ \varphi'(x) = \begin{cases} \varphi(x) \ \cup\ \{\langle t \succcurlyeq_{wrv} \hat{y}\rangle\} & \text{if } \langle ? \succcurlyeq_? t\rangle \in \varphi(x) \\ \varphi(x) & \text{otherwise} \end{cases}}{\varphi \overset{wrv_t(\hat{y})}{\Longrightarrow} \varphi'}$$

[INS VOLATILE READ]

$$\frac{\forall x.\ \varphi'(x) = \begin{cases} \varphi(x) \ \cup\ \{\langle \hat{y} \succcurlyeq_{rdv} t\rangle\} & \text{if } \langle ? \succcurlyeq_? \hat{y}\rangle \in \varphi(x) \\ \varphi(x) & \text{otherwise} \end{cases}}{\varphi \overset{rdv_t(\hat{y})}{\Longrightarrow} \varphi'}$$

[INS FORK]

$$\frac{\forall x.\ \varphi'(x) = \begin{cases} \varphi(x) \ \cup\ \{\langle t \succcurlyeq_{fork} u\rangle\} & \text{if } \langle ? \succcurlyeq_? t\rangle \in \varphi(x) \\ \varphi(x) & \text{otherwise} \end{cases}}{\varphi \overset{fork_t(u)}{\Longrightarrow} \varphi'}$$

[INS JOIN]

$$\frac{\forall x.\ \varphi'(x) = \begin{cases} \varphi(x) \ \cup\ \{\langle u \succcurlyeq_{join} t\rangle\} & \text{if } \langle ? \succcurlyeq_? u\rangle \in \varphi(x) \\ \varphi(x) & \text{otherwise} \end{cases}}{\varphi \overset{join_t(u)}{\Longrightarrow} \varphi'}$$

Figure 4.3: Instrumentation semantics

following trace:

$$wr_t(x, v); rel_t(m)$$

When we see $rel_t(m)$, the thread t has just accessed x, so

$$\varphi(x) = \{\langle t \succcurlyeq_{acc} t\rangle\}$$

The event $\langle t \succcurlyeq_{acc} t\rangle$ matches $\langle ? \succcurlyeq_? t\rangle$, so we add $\langle t \succcurlyeq_{rel} m\rangle$ to the set:

$$\varphi(x) = \{\langle t \succcurlyeq_{acc} t\rangle, \langle t \succcurlyeq_{rel} m\rangle\}$$

Intuitively, we added the release event because it happened after the access event, as defined by the ordering of actions executed by a single thread.

- On acquiring lock m, the rule [INS ACQUIRE] adds an event for the acquire to each *EventSet* in the instrumentation store where there is an event releasing m. For example, consider the instrumentation store for x after the release action described above:

  $$\varphi = \{\langle t \succcurlyeq_{acc} t\rangle, \langle t \succcurlyeq_{rel} m\rangle\}$$

  The set contains a release event for m ($\langle t \succcurlyeq_{rel} m\rangle$), so if thread u acquires m, then we add the event $\langle m \succcurlyeq_{acq} u\rangle$ to the set:

  $$\varphi(x) = \{\langle t \succcurlyeq_{acc} t\rangle, \langle t \succcurlyeq_{rel} m\rangle, \langle m \succcurlyeq_{acq} u\rangle\}$$

- On a volatile write by thread t, the rule [INS VOLATILE WRITE] adds an event representing the release to each *EventSet* containing an event executed by t.
- On a read to the volatile variable $\hat{x}$, the rule [INS VOLATILE READ] adds an event for the volatile read to every *EventSet* containing an event representing a write to $\hat{x}$.
- On the forking of thread u by thread t, the rule [INS FORK] adds an event representing the fork to each *EventSet* containing an event executed by t.
- On the joining of thread u by thread t, the rule [INS JOIN] adds an event representing the join to each *EventSet* containing an event executed by u.

At each step, all *EventSet*s are updated atomically according to these rules. Since this instrumentation store treatment is an extension of the Goldilocks algorithm, Hominy can detect and report race conditions that occur in the instrumented program.

4.2.2 Summary

Hominy employs *EventSet*s for each variable x to track the synchronization actions that impose ordering on each access to x. By updating these sets with synchronization actions that impose

ordering as the program executes, Hominy essentially records approximations of happens-before graphs between each pair of consecutive accesses to a variable. The end product of this process is an instrumentation trace:

$$\Sigma = \left(\varphi_0 \overset{a_1}{\Longrightarrow} \varphi_1 \overset{a_2}{\Longrightarrow} \ldots \overset{a_n}{\Longrightarrow} \varphi_n\right)$$

The instrumentation trace Σ records all access and synchronization actions that occurred in the program as well as how those actions affected the synchronized state of the program, as represented by the instrumentation stores φ_i.

4.3 Selecting a Per-Variable Trace

Hominy examines one variable at a time to construct that variable's synchronization discipline. To reason more easily about the synchronization discipline of a variable x, we first select only those actions in a trace which may directly affect access to x. The operator $\big|_x$ applied to a full instrumentation trace Σ returns the projection Ω of a variable x on Σ, which replaces the full instrumentation store φ_i at each state i in Σ by the *EventSet* for x:

$$\Omega = \left(\varphi_0 \overset{a_1}{\Longrightarrow} \varphi_1 \overset{a_2}{\Longrightarrow} \ldots \overset{a_n}{\Longrightarrow} \varphi_n\right) \big|_x = \left(S_0 \overset{a_1}{\Longrightarrow} S_1 \overset{a_2}{\Longrightarrow} \ldots \overset{a_n}{\Longrightarrow} S_n\right)$$

where $S_i = \varphi_i(x)$.

4.4 Inferring Ordering Between Accesses

We now have a projected trace Ω of synchronization actions and accesses to x. The next step is to infer how the program imposed ordering on each access in Ω. Suppose we have a trace of the following form:

$$\ldots \overset{a}{\Longrightarrow} S_i \overset{b_1}{\Longrightarrow} \ldots \overset{b_n}{\Longrightarrow} S_j \overset{a'}{\Longrightarrow} \ldots$$

where a and a' are accesses to x and none of $\{b_1, \ldots, b_n\}$ are accesses to x. The *EventSet* S_j encodes the synchronization devices used to impose an order between a and a'. It approximates the happens-before graph for all program events that have happened since the access a.

The ordering of the two accesses a and a' is given by the paths in the happens-before graph from a to a', as shown in Figure 4.4. The labels of edges in these paths are the synchronization devices that were used to order the pair of accesses. We call this set of edges the *OrderSet* M of a', where $M \subseteq$ *SyncDevice*. We therefore have a graph reachability problem. A simple algorithm for finding this *OrderSet* is a breadth-first search to find all edges between different threads on paths from a to a'.

For example, in the happens-before graph in Figure 4.4 all actions that have happened since the access a are shown in black. The path from a to a' is bold. The only edge between threads on this path is labeled m, so the *OrderSet* for access a' is $\{m\}$.

To summarize, we are trying to find how a given access is ordered from the previous access by examining a happens-before graph of events since the previous access. We do this by finding the

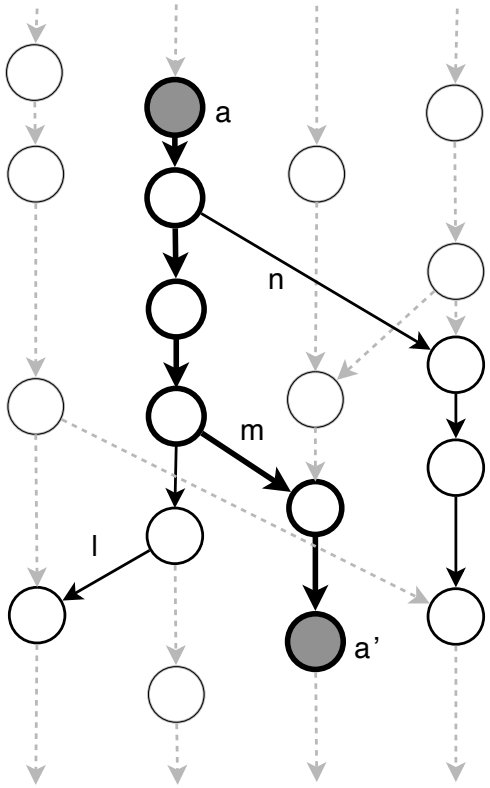


Figure 4.4: Consecutive accesses a and a' to variable x in the context of a full-program happens-before graph.

edges between threads on paths from one access to the next in the happens-before graph. These edges represent synchronization actions between threads.

Here, we assume that thread t executes action a'. To find an approximation of this *OrderSet* with the *EventSet* S_j, we employ two functions, *ancestors*(S_j, t) and *choose*(S_j, t), to select an approximation of those events in S_j that happen before action a'.

- The function *ancestors*(S_j, t) computes all events on paths from a to a' in the *EventSet* S_j.
- The function *choose*(S_j, t) chooses from the ancestors of a' in S_j those synchronization devices that imposed an order such that $a \rightarrow a'$. In essence, *choose* throws out synchronization actions that were present on paths between a and a' but which did not cause the ordering of a and a'.

We precisely define these functions below.

4.4.1 Choose

The *choose* function chooses the relevant edges in paths computed by *ancestors*(S_j, t). The *choose* function is defined as follows:

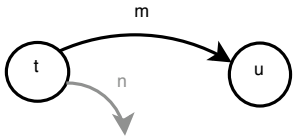
Definition 4.1. $choose(S,t) : EventSet \times Tid \rightarrow OrderSet$

$$\begin{aligned} choose(S,t) = \{&m \in Lock \mid \langle m \succcurlyeq_{acq} ? \rangle \in ancestors(S,t) \text{ and } \langle ? \succcurlyeq_{rel} m \rangle \in ancestors(S,t)\} \\ \cup\ &\{\hat{y} \in VolatileVar \mid \langle \hat{y} \succcurlyeq_{rdv} ? \rangle \in ancestors(S,t) \text{ and } \langle ? \succcurlyeq_{wrv} \hat{y} \rangle \in ancestors(S,t)\} \\ \cup\ &\{t^{\mathrm{F}} \in ForkingTid \mid \langle t \succcurlyeq_{fork} ? \rangle \in ancestors(S,t)\} \\ \cup\ &\{u^{\mathrm{J}} \in JoinedTid \mid \langle u \succcurlyeq_{join} ? \rangle \in ancestors(S,t)\} \end{aligned}$$

The function $choose(S,t)$ selects all parents and children of events in the ancestors set that imposed ordering from the last access in S to the next action of thread t. All locks and volatile variables that appear in events in the ancestors set imposed ordering between the accesses a and a'. Threads only act as synchronization devices in fork and join actions. We record *Tid*s as *ForkingTid*s (t^{F}) or *JoinedTid*s (t^{J}) to distinguish fork and join actions in subsequent steps of the analysis.

