Detecting Synchronization Defects in Multi-Threaded Object-Oriented Programs

Christoph von Praun

Diss. ETH No. 15524

Detecting Synchronization Defects in Multi-Threaded Object-Oriented Programs

A dissertation submitted to the
SWISS FEDERAL INSTITUTE OF TECHNOLOGY ZURICH
(ETH ZÜRICH)

for the degree of
Doctor of Technical Sciences

presented by
Christoph von Praun
Dipl.-Inf. TU-München
born January 20, 1972
citizen of Germany

accepted on the recommendation of
Prof. Dr. Thomas Gross, examiner
Prof. Dr. David Padua, co-examiner
Prof. Dr. Robert Stärk, co-examiner

2004

Abstract

This dissertation describes an efficient and automated approach to determine synchronization defects in multi-threaded object-oriented programs. The approach is based on the key observation that object-oriented programs provide guarantees about data confinement and encapsulation that can be leveraged by the design of a static analysis and a runtime checker. To be practical, the techniques are demonstrated on the Java programming language.

The static analysis operates on an abstract model of threads and data, and simulates the execution of a parallel program on these abstract domains. Thereby, precise information about locking, thread activity, object access, and object escape is recorded in a context-sensitive manner. This symbolic execution provides a general platform to assess properties of parallel programs. The focus of this dissertation is on the detection of three possible sources of errors: data races, methods that may not execute atomically, and deadlock. The static analysis of object-oriented languages is generally limited by the effects of aliasing and the resulting difficulty to disambiguate dynamically allocated data and locks. While approximations of the static analysis reduce the accuracy of the results, we have found the reports of potential synchronization defects to be highly useful in practice: Overreporting may occur, however at a rate that is amenable to manual inspection. True defects may be overlooked but such underreporting can be limited to certain cases that we observed rarely in practice.

Two alternative software mechanisms are developed to assess concurrency and locking at runtime: First, object race detection checks if access to shared objects follows a locking discipline. Second, object consistency guarantees that threads behave so that access to individual objects is serializable and happens without harmful interference. Both mechanisms are implemented as a sparse program instrumentation that is guided by the static analysis and optimized with standard compiler techniques. The runtime overhead is very low (on average 44% for object race checking and 25% for object consistency) and well spent in the light of the benefits.

The trend towards thread-level parallelism and multi-threaded computer systems make precise information about concurrency and synchronization indispensable for correct program translation, optimization, and execution. The techniques presented in this dissertation are promising steps towards providing this information and making the detection of synchronization defects a default option for compilers and runtime systems of multi-threaded object-oriented programs.

Kurzfassung

Die vorliegende Arbeit beschreibt einen effizienten und automatisierten Ansatz zur Erkennung von Synchronisationsfehlern in nebenläufigen objektorientierten Programmen. Der Ansatz basiert auf der Beobachtung, dass objektorientierte Programme gewisse Garantien über die Erreichbarkeit und Kapselung von Daten festlegen; solche Garantien können die Effizienz und Genauigkeit von statischen und dynamischen Verfahren zur Erkennung von Synchronisationsfehlern verbessern. Die in dieser Dissertation entwickelten Techniken werden an der Programmiersprache Java beispielhaft vorgestellt und evaluiert.

Die statische Analyse basiert auf einem symbolischen, d.h. simulierten, Programmablauf mit abstraktem Thread- und Datenmodell. Die Simulation registriert die Verwendung von Locks, Objektzugriffe und die Erreichbarkeit von Objekten durch nebenläufige Threads. Die Simulation ist kontext-sensitiv, d.h. bei der Analyse einzelner Methodenaufrufe wird der Programm- und Datenkontext, in welchem eine Methode zur Ausführung gelangt, berücksichtigt. Die symbolische Programmausführung ist die Grundlage für weitere Analysen. Der Schwerpunkt in der vorliegenden Arbeit liegt auf Analysen zur Erkennung von drei möglichen Fehlerquellen: Data Races, Methoden deren Ausführung nicht atomar sein könnte und Deadlock. Die Analyse von objektorientierten Sprachen wird generell durch Aliasing und durch die daraus resultierende Ungenauigkeit, dynamisch allozierte Objekte und Locks zu unterscheiden, erschwert. Obgleich die notwendigen Abstraktions- und Näherungsverfahren die Präzision reduzieren, sind die Ergebnisse der statischen Erkennung von Synchronisationsfehlern äußerst nützlich in der Praxis. Einige Warnungen haben keine Entsprechung in tatsächlichen Programmausführungen (Overreporting); solche Warnungen sind typisch, sie tauchen jedoch nur in relativ geringer Anzahl auf und können mit moderatem Aufwand vom Benutzer erkannt und selektiert werden. Es ist andererseits möglich, dass die entwickelten Analyseverfahren keine Warnung generieren für Programme, die tatsächlich Synchronisationsfehler aufweisen (Underreporting). Dieses Phänomen tritt jedoch nur in Situationen auf, die wir genau bestimmen können, die aber nur selten in der Praxis vorkommen.

Neben den statischen Analysen werden zwei software-basierte Techniken vorgestellt, welche Informationen über aktuelle Locks und Nebenläufigkeit zur Laufzeit verfügbar machen. Der erste Mechanismus, genannt Object Race Detection, überprüft, ob nebenläufige Threads beim Zugriff auf gemeinsame Objekte eine bestimmte Lockdisziplin einhalten. Der zweite Mechanismus, genannt Object Consistency, garantiert, dass die Effekte von Objektzugriffen serialisierbar sind und keine Interferenz von Threads auftaucht, die zu Inkonsistenz von Daten führen könnte. Beide Mechanismen sind in Form einer Programminstrumentierung implementiert, die durch die Ergebnisse der statischen Analyse gesteuert und mit gängigen Compilertechniken optimiert wird; Ziel ist es möglichst wenige Objektzugriffe zu instrumentieren. Die Laufzeitkosten dieser Instrumentierungsvarianten sind sehr gering (im Mittel 44% bei Ob-

ject Race Detection und 25% bei Object Consistency) und lohnenswert angesichts der Vorteile, die die Erkennung von Synchronisationsfehlern mit sich bringt.

Der Trend zu Multithreading in Software und Hardware Systemen erfordert die Verfügbarkeit von genauer Information über Nebenläufigkeit und Synchronisation zur korrekten Übersetzung, Optimierung und Ausführung eines Programms. Die Techniken, welche in dieser Dissertation vorgestellt werden, sind erste Schritte zur Bereitstellung dieser Information und zur Etablierung von Mechanismen zur Erkennung von Synchronisationsfehlern in Compiler- und Laufzeitsystemen für objektorientierte nebenläufige Programme.

Acknowledgment

This dissertation would not have been possible without the tireless efforts of my advisor, Professor Thomas Gross. He has taught me to work as an independent researcher and has been a steady source of wise encouragement and support. He will be a great example throughout my professional life and I would like to express my sincere thanks to him.

I am indebted to Professor David Padua and Professor Robert Stärk for their scrutinizing comments that helped me to improve this dissertation. My sincere thanks also go to Professor Gustavo Alonso who notably supported me as a graduate student at ETH Zurich.

I am extraordinarily grateful to all my friends and colleagues in the Computer Systems Institute at ETH Zurich. This exceptional group of people created a collaborative environment that is friendly, well-organized, and stimulating, allowing to focus on learning and research. A number of colleagues and students, past and present, have helped to develop the compiler infrastructure that is used in this dissertation. It is a great pleasure to thank them here for their commitment.

Contents

1 Introduction **1**
- 1.1 Motivation . . . 1
- 1.2 Scope . . . 2
- 1.3 Thesis . . . 2
- 1.4 Outline . . . 3

2 Background **5**
- 2.1 Terminology . . . 5
- 2.2 Data races . . . 5
- 2.3 Violations of atomicity . . . 12
- 2.4 Deadlock . . . 13
- 2.5 Trails to correct synchronization . . . 13

3 Static analysis **17**
- 3.1 Preliminaries . . . 17
- 3.2 Abstract threads . . . 19
- 3.3 Reference analysis . . . 21
- 3.4 Symbolic execution . . . 31
- 3.5 Experience . . . 38
- 3.6 Discussion . . . 45

4 Static detection of data races **49**
- 4.1 Object use graphs . . . 49
- 4.2 Building object use graphs . . . 52
- 4.3 Approximating happened-before . . . 60
- 4.4 Conflict detection . . . 66
- 4.5 Experience . . . 68
- 4.6 Extensions for weak memory models . . . 78
- 4.7 Discussion . . . 81

5 Static detection of atomicity violations **87**
- 5.1 Method consistency . . . 87
- 5.2 Algorithm . . . 91
- 5.3 Experience . . . 94
- 5.4 Discussion . . . 98

6 Static deadlock detection **105**
6.1 Resource deadlock 105
6.2 Algorithm 105
6.3 Experience 108
6.4 Discussion 109

7 Dynamic checking **111**
7.1 Object race detection 111
7.2 Detecting violations of object consistency 128
7.3 Method specialization 138

8 Conclusions **145**
8.1 Summary and contributions 146
8.2 Trends and future work 147

1

Introduction

1.1 Motivation

A common model for parallel programming is *multi-threading with shared memory*: Multi-threading is not only attractive for its potential to increase the performance of independent computations, but is also used as a concept for structuring tasks within a software system. Popular object-oriented programming languages like Java [53] and C♯ [38] have adopted multi-threading as a language feature.

Problem. In these languages, parallelism is specified and controlled explicitly by the programmer, inviting new sources of programming errors, called *synchronization defects*. Such errors are not known in sequential programming. In this dissertation, we consider three important classes of synchronization defects:

1. A *data race* [92] is a situation where events from different threads execute without ordering and read and write the same data. Data races can lead to data inconsistency and unintended nondeterminism.

2. A *violation of atomicity* [45] occurs if a sequence of shared data access of one thread is interleaved with access to the same data from other threads.

3. A *deadlock* [28] situation occurs at runtime if threads use synchronization so that a cyclic wait condition arises.

There are static and dynamic techniques for detecting particular classes of synchronization defects. Static tools are challenged by the complexity of thread interaction and the dynamic allocation of data and threads. Precise tools are typically very resource intensive. Tools with an approximative detection strategy depend on program annotations and user input to grasp the synchronization discipline and reduce the number of false reports. Purely dynamic tools can be precise but their findings are limited to particular program execution. In addition, these tools are confronted with a runtime overhead of typically a factor of 2 to 60; the overhead stems from program instrumentation or monitoring facilities. Most promising are combinations of static and dynamic techniques, e.g., a static analysis that guides and narrows the scope of a dynamic checker.

Approach. We have developed a static analysis framework that can approximate dynamic techniques for the detection of synchronization defects at compile-time. The framework is based

on a symbolic execution of an abstract model of threads and data, combining the availability of high-level program information in a compiler with the versatility of a dynamic checker. The framework is used to address the three classes of synchronization defects: data races, violations of atomicity (at the method level), and deadlock.

Research questions. The proposed model for detecting synchronization defects in multi-threaded object-oriented programs raises the following key questions:

- *Are multi-threaded object-oriented programs with a variety of synchronization patterns, indirect data access, polymorphism, and unstructured parallelism amenable to an efficient static detection of synchronization defects?*
- *What is the precision of the static detection in terms of underreporting and overreporting?*
- *How many residual cases that cannot be resolved through static analysis occur at runtime? How large is the runtime overhead if these cases are monitored by a dynamic checker?*

1.2 Scope

The static analyses developed in this dissertation assume the availability of the whole program; the presented algorithms are designed for a way-ahead compilation and link model. Our approach to synchronization fault detection has been developed with a certain application model in mind:

- Parallelism is unstructured, i.e., concurrency among statements cannot be easily determined from the static structure of the program. Threads are modeled as objects and their lifetime is mostly independent of the dynamic program scoping.
- Locks and monitors are the predominant synchronization mechanisms.
- Memory management is done automatically, e.g., through a garbage collector.
- Objects, not plain arrays, are the main abstraction for representing shared data.
- The language provides a clean object model so that objects are only accessed through corresponding references; pointers cannot be forged and there is no pointer arithmetic.

Programs that do not follow this model can still be treated correctly, however conservatism might deteriorate the precision of the analysis.

1.3 Thesis

- *Structural properties of object-oriented programs, such as data encapsulation and object confinement, offer new opportunities for the detection of synchronization defects.*

- *Static analysis of multi-threaded object-oriented programs can be precise enough to deliver useful information about potential synchronization defects to the programmer and to determine the absence of defects for large program parts.*
- *The execution overhead of the residual dynamic checking is typically low; certain guarantees about the absence of synchronization defects in a program execution and the compliance of a program execution with a high-level memory model can be given.*

1.4 Outline

The thesis is established as follows:

- We describe the design and implementation of a static analysis framework that enables a heap context-sensitive symbolic execution of multi-threaded object-oriented programs. The duration of the symbolic execution is demonstrated; optimizations are proposed and implemented.
- The framework is used to implement static analyses for the detection of three important classes of synchronization defects: (1) data races, (2) violations of atomicity at the method level, and (3) deadlock. The analyses are inspired by known dynamic detection procedures.
- The accuracy of the analyses is evaluated and compared with the runtime reality of benchmark programs that are representative for object-oriented applications and Internet services. To be practical, we use the Java programming language and common Java benchmark programs as a base for our study.
- We present two alternative software mechanisms to determine concurrency and locking at runtime: (1) Object race detection checks if shared object access follows a locking discipline. (2) Object consistency, verifies if threads behave so that access to individual objects happens in a serializable order.
- Both dynamic checkers are integrated with the static analysis and implemented as a sparse program instrumentation. After a discussion of the theoretical and practical detection capabilities of both systems, the runtime overhead is evaluated.

The dissertation is organized as follows: **Chapter 2** defines the notion of synchronization defects and summarizes issues around the detection of data races, violations of atomicity, and deadlock. We present our model to address the problem of synchronization defects and detail on the roles of programmers, compilers, and execution platform in this model. **Chapter 3** discusses the static analysis framework. First, various data structures for the abstract representation of programs and their data are introduced; we describe their construction and continue with the algorithm of the symbolic execution. **Chapters 4 to 6** describe the use of analysis framework for implementing algorithms that detect three classes of synchronization defects: data races, violations of atomicity, and deadlock. For each technique, we evaluate the resource requirements and the precision of the static analysis. **Chapter 7** describes two program instrumentations and runtime mechanisms that determine synchronization defects (object races) and memory consistency at the level of objects (object consistency). We illustrate how the context-sensitive

information from the static analysis is used to realize code specializations; code specialization is important to differentiate heap execution contexts at runtime which is the basis for the sparse runtime checking. An evaluation of the accuracy and cost of the system are given. **Chapter 8** addresses directions for future research and concludes this dissertation.