Examples

1. Let $S = \{\langle t \succcurlyeq_{acc} t \rangle, \langle t \succcurlyeq_{rel} m \rangle, \langle t \succcurlyeq_{rel} n \rangle, \langle m \succcurlyeq_{acq} u \rangle\}$, which represents the following graph.

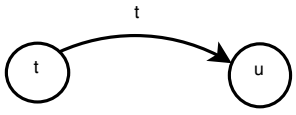


   In this example, no fork or join operations are present in S.

$$choose(S,u) = \{m\}$$

   No threads were involved in fork or join operations in S, thus the lock m is the only device labeling a happens-before edge in S.

2. If S includes fork or join operations, threads may act as synchronization devices. Let $S = \{\langle t \succcurlyeq_{acc} t \rangle, \langle t \succcurlyeq_{fork} u \rangle\}$, which represents the following graph.



$$choose(S,u) = \{t^{\mathrm{F}}\}$$

   Thread t forked u in S, so t^{F} is a happens-before edge label in S.

4.4.2 Ancestors

The ancestors of an access event in an *EventSet* are defined by the least fixed point of the function $ancestors(S,t)$, which approximates the breadth-first search solution to the graph reachability problem described above. We define $ancestors(S,t)$ as follows:

Definition 4.2. $ancestors(S, t) : EventSet \times Tid \rightarrow EventSet$

$$ancestors(S, t) = \{\langle ? \succcurlyeq_{?} s\rangle \in S \mid \langle s \succcurlyeq_{?} ?\rangle \in ancestors(S, t)\} \cup \{\langle t \succcurlyeq_{acc} t\rangle\}$$

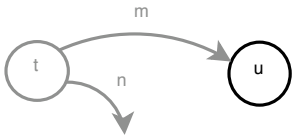
The computation of an example ancestors set is shown below.

Example Let $S = \{\langle t \succcurlyeq_{acc} t\rangle, \langle t \succcurlyeq_{rel} m\rangle, \langle t \succcurlyeq_{rel} n\rangle, \langle m \succcurlyeq_{acq} u\rangle\}$.

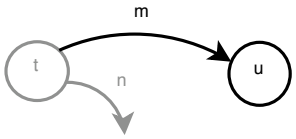
$$ancestors(S, u) = \{\langle t \succcurlyeq_{acc} t\rangle, \langle t \succcurlyeq_{rel} m\rangle, \langle m \succcurlyeq_{acq} u\rangle\}$$

This set is computed using a standard iterative fixed point algorithm as follows. At each step we show the ancestors set computed so far and a graph of the original *EventSet* with all threads (vertices) and edges in the ancestors set colored black.

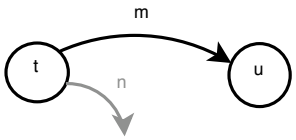
1. $\Rightarrow \{\langle u \succcurlyeq_{acc} u\rangle\}$



2. The event $\langle m \succcurlyeq_{acq} u\rangle$ has u as a child.
   $\Rightarrow \{\langle u \succcurlyeq_{acc} u\rangle, \langle m \succcurlyeq_{acq} u\rangle\}$



3. The event $\langle t \succcurlyeq_{rel} m\rangle$ has m as a child.
   $\Rightarrow \{\langle u \succcurlyeq_{acc} u\rangle, \langle m \succcurlyeq_{acq} u\rangle, \langle t \succcurlyeq_{rel} m\rangle\}$



4. The event $\langle t \succcurlyeq_{acc} t\rangle$ has t as a child.
   $\Rightarrow \{\langle u \succcurlyeq_{acc} u\rangle, \langle m \succcurlyeq_{acq} u\rangle, \langle t \succcurlyeq_{rel} m\rangle, \langle t \succcurlyeq_{acc} t\rangle\}$

5. No other events have children matching the parents accumulated thus far.
   $\Rightarrow \{\langle u \succcurlyeq_{acc} u\rangle, \langle m \succcurlyeq_{acq} u\rangle, \langle t \succcurlyeq_{rel} m\rangle, \langle t \succcurlyeq_{acc} t\rangle\}$

Note that lock n is not a parent of any event in the set constructed so far, so the event $\langle t \succcurlyeq_{rel} n\rangle$ does not enter the set.

4.5 Ordered Access Traces

At this point, we have inferred how a given access is ordered from the last access. From this, we can reduce a per-variable instrumentation trace Ω for a variable x to an *ordered access trace* that describes how the sequence of accesses to x were ordered.

For each access and its associated ordering information, Hominy generates an ordered access K. Recall from Section 3.4 that K has the form:

$$K ::= M \rhd rd_t \mid m \rhd wr_t \mid M \rhd acc_t$$

where K consists of the following parts:

- An *OrderSet* M of synchronization devices which ordered the access from previous accesses.
- The type of access. This may be a read access (rd), a write access (wr), or acc, representing a read or write access.
- The thread t that executed the access.

Hominy constructs an ordered access trace O that is consistent with a trace Ω computed in Section 4.3. To do this, Hominy applies the *choose* function for each access to x to translate a projected trace Ω to an ordered access trace that encodes this information. Recall from Section 3.4 that the structure of an ordered access trace O is defined as follows:

$$O ::= \varepsilon \mid O; K$$

We use the judgment $O \vdash_x \Omega$ to describe when an ordered access trace O is consistent with a trace Ω for x. It is defined in Figure 4.5. The judgment works as follows:

- The rule [ORDER BEGIN] states that an empty ordered access trace is consistent with an empty trace, the initial program state. Intuitively, we can represent an empty trace with and empty ordered access trace.
- The rule [ORDER NON-ACCESS] states that if an ordered access trace O is consistent with projected trace Ω then it is also consistent with the trace that results by adding one more non-access action a to the end of Ω. Intuitively, this means that the action a happens after the last access to x in the trace Ω and therefore cannot affect the last access or any actions that ordered it.
- The rule [ORDER ACCESS] states that if an ordered access trace O is consistent with the trace Ω for x then an access action b following Ω is ordered by the synchronization devices used along the path in the happens-before graph from the last access to b, represented by $choose(S, t)$.

[ORDER BEGIN]

$$\frac{}{\varepsilon \vdash_x \varnothing}$$

[ORDER NON-ACCESS]

$$\frac{O \vdash_x \Omega \qquad \forall t, v.\ a \notin \{rd_t(x,v), wr_t(x,v)\}}{O \vdash_x \Omega \overset{a}{\Longrightarrow} S}$$

[ORDER ACCESS]

$$\frac{\Omega = (\ldots \overset{a}{\Longrightarrow} S) \qquad O \vdash_x \Omega \qquad b \in \{rd_t(x,v), wr_t(x,v)\}}{(O; \mathit{choose}(S,t) \rhd \mathit{acc}_t) \vdash_x (\Omega \overset{b}{\Longrightarrow} \{\langle t \succcurlyeq_{acc} t \rangle\})}$$

Figure 4.5: Ordered access trace construction rules

4.6 Simplifying Ordered Access Traces

We can now derive an ordered access trace representing the treatment of a variable in a program. Such a trace may be extremely long and contain small details which interfere in the recognition of a larger-scale synchronization discipline. For example, consider the following section of an ordered access trace:

$$\ldots \{m\} \rhd rd_t; \varnothing \rhd wr_t; \{m\} \rhd rd_u; \ldots$$

We assume that thread t acquired the lock m, read x, and, still holding m, wrote to x. It may not be immediately obvious that t still holds m at the write access by t. For example, t could have released m between the two accesses without reacquiring it for the second access. However, we see that thread u accesses x next, ordered by m. This means that thread t released the lock m after writing to x. While thread t could have acquired and *then* released lock m after writing to x, narrowly missing a race condition, we assume that Hominy is inferring synchronization disciplines for race-free programs. The full projected trace for x does contain enough information to determine the exact sequence of locking operations.

This pattern of access occurs in many programs. To avoid dealing with local trace details like these in the definitions of high-level synchronization disciplines, and to more easily characterize the ordered access trace for a variable, we simplify the ordered access trace by merging consecutive accesses that share ordering properties. To represent merged accesses, we extend the structure of ordered access traces to include a simple representation of repeated patterns as follows:

$$O ::= \ldots \mid (O)^+$$

We simplify an ordered access trace O using the rewrite rules defined in Figure 4.6.

Rewrite rule 4.1 Two consecutive accesses executed by the same thread and ordered by the same set of synchronization devices merge into one logical access ordered by this set.

$$M \rhd acc_t; M \rhd acc_t \;\leadsto\; M \rhd acc_t \tag{4.1}$$
$$M \rhd acc_t; \varnothing \rhd acc_t \;\leadsto\; M \rhd acc_t \tag{4.2}$$
$$O; (O)^+ \;\leadsto\; (O)^+ \tag{4.3}$$
$$(O)^+; O \;\leadsto\; (O)^+ \tag{4.4}$$
$$O_0; O_1; \ldots; O_n; O_0; O_1; \ldots; O_n \;\leadsto\; (O_0; O_1; \ldots; O_n)^+ \tag{4.5}$$
$$((O)^+)^+ \;\leadsto\; (O)^+ \tag{4.6}$$
$$\frac{O \;\leadsto\; O'}{E[O] \;\leadsto\; E[O']} \tag{4.7}$$

Figure 4.6: Ordered access trace rewrite rules

Rewrite rule 4.2 Two consecutive accesses executed by the same thread, where the second access has no ordering besides thread order merge into one logical access. We assume that the program is free of race conditions and therefore that thread ordering is sufficient to prevent race conditions here.