2

Background

The goal of this dissertation is to develop a compilation and execution model for parallel programs that allows to detect and reject programs that are afflicted with synchronization defects. This chapter lays the foundations for our work and describes principal aspects of parallel programs. We define a formalism for describing parallel programs and program executions in Section 2.1. Sections 2.2 to 2.4 discuss different classes of error conditions and methods for their detection. Related work is described in this chapter as far as it contributes to the foundations; subsequent chapters discuss related work on individual aspects in more detail. Finally, common methodical approaches to the development of parallel programs (Section 2.5) are described. These approaches are characterized by the roles and obligations of programmer, compiler, and runtime system to determine the correctness of the synchronization.

2.1 Terminology

We adopt the terminology and notation from Choi et al. [25, 109]. A *program execution* is defined as a sequence $e_0, \ldots, e_n$ of *events* where e_i is defined as a tuple $\langle o, f, t, L, k \rangle$:

- i is a unique id.
- o is the object instance (or class) that is being accessed.
- f is the field variable that is accessed inside the object.
- t is the thread accessing the object.
- L is the set of locks being held during the access.
- k is the kind of event (one of $\{\text{READ}, \text{WRITE}, \text{LOCK}, \text{UNLOCK}, \text{START}, \text{JOIN}\}$).

Events shall not only be used to model variable access, but also lock and unlock, as well as thread start and join; for such events, the accessed field variable f is void.

2.2 Data races

A *critical section* is a statement sequence that should execute without interference of other threads. The concept is useful to guarantee that access from different threads to the same data is

ordered, avoiding inconsistency and data corruption. *Races* are used to characterize situations where the guarantee about non-interference of threads accessing shared data is violated. Netzer and Miller [92] distinguish *data races* that refer to unsynchronized access of shared data and *general races* that are sources of nondeterminism in parallel programs in general. Our discussion in this section focuses on data races, while Section 2.3 discusses synchronization defects related to general races.

Intuitive definition. An intuitive definition of a data race is that different threads access the same data, at least one access is a write, and the access events are not ordered in the flow of the execution.

Static detection. The set of data races that are possible by the nature of a program (i.e., all possible executions) is called *feasible data races* [92]. In an ideal world, one would like to define and detect precisely the feasible data races based on a static program representation (*static detection*). The multitude of patterns and mechanisms for data sharing and thread synchronization make it however difficult to conceive such a data race definition that matches the intuitive concept.

The following results from complexity and computability analysis show that a precise analysis of the synchronization structure in concurrent programs is computationally intractable or even impossible: Taylor [117] shows that various synchronization-sensitive analyses (those that consider only the possible interleavings and execution paths of a parallel program with rendezvous style synchronization) are NP-complete for programs without procedures and recursion. Ramalingam [98] extends this result for programs with (recursive) procedures and shows that any context-sensitive, synchronization-sensitive static analysis is undecidable.

Hence, practical static data race checkers provide an approximation and have the potential of covering all *feasible data races* however at the risk of reporting incidents that do not correspond to feasible program executions (*overreporting*, *spurious data races*); the reports of such a static tool are called *apparent races* [92]. Figure 2.1 illustrates the conceptual relationship between data races and the findings of a typical static detection mechanism; the extent of the boxes does not reflect the multitude of incidents or reports.

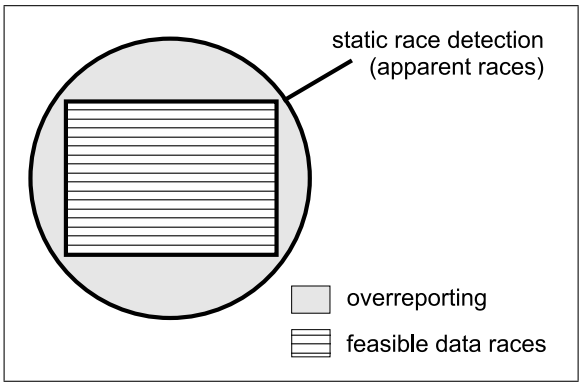

Figure 2.1: Detection capabilities of static data race detection.

According to [29, 44], an ideal static checker should have the following properties:

- *Soundness.* A sound checker reports an error if there is some error in the program (no *underreporting*). Not all errors need to be reported according to this definition, e.g., a race detection tool is still sound if it omits reports of races that are a consequence of another race.
- *Completeness.* A checker that is complete reports only genuine errors, not spurious incidents (no *overreporting*).

Dynamic detection. Due to the difficulty of defining and analyzing data races based on a program representation, common data race definitions and detection mechanism have focused on program executions. An *actual data race* [92] is the occurrence of a data race in a program execution. *Dynamic detection* determines actual data races. Current techniques of dynamic data race detection are afflicted with two sources of inaccuracy: *Underreporting* means that actual races are omitted in the reporting (*omitted data race*); *overreporting* means that there are reports that do not correspond to real data races (*spurious data race*); Figure 2.2 illustrates this discrepancy between reported and actual respectively feasible data races.

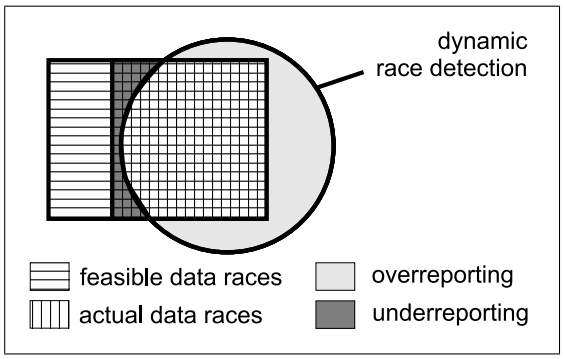

Figure 2.2: Detection capabilities of dynamic data race detection.

We discuss three important variants of data race definitions (Sections 2.2.1 to 2.2.3) that are relevant in this dissertation. The definitions differ in their approximation of ordering among execution events and in the granularity of data that is subsumed by an access event. For each data race definition, static and dynamic detection mechanisms are presented and assessed according to their *accuracy* and *runtime efficiency*.

2.2.1 Happened-before data races

Definition. Lamport [73] defines an irreflexive partial ordering among events in a program execution called *happened-before*, written $\rightarrow$. Two events e_i, e_j of a particular execution are ordered, i.e., $e_i \rightarrow e_j$, if (1) they belong to the same thread and are ordered in the control flow of this thread, or (2) e_i and e_j belong to different threads and some inter-thread synchronization (e.g., locking, thread start or join) forced e_i to occur before e_j.

According to Netzer and Miller [92], two events e_i and e_j participate in a data race, if (1) the events belong to different threads, (2) the events access the same variable, and (3) at least one

Program 2.1: Execution interleaving with a happened-before data race.

```
1     // Thread 1                  // Thread 2
2
3     t2 = new Thread2();
4     t2.start();
5                                 // start
6                                 ... = x;
7     x = ...;
8     t2.join();
9     // waiting
10                                y = ...;
11                                // terminate
12    // Thread 2 joined
13    ... = y;
```

event corresponds to a write and (4) the events are not ordered according to the happened-before relation. A *happened-before data race* is thus defined as

$$\begin{aligned} happenedBeforeRace(e_i, e_j) \quad \Leftrightarrow \quad & (e_i.t \neq e_j.t) \wedge \\ & (e_i.o = e_j.o) \wedge (e_i.f = e_j.f) \wedge \\ & (e_i.a = \text{WRITE} \vee e_j.a = \text{WRITE}) \wedge \\ & (e_i \not\rightarrow e_j) \wedge (e_j \not\rightarrow e_i). \end{aligned} \tag{2.1}$$

Example. There are two shared variables in the execution scenario in Program 2.1, `x` and `y`. The order of access to `x` is not determined by the program (i.e., not ordered by the happened-before relation), there is a write (Thread 2), and hence there is a data race. The situation is different for variable `y` which is written by Thread 1 and read by Thread 2 only after Thread 1 terminated.

Detection. There are basically two implementation alternatives for dynamic data race detection: Trace-based techniques record relevant events at runtime and determine data races during an offline analysis of the execution trace [91]. Online techniques limit the amount of trace data and verify condition 2.1 at runtime [36, 82]. Due to the limited context for inferring the happened-before relation (only a limited window of access events can be tracked by an online checker), online techniques are usually less precise than trace-based systems; moreover, the temporal ordering of access events is approximated through the occurrence of synchronization events at runtime, which introduces scheduling dependence into the detector and might lead to omitted reports [36]. An important approach to improve the accuracy of online checkers for programs with lock-based synchronization has been proposed in [36] and has led to the lock-based definition of data races (see Section 2.2.2).

Besides accuracy, the execution overhead of online checkers, which is typically a factor of 5 to 100 of the original execution, is a major concern. Mellor-Crummey [83] uses compile-time information to avoid unnecessary checking of accesses to thread-local data in Fortran programs. Christiaens and de Bosschere [27] have investigated object-oriented programs and avoid the checking of accesses to objects that are reachable only from a single thread; in their system, object reachability with respect to threads is determined dynamically. Compiler optimizations

and runtime tuning improve the overhead of online happened-before data race checkers to a factor of 2 to 20 [27].

2.2.2 Lock-based data races

The difficulty to infer the happened-before relation from a program execution has led to another definition of data races that is not based on the temporal ordering of events but on a *locking policy* [103]. The rationale behind *unique-lock data races* is that accesses to shared mutable variables that are consistently protected by a unique lock cannot be involved in a data race.

Definition. Let $E(o, f)$ be the set of events that access field f on object o. A *unique-lock data race* on the variable $o.f$ is defined as

$$\begin{aligned} \mathit{uniqueLockRace}(o, f) \quad \Leftrightarrow \quad & \exists e_i, e_j \in E(o, f) : e_i.t \neq e_j.t \wedge \\ & \exists e_i \in E(o, f) : e_i.a = \text{WRITE} \wedge \\ & \cap_{e \in E(o,f)} e.L = \emptyset. \end{aligned} \tag{2.2}$$

This data race definition is suitable for programs that base their synchronization on locks and monitors [63] which is common for object-oriented parallel programs.

Formula 2.2 characterizes a conservative set of actual data races in a program execution: A complete set of access events to the same variable that are not ordered through monitor-style synchronization is identified; there could be overreporting because the events might be ordered through other means of synchronization. This definition is however not practical, because several common and correct synchronization patterns violate the locking policy: initialization without lock, shared read-only data, and read-write locks. Consequently, implementations that check for unique lock data races extend the unique-lock data race definition and delay the checking of the locking policy, until a variable is accessed by a second thread. This extension might lead to underreporting because the thread scheduler could arrange the access events of different threads such that the data race is hidden; the technique is however frequently applied in practice because it has been demonstrated that the risk of underreporting is low and worthwhile in the light of a significant reduction of spurious reports [103].

The implementation of the more practical locking policy associates a *lockset* and *state* with each variable. The state specifies if the variable has been accessed by more than one thread (shared) and whether the second thread performed a write; the lockset records the locks that have been held during all access and is initialized as soon as a variable is accessed by more than one thread. At every access e, the lockset associated with variable is refined through intersection with set of locks $e.L$ held during e. A report is generated if the lockset got empty, if the variable is in the shared state, and a write has been seen since the variable has become shared.

Example. The execution scenario in Program 2.2 accesses two shared variables `x` and `y` in different locking contexts: Access to `y` happens in the scope of a unique lock, hence the access events are ordered. Access to `x` happens outside the scope of a lock, hence `x` is target of a lock-based data race. This error would not be recognized according to definition of a

Program 2.2: Execution with a data race that is reported by a lock-based, but not by a happened-before based checker.

```
1     // Thread 1                  // Thread 2
2
3     x++;
4     synchronized (lock) {
5         y++;
6     }
7                                 synchronized (lock) {
8                                     y++;
9                                 }
10                                x++;
```

happened-before data race because the synchronization intervening the access events of `x` would be wrongly assumed to enforce ordering (in this interleaving). In the scenario of Program 2.1, the lock-based race definition would determine a data race on variable `y` because the access statements are not executed under unique lock protection. The example scenarios indicate that happened-before and lock-based data race detection have different capabilities. A detailed comparison is given by O'Callahan and Choi in [93].

Detection. The original implementation of a lock-based data race detection (Eraser) [103] is based on a binary instrumentation of all memory access instructions that may target global or heap memory; this instrumentation causes slow down by factor of 10 and 30. Subsequent research has developed systems that exploit program analysis to reduce the amount of instrumentation and the runtime overhead. A significant improvement has been achieved by our early work on object race detection [122] that used a simple escape analysis to reduce the program instrumentation and caused a runtime overhead of 16% to 129%. A further improvement in efficiency is achieved by Choi et. al. [25]; their system actually checks for a refined version of unique-lock data races (common-lock data races [36]) and uses several static and dynamic techniques to reduce the overhead of dynamic data race detection for typical Java programs below 50%. One importants part of the system is the static data race detection [26] that we discuss, along with other static analyses for race detection, in Section 4.7.3.

2.2.3 Object races

The data race definitions in previous sections defined the granularity of the detection as the smallest memory-coherent unit or individual variable. *Object (data) races* coarsen the granularity of this view for object-oriented programs and define locations as objects. This is justified because the fields of shared objects are typically accessed in the same locking context. The coarsening of the memory granularity in the view of the checker enables runtime optimizations that reduce the frequency of dynamic checks [122].

Definition. Let $E(o)$ be the set of events that access some field on object o. Object races are defined as a variant of unique-lock data races:

$$\begin{aligned} objectRace(o) \quad \Leftrightarrow \quad & \exists e_i, e_j \in E(o) : e_i.t \neq e_j.t \wedge \\ & \exists e_i \in E(o) : e_i.a = \text{WRITE} \wedge \\ & \cap_{e \in E(o)} e.L = \emptyset. \end{aligned} \tag{2.3}$$

Detection. We describe and evaluate the system for object race detection in detail in Section 7.

2.2.4 Conclusion

The data race definitions in Formulae 2.1, 2.2, and 2.3 identify a conservative set of actual data races in a program execution. The definitions are however not operational, i.e., some of the predicates must be approximated from the observations in an execution trace. This can lead to overreporting.

Practical mechanisms for data race detection use heuristics to assess the ordering among accesses. The heuristics may err (even if this is uncommon) and introduce hence unsoundness: Potential of underreporting is accepted in favor of a significant reduction of spurious reports that would be given if a conservative approximation of access ordering was used.

Two important sources of inaccuracy of dynamic data race checkers are (1) their approximation of the temporal ordering relation (happened-before based checkers) and (2) the delayed checking until some data is accessed by more than one thread (unique-lock and object race checkers). Figure 2.3 illustrates the detection capabilities of different techniques for dynamic data race checking. None of the approaches covers all actual data races, and all approaches have the potential of overreporting. The choice of data race definition and the design of the dynamic checker should be adjusted to the synchronization mechanisms that are used in a program. Generally, happened-before based techniques are preferable for programs with post-wait, rendez-vous, or fork-join style synchronization; lock-based detectors are more appropriate for programs with monitor style synchronization.

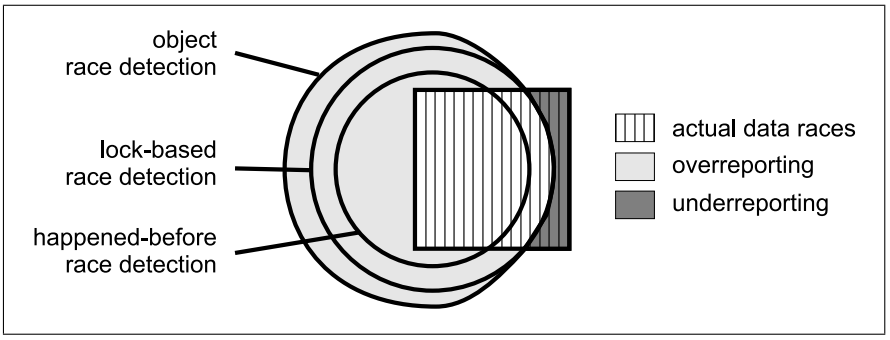

Figure 2.3: Detection capabilities of different approaches to dynamic race detection.

Static data race detection can provide a conservative approximation of feasible data races in a program and is hence useful to narrow the scope of dynamic detection mechanisms.

The use of data races as a *correctness criterion* for programs or program executions [52, 3, 1] is not attractive in practice due to the absence of a precise detection mechanism.

Program 2.3: Class `Account` with non-atomic `update` method.

```
class Account {
    int balance;

    synchronized int read() {
        return balance;
    }

    void update(int a) {
        int tmp = read();
        synchronized(this) {
            balance = tmp + a;
        }
    }
}
```

2.3 Violations of atomicity

Synchronization is not only used to prevent data races at the level of individual variables but also used to ensure non-interference for an access sequence. Access statements that ought to execute atomically form a *critical section* that can be understood as a 'macro-statement' in the view of a thread scheduler. Critical sections as a unit execute atomically but, like statements, without order. Netzer and Miller refer to this phenomenon, which is the principal source of non-determinism in concurrent programs, as *general race* [92]. If critical sections are too fine-grained, undesired interference on shared data can occur – although there is no data race.

Definition. A method is *atomic* if the effect of its execution is independent of the effects of concurrent threads. In particular, shared data that is accessed by a thread t in the dynamic scope of a method m must not be updated by concurrent threads as long as t executes m.

Examples. Consider the example of a bank account in Program 2.3: The shared variable `balance` is accessed under common lock protection and hence there is no data race. The structure of locking specifies that the lock associated with the `Account` instance protects *either* a read *or* a write of field `balance`. Method `update` applies this synchronization discipline, however it performs a read *and* a write and hence cannot be atomic. A lost-update problem can occur if multiple threads access an `Account` instance and execute method `update`, which is itself not a critical section, concurrently.

Program 2.4 shows the example of a data structure that has atomic methods but includes a potential data race on variable `size`. Programs 2.3 and 2.4 demonstrate that the properties of race freedom and atomicity are *incomparable* [126].

Detection. We describe and evaluate a static technique for determining atomicity of methods in Chapter 5.

Program 2.4: Class `Stack`, which has atomic methods but allows a data race on variable `top`.

```
class Stack {
    int top;
    Object[] arr;

    int size() {
        return top;
    }

    synchronized void push(Object o) {
        // assert (top < arr.length);
        arr[top++] = o;
    }

    synchronized Object pop() {
        // assert(top > 0);
        return arr[--top];
    }
}
```

2.4 Deadlock

Definition. A *(resource) deadlock* situation occurs at runtime if threads use synchronization so that a mutual wait condition arises [28]. Some deadlock definitions are more general and consider any situation without progress, e.g., the awaiting of an external communication interaction, as deadlock. Our focus is on resource deadlock.

Detection. Dependencies among processes and resources can be modeled in a bipartite *resource allocation graph* [106]. Nodes correspond to processes and exclusive resources, edges from processes to resources stand for a wait-for relation, and edges between resources and processes represent current-use relations. The basic principle of dynamic deadlock detection is to record and update such a graph at runtime. If the graph becomes cyclic, a deadlock is detected. Variations of this simple algorithm have been developed for distributed systems where a unique global view of the system in a resource allocation graph is difficult to determine and maintain at runtime.

In Java, resource deadlock can occur through mutual dependencies among threads in the usage of locks. Recent Java runtime environments [69] track the monitor ownership of threads in a data structure that resembles the resource allocation graph and determine deadlock through cycle detection.

We describe and evaluate a static technique to detect potential resource deadlock in Chapter 6.

2.5 Trails to correct synchronization

This section discusses different development approaches of concurrent software. First, we categorize the approaches and discuss their strategies to address the problem of synchronization

defects. Second, we argue about the correctness criteria that are applicable to assess "correct synchronization" in each approach. Third, we emphasize and discuss the approach that is pursued in this dissertation.

Canonical approaches. Figure 2.4 illustrates canonical approaches along their flexibility to express synchronization patterns and their susceptibility to synchronization defects.

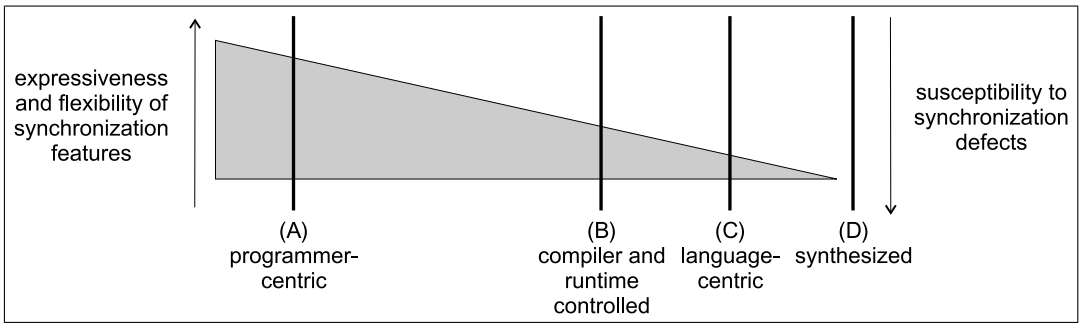

Figure 2.4: Canonical approaches to the development of parallel programs.

(A) In the *programmer-centric* approach, the programmer is exposed to a variety of language features that allow to explicitly control multi-threading and synchronization. The correct application of these features is left to the programmer. While this approach offers the widest flexibility in the design of concurrent systems, it has important shortcomings: Compiler and runtime technology are mostly adopted from sequential environments such that tools that assess concurrency and synchronization defects are not integrated in the development process. The debugging of synchronization defects is mostly done with conventional techniques. The programmer-centric approach is current practice for the development of multi-threaded Java and C♯ applications today.

(B) The *compiler and runtime controlled* approach offers the same flexibility for the programmer in the use of concurrency features as the *programmer-centric approach*. However, the compiler and runtime system reason about the structure of parallelism and synchronization, and determine potential synchronization defects.

For the example of data races, the assessment through the compiler can lead to three situations: (1) The report of the compiler reflects an actual synchronization defect that is corrected by the programmer. (2) The compiler issues a report that is identified as benign by the programmer. The programmer communicates the synchronization intention through an annotation to the compiler, which in turn relaxes its rules for checking in this situation. (3) The compiler report is ignored by the programmer and a dynamic checker determines the occurrence of synchronization defect at runtime.

(C) In the *language-centric* approach, certain classes of synchronization defects (data races, deadlock) are ruled out by the design of the type system. Examples are the programming language Guava [9], and type extensions of Java like [41, 45] or [16]. These languages require that the programmer specifies the synchronization discipline explicitly through type annotations and declaration modifiers. Some systems allow to infer a large number of annotations automatically (e.g., [42]), or provide appropriate defaults to match the common

case [16]. The *language-centric approach* is a promising and attractive direction to address the problem of synchronization defects. However, these systems force the software designer to use the synchronization models that can be type-checked, and some popular and efficient, lock-free synchronization patterns are not accommodated.[1] In addition, it is unclear if the specific locking discipline that is imposed by the type system and requires, e.g., lock protection for access to all potentially shared mutable data structures, is well amenable to program optimization and high performance in concurrent software systems. Hence it will be a while until language-centric approaches to concurrency control become widely accepted.

(D) In the *synthesized* approach, the view of the programmer is confined to sequential programming and parallelism is automatically synthesized through the compiler. Such auto-parallelizing compilers employ dependence analysis, discover computations without data interference that can be executed concurrently, and finally generate efficient parallel programs. Vectorization, e.g., has been successful along this procedure: Vectorizing compilers typically focus on loops and exploit concurrent execution features of synchronous processor architectures (SIMD). The transfer of auto-parallelization to asynchronous multiprocessors (MIMD) has however brought new challenges. Parallel execution is done in separate threads, and current hardware and OS systems cause a critical overhead in managing threads and shared data. The focus on loops is often not sufficient and compilers are compelled to unveil new opportunities and larger scopes for parallel execution. For object-oriented programs, dependence analysis is complicated through the use of dynamic data structures and the effects of aliasing. Hence speculative approaches have been conceived that can be successful to improve the performance of programs where static dependence analysis fails [94].

The *synthesized approach* to parallelism is conceptually appealing because synchronization defects cannot occur by design. The challenges posed through modern processor architectures and applications are however hard and manifold. Hence automated or speculative parallelization are often not as effective as techniques where parallelism is specified explicitly by the programmer. The scope of the dependence analysis and speculation is usually limited to local program constructs, and hence automated parallelization is not successful for programs where independent computational activities can be separated at a high-level in the system design. Kennedy [47, Section 8.1.1] argues that "*... it is now widely believed that (auto-)parallelization, by itself, is not enough to solve the parallel programming problem*".

Correct synchronization. We have presented three important classes of synchronization defects (data races, violations of atomicity, and deadlock). Based on their definitions, we would like to conclude with a notion of correctness for parallel programs that reflects the absence of such synchronization defects.

For the *synthesized* approach (D), the correctness criterion is to preserve the semantics of the initial sequential program. The presence of data races or atomicity violations is an implementation aspect.

[1] Flanagan and Freund [41] use program annotations that allow to escape the type system in such cases.

In the *language-centric* approach (C), correctness criteria are explicitly stated by the programmer and verified by the type checker. Most type systems focus on specific synchronization defects, and hence the scope of the check is limited, e.g., to the absence of data races and deadlock.

For the *programmer-centric* (A) and the *compiler- and runtime controlled* (B) approach, a precise notion of correctness is difficult to achieve: First, there is no explicit definition of the synchronization discipline [116] in a program, and the structure of synchronization is generally difficult to determine from the program text. Current object-oriented programming languages like Java and C♯, e.g., offer low-level features that allow for a large variety of inter-thread synchronization mechanism; such mechanisms and their synchronization effect are not easily recognized by a static analysis. Second, the given definitions do not allow to draw a clear line between benign and harmful incidents: The different detection mechanisms of data races have incomparable reporting capabilities that are neither sound nor complete. These aspects make the three classes of synchronization defects a guideline rather than a correctness criterion.

Compiler and runtime controlled concurrency. This dissertation pursues the compiler and runtime controlled approach to concurrency. The approach is founded on a cooperation of programmer, compiler, and runtime system; the roles and challenges of the individual participants respectively constituents are discussed in the following:

- The *programmer* specifies parallelism and synchronization explicitly, and preferably follows certain synchronization patterns that are recognized by the compiler. In cases where the compiler is unable to infer the intention of the programmer, the programmer can provide annotations (e.g., type modifiers like `final` or `volatile`) to avoid unnecessary conservatism in the downstream tool chain (compiler and runtime).
- The *compiler* analyzes the program and reports potential incidents for different classes of synchronization defects. The challenges are to avoid underreporting, and at the same time to minimize the number of false positives due to conservatism. Reports presented to the user should be aggregated and concisely specify the class and source of the problem.
- The *runtime* system is responsible to check residual cases of potential faults that are not resolved by the programmer. The challenge is to avoid underreporting while making the checker efficient.

3

Static analysis

This chapter describes a general static analysis for multi-threaded object-oriented programs. The analysis provides the platform for various algorithms that detect synchronization defects; these algorithms are described in Chapters 4 to 6.

3.1 Preliminaries

The purpose of the analysis is to determine statements that access potentially shared data. Access is distinguished according to the target object, the accessed field, and the thread and locking context.

Classification. The static analysis can be classified along the dimensions proposed by Ryder [102]:

- *Flow sensitivity.* The analysis is *flow-insensitive*, i.e., it does not account for the execution order of individual statements inside a procedure. This model is simple and well suited for the analysis of multi-threaded programs, where the interleaving of threads and the execution order of statements is not known at compile-time.
- *Context sensitivity.* The analysis is *context-sensitive*, i.e., it distinguishes different calling contexts. The notion of context is *functional*, i.e., the context is determined through the state of the computation at a call site rather than the hierarchy of callers [129]; this means that the analysis of a method is tailored to the data on which the method operates, the thread executing the method, and the locking context.
- *Program representation (i.e., calling structure).* The analysis is done in several phases. First, the calling structure of the program is approximated through a variable-type analysis (VTA) [115]. This structure is used and refined in subsequent phases that determine an abstract model for dynamically allocated data (heap shape graph) and perform a symbolic program execution.
- *Object representation.* Object representations are created for each class in the program and at each site that allocates an object instance. Sites are distinguished according to their calling context.
- *Field sensitivity.* Fields are explicitly represented in the object reference model, hence the analysis is *field-sensitive*.