Rewrite rule 4.3 A sequence of ordered accesses followed by one or more repetitions of the same sequence merges to one or more repetitions of the sequence.

Rewrite rule 4.4 A sequence of ordered accesses preceded by one or more repetitions of the same sequence merges to one or more repetitions of the sequence.

Rewrite rule 4.5 A repeated sequence of ordered accesses is merged to an expandable form of the sequence representing one or more repetitions of the sequence.

Rewrite rule 4.6 One or more repetitions of one or more repetitions of a sequence of ordered accesses is equivalent to one or more repetitions of that sequence.

Rewrite rule 4.7 This rewrite rule equates traces differing on equivalent sub-traces. Here E is a context defined as follows:

$$E ::= [\,] \mid E; O \mid O; E \mid E^+$$

We apply these rules to rewrite a full ordered access trace for a variable x to its simple form.

4.7 Recognizing Synchronization Disciplines

Given a merged access trace for a variable, we identify the variable's synchronization discipline by matching the discipline's access pattern (as defined in Section 3.3) against the trace in the style of regular expressions. To infer compound synchronization disciplines, we use the recursive matching algorithm described in Algorithm 4.1. This algorithm chooses the discipline whose access pattern

(as defined in Table 3.2) that matches the longest prefix of the trace and recursively determines the discipline for the unmatched remainder of the trace.

Algorithm 4.1 Simple recursive synchronization discipline matching algorithm.

```
function MATCH(trace)
    if trace is not empty then
        Let d be the discipline matching the longest prefix of trace.
        Let rest be the suffix of trace not matched by d.
        return d; MATCH(rest)
    else
        return ε
    end if
end function
```

Ties in match length are broken in favor of disciplines with higher precedence (a lower number) according to the following:

1. `fork`
2. `join`
3. `guarded-by` m
4. `vol` $\hat{x}$
5. `thread-local`

This precedence ensures that if two disciplines such as `thread-local` and `guarded-by` m tie in a match, the discipline more likely to cause ordering (`guarded-by` m) is not masked by a more general discipline such as `thread-local`.

As an example of the matching algorithm, consider the following trace:

$$\begin{array}{lllll} \text{Thread } t: & wr_t(x,1); fork_t(u); & & acq_t(m); wr_t(x,2); rel_t(m); & \\ \text{Thread } u: & & acq_u(m); rd_u(x,1); rel_u(m); & & acq_u(m); rd_u(x,2); rel_u(m) \end{array}$$

which is represented by the following ordered access trace:

$$\varnothing \rhd wr_t;\ \{t^{\mathrm{F}}\} \rhd rd_u;\ \{m\} \rhd wr_t;\ \{m\} \rhd rd_u$$

The only discipline to match the trace starting at the beginning is `thread-local`. It matches the first access ($\varnothing \rhd wr_t$) but not the second ($\{t^{\mathrm{F}}\} \rhd rd_u$), so we start matching against the rest of the trace starting with the second access.

The longest match starting at the second access is a tie between `fork` and `thread-local`. The `thread-local` discipline clearly has lowest precedence, so we choose `fork`. The remainder of the trace is matched by `guarded-by` m. The complete inferred specification is therefore:

`thread-local; fork; guarded-by` m

4.7.1 Preference to Fork and Join

Another exception must be made for the `fork` and `join` disciplines to avoid a problem with the approach that treats `fork` and `join` equally with other disciplines. Consider the following trace:

$$\begin{array}{ll}\text{Thread } t: & wr_t(x,1); fork_t(u);\\ \text{Thread } u: & \qquad\qquad acq_u(m); rd_u(x,1); rel_u(m); acq_u(m); wr_u(x,2); rel_u(m); acq_u(m); rd_u(x,2); rel_u(m)\end{array}$$

which is represented by the following simplified ordered access trace:

$$\varnothing \rhd wr_t;\ \{t^{\mathrm{F}}\} \rhd rd_u;\ \{m\} \rhd acc_u$$

By the simple algorithm presented above, we infer the following specification for this trace:

$$\texttt{thread-local}_t\texttt{; thread-local}_u$$

This specification does describe the trace, but it does not describe a valid ordering for the trace. Specifically, a transition from `thread-local` in one thread to `thread-local` in another must be ordered by some synchronization action, typically a fork or join as in this case. The specification which should be inferred here is the following:

$$\texttt{thread-local; fork; guarded-by } m$$

The first access is matched by `thread-local` for thread t, as we expect. Starting at the second access, all remaining accesses in the trace are executed by thread u, so `thread-local` matches the remainder of the trace. However, $\texttt{thread-local}_u$ masks the ordering between the first and second accesses that is provided by the `fork` discipline.

To avoid the masking of this ordering, we make the following exception to our simple matching algorithm:

> If `fork` or `join` matches the ordered access trace, it is chosen without considering other disciplines, even if other disciplines match longer prefixes of the trace.

The resulting algorithm shown in Algorithm 4.2 resolves the masking problem.

4.8 Summary

We now have a complete transformation from a full program trace to a synchronization discipline specification for a single variable. By applying these steps of recording an instrumentation trace, extracting a per-variable trace, inferring and simplifying an ordered access trace, and matching synchronization discipline specifications against the trace for each variable in a program, Hominy generates a full specification for the program. The Hominy analysis can easily be extended to verify synchronization specifications for variables.

Algorithm 4.2 Recursive synchronization discipline matching algorithm with preference for `fork` and `join`.

```
function MATCH(trace)
    if trace is not empty then
        if fork matches trace then
            Let d = fork
        else if join matches trace then
            Let d = join
        else
            Let d be the discipline matching the longest prefix of trace.
        end if
        Let rest be the suffix of trace not matched by d.
        return d; MATCH(rest)
    else
        return ε
    end if
end function
```

Chapter 5

Hominy Implementation

We have implemented the Hominy synchronization discipline specification inference analysis described in Chapter 4 with the extensions for concurrent read accesses described in Appendix A. Our command-line tool takes the main class of a compiled Java program as a command-line argument:

```
$ hominy ChangeLocks
```

The Hominy tool instruments and runs the program, observing the program trace and applying specification matching online for all variables. When the instrumented program has finished executing, Hominy writes a specification of the synchronization discipline for every variable to a log, excluding those variables which are declared to be `final` or `volatile`, since race conditions cannot occur on these types of variables. The synchronization disciplines logged by Hominy are fairly similar in appearance to those defined in Chapter 3. Ideally, Hominy would return annotated source code such as that shown in the program ChangeLocks in Figure 5.1. However, to accomplish this, it is necessary to map memory addresses back to variable names in program source code. This program heap abstraction is left as an extension to this work. A simple approach used by the FindLocks tool [18] could be applied similarly in Hominy.

In Hominy's actual output, each variable (or more precisely, each memory location) used by a program is uniquely identified by the object address, class name, and field name corresponding to that location, as in `0x0120A47E BankAccount.balance`. The current version of Hominy does perform some simple and localized heap abstractions to make its output more intuitive. Hominy recognizes static fields (static class variables) and relations between fields within the same instance of an object. Static fields appear to be members of an object address of "`static`." In specifications for a field f, references to the memory address of an object containing f appear as "`this`" instead of the literal memory address of the object. The Hominy specification for ChangeLocks is shown at the bottom of Figure 5.1.

Annotated source code:

```
public class ChangeLocks extends Thread {
  // First lock used to guard field.
  static final Object m = new Object();
  // Second lock used to guard field.
  static final Object n = new Object();
  static int field;  /*## spec: thread-local;
                                fork;
                                guarded-by m;
                                vol barrier;
                                guarded-by n ##*/
  // Barrier used to syncrhonize switch from lock m to n
  static volatile int barrier = 3;

  public void run() {
    // field is first guarded by m.
    synchronized(m) {
      field++;
    }
    // Now we wait for all Threads to reach this point.
    barrier--;
    while(barrier > 0); // spin!
    // field is now guarded by n.
    synchronized(n) {
      field++;
    }
  }

  public static void main(String[] args) {
    field = 0;
    for (int i = 0; i < 3; i++) {
      new ChangeLocks().start();
    }
  }
}
```

Specification:

```
Object: static
 Field: ChangeLocks.field
  Spec: thread-local T0;
        fork {T0};
        guarded-by {Lock:0x006F7CE9};
        vol {Vol:this:ChangeLocks.barrier};
        guarded-by {Lock:0x01AD77A7}
```

Figure 5.1: The ChangeLocks program, annotated with synchronization discipline specifications, and the specification Hominy infers for it.

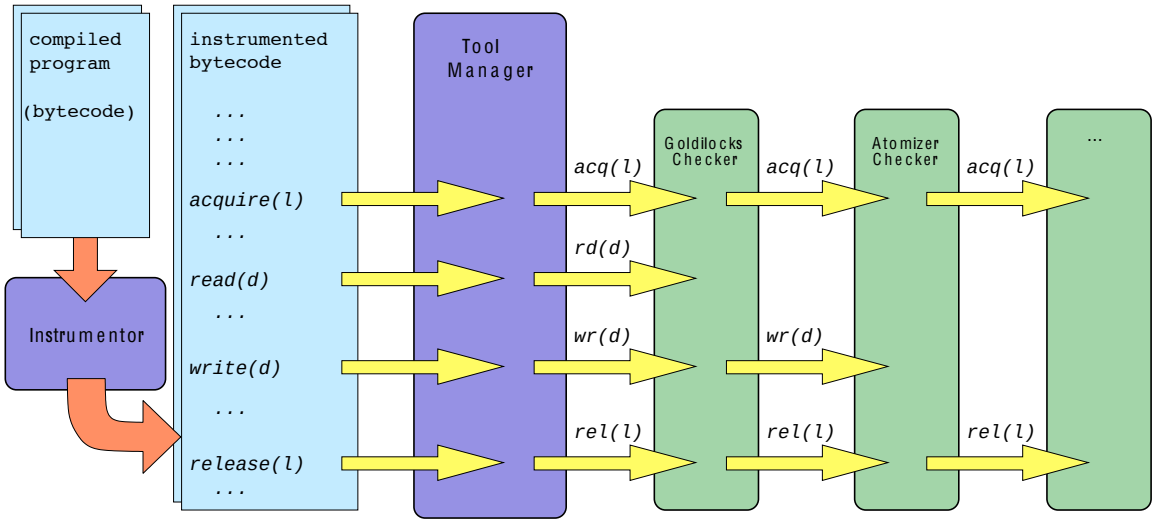


Figure 5.2: A schematic of the RoadRunner framework

5.1 The RoadRunner Program Analysis Framework

Our implementation of Hominy is built on top of RoadRunner, a dynamic program analysis framework currently under development at Williams College and UC Santa Cruz. The framework is designed for building and comparing analyses of concurrent programs in Java. RoadRunner can also compose analyses to mitigate the imprecision or poor performance of a single analysis. For example, composing two race condition checkers such as Eraser (fast, but imprecise) and Goldilocks (precise but slower), RoadRunner can produce an analysis which is more precise than Eraser, but faster than Goldilocks.