- *Reference representation.* For each method and context in which a method executes, individual reference variables are mapped to objects or groups of objects they refer to.
- *Directionality.* The reference analysis is unification-based, i.e., an assignment `x = y` is treated symmetrically and makes the reference sets of both variables equal [110].

Example. Program 3.1 shows a monitor-based Java implementation of the Dining Philosopher problem [34]. This program is used to illustrate the static analysis.

Program 3.1: Dining Philosopher program.

```
class Table {
    final boolean forks[];
    final int numPhil;
    int numSnacks;

    Table(int num_phil, int num_snacks) {
        numPhil = num_phil;
        numSnacks = num_snacks;
        forks = new boolean[num_phil];
        for (int i = 0; i < num_phil; ++i)
            forks[i] = true;
    }

    synchronized boolean getForks(int id) {
        int id2 = (id + 1) % numPhil;
        while(! (forks[id] && forks[id2]))
            wait();
        if (numSnacks > 0) {
            numSnacks --;
            forks[id] = forks[id2] = false;
            return true;
        } else {
            notify();
            return false;
        }
    }

    synchronized void putForks(int id) {
        int id2 = (id + 1) % numPhil;
        forks[id] = forks[id2] = true;
        notify();
    }
}


class Philo extends Thread {
    final int id;
    final Table tab;
    int snacksEaten;

    Philo(int i, Table t) {
        id = i;
        tab = t;
        snacksEaten = 0;
    }

    void run() {
        while (tab.getForks(id)) {
```

```
            long l = (int) (Math.random() * 50);
            sleep(l);
            snacksEaten++;
            tab.putForks(id);
        }
    }

    void report() {
        System.out.println("Philo" + id + "ate" + snacksEaten + "snacks.");
    }
}


class Main {
    static void main(String args[]) {
        int NUM_PHILS = 2;
        int NUM_SNACKS = 200;
        Table tab = new Table(NUM_PHILS, NUM_SNACKS);
        Philo[] p = new Philo[NUM_PHILS];
        for (int i=0; i < NUM_PHILS; ++i) { // start
            p[i] = new Philo(i, tab);
            p[i].start();
        }
        try {
            for (int i=0; i < NUM_PHILS; ++i) { // join, report
                p[i].join();
                p[i].report();
            }
        } catch (InterruptedException e) {}
    }
}
```

Class `Table` models the shared resources that are protected through monitor-style synchronization. Class `Philo` implements the activities of individual philosophers. Class `Main` is the entry point of the program; it instantiates the setting, forks and joins individual `Philo` threads, and finally reports the share that each philosopher obtained (field variable `numSnacks`).

Outline. Subsequent sections describe the phases of the analysis in more detail:

1. The classes used in the program, the threads, and their call graphs are determined (Section 3.2).
2. A model of the dynamic data in the program (heap shape graph), is computed (Section 3.3).
3. A symbolic execution of threads on the abstract data domain registers access to potentially shared objects in a context-sensitive manner (Section 3.4).

3.2 Abstract threads

Threads are separate control flows in a parallel program execution. In Java programs, threads correspond to the execution of the `main` method at program start and further dynamically created control flows associated with `Thread` objects. The static analysis uses *abstract threads* to

represent such potentially concurrent control flows. Three different kinds of abstract threads are distinguished:

- The *main* thread represents the initial execution at program start. For Java, the entry method of the main thread has the signature `void main(String[])`. This abstract thread has one instance at runtime, i.e., it is *unique*.

- *Run threads* stand for control flows that start on objects that implement the interface `java.lang.Runnable`, more precisely at the method with signature `void run()`. Such objects may have the type `java.lang.Thread` or are associated with an object of that type. The static analysis can determine run threads based on the allocation sites of thread objects and possibly associated runnable objects. *Run* threads are characterized by the class of the object that implements the `run` method.

  The actual number of runtime instances of a *run* thread is usually difficult or impossible to determine for a compiler. If a run thread has multiple or multiply executed allocation sites, then multiple instances are assumed to execute concurrently, otherwise the run thread is *unique*. An allocation site is *multiply executed* if it is inside a loop or recursion or the defining method is multiply executed. A method is multiply executed, if it is called inside a loop, it has multiple call sites, one of its callers is multiply executed, or it is executed by a non-unique thread.

- *Init threads* correspond to control flows that occur through the implicit invocation of class initializers in Java. The invocation occurs at runtime if a class is first used (details in [53, Section 12.4.1]). As the first use of class can often not be attributed to a particular thread, we create a separate abstract thread for each class that has a static initializer. The entry point of such an init thread is the class initialization method with signature `void <clinit>()`.

The main and run threads are subsumed under the term *user threads*. An abstract thread t_i is a tuple $\langle k, m, C \rangle$:

- i is a unique id.

- k is the kind of thread (one of $\{\text{INIT}, \text{MAIN}, \text{RUN}\}$).

- m is the multiplicity, i.e., a boolean value that specifies if the abstract thread is *unique*; main and init threads are always unique.

- C is the set of defining classes; main and init threads have one defining class; run threads that correspond to runnable objects for which the type cannot be uniquely determined can have several defining classes. The entry method(s) of an abstract thread are determined from the thread kind and the defining class(es).

Example. In the example of Program 3.1, there are the following abstract threads: $t_1 = \langle \text{MAIN}, \text{true}, \{\texttt{Main}\} \rangle$, $t_2 = \langle \text{RUN}, \text{false}, \{\texttt{Philo}\} \rangle$, $t_3 = \langle \text{INIT}, \text{true}, \{\texttt{System}\} \rangle$, and further init threads corresponding to library classes that are omitted for brevity.

3.3 Reference analysis

The heap shape graph (HSG) represents a flow-insensitive model of global data and their reference relations, approximating object connectivity at all program points. Flow-insensitivity makes the HSG suitable for the analysis of multi-threaded programs because the model is independent of the flow and progress of individual threads, which is generally not known at compile-time.

3.3.1 Definition

The HSG is a directed labeled graph. Each node represents a class, an individual instance, or sets of instances. Edges correspond to field reference relations. Representatives are initially created for all classes and augmented during the analysis at instance allocation sites in different contexts.

The runtime values of reference variables are approximated through *alias sets*, i.e., sets of object representatives that are combined along the course of the unification-based analysis (described in Section 3.3.2). After the static analysis, alias sets express may-alias information: if two variable may refer to the same instance at runtime, both variables refer to the same alias set.

We refer to the set of runtime objects that are subsumed under an alias set as *abstract object*. An *abstract object* o_i is a tuple $\langle f, p, t \rangle$:

- i is a unique identifier; the identifier is only used to refer to abstract objects in this presentation; it has no meaning in the algorithms we describe.
- f is a map from fields to abstract objects that are directly reachable through those fields.
- p is a mask that denotes properties of the abstract object. Possible properties are

class (C)	Expresses that this abstract object stands for a class.
global (G)	Means the abstract object is reachable (through chains of field dereferences) from an abstract object that stands for a class or thread root. Abstract objects that are global are candidates for concurrent access.
thread (R)	Means that the abstract object has an associated abstract thread; such objects constitute the initial context for the thread entry method (e.g., the object reachable through the `this`-reference of a `run` method).
shared (S)	The abstract object is potentially accessed by multiple runtime threads.
escape (E)	The abstract object may be accessible outside its allocating scope.
object-specific (O)	The abstract object is created and accessed only in the scope of another object called the *hosting* object.
thread-specific (T)	The abstract object is affiliated with a thread instance and only accessed during the construction of the thread or by the thread itself.
unique (U)	The abstract object corresponds to a single runtime entity (class or instance).

- t is a mask denoting the abstract threads that create or access the abstract object.

The HSG is computed in several steps that are discussed in the following sections. Section 3.3.2 describes the inter- and intra-procedural steps that set up the fundamental structure of the graph. Several associated analyses (Sections 3.3.4 to 3.4.3) extend this information and compute properties of individual abstract objects.

Example. Figure 3.1 shows the HSG for Program 3.1. The nodes are annotated with a unique identifier (i), the properties (p), and a class name. The class name is for illustration purpose and corresponds to the represented class (if the node stands for a class), or the class from which an abstract object is instantiated (if the node stands for an instance; in this simple example, one class can be specified for each instance). Array types are specified by an opening square bracket `[`, followed by the element type (which may be a primitive type). Edges are annotated by the field they model. Nodes 1 to 5 represent classes (C). All nodes in the HSG have the global (G) property by definition (Section 3.3.1).

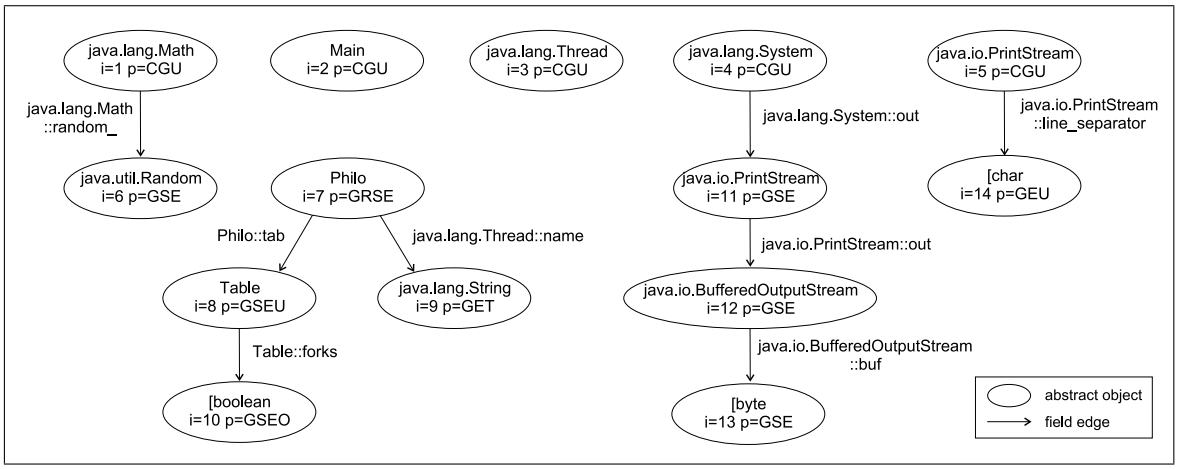

Figure 3.1: HSG of the Dining Philosopher program.

The graph is simplified for the purpose of illustration: Several nodes that are only allocated and accessed by init threads are not displayed; e.g., the `java.io.PrintStream` object reachable through the field `java.lang.System::err` is not accessed by user threads and hence omitted.

3.3.2 Algorithm

The HSG is computed according to Ruf's unification-based analysis [100]. This analysis is *compositional* because every method is analyzed independently and *method summaries* are used to transfer the effects of the method to individual call sites. A method summary is generic with respect to the caller context (parameters, return value, and thrown exception).

Alias sets are represented by a union-find data structure. The *unify*() operation merges two alias sets and combines their field maps. Alias sets that are reachable through common fields are unified recursively. In addition, the properties are merged such that the resulting alias set has a property if at least one of the original alias sets has the property (logical OR; not so for

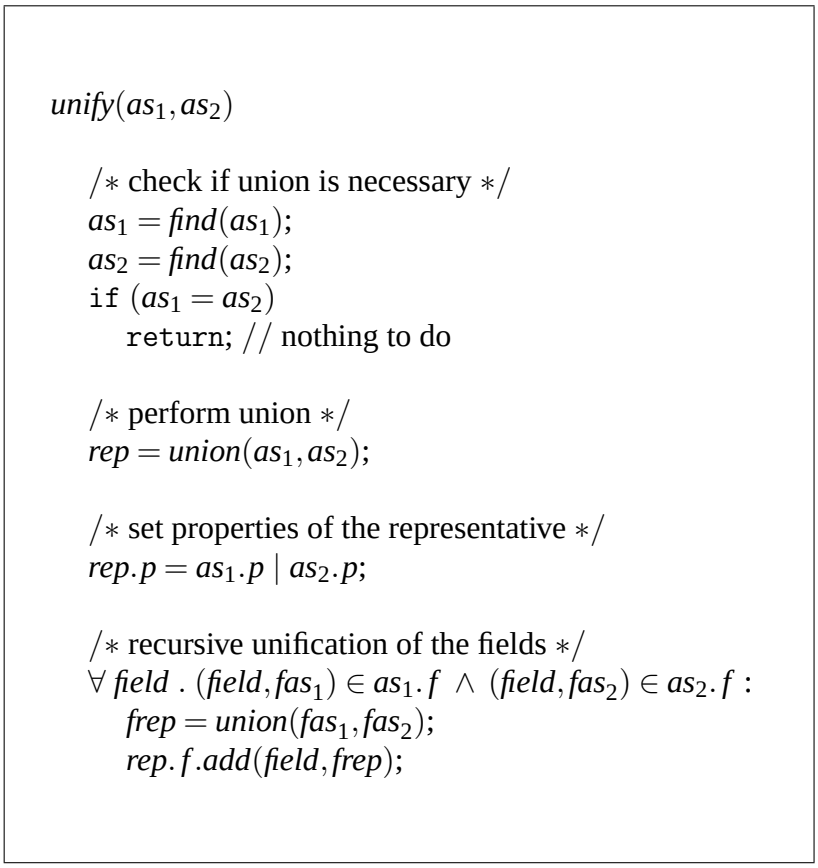

Figure 3.2: Method *unify* for the recursive unification of alias sets as_1 and as_2. We assume that alias sets are based on a union-find data structure that implements the methods $union(u_1, u_2)$ that merges u_1, u_2 and returns the common representative, and $find(u)$ to determine the representative of u.

.

the properties *thread-specific* and *object-specific* but these properties are set after the HSG is complete). Figure 3.2 defines the recursive unification of alias sets. Note that the defintion of *unify* does not describe the treatment of the unique identifier *i*; we use the identifier only to refer to abstract objects in this presentation.

An intra-procedural analysis associates reference variables with alias sets and unifies alias sets in a stepwise process along the statement sequence of a method. The inter-procedural analysis handles the effects of method invocation.

Inter-procedural analysis

First, alias sets for classes and abstract threads are created. These alias sets are the roots of the HSG. Further alias sets are connected to these roots as a result of a sequence of intra-procedural steps. Nodes that are transitively reachable through these root nodes are classified as *global*.