As illustrated in Figure 5.2, RoadRunner instruments a program to report accesses and synchronization events via hooks into an analysis back end. As the instrumented program runs, the back end publishes events to a chain of analysis tools. Each tool passes on all synchronization events to the next. If the first tool in the chain detects an error on an access, it passes the access on for further evaluation by subsequent tools. The tool interaction allows us to compare analyses and compose them together to find other kinds of program errors.

5.2 Instrumentation

RoadRunner provides a built-in per-variable instrumentation store. A variable in Hominy terms is equivalent to a field in an object (the cross product of the object's memory address and the name of the field, which in turn maps to an absolute memory address) in a Java program. Every object field is shadowed by a *GuardState*, in which RoadRunner analysis tools can store information about the state of that field. On each access to a variable x, the analysis tool receives the *GuardState* for x.

Hominy uses the *GuardState* to store a map from memory location to *EventSet*. On each synchronization action in the instrumented program Hominy applies updates to every active *EventSet* according to the rules in Section A.2.1, which extend those presented in Section 4.2. Hominy currently uses a single lock to guard all internal data structures. This is not ideal since it eliminates most opportunities for concurrency, but it does simplify the implementation.

Although we assume that any program analyzed by Hominy must be free of race conditions, Hominy verifies this assumption by detecting and reporting any race conditions that occur in the instrumented program. This is trivial functionality, since the *EventSet* instrumentation store and update algorithm are extensions to Goldilocks. If a race condition occurs on a variable x, Hominy stops monitoring x because it is clear that the program does not follow a valid synchronization discipline for x. Although the lack of a race condition report from Hominy guarantees that there were no race conditions on the trace observed, it does not guarantee that all execution paths in the program are free of race conditions.

5.3 Simplifying Traces Online

We apply the rewrite rules from Section 4.6 to simplify a trace. However, to simplify repeated patterns of arbitrary length at this stage, we need the whole trace. Since many full ordered access traces consisting of very large numbers of accesses simplify to shorter traces of only a few accesses, waiting to simplify a full trace requires a higher memory overhead than would be necessary if access traces could be reduced online. For any program of more than moderate running time, the amount of memory required to store full ordered access traces for each variable is prohibitive.

One solution to the access trace storage problem is to log access traces to disk at a certain interval. This alleviates the memory bottleneck, but performance suffers from an increase in the amount of synchronization required and the speed penalties of disk access. Instead, we observe that storing an un-reduced access trace of at most n accesses in addition to the current access allows us to simplify repeated patterns of up to n accesses, as formalized in the following rewrite rules:

$$\frac{|O| < n}{O; O \rightsquigarrow (O)^+} \qquad \frac{|O| < n}{(O)^+; O \rightsquigarrow (O)^+}$$

In the current implementation, we choose the maximum number of un-reduced accesses to be two. This is a simple and conservative approach which effectively prohibits the reduction of repeated access patterns of more than one access, but in our experience, most synchronization disciplines do not exhibit longer access patterns. Note that this restricts the length of the pattern to one access but it does not restrict the number of repetitions of that pattern that can be recognized, since each repetition in turn will be reduced with the pattern.

We do not use the rewrite rule $O; O^+ \rightsquigarrow O^+$. Our online analysis only receives one access at a time and compares it to a history of at most one simplified ordered access, so no partially simplified

traces of the form $O; O^+$ can appear. Similarly, the congruence

$$\frac{O \rightsquigarrow O'}{E[O] \rightsquigarrow E[O']}$$

does not map to our online trace simplification algorithm. We replace it with left-most matching.

The Hominy architecture permits various simplification strategies to be plugged into the analysis. This will enable us to explore other approaches to simplifying traces. Additionally, specification matchers are implemented as simple state machines, so a specification can match complex patterns of accesses, including arbitrary repetitions.

5.4 Evaluation

Our experience with Hominy shows that the Hominy inference process successfully identifies a number of synchronization disciplines beyond conventional lockset based disciplines. Implementation choices have limited the efficiency of the initial version of Hominy, but the specifications inferred by the tool are correct and concise.

5.4.1 Effectiveness

We ran Hominy on a number of small benchmarks to check the accuracy of the specifications it infers. In each of the following code examples, exception handling code and some scheduler-perturbing code (`Thread.sleep()`) has been removed for clarity. Hominy inferred correct specifications for each of these benchmarks, as discussed below.

SyncObject

The SyncObject program, shown in Figure 5.3, runs five threads, each of which call the synchronized method `f()` of a single SyncObject. Since the field `i` of a SyncObject o is protected by the lock o, all accesses to `i` are `guarded-by` o. Hominy correctly infers this specification, as shown in Figure 5.3. Technically, the specification should be "`thread-local; guarded-by this`," since the first access is not ordered from any previous accesses by the lock acquire operation. For a more concise specification, Hominy uses information about what locks are held at the first access to extend the `guarded-by` discipline to include the first access if possible.

ForkLock

The ForkLock program, shown in Figure 5.4, writes zero to the shared variable `field` in the main thread and then forks two other threads. These two threads each access `field` while holding the lock `lock`. Finally, the main thread waits for threads 1 and 2 to terminate and then prints the value of `field`. The synchronization discipline for `field` should therefore be:

```
thread-local; fork; guarded-by lock; join
```

Source code:

```
public class SyncObject extends Thread {

  protected static final SyncObject so = new SyncObject();

  protected int i = 0;

  public synchronized void f() {
    i++;
  }

  public void run() {
    so.f();
  }

  public static void main(String[] args) {
    for (int i = 0; i < 5; i++) {
      new SyncObject().start();
    }
  }

}
```

Specification:

```
Object: 0x01893EFE
 Field: SyncObject.i
  Spec: guarded-by {Lock:this}
```

Figure 5.3: The SyncObject program and the specification Hominy infers for it..

Hominy correctly infers this specification, as shown in Figure 5.4.

ChangeLocks

The program ChangeLocks illustrates a simple example of a nontrivial synchronization discipline successfully inferred by Hominy. The program has a field `field` that is initialized by the main thread and subsequently accessed by three threads forked by the main thread. The threads first use the lock `m` to synchronize access to `field`. Next, they all reach a volatile barrier `barrier` and proceed once all threads have arrived at the barrier. After leaving the barrier, all threads use lock `n` to synchronize access to `field`. Given this behavior, it is clear that the synchronization discipline of `field` is the following:

```
thread-local; fork; guarded-by m; vol barrier; guarded-by n
```

The `hominy` tool successfully infers the synchronization discipline beginning at `guarded-by m`, but cannot recognize the initial `thread-local` discipline and `fork` transition, due to limitations in the current version of RoadRunner (the instrumentation and analysis framework on which Hominy is based) that prevent actions executed in the constructor of an object or the initialization of static variables. Modification of RoadRunner to handle this case correctly may be possible, but we have not yet found a workaround for this version of Hominy. The source code and the output from Hominy for ChangeLocks are shown in Figures 5.1.

ReadShared

The program ReadShared, shown in Figure 5.5, exhibits the `read-shared` discipline for the `constant` variable. The main thread writes to `constant` and then forks three new threads. Each of these threads reads `constant` ten times. After all of these threads have terminated, the main thread joins each and accesses `constant` again. The specification of `constant`'s synchronization discipline is therefore:

```
thread-local; fork; read-shared; join; thread-local
```

The final `thread-local` we expect is correctly absorbed into the `join` discipline by Hominy, which infers the specification shown in Figure 5.5.

5.4.2 Larger Benchmarks

We also ran Hominy on a collection of larger and more realistic benchmarks to test the correctness of Hominy's inferred specifications in real world programs and to measure the performance of the analysis.

The Elevator [20] benchmark is a multithreaded elevator simulation program that uses mutual exclusion locks to synchronize access to shared data. Hominy monitored 376 variables in Elevator. Elevator pauses threads as part of its simulation, so the running time is not compute or IO-bound. The SOR benchmark is a scientific computing program that makes heavy use of cyclic barriers [20].

Source code:

```
public class ForkLock extends Thread {

  // lock is used to protected field once all threads are running.
  private static final Object lock = new Object();

  private static int field;

  public void run() {
    for (int i = 0; i < 2; i++) {
      synchronized(lock) {
        // Access guarded by lock
        field++;
      }
    }
  }

  public static void main(String[] args) {
    // The main thread initializes field.
    field = 0;
    // The main thread forks two other threads.
    Thread t1 = new ForkLock();
    Thread t2 = new ForkLock();
    t1.start();
    t2.start();
    t1.join();
    t2.join();
    // Now all other threads have terminated and been joined
    // by the main thread. The main thread accesses field again.
    System.out.printf("Field is %d, should be 4.", field);
  }
}
```

Specification:

```
Object: static
 Field: ForkLock.field
  Spec: thread-local T0;
        fork {T0};
        guarded-by {Lock:0x0171BBC9};
        join {T2}
```

Figure 5.4: The ForkLock program and the specification Hominy infers for it.