The inter-procedural analysis processes each abstract thread individually and analyses the methods in the call graph of each thread in the reverse order of their invocation (bottom up traversal of the strongly connected components (SCCs)). The analysis computes one summary per method.

Intra-procedural analysis

Given a method, the goal of the intra-procedural analysis is to establish a *method summary* that models the effects of the method execution independent of the calling context. A method summary captures aliasing that is created through method execution, including data that is shared between the caller and the callee (i.e., parameters, return value, and thrown exceptions). In addition, the method summary records object allocations, read and write access. The summary of a method m is defined as a tuple of alias sets $MS[m] = \langle\langle f_0, \ldots, f_n\rangle, \langle l_0, \ldots, l_m\rangle, r, e, A, R, W, E\rangle$:

- $f_0, \ldots, f_n$ are alias sets of formal parameters that hold reference values.
- $l_0, \ldots, l_m$ are alias sets of local variables that hold reference values.
- r is the alias set of the return value if method m returns a reference value.
- e is the alias set of the exception value if method m throws or conveys exceptions [53]; if different exceptions are thrown by a method, they are combined to the same alias set in the summary.
- A is the set of alias sets of objects that are allocated in the scope of m.
- R is the set of alias sets of objects that are read in the scope of m.
- W is the set of alias sets of objects that are written in the scope of m.
- E is the set of alias sets of objects that escape from the scope of method m.

The computation of the method summaries starts with the creation of alias sets for all formal reference parameters $f_0, \ldots, f_n$ and local variables. Formal parameters are not aliased at this point, and caller-side aliasing is taken into account when a method summary is instantiated at a call site. The method summary is gradually built along a flow-insensitive traversal of the method's statements. We assume that the method is given in Singele Static Assignment form (SSA), i.e., ϕ-functions are used to select values at control-flow join points. Figure 3.3 lists the analysis rules; only statements that access objects or handle references need to be considered.

Assignment combines alias sets; field and array access creates field-reference relations (all slots of an array are represented by the same conceptual field '$').

At a call site, polymorphism is resolved through type information computed by the variable-type analysis (VTA, [115]) and each target method is treated separately. There are two cases:

1. *Non-recursive target*.

   If a method summary for the callee has not yet been computed, then the analysis descends eagerly to compute it (see inter-procedural analysis). Function *MC* (Figure 3.3) creates a *method context* from the existing *method summary*. Thereby, the alias sets of the method summary are cloned and associated with the method context; global alias sets are retained. The resulting method context is embedded into the method summary of the caller by unifying the alias sets of formal and actual parameters, return value, and thrown exceptions. The creation of a method context avoids the infiltration of call site specific effects into the method summary and hence enables context sensitivity of the analysis.

Variables and domains		**Analysis state**	
$m \in \mathcal{M}$	methods	$CAS : \mathcal{C} \to O$	alias set lookup for classes
$c \in \mathcal{C}$	classes	$AS : \mathcal{V} \to O$	alias set lookup for local vars
$f \in \mathcal{F}$	fields	$MC : \mathcal{M} \to \mathcal{B}$	method context creation
$v \in \mathcal{V}$	local variables	$HA :\to 2^O$	alias sets for exception variables
$o, e, r, h \in O$	alias set		in matching handlers of the
$b \in \mathcal{B}$	method context		currently analyzed method

Statement	**Action**
$v_0 = v_1$	$unify(AS(v_0), AS(v_1))$;
$v = \phi(v_0, \ldots, v_n)$	$\forall i \in \{0, \ldots, n\} : unify(AS(v), AS(v_i))$;
$v = c.f$	$unify(AS(v), CAS(c).f[f])$;
$c.f = v$	$unify(AS(v), CAS(c).f[f])$;
$v_0 = v_1.f$	$unify(AS(v_0), AS(v_1).f[f])$;
$v_1.f = v_0$	$unify(AS(v_0), AS(v_1).f[f])$;
$v = m(v_0, \ldots, v_n)$	`let` $MC(m) = \langle\langle f'_0, \ldots, f'_n\rangle, \langle l'_0, \ldots, l'_m\rangle, r', e', A', R', W', E'\rangle$:
	$\forall i \in \{0, \ldots, n\} : unify(AS(v_i), AS(f'_i))$;
	$unify(AS(v), AS(r'))$;
	$\forall h_i \in HA() : unify(h_i, AS(e'))$;
$v =$ `new` c	create new entry for allocation in $AS(v)$;
`return` v	$unify(r, AS(v))$;
`throw` v	$\forall h_i \in HA() : unify(h_i, AS(v))$;
	$unify(e, AS(v))$;
`catch` v	$unify(e, AS(v))$;

Figure 3.3: Transfer rules for creating method summaries.

2. *Recursive target.* In the recursive case, the method summary of the target method is about to be computed. In this case, function *MC* does not clone alias sets but returns the (yet) incomplete method summary itself which is consequently embedded into the callee (see non-recursive case). This procedure leads to a loss of context sensitivity inside a recursion. Alternatively the method summary could be copied like in the non-recursive case, and the inter-procedural analysis would iterate over all members of an SCC in the call graph until a fixed point is reached. A more detailed discussion is given in [100]. The latter procedure can be more precise, but also increases the cost of the analysis. We choose the first implementation alternative because it simplifies the handling of recursion also in downstream analyses (Section 3.4).

The flow of references to exception objects is considered at `throw` and `catch` statements that unify the alias set of the handled reference $AS(v)$ with the exception e of the current method. At method invocation sites, possible exceptions e' thrown by the callee are matched with exceptions e of the caller.

Program 3.2: Detailed `main` method of the Dining Philosopher program.

```
1  static void main(String args[]) {          // f0
2      int NUM_PHILS = 2;
3      int NUM_SNACKS = 200;
4      Table tab;                              // l0
5      Philo[] p;                              // l1
6      int i;
7      Philo tmp;                              // l2
8
9      tab = new Table(NUM_PHILS, NUM_SNACKS);
10     p =  new Philo[NUM_PHILS];
11
12     for (i=0; i < NUM_PHILS; ++i) {
13         tmp =  new Philo(i, tab);
14         p[i] = tmp;
15         tmp.start();
16     }
17     try {
18         for (i=0; i < NUM_PHILS; ++i) {
19             tmp = p[i];
20             tmp.join();
21             tmp.report();
22         }
23     } catch (InterruptedException e) {}     // l3
24 }
```

Example. Program 3.2 illustrates the formal parameters and local variables that hold reference values in method `Main::main`. Figure 3.4 shows the corresponding method shape graph with abstract objects that are reachable through formal parameters and local variables. The method summary is of `main` is $\langle\langle f_0\rangle, \langle l_0, l_1, l_2, l_3\rangle, r, e, A, R, W, E\rangle$ with formal parameter $\langle 100\rangle$, local variables $\langle 8, 101, 7, 102\rangle$, no return value, and no checked exception. The ids of abstract objects that are global (id < 100) correspond to those in the global HSG in Figure 3.1; the ids of abstract objects that are local to the execution scope of the method or vary with the caller context have id ≥ 100. The computation of the sets A, R, W, and E is discussed in the corresponding example in Sections 3.3.3 and 3.3.5.

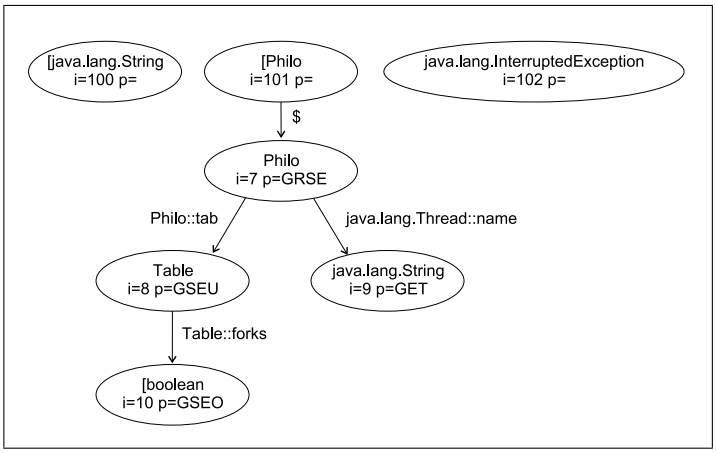

Figure 3.4: Method summary for `Main:main` in the Dining Philosopher program.

3.3.3 Read, write, and allocation analysis

The purpose of the read, write, and allocation analysis is to record abstract objects that are read, written, or allocated in the dynamic scope of a method. This information is collected along the intra-procedural analysis and stored in the method summary. Figure 3.5 specifies the rules that updates read, write and allocation sets along the analysis. The integration of read, write and allocation information into the method summaries allows to approximate the effects of a method at a call site during the symbolic execution (Section 3.4).

Variables and domains		**Analysis state**	
$m \in \mathcal{M}$	methods	$CAS : \mathcal{C} \to O$	alias set lookup for classes
$c \in \mathcal{C}$	classes	$AS : \mathcal{V} \to O$	alias set lookup for local vars
$f \in \mathcal{F}$	fields	$MC : \mathcal{M} \to \mathcal{B}$	method context creation
$v \in \mathcal{V}$	local variables		
$o \in O$	alias sets		
$b \in \mathcal{B}$	method context		

Statement	**Action**
$v = c.f$	$R = R \cup \{CAS(c)\};$
$c.f = v$	$W = W \cup \{CAS(c)\};$
$v_0 = v_1.f$	$R = R \cup \{AS(v_1)\};$
$v_1.f = v_0$	$W = W \cup \{AS(v_1)\};$
$v = m(v_0, \ldots, v_n)$	`let` $MC(m) = \langle\langle f_0, \ldots, f_n\rangle, \langle l_0, \ldots, l_m\rangle, r, e, A', R', W', E'\rangle$: $R = R \cup R';$ $W = W \cup W';$ $A = A \cup A';$
$v =$ `new` c	$A = A \cup \{AS(v)\};$

Figure 3.5: Transfer rules for the read, write and allocation analysis. R, W, and A refer to the read, write, and allocation sets of the method being analyzed.

Example. For method `main` (Program 3.2 and Figure 3.4), the following sets are computed: allocation set $A = \{o_7, o_8, o_9, o_{10}, o_{101}, o_{102}\}$, read set $R = \{o_7, o_{101}\}$, and write set $W = \{o_7, o_8, o_{10}, o_{101}, o_{102}\}$. The sets contain abstract objects that are allocated/read/written in `main` itself or its callees. E.g., an `java.lang.InterruptedException` (o_{102}) may be allocated and thrown by method `java.lang.Thread::join`, called in line 20 of Program 3.2. The `Philo` instance (o_7), more precisely the field `priority` it inherits from class `java.lang.Thread`, is read in method `java.lang.Thread::start`, which is called in line 15; hence $o_7 \in R$. Both methods, `start` and `join`, are not shown in Program 3.2 but are are part of the standard library. the read, write, allocate behavior of native methods is modeled explicitly. Read and writes of final variables do not account.

3.3.4 Shared analysis

The inter-procedural analysis processes each abstract thread separately and establishes method contexts along the call graph of each thread. Whenever a method context is created, the abstract thread is noted in the mask of all abstract objects that are read, written and allocated (Section 3.3.3). This information is accumulated along the unification of alias sets such that finally all global abstract objects have the complete set of accessing threads denoted in their abstract thread mask t.

The complete HSG contains only nodes that are *global*, reflecting a conservative set of objects that are subject to shared access. The thread access information associated with abstract objects can be used to narrow the set of potentially shared objects to those that indicate access from a non-unique thread or multiple user threads; such abstract objects are marked with property *shared*. The subtle consequences of omitting init threads from the consideration are discussed in Section 3.6.1. Subsequent synchronization analyses can focus on shared abstract objects.

Example. Some abstract objects in the HSG in Figure 3.1 are marked as shared: e.g., the `Philo` object (o_7), the `Table` object (o_8), and its associated array (o_{10}) are shared due to access from the non-unique `Philo` and the `main` thread. The unique `main` thread is the only accessor of the `System` class (o_4) and its associated `java.io.PrintStream` object (o_{11}), and hence these abstract objects are not shared.

3.3.5 Escape analysis

A reference is said to *escape* from a method m if the method stores the reference to a variable such that the reference is available beyond the execution of m. We use a simple escape analysis that determines if local reference variables contain values that escape.

Escape information is determined separately for each method after the computation of the method summary. At this moment, escape information is available for all callees. The analysis is flow-insensitive and repeatedly iterates over the statements of a method and applies the transfer rules in Figure 3.6 until a fixed point is reached. There is a boolean flag *escape* for each local reference variable that records if the the value is made available in a variable that outlives the incarnation of the method. Initially, all *escape* flags are set to `false`. Finally, the escape information of local variables is transferred and denoted as a property of the corresponding alias sets, i.e., $E = \{AS(v) : v.escape = \texttt{true}\}$. Now, the set E of escaping abstract objects is initialized in the method summary.

Example. Several abstract objects in the method summary (Figure 3.4) of `main` are escaping: The `Philo` object escapes because it implements the `java.lang.Runnable` interface and is associated with a user thread; a reference to this object is naturally available in the context of the thread it implements, which is different from its allocating thread. Subsequently, all objects that are transitively reachable through the `Philo` instance escape; the escape set of `main`'s method summary is $E = \{o_7, o_8, o_9, o_{10}\}$. The `java.lang.String` array passed as parameter (o_{100}) and the array holding references to the thread objects (o_{101}) do not escape.

Variables and domains		**Analysis state**	
$m \in \mathcal{M}$	methods	$ESC : O \rightarrow \text{bool}$	escape flag for abstract objects
$v \in \mathcal{V}$	local variables	$ARG : \mathcal{V} \rightarrow \text{bool}$	true, if v is formal parameter
$o \in O$	alias sets	$MS : \mathcal{M} \rightarrow \mathcal{S}$	method summary lookup
$s \in \mathcal{S}$	method summary		

Statement	**Action**
$v_0 = v_1$	$v_0.escape \equiv v_1.escape$;
$v = \phi(v_0, \ldots, v_n)$	$\forall i \in \{0, \ldots, n\} : v.escape \equiv v_i.escape$
$v = c.f$ $c.f = v$	$v.escape = \texttt{true}$;
$v_0 = v_1.f$ $v_1.f = v_0$	$v_0.escape = v_0.escape \vee v_1.escape \vee ARG(v_1)$;
$v = m(v_0, \ldots, v_n)$	`let` $MS(m) = \langle\langle f_0, \ldots, f_n\rangle, \langle l_0, \ldots, l_m\rangle, r, e, A', R', W', E'\rangle$: $\forall i \in \{0, \ldots, n\} : v_i.escape = v_i.escape \vee ESC(f_i)$ $v.escape = \texttt{true}$;
$v =$ `new` c	`if` (c is a `java.lang.Runnable`) $v.escape = \texttt{true}$;
`return` v	$v.escape = \texttt{true}$;
`throw` v	$v.escape = \texttt{true}$;

Figure 3.6: Transfer rules for the escape analysis. The equivalence-operator $\equiv$ is bidirectional and transfers the escape property of any of its arguments to both arguments.

3.3.6 Object-specific analysis

A frequent pattern in object-oriented programs is to implement one object through other subordinate objects that are created, managed, and accessed only in the scope of the encapsulating object. We say that the subordinate objects are *object-specific* to the encapsulating object. This notion of confinement allows the compiler to determine lock-protection in the static data race-detection (Section 4.4).

The goal of the object-specific analysis is to identify abstract objects that are *object-specific* and to relate them to their encapsulating abstract object. The analysis proceeds as follows: First, the usage of references inside procedures is inspected and those reference variables respective alias sets that are used such that their value is not confined to an encapsulating instance are marked as *object-escaping* (*oescape*). This information is propagated along the creation of the HSG. Finally, all abstract objects that are not object-escaping do never occur in contexts that violate the assumptions of confinement and hence are object-specific.

The intuitive idea behind the conservative approximation of the object-specific property is as follows: Instances with the property that references only appear in local variables or are assigned/read from fields that are accessed through the `this`-reference are object-specific. The transfer rules in Figure 3.7 reflect this idea; two additional benign situations are accommodated:

References that are passed into methods that do not escape the value can be object-specific. References read and written to object-specific objects can be object-specific.

Similar to the escape analysis, the object-escape property of alias sets is computed immediately after a method summary is created along a flow-insensitive traversal of the statements. The transfer rules in Figure 3.7 are applied until a fixed point is reached. It is assumed that escape information has been already computed at this moment. Initially, all non-`this` formal parameters of the analyzed method are marked as *oescape*.

Variables and domains		**Analysis state**	
$m \in \mathcal{M}$	methods	$ESC : O \rightarrow \text{bool}$	escape flag for abstract objects
$v \in \mathcal{V}$	local variables	$AS : \mathcal{V} \rightarrow O$	alias set lookup for local vars
$o \in O$	alias sets	$MS : \mathcal{M} \rightarrow \mathcal{S}$	method summary lookup
$s \in \mathcal{S}$	method summary		

Statement	**Action**
$v_0 = v_1$	$v_0.oescape \equiv v_1.oescape$;
$v = \phi(v_0, \ldots, v_n)$	$\forall i \in \{0, \ldots, n\} : v.oescape \equiv v_i.oescape$;
$v = c.f$	$v.oescape =$ `true`;
$c.f = v$	
$v_0 = v_1.f$	`if` $(v_1 \neq$ `this`$)$
	$\quad v_0.oescape = v_0.oescape \vee v_1.oescape$;
$v_1.f = v_0$	`if` $(v_1 \neq$ `this` $\wedge\ AS(v_0) \neq AS(v_1))$
	$\quad v_0.oescape =$ `true`;
$v = m(v_0, \ldots, v_n)$	`let` $MS(m) = \langle\langle f_0, \ldots, f_n\rangle, \langle l_0, \ldots, l_m\rangle, r, e, A', R', W', E'\rangle$:
	$\quad \forall i \in \{0, \ldots, n\} : v_i.oescape = v_i.oescape \vee ESC(f_i)$;
	$\quad v.oescape =$ `true`;
`return` v	$v.oescape =$ `true`;
`throw` v	$v.oescape =$ `true`;

Figure 3.7: Transfer rules for the object-specific analysis. The equivalence-operator $\equiv$ is bidirectional and transfers the *oescape* property of any of its arguments to both arguments.

After a fixed point is reached, the object-escape information is transferred from the local variables and denoted as a property of the corresponding alias sets, which is conveyed by the unification algorithm. Finally, after all abstract threads are processed and the structure of the HSG is complete, the analysis marks those global abstract objects as *object-specific* that are not object-escaping. Following the incoming field edges backward in the HSG unveils the hosting abstract object (there must be exactly one).

Example. The HSG in Figure 3.1 specifies that the boolean array (o_{10}) is object-specific to the `Table` instance (o_8). The methods of class `Table` handle the reference to this array object such it is not leaked from the scope of the hosting object.

3.4 Symbolic execution

The symbolic execution simulates the execution of a parallel program on the abstract domains of threads (Section 3.2) and global data (Section 3.3). At this point, the sole purpose of the simulation is to enumerate a conservative set of (abstract) access events to shared data. We do not discuss how the computed information is used for the detection of synchronization conflicts in this section but focus on the discussion of the general algorithm that computes these abstract events. Chapters 4 to 6 adopt this algorithm and elaborate on different uses of the generated information for the purpose of detecting different kinds of synchronization defects.

An access event is a tuple $\langle s, o, f, t, L \rangle$:

- s is the statement performing the access.
- o is the accessed abstract object.
- f is the accessed field.
- t is the accessing abstract thread.
- l is the set of abstract objects that are locked at the time of the access.

In a first approximation, the analysis is flow-insensitive and hence does not relate access events according to their control flow and synchronization dependences. This simplification will be offset by the data structure of OUGs described in Section 4.1.

3.4.1 Algorithm

The symbolic execution processes each abstract thread separately. The analysis starts at the entry methods of a thread, creates an initial method context, and applies the transfer rules in Figure 3.8 along the statement sequence.

At a call site, the method summary of the callee is determined, cloned, and embedded into the calling context: operation *embed* unifies actual parameters, return value and exception with the context of the callee (Section 3.3.2, Figure 3.3). The operation *process* guides the analysis straight into the callee (later, in Section 3.4.2, we show two optimizations that refine this behavior and improve the efficiency of the symbolic execution). At polymorphic call sites, each alternative is processed separately. The rule for method invocation in Figure 3.8 only specifies the analysis of instance methods; class methods are handled correspondingly. At this point of the analysis, context-sensitive type information associated with alias sets can be exploited to bound polymorphism. It is not necessary to follow recursive method invocations (i.e., call sites that are already being analyzed), because the contexts inside the recursion have been encountered earlier (see loss of context sensitivity in Section 3.3.2) and hence the access events that would be created by descending into the recursion would not add any new information.

The rules for field access specify the invocation of operation *registerEvent*() that basically records an access event. As mentioned earlier, this section does not elaborate on the use of the recorded information.

The traversal of methods with block monitors is handled specially and follows the basic block structure such that the locking context is determined correctly. At the beginning of a protected code block, the rule for `monitorenter` is triggered, at the end the rule for `monitorexit`.[1] The structure of locking in Java allows the symbolic execution to track abstract objects that are locked along the execution path in a stack. The operations *acquireLock*() and *releaseLock*() maintain this stack; the operation $set(L)$ returns the set of abstract objects on the lock stack L.

Variables and domains		**Analysis state**	
$m \in \mathcal{M}$	methods	$CAS : \mathcal{C} \to O$	alias set lookup for classes
$v \in \mathcal{V}$	local variables	$AS : \mathcal{V} \to O$	alias set lookup for local vars
$o \in O$	alias sets	$MC : \mathcal{M} \to \mathcal{B}$	method context creation
$b \in \mathcal{B}$	method context		
$c \in \mathcal{C}$	classes		
$f \in \mathcal{F}$	fields		
$t \in \mathcal{T}$	abstract threads		
$s \in \mathcal{S}$	statements		

Statement	**Action**
$v = c.f$ $c.f = v$	`if` $(CAS(c).p$ is *shared*$)$ *registerEvent*$(\langle s, CAS(c), f, t, set(L)\rangle)$;
$v_0 = v_1.f$ $v_1.f = v_0$	`if` $(AS(v_0).p$ is *shared*$)$ *registerEvent*$(\langle s, AS(c), f, t, set(L)\rangle)$;

$v = m(v_0, \ldots, v_n)$	`if` $(m$ is `synchronized`$)$
	  acquireLock$(AS(v_0))$;
	$b = embed(MC(m))$;
	process(b);
	`if` $(m$ is `synchronized`$)$
	  releaseLock$(AS(v_0))$;

`monitorenter` v	*acquireLock*$(AS(v))$;
`monitorexit` v	*releaseLock*$(AS(v))$;

Figure 3.8: Transfer rules for the symbolic execution. t is the current abstract thread, s is the current statement, L is the stack of currently locked objects.

So far, we have discussed the analysis from the perspective of the control flow, i.e., as a traversal of the call structure and individual statements. Note however that the symbolic execution does not only track the control flow but also the data environment provided by the method contexts. Hence in an object-oriented program, the symbolic execution maps nicely onto a traversal of the HSG determined in the previous analysis phase (Section 3.3.2): The current position of the traversal reflects the currently accessed abstract object.

[1] At the bytecode level, the actual situation is different: due to exception handling, there can be several `monitorexits` per `monitorenter`.

Example. Figure 3.9 illustrates the first steps of the symbolic execution of the `Philo` thread.

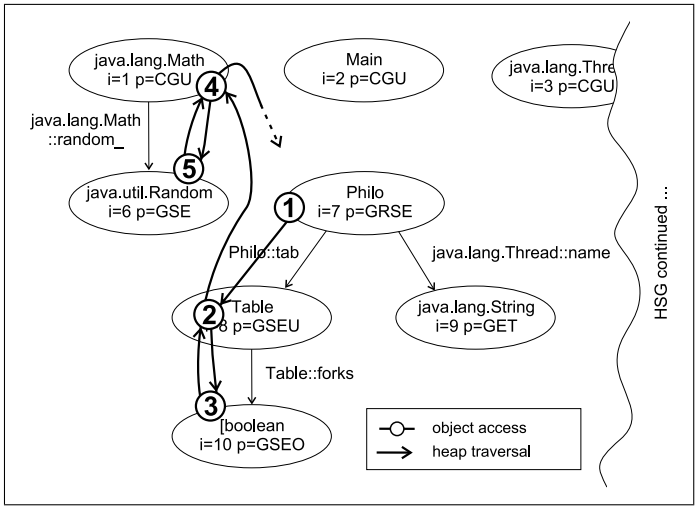

Figure 3.9: Fragment of the HSG with prefix of heap traversal.

1. The symbolic execution starts at the `Philo::run` method. The initial method context is given by the thread root object (o_7), the set of locks is initially empty. Two read events of the final fields `id` and `tab` are registered.

2. The execution branches into method `Table::getForks`. The target object (o_8) is added to the set of current locks because the method is synchronized. A series of access events to the fields `forks`, `numPhil` and `numSnacks` are registered. The processing of methods `java.lang.Object::wait` and `java.lang.Object::notify` does not create any relevant access event and hence they are not considered in this description.

3. Read and write access to the boolean array that is associated with the `Table` instance.

4. After the return of method `Table::getForks` (and the release of lock o_8), branch into method `java.lang.Math::random`. The method is synchronized, hence the target class (o_1) is added to the set of current locks.

5. Creation and initialization of an instance of type `java.util.Random` (o_6). The call to method `java.util.Random::nextDouble` and subsequent access to private synchronized methods on that object lead to a series of field access events.

step	1	2	3	4	5
set of current locks	$\emptyset$	$\{o_8\}$	$\{o_8\}$	$\{o_1\}$	$\{o_1, o_6\}$

Table 3.1: Locks held at the initial steps of the symbolic execution.

Table 3.1 shows the set of current locks held at each step of the heap traversal in Figure 3.8.

3.4.2 Optimizations

The symbolic execution can be the most expensive phase of the analysis because it considers all possible flows through the call structure of a program. Conceptually, all non-recursive call sites are followed in a straightforward manner. This naïve procedure is impractical for larger benchmark programs. Hence, we have developed two optimizations that reduce the number of descents into calls.

Call clipping

The first optimization avoids descents into methods that do not affect the state of shared data. At a call site of method m, the analysis determines from the method context $MC(m) = \langle\langle f_0, \ldots, f_n\rangle, \langle l_0, \ldots, l_m\rangle, r, e, A, R, W, E\rangle$ if shared data is allocated, read or written, i.e., there are abstract objects that have the property *shared* in one of the sets A, R, and W.

Call caching

The second optimization avoids repeated descents into calls with equivalent method, thread, and locking contexts. These three aspects of context in the symbolic execution are encoded in a *site context* $\langle m, \langle a_0, \ldots, a_n\rangle, t, L\rangle$:

- m is the callee.
- $a_0, \ldots, a_n$ are the abstract objects that are passed as actual parameters.
- t is the current abstract thread.
- L is the stack of currently locked abstract objects.

The symbolic execution memorizes all processed call sites it encounters in terms of site contexts. At a call site, actual parameters, abstract thread and set of current locks are matched with site contexts of earlier invocations of m. In the matching, global alias sets are identified with their unique identifier, other alias sets can be determined as *fully local* if only local alias sets are reachable through their fields. A match means that the method m, the abstract thread, all locked abstract objects and all alias sets that are not fully local are equal; in that case, the symbolic execution does not descend into the call (operation *process* in Figure 3.8). This mechanism resembles call caching in functional languages [59], although our mechanism for matching is simpler.

3.4.3 Thread-specific analysis

Abstract objects are *thread-specific* if they are accessed only during the creation of the thread, and consequently by the thread itself. This is an extended notion of objects that are object-specific to thread objects. Thread-specific objects are common in Java and provide – besides the determination of *global* and *shared* objects – further opportunities to narrow the scope of search for synchronization defects.

We present two algorithms that determine thread-specificness. Both algorithms are complementary and may conservatively classify different sets of abstract objects as thread-specific. The thread-specific analysis executes both algorithms and combines their results.

Access-based algorithm

Choi et. al. [25] propose an approximation algorithm for determining thread-specific objects; the algorithm determines thread-specificness based on the access statements and their execution contexts. Two concepts are necessary to determine if an object is accessed only during the construction of a thread and consequently by the thread itself:

- *Safe thread*. A user thread is safe if its reference does not escape from the constructor and the thread is started only after the constructor completed.
- *Safe method*. (1) The constructor methods of a thread object and the `run` method (if only invoked implicitly by `java.lang.Thread::start`) are safe. (2) All methods that are only called by safe methods through the `this`-reference are safe.
- *Safe fields*. A field is safe, if it belongs to a thread object and is accessed only inside safe methods and through the `this`-reference.