Source code:

```
public class ReadShared extends Thread {

  // After initialization threads only read constant.
  static int constant;

  public void run() {
    for (int i = 0; i < 10; i++) {
      // Read-only access
      System.out.println(constant);
    }
  }

  public static void main(String[] args) {
    // The main thread initializes constant.
    constant = 8;
    Thread[] ts = new Thread[3];
    // The main thread forks several other threads.
    for (int i = 0; i < 3; i++) {
      ts[i] = new ReadShared();
      ts[i].start();
    }
    // The main thread joins several other threads.
    for (Thread t : ts) {
      t.join();
    }
    // All other threads have terminated.
    // The main thread reads and writes constant.
    constant++;
    System.out.println("Constant is: " + constant);
  }

}
```

Specification:

```
Object: static
 Field: ReadShared.constant
  Spec: thread-local T0;
        fork {T0};
        read-shared;
        join {T1,T2,T3}
```

Figure 5.5: The ReadShared program and the specification Hominy infers for it.

```
Object: 0x0084DA23
 Field: ButtonPress.onFloor
  Spec: thread-local T0

Object: 0x002D9C06
 Field: Floor.upPeople
  Spec: read-shared

Object: 0x01E1A408
 Field: Vector.elementCount
  Spec: guarded-by {Lock:this}

...
```

Figure 5.6: Sample output from Hominy for the Elevator benchmark.

```
Object: static
 Field: SOR.iterations
  Spec: thread-local T0;
        fork/join {T0}

Object: static
 Field: SOR.red_
  Spec: thread-local T0

Object: 0x00BB7465
 Field: CyclicBarrier.barrierCommand_
  Spec: guarded-by {Lock:this}

...
```

Figure 5.7: Sample output from Hominy for the SOR benchmark.

Hominy monitored 20 variables in SOR. The TSP benchmark is a traveling salesman problem solver [20]. The MonteCarlo benchmark is a financial simulator [5]. Hominy monitored roughly 500,000 variables in MonteCarlo.

Hominy successfully inferred specifications for all race-free variables running online specification matching for each of these benchmarks except MonteCarlo, where two variables did not match any specification, although no races were reported. These variables had no accesses beyond their initialization. Since `thread-local` initializations in object constructors are missed by the RoadRunner framework, Hominy sees only the accesses after the initialization. For variables that are never accessed again, it is therefore impossible to infer a specification. (See Figure 5.6, Figure 5.7, Figure 5.8, and Figure 5.9 for sample specifications.) The TSP benchmark has one field which experiences race conditions, as reported by Hominy and verified with Goldilocks.

```
Object: 0x0192B996
 Field: jgfutil/JGFTimer.time
  Spec: guarded-by {Lock:0x00A8C488}

Object: 0x01C4B997
 Field: montecarlo/RatePath.nAcceptedPathValue
  Spec: thread-local T0

...
```

Figure 5.8: Sample output from Hominy for the MonteCarlo benchmark.

```
Object: static
 Field: TspSolver.MinLock
  Spec: read-shared

Object: static
 Field: TspSolver.TourStackTop
  Spec: thread-local T0;
        fork {T0};
        guarded-by {Lock:0x00F7F540}

Object: 0x01922221
 Field: PrioQElement.priority
  Spec: guarded-by {Lock:0x00F7F540}

...
```

Figure 5.9: Sample output from Hominy for the TSP benchmark.

Benchmark	LOC	Field Instances	Run Time (s)	RoadRunner Slowdown (x)	Hominy			
					Slowdown (x)		Memory (MB)	
					Online	Offline	Online	Offline
ChangeLocks	33		1.2	1.7	1.7			
ForkLock	45		1.2	1.3	1.3			
SyncObject	16		0.2	1.5	1.5			
ReadShared	30		10.2	1.0	1.1			
Elevator	529	376	6.1	1.7	2.2	2.8	64	76
MonteCarlo	3K	500K	2.5	2.2	1,498.6	-	1180	-
SOR	17K	20	0.5	2.6	610.0	38,484.0	37	20
TSP	706	136	0.2	8.5	409.5	1,405.0	27	-

Table 5.1: Performance of Hominy

These results suggest that Hominy is very effective at inferring synchronization disciplines for a variety of types of concurrent programs. Additionally, the large proportion of fields protected by simple synchronization disciplines and the limited length of the compound synchronization disciplines that Hominy observed shows that these programs generally following simple conventions to protected shared data against race conditions.

5.4.3 Performance

Table 5.1 shows performance statistics for Hominy in comparison to the performance of an instrumented program running under the RoadRunner framework with no analysis attached and the raw running time of each benchmark. All tests were run on a Dell Precision 490 with dual core 5000x series Intel Xeon processors and 2GB of memory, running FreeBSD 6.2. For both MonteCarlo and TSP, Hominy ran out of memory storing full per-variable ordered access traces in memory for offline specification matching. On the TSP benchmark, Hominy finished running the program, but ran out of memory analyzing the stored traces.

Hominy causes very little slowdown on our small benchmarks, as is expected. The computation involved in these benchmarks is trivial, so the overhead introduced by Hominy beyond that of RoadRunner itself is extremely small. For non-compute-bound programs such as Elevator, Hominy also performs well, with only a 2.2x slowdown over the running time of the uninstrumented Elevator program using the online specification matching algorithm.

Conservative implementation choices and a non-trivial analysis process make the Hominy tool relatively slow on large programs. However, several possible optimizations may boost performance. As our results for the MonteCarlo, SOR, and TSP benchmarks show, using an online specification matcher provides a significant performance improvement over an offline specification matcher that saves the entire program trace and analyzes it after the program has finished executing, especially for larger programs. Under the eager update model, large numbers of fields (MonteCarlo has roughly 500,000) introduce high cost for instrumentation, since the instrumentation store for every field must be updated on every synchronization action. However, the slowdown introduced by Hominy is clearly not correlated with only the number of fields, since SOR, with only 20 fields has a significantly higher slowdown under Hominy than does TSP, which has 136 fields.

From our limited results on compute-bound programs, the slowdown introduced by Hominy seems linked most closely with the running time of the program itself. This is unsurprising, given that a compute-bound program spends most of its time working with data in memory. A longer-running program generally performs proportionally more memory accesses than a short program.

Chapter 6

Conclusions

6.1 Contributions

We have developed the Grits specification language for specifying the synchronization disciplines a program follows to control access and prevent race conditions on each shared variable in the program. Grits specifications essentially describe the happens-before model of a program. As such, specifications in Grits can describe complex synchronization disciplines, but our experience shows that most programs use a small collection of simple synchronization disciplines. Though annotating code with synchronization disciplines can serve as useful human-readable documentation of a program in addition to a verifiable specification of the program, most code is not annotated. To find specifications for legacy code, lazy programmers, or complex programs, we have a developed Hominy, a dynamic specification inference tool that infers Grits synchronization discipline specifications of the shared variables in a program. Hominy is based on the same observations about the ordering of accesses to a variable under the happens-before model as is Grits. Our experience has shown that Hominy effectively infers specifications for a range of synchronization disciplines in real programs.

6.2 Future Work

6.2.1 Heap Abstraction

The current version of Hominy infers a specification for each variable (memory address) used in the execution of a program. To annotate source code with specifications of the synchronization discipline for each field in a class, it is necessary to merge the specifications for several instances of a class to a single specification. In some cases, there may be multiple distinct synchronization disciplines for fields in instances of the same class used in different contexts in the program. To map the memory addresses recorded by Hominy back to program names, it is necessary to perform heap abstraction similar to that done in FindLocks [18].

6.2.2 Precision of Inference

Although our experience shows that the specifications inferred by Hominy are correct in most cases, there are possible cases where the *EventSet* approximation of a happens-before graph can produce imprecise ancestors sets. Since an *EventSet* collapses all events by a single thread into a single logical event, ordering of events within a thread is lost. For example, consider the following trace:

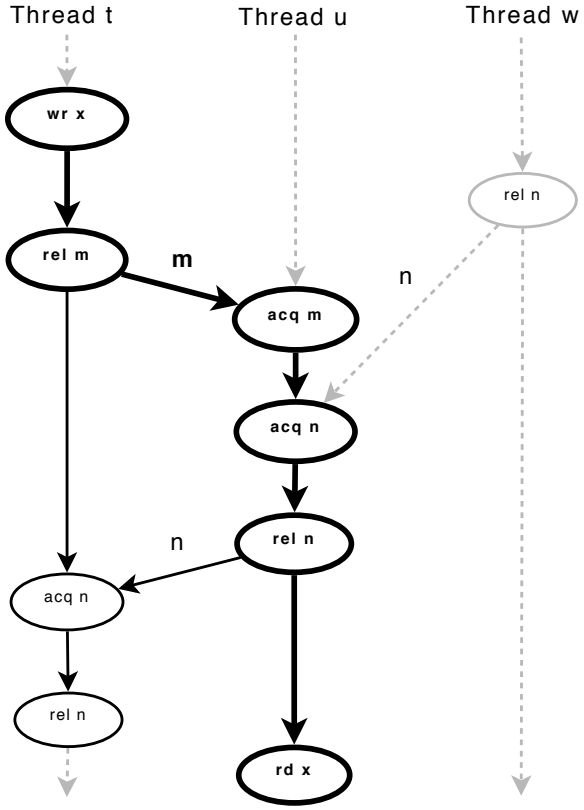
$$\begin{array}{lllll} \text{Thread } t: & wr_t(x,v); & rel_t(m); & & acq_t(n); rel_t(n); \\ \text{Thread } u: & & & acq_u(m); acq_u(n); rel_u(n); & rd_u(x,v) \\ \text{Thread } w: & rel_w(n); & & & \end{array}$$

The happens-before graph of this trace and its *EventSet* approximation are shown in Figure 6.1 along with the ancestors sets we derive from each. It is clear from the happens-before graph that lock n does not order the access by thread u. However, in the *EventSet*, the lack of thread-internal ordering causes Hominy to falsely assume that the event $rel_t(n)$ happened before the event $acq_u(n)$. As a result, the *OrderSet* computed by Hominy for the access by thread u includes lock m and lock n, although it should only include lock m.

Even if this situation does occur, the *OrderSet* inferred for this particular access is at worst a superset of the correct *OrderSet* and we do not lose any ordering information. Since these types of imprecision are not common, they are typically handled without extra work at the specification matching stage, where superset relation means that a specification that matches the correct *OrderSet* will also match the imprecise *OrderSet*. It is still possible that a specification that does not otherwise match a trace could match a trace with imprecise *OrderSet*s. Since Hominy is not a synchronization discipline checker and because it is a dynamic analysis, occasional imprecision is acceptable and can often be attributed to peculiarities of a single observed trace. Our experience suggests that imprecision due to the *EventSet* abstraction is uncommon and has little effect on the overall precision of Hominy's inferred specifications. In applications that cannot tolerate the possibility of imprecision of this form, a full happens-before graph can be used to compute orderings of accesses, but they must pay for increased memory and computational overhead.

6.2.3 Performance

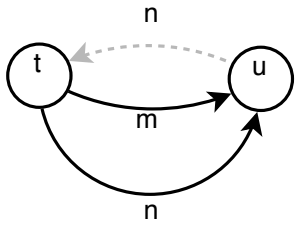
Hominy's relatively high slowdown and memory footprint are not surprising given the conservative implementation of this version of Hominy. Achieving good performance was not a central goal of this implementation, but a more optimistically concurrent implementation may make this implementation more efficient. Currently, every synchronization action reported by the instrumented program triggers a full traversal of all active *GuardState*s in the program. For large programs like MonteCarlo, in which Hominy instrumented 500,000 individual variables, the cost of such a full traversal is great. To avoid the cost of global updates, Goldilocks uses a lazy approach to updating its instrumentation store. Rather than updating each *GoldiSet* for every synchronization action, Goldilocks maintains a *synchronization event queue* to record each synchronization event. When an access to a variable x occurs, Goldilocks updates the *GoldiSet* for x and then analyzes the access to determine whether it



(a) The happens-before graph for the trace yields an *OrderSet* $\{m\}$.

$$\{\langle t \succcurlyeq_{rel} m\rangle, \langle w \succcurlyeq_{rel} n\rangle, \langle m \succcurlyeq_{acq} u\rangle, \langle n \succcurlyeq_{acq} u\rangle, \langle u \succcurlyeq_{rel} n\rangle, \langle n \succcurlyeq_{acq} t\rangle, \langle t \succcurlyeq_{rel} n\rangle\}$$

(b) *EventSet* constructed from the trace.



(c) Graph representation of the *EventSet*. The *EventSet* yields an *OrderSet* $\{m, n\}$.

Figure 6.1: An imprecise *EventSet* approximation of a happens-before graph.

is race-free. This lazy update policy makes the system much more complicated, but also significantly improves the performance of Goldilocks [8]. Applying a lazy update policy to Hominy would likely provide similar performance improvements.

6.2.4 Checking Synchronization Disciplines

Ultimately, we would like to be able to check that a program conforms to a set of synchronization disciplines. Hominy can easily be converted to a program checker by replacing inference steps with verification. However, as a dynamic analysis which approximates a happens-before model of a program, Hominy cannot guarantee the soundness we expect from program checkers. The model for Grits is not well-suited to static checking, since it is based on happens-before relations that are fully manifest only at run time. Nonetheless, some synchronization disciplines can be checked statically—`guarded-by` is an obvious choice—and a static checker which inserts run-time checks may be the most effective approach for balancing performance and soundness. Recent advances in automated testing techniques for better code coverage are another promising platform for a Grits specification checker.

Appendix A

Extensions for Concurrent Read Access

The definition of race-freedom presented in Chapter 3 requires that all accesses to a variable x be totally ordered. However, many race-free programs do not meet this requirement because they allow multiple threads to read constant values without ordering. Since read accesses to the variable x do not affect the value of x, no race conditions can exist among a set of read accesses. Because of this we make an exception to the requirement that all accesses to a variable be totally ordered. Reads to the same variable can be concurrent as shown in Figure A.1. We extend the Grits specification language and the Hominy specification analysis to handle this model.

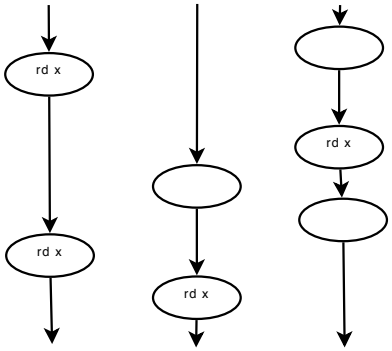


Figure A.1: Concurrent read accesses under the `read-shared` synchronization discipline.

A.1 Extensions to Grits

To allow concurrent read access to variables, we extend Grits as follows with the `read-shared` discipline:

SimpleDiscipline ::= `thread-local` | `fork` | `join` | `guarded-by` *Lock* | `vol` *VolatileVar* | `read-shared`

Discipline ::= *Discipline*; *SimpleDiscipline* | *SimpleDiscipline*

A variable x that is protected by the synchronization discipline `read-shared` may be read repeatedly by any number of threads, but must not be the target of any write accesses in the trace. The `read-shared` discipline is defined by an access pattern in which every access is a read and each access may be ordered by any set of synchronization devices, including the empty set.

$$\texttt{read-shared} \equiv (? \rhd rd_?)^+$$

The following trace is consistent with the synchronization discipline `read-shared` for x.

Thread t :	$wr_t(x,v); fork_t(u); fork_t(w);$		$rd_t(x,v);$
Thread u :		$rd_u(x,v);$	
Thread w :			$rd_w(x,v); rd_w(x,v)$

A.1.1 Additional Ordering Requirements

In the context of a larger concatenated synchronization discipline, additional ordering requirements are imposed on the concurrent read accesses allowed under the `read-shared` discipline. Although concurrent read accesses to the same variable are harmless, a read access executed concurrently with a write access causes a race condition and therefore must be prevented.

To accommodate write access and concurrent read access to variables, we redefine the requirement for race freedom presented in Section 3.1 with these two rules:

1. Every read of x in a program trace must be ordered with respect to every write to x in the trace.

2. Every write to x must be ordered with respect to other accesses (both reads and writes) to x in the trace.

For each write access a to the variable x in a trace, there must exist a happens-before relation between a and every read or write access $b \neq a$ in the trace. Read accesses may be concurrent to other read accesses, but by the write-ordering requirement, it is clear that no read access to x may happen concurrently with a write access to x.

By the transitivity of the happens-before relation, it is clear that we need only consider the previous and next accesses in a given case rather than all accesses. For any read access a to a variable x in a program, we require that there exist a happens-before relation from the last write

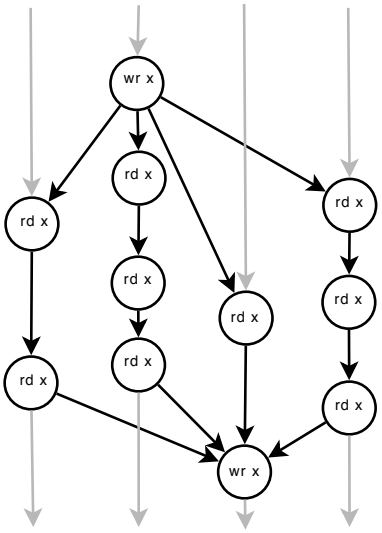


Figure A.2: Example happens-before graph for environments with concurrent read access.

to x to a. For each write access b to x, there must be a happens-before ordering from the last read access to x by each thread to b.

Figure A.2 is an example of a happens-before graph including concurrent read accesses preceded and followed by write accesses. Synchronization actions are not shown; edges represent happens-before orderings transitively closed over multiple happens-before edges representing thread-internal ordering and synchronization. Reads by different threads are concurrent, but each read happens after the previous write access and happens before the next write access.

In practice, `read-shared` is not typically concatenated with other synchronization disciplines besides `thread-local`; `fork`, since a typical `read-shared` variable is written only once at initialization. Commonly the main thread initializes these constants and then forks a number of other threads. Subsequent read accesses are therefore ordered by the `fork` operation.

A.2 Extensions to Hominy

To allow multiple threads to read a variable x simultaneously while no threads are writing to x, we extend the instrumented semantics to distinguish between read and write access, following the model used by Goldilocks.

The instrumentation store is redefined to include two parts. The instrumentation store φ_{wr} is a store which maps a program variable to an *EventSet*. The *EventSet* $\varphi_{wr}(x)$ is only reset to the accessing thread on write accesses to x. The instrumentation store φ_{rd} maps each variable-thread pair to an *EventSet*. The *EventSet* $\varphi_{rd}(x, t)$ is only reset to the accessing thread on read accesses

to x by thread t.

$$\varphi_{wr} : \mathit{Variable} \rightarrow \mathit{EventSet}$$
$$\varphi_{rd} : (\mathit{Variable} \times \mathit{Tid}) \rightarrow \mathit{EventSet}$$
$$\varphi : \varphi_{wr} \times \varphi_{rd}$$

The *EventSet* stored in $\varphi_{rd}(x,t)$ tracks the synchronization actions that have happened after the last read by thread t of variable x. The *EventSet* stored in $\varphi_{wr}(x)$ tracks the synchronization actions that have happened after the last write to x by any thread.

A.2.1 Recording Program Traces

Updates to *EventSet*s are performed according to the following rules:

1. On a write to x by thread t, $\varphi_{wr}(x)$ is reset to $\{\langle t \succcurlyeq_{wr} t\rangle\}$ and $\varphi_{rd}(x,t)$ is set to the empty set for all threads t.

2. On a read from x by thread t, $\varphi_{rd}(x,t)$ is reset to $\{\langle t \succcurlyeq_{rd} t\rangle\}$.

3. On a synchronization action a_s, for all x, an *Event* e representing a is added to $\varphi_{wr}(x)$ if $\varphi_{wr}(x)$ contains an event whose child is the parent of e. The same update is applied to $\varphi_{rd}(x,t)$ for all t.

These rules are formalized in Figure A.3 and Figure A.4. They follow the same form as the rules in Figure 4.3.