The access-based thread-specific analysis is done in three steps: First, safe methods and threads are identified. Second, candidate objects are identified that are only reachable (in the HSG) through safe fields of a safe thread object. The third step checks if the allocation and member access through non-`this`-references (determined by the symbolic execution) to candidate objects occur inside safe methods. At this point, thread-specific objects that are immediately reachable through the field of a thread are determined.

A simple extension is to determine objects that are object-specific to thread-specific objects also as thread-specific. This step is based on the results of the *object-specific analysis* (Section 3.3.6).

Example. There is one safe thread in the example Program 3.1: the `Philo` thread. Candidate for safe fields are `Philo::tab` and `java.lang.Thread::name`. The `java.lang.String` object that is reachable through the latter field is indeed thread-specific because it is – in the example program – only accessed by safe methods, i.e., in `java.lang.Thread::<init>`. The situation is different for the `Table` object (o_8); this object is created and initialized in the `Main::main` method which is not safe, and hence the abstract object is not thread-specific either.

Confinement-based algorithm

An alternative algorithm to assess thread-specificness is to determine if references are confined to a specific thread instance. Two concepts are necessary to determine confinement with respect to a thread instance:

- *Confining thread*. An abstract thread t_i is confining if its root object is only accessed by its creator thread and by the thread instance itself.

- *Thread-confined object.* Abstract objects that are created and accessed only by a confining thread and that are reachable in the HSG only from a confining thread root or other thread-confined objects are themselves thread-confined.

We assume in the following that the creating and the created thread can be distinguished at the abstract level, i.e., they are represented by different abstract threads. This property can be checked during the computation of abstract threads (Section 3.2); if the property is not met for some abstract thread t, then t cannot be confining.

Confining threads are determined according to the reachability of their thread root object in the HSG: First, the root object of thread t_i must not be reachable from abstract objects with creator/access thread-mask different from i. This guarantees that the reference to the thread-root is not *leaked* (i.e., written to a shared variable) by the creating thread t_j, $j \neq i$. This property is checked by inspection of the HSG. Secondly, it must be ensured that a thread instance does not leak a reference to itself: An abstract thread t_i must not write its `this`-reference to the field of a 'traitor object' which is accessible from the creating thread t_j and made available to different runtime instances of t_i. This property is checked during the symbolic execution.

The analysis provides two results if confinement with respect to a thread t_i can be determined: (1) The thread root object of t_i is accessed only by the allocating thread and a single thread instance of t_i. (2) Objects that are confined to t_i cannot have access conflicts even if t_i is not unique.

3.4.4 Uniqueness analysis

An abstract object is *unique* if the static analysis can determine that it corresponds to a single runtime instance. Uniqueness is an important characteristic, e.g., for run threads and abstract objects that are used as locks (Section 4.4).

Section 3.2 describes how the notion of 'one-time execution' can be approximated for individual statements and in particular for allocation sites of run threads (i.e., unique run threads). This procedure is conservative and does not distinguish between different contexts in which an allocation site may execute.

The symbolic execution allows to do a more precise, context-sensitive computation of uniqueness for abstract objects. At every statement, the symbolic execution has the following information available: the current abstract thread, the caller hierarchy above the current method, the current statement and whether it executes in a loop. The transfer rules in Figure 3.10 use this information to count the number of allocations *nalloc* of global abstract objects.

At allocation sites, *nalloc* is incremented in the alias set that corresponds to the allocated objects. The increment accounts for the uniqueness of the allocating thread and whether the allocation or one of the call sites on the stack of the symbolic execution is in a loop or recursion. In situations where calls are not followed due to the reuse of a method context (Section 3.4.2), the counters of those abstract objects that the reused method context specifies in its allocation set A are adjusted explicitly.

Variables and domains		**Analysis state**	
$m \in \mathcal{M}$	methods	$CAS : \mathcal{C} \to O$	alias set lookup for classes
$c \in \mathcal{C}$	classes	$AS : \mathcal{V} \to O$	alias set lookup for local vars
$v \in \mathcal{V}$	local variables		
$o \in O$	alias sets		
$b \in \mathcal{B}$	method context		

Statement	**Action**
$v =$ `new` c	`if` (t is not *unique* $\vee$ s is in a loop or recursion $\vee$ some site on the call stack is in a loop or recursion) $AS(v).nalloc \mathrel{+}= 2$; `else` $AS(v).nalloc \mathrel{+}= 1$;

Figure 3.10: Transfer rules for the uniqueness analysis. t is the current abstract thread, s is the current statement.

Program 3.3: Lazy initialization of field `random_` in class `java.lang.Math`

```
static Random random_;

static synchronized double random () {
    if (random_ == null)
        random_ = new Random ();
    return random_.nextDouble ();
}
```

The uniqueness analysis is intertwined with the symbolic execution. After the symbolic execution, all global abstract objects that have an allocation counter $nalloc \leq 1$ are marked as unique.[2]

Example. There are several unique abstract objects in the HSG of the Dining Philosopher program (Figure 3.1 and Program 3.1). The allocations of the `java.io.PrintStream` instance (o_{11}) and the character array (o_{14}) are executed once in the scope of a class initializer. Similarly the instance of class `Table` (o_8) that is allocated in the `main` method. Some objects that are actually unique are not recognized by our conservative algorithm: The allocation of the `java.util.Random` object (o_6) is done lazily in method `java.lang.Math::random` (Program 3.3). At runtime, only a single instance is created and associated with class `java.lang.Math`. The static analysis determines that the call to `java.lang.Math::random` is inside a loop and issued by a non-unique abstract thread; hence multiple executions of the method and multiple allocations of the `java.util.Random` object are assumed.

[2] Abstract objects that represent classes, `java.lang.String` constants, or `null` have $nalloc = 0$.

3.5 Experience

We first introduce the infrastructure (Section 3.5.1) and the benchmarks (Section 3.5.2) that are used for the evaluation in this and subsequent chapters. Afterwards, we report on our experience with the HSG (Section 3.5.3) and the symbolic execution (Section 3.5.4).

3.5.1 Infrastructure and runtime system

Our runtime system is based on GNU libgcj [49] (version 2.96). The numbers we present in the static and dynamic assessment refer to the overall program including library classes. The effect of native code on aliasing, object creation, and object access is modeled explicitly. We use a Pentium IV 1.4 GHz multiprocessor system for the compilation and a Pentium III 930 MHz uniprocessor for the execution of the compiled programs.

3.5.2 Benchmarks

- philo is a simple, monitor-based implementation of the Dining Philosopher problem (see Section 3.1). The used configuration executes five philosopher threads.
- elevator is a real-time discrete event simulator that is used as an example in a course on concurrent programming. Elevators are implemented as threads that poll directives from a central control board. Communication through the control board is synchronized through locks. The configuration we use simulates four elevators.
- mtrt is a multi-threaded raytracer from the JVM98 benchmark suite [108] configured with two threads. The source code is slightly modified to remove the compile-time dependence with the Java-awt-library (not supported by GNU libgcj [49]).
- sor (Successive Over-Relaxation over a 2D grid), and tsp (Traveling Salesman Problem) are data- and task-parallel applications with data access patterns of scientific codes. Synchronization among threads is based on fork-join rather than locks. The used configuration operates with two threads and performs 50 iterations over a 500 matrix.
- hedc is a warehouse for scientific data developed at ETH Zurich [113]. This benchmark represents an application kernel that implements a meta crawler for searching multiple Internet archives in parallel. In the benchmark configuration, four principal threads issue random queries to two archives each. The individual queries are handled by reusable worker threads. The workload of this application kernel is typical for Internet server applications and similar to applications based on alternative mechanisms, such as Java Servlets.
- The programs mold(yn), ray(tracer), and monte(carlo) are multi-threaded numeric application kernels from the Java Grande benchmark suite [68]. All benchmarks are executed with two threads in their 'size A' configuration.
- specjbb is an e-commerce benchmark [107]. Due to a limitation of our runtime system, we tested only a configuration with one warehouse.

- jigsaw is an Open Source web-server [130] (version 1.0alpha5). The workload for runtime experiments is created by two http-clients that each fetch 50 random web-pages of 1024 bytes.

	LOC	*bytecodes*	*methods*	*classes*		*abstract threads*	
	appl	*in CG*	*in CG*	*lib*	*appl*	*init*	*user*
philo	81	3603	192	129	2	21	2
elevator	528	6818	311	142	5	23	2
mtrt	11298	20135	719	158	34	33	2
sor	251	4481	206	132	7	25	3
tsp	706	6479	299	141	4	24	2
hedc	27952	24371	1021	208	48	47	3
mold	1402	6529	224	129	11	24	2
ray	1972	5980	270	131	19	27	2
monte	3674	8159	441	146	19	35	2
specjbb	31903	47742	1398	182	73	64	2
jigsaw	31596	32642	1396	168	129	60	6

Table 3.2: Size and thread characteristics of the benchmark programs.

Table 3.2 reports terms of lines of application code (*LOC*), and the number of *bytecodes* and *methods* in the call graph (CG) for each benchmark. The number of abstract *user* threads includes the main thread. For all benchmarks, only the main thread can be determined as unique; for other user threads, multiple concurrent runtime instances are assumed.

3.5.3 Computation of the heap shape graph

Resource requirements. The duration and memory requirements of the HSG computation are detailed in Table 3.3. Overall, the cost is moderate for the benchmarks. For each method in the CG, a method summary is computed. Method summaries are typically small, their size is however related to the depth of the call graph and the size of the benchmark: Larger benchmarks like hedc, jigsaw, and specjbb usually operate on larger, polymorphic data structures and hence have larger method summaries. In addition, for hedc and specjbb, recursions over 31, respectively 24 methods lead to a loss of context sensitivity. The summaries of methods inside the recursion might be charged with nodes that actually originate from calling context(s) of the method. Unfortunately, frequently called methods like `java.lang.String::valueOf(java.lang.Object)` are found inside the recursion, leading to unwanted aliasing among objects that flow as actual parameter into this method.

Table 3.4 reports the number of nodes in the HSG of each benchmark and classifies them according to their runtime correspondence into nodes that represent *classes*, *instances*, and *arrays*.

Shared analysis. The results of the shared analysis (Section 3.3.4) are reflected by the number of *shared* objects. The analysis is effective and shared nodes are typically a small fraction of the global nodes. The more nodes are excluded from being shared at runtime, the more effective are the optimizations of the symbolic execution (Section 3.4.2). Note that a large number of class nodes corresponds to classes that do not have static variables, or classes with only final

	time		*mem*	*method summary*	
	[s]	*[kStmts/s]*	*[MB]*	*max nodes*	*avg nodes*
philo	0.7	1.7	1	26	3.1
elevator	0.9	2.7	1	53	5.0
mtrt	2.5	2.9	1	169	8.3
sor	0.8	2.0	1	31	3.1
tsp	0.9	2.5	1	41	4.8
hedc	5.0	1.9	5	285	14.3
mold	0.8	2.9	1	25	3.5
ray	0.9	2.4	1	36	4.0
monte	1.0	3.1	1	35	4.5
specjbb	13.2	1.4	9	444	27.1
jigsaw	9.0	1.4	9	447	44.6

Table 3.3: Runtime characteristics of the reference analysis.

fields that are written during the class initialization and subsequently read. Such classes are not *shared*.

Uniqueness analysis. The uniqueness analysis (Section 3.4.4) is effective and can determine that a significant fraction of abstract objects that origin in dynamic allocations (column *unique* for instances and arrays) have a unique runtime counterpart.

Object-specific analysis. The results of the object-specific analysis (Section 3.3.6) and thread-specific analysis (Section 3.4.3) are given in columns *o/t-spec.* A considerable fraction of abstract objects that represent arrays are local to another object; this is due to Java's jagged array model (multi-dimensional arrays are represented as arrays of arrays) and due to the encapsulation of arrays in other objects which is typical in object-oriented design.

Thread-specific analysis. The thread-specific analysis is less successful: only few instances and arrays are identified as thread-specific. There are two reasons for this: First, there are few candidate objects in the benchmark programs. Second, the effects of false aliasing in combination with the conservatism of the thread-specific analysis defeat several actual thread-specific objects from being classified as such. In hedc, e.g., the imprecision of the call graph identifies the `run` methods of all user threads as being invoked explicitly (although this is only the case for one of the user threads) and hence all abstract objects accessed by these methods and their callees are not classified as thread-specific.

Complexity. Conceptually, the number of nodes in the HSG can be at least exponential in the program size. The following example is given by Ruf [100]: Imagine a method m_0 that allocates nodes in the lowermost hierarchy of a tree structure, i.e., it returns an abstract object a_0 with fieldmap $\{(\texttt{left}, a_1), (\texttt{right}, a_2)\}$. Similarly method m_1 allocates the lowermost but one hierarchy and returns abstract object b_0 with fieldmap $\{(\texttt{left}, b_1), (\texttt{right}, b_2)\}$, where b_1 and b_2 are created through method m_0. A cascade of k methods m_k would create $O(2^k)$ shape graphs nodes. Such a scenario occurs however rarely in practice, because inductive data structures are usually created and maintained by recursive or iterative methods.

	class		*instance*				*array*			
	global	*shared*	*global*	*shared*	*unique*	*o/t-spec*	*global*	*shared*	*unique*	*o/t-spec*
philo	131	4	39	6	20	2/3	27	3	19	13/0
elevator	147	2	62	11	20	15/1	36	8	19	12/0
mtrt	192	16	193	75	47	44/3	45	25	18	16/0
sor	139	1	40	2	20	2/0	32	4	21	13/0
tsp	145	5	51	8	23	8/3	38	12	24	14/1
hedc	256	36	457	155	143	110/0	72	57	27	29/0
mold	140	5	45	11	23	5/3	42	14	26	22/1
ray	150	5	65	25	24	6/5	34	6	20	18/3
monte	165	13	74	16	33	15/1	34	7	20	12/0
specjbb	255	29	393	57	122	116/0	79	33	22	33/2
jigsaw	297	46	432	147	164	116/0	90	63	29	36/0

Table 3.4: HSG nodes and their classification.

In practice, we observe that the number of nodes in the final HSG is roughly proportional to the size of the analyzed code. Figure 3.11 illustrates the relationship between the size of the analyzed program (featured by the number of *bytecodes in CG*) and the total number of nodes in the HSG (sum of *class*, *instance*, and *array* in Table 3.5).