A.2.2 Selecting a Per-Variable Trace

We redefine the operation $\big|_x$ to return the projection of a variable x on an instrumentation trace Σ that replaces φ_i at each state i by $\varphi_i(x)$. We use the notation $\varphi(x)$ to abbreviate $(\varphi_{wr}(x) \times \varphi_{rd}(x))$.

$$\Omega = \big(\varphi_0 \xrightarrow{a_1} \varphi_1 \xrightarrow{a_2} \ldots \xrightarrow{a_n} \varphi_n\big) \big|_x = \big(\varphi_0(x) \xrightarrow{a_1} \varphi_1(x) \xrightarrow{a_2} \ldots \xrightarrow{a_n} \varphi_n(x)\big)$$

A.2.3 Inferring Ordering Between Accesses

To select the set of synchronization devices that order a certain access in an environment which allows concurrent reads, we redefined the *choose* and *ancestors* functions.

The two functions choose_{rd}, used for determining ordering of read accesses, and choose_{wr}, for determining ordering of write accesses, use the $\mathit{ancestors}_{rd}$ and $\mathit{ancestors}_{wr}$ sets, respectively. Otherwise, both functions select all the edge labels from the ancestors set.

[EXT INS READ BEFORE FIRST WRITE]

$$\frac{\forall y, u.\ \varphi'_{rd}(y,u) = \begin{cases} \varphi_{wr}(x) = \varnothing & \\ \{\langle u \succcurlyeq_{rd} u\rangle\} & \text{if } u = t \text{ and } y = x \\ \varphi_{rd}(y,u) & \text{otherwise} \end{cases}}{(\varphi_{wr}, \varphi_{rd}) \xrightarrow{rd_t(x,v)} (\varphi_{wr}, \varphi'_{rd})}$$

[EXT INS READ AFTER FIRST WRITE]

$$\frac{\forall y, u.\ \varphi'_{rd}(y,u) = \begin{cases} \langle ? \succcurlyeq_? t\rangle \in \varphi_{wr}(x) & \\ \{\langle u \succcurlyeq_{rd} u\rangle\} & \text{if } u = t \text{ and } y = x \\ \varphi_{rd}(y,u) & \text{otherwise} \end{cases}}{(\varphi_{wr}, \varphi_{rd}) \xrightarrow{rd_t(x,v)} (\varphi_{wr}, \varphi'_{rd})}$$

[EXT INS FIRST WRITE]

$$\frac{\begin{array}{c}\varphi_{wr}(x) = \varnothing \\ \forall u.\ (\varphi_{rd}(x,u) = \varnothing \text{ or } \langle ? \succcurlyeq_? t\rangle \in \varphi_{rd}(x,u)) \\ \forall y, u.\ \varphi'_{rd}(y,u) = \begin{cases} \varnothing & \text{if } y = x \\ \varphi_{rd}(y,u) & \text{otherwise} \end{cases}\end{array}}{(\varphi_{wr}, \varphi_{rd}) \xrightarrow{wr_t(x,v)} (\varphi_{wr}[x := \{\langle t \succcurlyeq_{wr} t\rangle\}], \varphi'_{rd})}$$

[EXT INS WRITE]

$$\frac{\begin{array}{c}\langle ? \succcurlyeq_? t\rangle \in \varphi_{wr}(x) \\ \forall u.\ (\varphi_{rd}(x,u) = \varnothing \text{ or } \langle ? \succcurlyeq_? t\rangle \in \varphi_{rd}(x,u)) \\ \forall y, u.\ \varphi'_{rd}(y,u) = \begin{cases} \varnothing & \text{if } y = x \\ \varphi_{rd}(y,u) & \text{otherwise} \end{cases}\end{array}}{(\varphi_{wr}, \varphi_{rd}) \xrightarrow{wr_t(x,v)} (\varphi_{wr}[x := \{\langle t \succcurlyeq_{wr} t\rangle\}], \varphi'_{rd})}$$

Figure A.3: Extended update rules for accesses.

[INS RELEASE]

$$\frac{\begin{array}{c} a \in \{rel_t(m)\} \\ \forall x.\ \varphi'_{wr}(x) = \begin{cases} \varphi_{wr}(x) \ \cup\ \{\langle t \succcurlyeq_{rel} m\rangle\} & \text{if } \langle ? \succcurlyeq_? t\rangle \in x \\ \varphi_{wr}(x) & \text{otherwise} \end{cases} \\ \forall x, u.\ \varphi'_{rd}(x,u) = \begin{cases} \varphi_{rd}(x,u) \ \cup\ \{\langle t \succcurlyeq_{rel} m\rangle\} & \text{if } \langle ? \succcurlyeq_? t\rangle \in x \\ \varphi_{rd}(x,u) & \text{otherwise} \end{cases} \end{array}}{(\varphi_{wr}, \varphi_{rd}) \xRightarrow{a} (\varphi'_{wr}, \varphi'_{rd})}$$

[INS ACQUIRE]

$$\frac{\begin{array}{c} a \in \{acq_t(m)\} \\ \forall x.\ \varphi'_{wr}(x) = \begin{cases} \varphi_{wr}(x) \ \cup\ \{\langle m \succcurlyeq_{acq} t\rangle\} & \text{if } \langle ? \succcurlyeq_? m\rangle \in x \\ \varphi_{wr}(x) & \text{otherwise} \end{cases} \\ \forall x, u.\ \varphi'_{rd}(x,u) = \begin{cases} \varphi_{rd}(x,u) \ \cup\ \{\langle m \succcurlyeq_{acq} t\rangle\} & \text{if } \langle ? \succcurlyeq_? m\rangle \in x \\ \varphi_{rd}(x,u) & \text{otherwise} \end{cases} \end{array}}{(\varphi_{wr}, \varphi_{rd}) \xRightarrow{a} (\varphi'_{wr}, \varphi'_{rd})}$$

[INS VOLATILE WRITE]

$$\frac{\begin{array}{c} a \in \{wr_t(\hat{y}, v)\} \\ \forall x.\ \varphi'_{wr}(x) = \begin{cases} \varphi_{wr}(x) \ \cup\ \{\langle t \succcurlyeq_{wrv} \hat{y}\rangle\} & \text{if } \langle ? \succcurlyeq_? t\rangle \in x \\ \varphi_{wr}(x) & \text{otherwise} \end{cases} \\ \forall x, u.\ \varphi'_{rd}(x,u) = \begin{cases} \varphi_{rd}(x,u) \ \cup\ \{\langle t \succcurlyeq_{wrv} \hat{y}\rangle\} & \text{if } \langle ? \succcurlyeq_? t\rangle \in x \\ \varphi_{rd}(x,u) & \text{otherwise} \end{cases} \end{array}}{(\varphi_{wr}, \varphi_{rd}) \xRightarrow{a} (\varphi'_{wr}, \varphi'_{rd})}$$

[INS VOLATILE READ]

$$\frac{\begin{array}{c} a \in \{rd_t(\hat{y}, v)\} \\ \forall x.\ \varphi'_{wr}(x) = \begin{cases} \varphi_{wr}(x) \ \cup\ \{\langle \hat{y} \succcurlyeq_{rdv} t\rangle\} & \text{if } \langle ? \succcurlyeq_? \hat{y}\rangle \in x \\ \varphi_{wr}(x) & \text{otherwise} \end{cases} \\ \forall x, u.\ \varphi'_{rd}(x,u) = \begin{cases} \varphi_{rd}(x,u) \ \cup\ \{\langle \hat{y} \succcurlyeq_{rdv} t\rangle\} & \text{if } \langle ? \succcurlyeq_? \hat{y}\rangle \in x \\ \varphi_{rd}(x,u) & \text{otherwise} \end{cases} \end{array}}{(\varphi_{wr}, \varphi_{rd}) \xRightarrow{a} (\varphi'_{wr}, \varphi'_{rd})}$$

[INS FORK]

$$\frac{\begin{array}{c} \forall x.\ \varphi'_{wr}(x) = \begin{cases} \varphi_{wr}(x) \ \cup\ \{\langle t \succcurlyeq_{fork} w\rangle\} & \text{if } \langle ? \succcurlyeq_? t\rangle \in x \\ \varphi_{wr}(x) & \text{otherwise} \end{cases} \\ \forall x, u.\ \varphi'_{rd}(x,u) = \begin{cases} \varphi_{rd}(x,u) \ \cup\ \{\langle t \succcurlyeq_{fork} w\rangle\} & \text{if } \langle ? \succcurlyeq_? t\rangle \in x \\ \varphi_{rd}(x,u) & \text{otherwise} \end{cases} \end{array}}{(\varphi_{wr}, \varphi_{rd}) \xRightarrow{fork_t(w)} (\varphi'_{wr}, \varphi'_{rd})}$$

[INS JOIN]

$$\frac{\begin{array}{c} \forall x.\ \varphi'_{wr}(x) = \begin{cases} \varphi_{wr}(x) \ \cup\ \{\langle w \succcurlyeq_{join} t\rangle\} & \text{if } \langle ? \succcurlyeq_? w\rangle \in x \\ \varphi_{wr}(x) & \text{otherwise} \end{cases} \\ \forall x, u.\ \varphi'_{rd}(x,u) = \begin{cases} \varphi_{rd}(x,u) \ \cup\ \{\langle w \succcurlyeq_{join} t\rangle\} & \text{if } \langle ? \succcurlyeq_? w\rangle \in x \\ \varphi_{rd}(x,u) & \text{otherwise} \end{cases} \end{array}}{(\varphi_{wr}, \varphi_{rd}) \xRightarrow{\langle w \succcurlyeq_{join} t\rangle} (\varphi'_{wr}, \varphi'_{rd})}$$

Figure A.4: Extended update rules for synchronization using locks, volatile variables, and thread control.

Definition A.1. *choose*$(\varphi(x), t)$

$$
\begin{aligned}
choose_{rd}(\varphi(x),t) = \{&m \in Lock \mid \langle m \gg_{acq} ? \rangle \in ancestors_{rd}(\varphi(x),t) \text{ and} \\
&\langle ? \gg_{rel} m \rangle \in ancestors_{rd}(\varphi(x),t)\} \\
\cup\ \{&\hat{y} \in VolatileVar \mid \langle \hat{y} \gg_{rdv} ? \rangle \in ancestors_{rd}(\varphi(x),t) \text{ and} \\
&\langle ? \gg_{wrv} \hat{y} \rangle \in ancestors_{rd}(\varphi(x),t)\} \\
\cup\ \{&t^{\mathrm{F}} \in ForkingTid \mid \langle t \gg_{fork} ? \rangle \in ancestors_{rd}(\varphi(x),t)\} \\
\cup\ \{&u^{\mathrm{J}} \in JoinedTid \mid \langle u \gg_{join} ? \rangle \in ancestors_{rd}(\varphi(x),t)\} \\
choose_{wr}(\varphi(x),t) = \{&m \in Lock \mid \langle m \gg_{acq} ? \rangle \in ancestors_{wr}(\varphi(x),t) \text{ and} \\
&\langle ? \gg_{rel} m \rangle \in ancestors_{wr}(\varphi(x),t)\} \\
\cup\ \{&\hat{y} \in VolatileVar \mid \langle \hat{y} \gg_{rdv} ? \rangle \in ancestors_{wr}(\varphi(x),t) \text{ and} \\
&\langle ? \gg_{wrv} \hat{y} \rangle \in ancestors_{wr}(\varphi(x),t)\} \\
\cup\ \{&t^{\mathrm{F}} \in ForkingTid \mid \langle t \gg_{fork} ? \rangle \in ancestors_{wr}(\varphi(x),t)\} \\
\cup\ \{&u^{\mathrm{J}} \in JoinedTid \mid \langle u \gg_{join} ? \rangle \in ancestors_{wr}(\varphi(x),t)\}
\end{aligned}
$$

The simple ancestor-tracing approach to inferring ordering for a given access must be augmented for this environment. To properly handle concurrent reads, we redefine the *ancestors* and *choose* functions to consider the ordering rules for concurrent read accesses. For read accesses to x, which must be ordered from the last write access to x but not other read accesses $ancestors(S,t)$ uses only the *EventSet* for the last write to x. For write accesses to x, which must be ordered from the last write to x and all of the reads of x since the last write to x, $ancestors(S,t)$ finds the intersection of the *EventSet*s for the last write and all last reads in the case of a write access.

The ancestors of an access *Event* are defined by the least fixed point of the function *ancestors*.

Definition A.2. *ancestors*$(\varphi(x), t)$

$$
\begin{aligned}
ancestors_{rd}(\varphi(x),t) &= \{\langle ? \gg_{?} s \rangle \in \varphi_{wr}(x) \mid \langle s \gg_{?} ? \rangle \in ancestors_{rd}(\varphi(x),t)\} \ \cup\ \{\langle t \gg_{rd} t \rangle\} \\
ancestors_{wr}(\varphi(x),t) &= \{\langle ? \gg_{?} s \rangle \mid \exists u \neq t.\ (\varphi_{rd}(x,u) = \varnothing \text{ or } \langle ? \gg_{?} s \rangle \in \varphi_{rd}(x,u)) \\
&\qquad \text{or } \langle s \gg_{?} ? \rangle \in ancestors_{wr}(\varphi(x),t)\} \ \cup\ \{\langle t \gg_{wr} t \rangle\}
\end{aligned}
$$

An *Event* e is in the least fixed point of $ancestors_{wr}(\varphi(x), t)$ if any of the conditions below are met:

1. *Event* e is the *Event* $\langle t \gg_{wr} t \rangle$, representing the next action by t (presumably a write access to x, since this is the $ancestors_{wr}$ function).
2. *Event* e is on a happens-before path from the last write access to x to the next action of t.
3. *Event* e is on a happens-before path from the last read access to x by a thread other than t to the next action of t, where this read access happens after the last write access to x.

[EXT ORDER BEGIN]

$$\frac{}{\top \vdash_x \varphi}$$

[EXT ORDER NON-ACCESS]

$$\frac{O \vdash_x \Omega \qquad \forall t, v.\ a \notin \{rd_t(x,v), wr_t(x,v)\}}{O \vdash_x \Omega \overset{a}{\Longrightarrow} \varphi}$$

[EXT ORDER READ]

$$\frac{\begin{array}{c}\Omega = (\ldots \overset{a}{\Longrightarrow} \varphi) \\ O \vdash_x \Omega \\ O' = choose_{rd}(ancestors_r(\varphi, x, t), \varphi, x) \rhd rd_t \\ \varphi' = (\varphi_{wr}, \varphi_{rd}[(x,t) = \{t\}])\end{array}}{O; O' \vdash_x (\Omega \overset{rd_t(x,v)}{\Longrightarrow} \varphi')}$$

[EXT ORDER WRITE]

$$\frac{\begin{array}{c}\Omega = (\ldots \overset{a}{\Longrightarrow} \varphi) \\ O \vdash_x \Omega \\ O' = choose_{wr}(ancestors_w(\varphi, x, t), \varphi, x, t) \rhd wr_t \\ \varphi'_{wr} = \varphi_{wr}[x := t] \\ \forall y, u, v.\ \varphi'_{rd}(y,u) = \begin{cases} \varnothing & \text{if } y = x \\ \varphi_{rd}(y,u) & \text{otherwise} \end{cases} \\ \varphi' = (\varphi'_{wr}, \varphi'_{rd})\end{array}}{O; O' \vdash_x (\Omega \overset{wr_t(x,v)}{\Longrightarrow} \varphi')}$$

Figure A.5: Extended ordered access trace construction rules

A.2.4 Ordered Access Traces

The rules [EXT ORDER BEGIN], [EXT ORDER NON-ACCESS], [EXT ORDER READ] and [EXT ORDER WRITE], defined in Figure A.5, apply these new *ancestors* and *choose* functions build an ordered access trace O for a trace Ω that may contain concurrent read accesses to a single variable. These rules form an access trace that reflects ordered read and write accesses, allowing for concurrent reads.

A.2.5 Simplifying Ordered Access Traces

Ordered accesses and traces are simplified via the rewrite rules defined in Section 4.6. No other rules are necessary.

A.2.6 Recognizing Synchronization Disciplines

In addition to the disciplines described in Section 4.7, we add the `read-shared` discipline, presented in Section A.1. The `read-shared` discipline is recognized as a sequence of read accesses by any thread and with any ordering with no write accesses, as described by the access pattern for `read-shared`:

$$\left(? \rhd rd_?\right)^+$$

Bibliography

[1] Abadi, M., Flanagan, C., and Freund, S. N. Types for safe locking: Static race detection for Java. *ACM Transactions on Programming Languages and Systems 28*, 2 (2006), 207–255.

[2] Agarwal, R., and Stoller, S. Type inference for parameterized race-free Java. In *Conference on Verification, Model Checking and Abstract Interpretation* (Jan. 2004), vol. 2937 of *Lecture Notes in Computer Science*, pp. 149–160.

[3] Banerjee, U., Bliss, B., Ma, Z., and Petersen, P. A theory of data race detection. In *Workshop on Parallel and Distributed Systems: Testing and Debugging* (2006), pp. 69–78.

[4] Boyapati, C., and Rinard, M. A parameterized type system for race-free Java programs. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications* (2001), pp. 56–69.

[5] Bull, J. M., Smith, L. A., Westhead, M. D., Henty, D. S., and Davey, R. A. A benchmark suite for high performance Java. *Concurrency: Practice and Experience 12*, 6 (May 2000), 375–388.

[6] Choi, J.-D., Lee, K., Loginov, A., O'Callahan, R., Sarkar, V., and Sridharan, M. Efficient and precise datarace detection for multithreaded object-oriented programs. In *Conference on Programming Language Design and Implementation* (2002), pp. 258–269.

[7] Christiaens, M., and Bosschere, K. D. Trade, a topological approach to on-the-fly race detection in Java programs. In *Symposium on Java Virtual Machine Research and Technology* (2001), pp. 15–15.

[8] Elmas, T., Qadeer, S., and Tasiran, S. Goldilocks: A race and transaction-aware Java runtime. In *Conference on Programming Language Design and Implementation* (2007), pp. 245–255.

[9] Flanagan, C., and Freund, S. N. Type-based race detection for Java. In *Conference on Programming Language Design and Implementation* (2000), pp. 219–232.

[10] Flanagan, C., and Freund, S. N. Type inference against races. *Science of Computer Programming 64*, 1 (2007), 140–165.

[11] Lamport, L. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM 21*, 7 (July 1978), 558–565.

[12] Leveson, N. G., and Turner, C. S. An investigation of the Therac-25 accidents. *IEEE Computer 26*, 7 (July 1993), 18–41.

[13] O'Callahan, R., and Choi, J.-D. Hybrid dynamic data race detection. In *Symposium on Principles and Practice of Parallel Programming* (2003), pp. 167–178.

[14] Poulsen, K. Tracking the blackout bug. http://www.securityfocus.com/news/8412, retrieved 11 May 2008.

[15] Pratikakis, P., Foster, J. S., and Hicks, M. LOCKSMITH: context-sensitive correlation analysis for race detection. In *Conference on Programming Language Design and Implementation* (2006), pp. 320–331.

[16] Raynal, M. About logical clocks for distributed systems. *Operating Systems Review 26*, 1 (1992), 41–48.

[17] Ronsse, M., and Bosschere, K. D. Recplay: a fully integrated practical record/replay system. *ACM Transactions on Computing Systems 17*, 2 (1999), 133–152.

[18] Rose, J., Swamy, N., and Hicks, M. Dynamic inference of polymorphic lock types. *Science of Computer Programming 58*, 3 (December 2005), 366–383.

[19] Savage, S., Burrows, M., Nelson, G., Sobalvarro, P., and Anderson, T. E. Eraser: A dynamic data race detector for multi-threaded programs. In *Symposium on Operating Systems Principles* (1997), pp. 27–37.

[20] von Praun, C., and Gross, T. Static conflict analysis for multi-threaded object-oriented programs. In *Conference on Programming Language Design and Implementation* (2003), pp. 115–128.