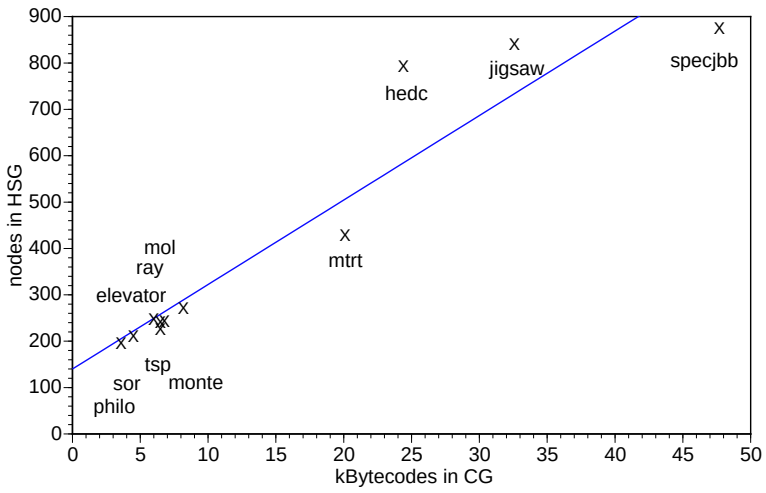

Figure 3.11: Size of the HSG (number of nodes).

We observe that the runtime of the reference analysis is almost linear in the program size (Table 3.3, between 1.4 and 3.1 kStmts/s). Larger programs perform worse, due to the sizes of the individual method shape graphs and the duplication of nodes established in the callees. Figure 3.12 illustrates the relationship between the size of the analyzed program (*bytecodes in CG*) and the duration of the reference analysis.

3.5.4 Symbolic execution

Table 3.5 reports the execution times and memory requirements of the symbolic execution. The resource demands increase rapidly but remain manageable even for our largest benchmark

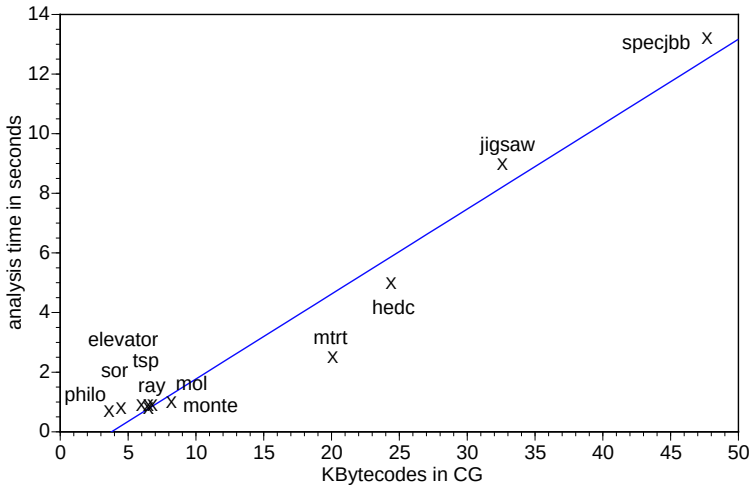

Figure 3.12: Duration of the reference analysis.

programs. One important factor that contributes to the long duration of the symbolic execution for hedc, specjbb, and jigsaw is the imprecision of available type and alias information. For these benchmarks, conservative assumptions are made in the case of objects that are instanced from dynamically loaded classes (all of them fall into a single type and alias set). Moreover, 31 methods in hedc (some of which are frequently used in different contexts) are found in one strongly connected component of the call graph. Due to the loss of context sensitivity, spurious aliasing is created in the HSG among unrelated objects, leading to further imprecision and conservative assumptions in the downstream analyses. If objects that are actually local to methods or threads become global or shared due to false aliasing, there are fewer optimization opportunities and hence more work needs to be done during the symbolic execution.

Column *time, [kStmts/s]* reports the frequency at which the program, more precisely the statements that are relevant to the symbolic execution (e.g., object, array, and class access and method invocation), are processed. Note that the symbolic execution might process the same statement several times in different contexts. The rate is higher for larger programs, i.e., hedc, specjbb, and jigsaw, than for smaller programs. This is because the overall duration of the analysis is used to calculate the rate; the overall duration includes a certain fixed ramp-up time where global data structures for the analysis are initialized. For smaller benchmarks, this initial cost contributes relatively more to the overall cost and hence, for the smaller programs, the rate calculated in *kStmts/s* looks lower than it actually is.

The efficiency of the symbolic execution depends on the effectiveness of the optimizations discussed in Section 3.4.2. For the measurements in Table 3.5 two optimizations are enabled: (1) The analysis does not descend into call sites that have no effect (alloc, read, write, escape) on shared data. (2) The analysis does not descend into calls if the callee has been processed in the same lock and object context earlier. One important aspect of determining equivalent contexts is the classification of non-global alias sets as *fully local* (Section 3.4.2).

Column *proc* in Table 3.5 specifies the number of method incarnations that are processed, *no eff*, the number of call sites that are skipped due to optimization (1), and column *cache* specifies

	time		*mem*	*call sites*		
	[s]	*[kStmts/s]*	*[MB]*	*proc*	*no eff*	*cache*
philo	0.8	1.1	1	98	145	34
elevator	0.9	1.7	2	186	251	81
mtrt	1.7	4.0	7	1049	1574	839
sor	0.8	1.4	1	47	105	13
tsp	1.0	1.5	2	158	213	56
hedc	10.4	11.2	28	16475	35268	17836
mold	1.0	2.2	1	153	289	87
ray	1.0	2.0	1	208	339	114
monte	1.0	2.7	3	399	596	185
specjbb	19.3	14.4	26	24197	109813	42381
jigsaw	29.6	11.9	24	38872	122296	87986

Table 3.5: Characteristics of the symbolic execution (*normal case*).

calls sites skipped due to optimization (2). The optimizations result in more than linear savings of analysis effort for larger programs (hedc, mtrt). The total saving is not just the numbers reported under *no eff* and *cache*, but also includes nested calls that would have been followed if the analysis had descended and processed these calls. We have experimented with different configurations of call caching to estimate the potential of more elaborate implementations. Tables 3.5 and 3.6 report four configurations:

- The *normal case* (Table 3.5) approximates the *fully local* property and determines an alias set as fully local if all connected alias sets are reachable through at most two field indirections and all these alias sets are not global. This heuristic is the default implementation to determine recurring site contexts.

- The *best case* (Table 3.6) assumes that all non global alias sets are *fully local*, i.e., no global objects are reachable through field references of alias sets that are not global. In this setting, more *site contexts* can be matched with already processed site contexts and hence call caching is more effective. While this (best case) assumption is not valid in general (it can lead to incorrect results of the symbolic execution), it expresses an upper bound on the potential of determining *fully local* alias sets. The results show, that specjbb and jigsaw may benefit from a more precise implementation for determining the *fully local* property of alias sets.

- The *worst case* (Table 3.6) assumes that there are no *fully local* alias sets, hence less site contexts are recognized as recurring. This configuration serves to rate the performance of our implementation of call caching against the most conservative assumptions to determine *fully local*. The results show, that the approximation of *fully local* in the *normal case* is quite effective to determine recurring site contexts that are not matched in the *worst case*.

- *no cache* (Table 3.6) assumes that call caching is disabled altogether. This configuration shows the importance of call caching to make the symbolic execution practical: The execution times increase most rapidly, even for programs of moderate size. We give no reports (–) for mtrt, hedc,specjbb, and jigsaw, because the analysis time exceeded two hours.

	best case				*worst case*				*no cache*		
	[s]	*proc*	*no eff*	*cache*	*[s]*	*proc*	*no eff*	*cache*	*[s]*	*proc*	*no eff*
philo	0.8	98	145	34	0.9	130	169	66	1.0	772	432
elevator	0.9	185	251	81	0.9	215	280	105	1.1	1247	768
mtrt	1.7	1034	1541	838	1.8	1304	2103	1128	–	–	–
sor	0.8	47	105	13	0.8	47	105	13	0.9	93	145
tsp	1.0	157	210	56	1.1	188	240	88	1.3	1016	635
hedc	9.8	15535	31496	18067	13.2	21794	52194	27864	–	–	–
mold	1.0	153	289	87	1.0	241	383	151	1.3	2363	1337
ray	1.0	208	339	114	1.0	307	436	177	1.4	3404	1908
monte	1.0	387	589	188	1.1	702	948	348	35.5	168682	63388
specjbb	6.0	5452	9659	7125	27.5	48530	143916	56425	–	–	–
jigsaw	5.8	7146	9882	5109	36.9	55464	166430	87315	–	–	–

Table 3.6: Effect of different call caching strategies on the symbolic execution.

Complexity. The complexity of the symbolic execution is at least as high as the complexity of computing the HSG. If all call sites are followed in a straightforward manner, methods are processed several times in different contexts and hence the complexity is exponential in the number of call sites. Figure 3.13 illustrates the execution times of the symbolic execution with different call caching configurations. The curves are exponential fits of the data points; all configurations exhibit a super-linear correlation among program size and analysis time; the *no cache* scenario definitely exhibits an exponential pattern. The other configurations show that call caching is the key to enable the symbolic execution for larger programs. The cost of the symbolic execution with call caching does not only depend on the program size, but is also strongly related to the effectiveness of the call caching and the characteristics of the application in its interaction with shared data (e.g., the number of allocations and access sites to data that is classified as shared).

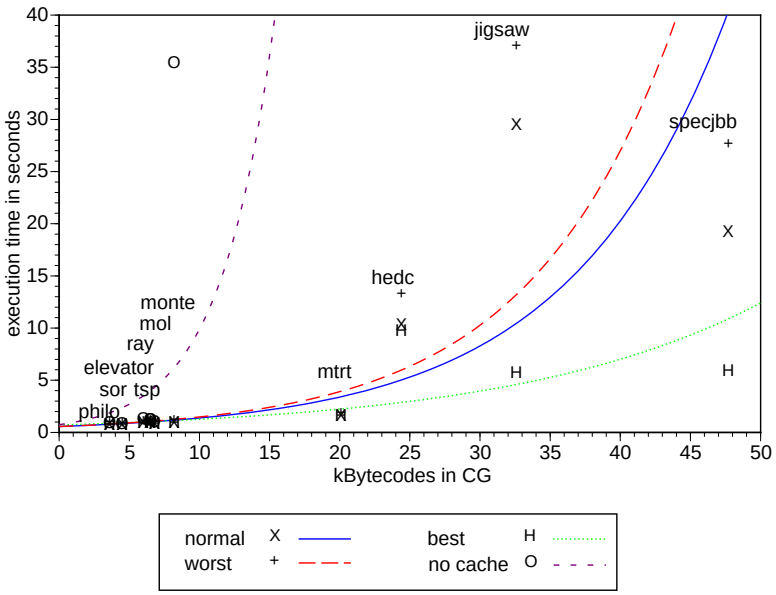

Figure 3.13: Duration of the symbolic execution with different configurations of call caching.

Applications. In Chapters 4 to 6, static analyses are described that determine three classes of synchronization defects: data races (Chapter 4), violations of atomicity (Chapter 5), and deadlock (Chapter 6). All analyses are based on the symbolic execution and its capability to track access to shared objects in different contexts. There are basically two phases in each analysis: First, data about the access/locking behavior of the program is collected along the symbolic execution; then, these data are analyzed for potential synchronization faults.

3.6 Discussion

3.6.1 Sources of unsoundness

The shared analysis does not consider init threads as separate runtime entities and thus may not classify objects as shared although they are accessed in the scope of a class initializer and a user thread (which may be different from the thread that executes the class initializer). This simplification makes the designation of abstract objects as *shared* unsound. In practice, there are two reasons why this source of unsoundness is not a significant problem for the detection synchronization defects: First, the situation in which certain data is accessed by a thread in the scope of a static initializer and otherwise only by threads different from the initializing thread is rare (we have not observed such a scenario in the benchmarks we investigated). Second, we have observed that static initializer methods typically create and initialize objects that are consequently available to other threads. Ordering between initialization and subsequent uses is either guaranteed through Java's class initialization mechanism (see [53, Chapter 12.4]), or through the fact that the initializer makes the reference to some object available only after the initialization is performed. Hence, abstract objects that are mistakenly not classified as shared are typically not candidates for synchronization defects.

Our model does not consider the execution of `finalize` methods that are typically invoked by a separate *finalizer thread*. This indifference could lead to a situation where an object is accessed by two runtime threads (one of which is the finalizer thread) but is not classified as shared at the abstract level by our static analysis. Finalization in Java, is generally regarded as a source of errors that is hardly amenable to static and dynamic analysis [14]. Finalizers should be used sparsely and specified with care; the benchmarks we evaluated do not make use of finalization.

3.6.2 Sources of incompleteness

As mentioned earlier, spurious aliasing can be a source of imprecision in the sense that a number of allocation and access sites that actually operate on thread-local data may be deemed to operate on shared data just because one of the abstract objects merged into the bloated common abstraction happens to be shared.

On the example of the thread-specific analysis, we notice that imprecision in the call graph and heap shape graph can spoil the recognition of thread-specific objects. Such objects might falsely be reported as target of a synchronization defect in downstream analyses (e.g., Chapter 4).

Hence, precise alias and type information is important to bound the scope of the symbolic execution and to direct the search for synchronization defects. A pragmatic design choice is to consider flow-insensitive algorithms for reference analysis because they abstract from statement ordering and interleaving in multi-threaded programs. However, this choice reduces the precision of alias information compared to flow-sensitive analyses. Moreover, our reference analysis is based on unification. While this choice offers attractive performance (the complexity is typically linear in the size of the program) and ease of implementation, Ruf [100] notes inaccuracies in this schema. In some cases, false aliasing can be avoided through a manual program transformation. Alternatively, a more precise, subset-based scheme [6] could be used. Liang et al. [77] compare unification and subset-based reference analyses for Java programs and observe that subset-based analyses can be significantly more precise. A subset-based, context-sensitive reference analysis for Java programs has been developed by Milanova et al. [86]. Similar to our work, the notion of context is based on objects, not call strings like in previous work, e.g., [55]. The technique in [86] avoids the redundant analysis at method invocations with different call sites but the same target object. The analysis is generic such that not only the target of a call, but also other variables can be used for context disambiguation; in our analysis, the global objects that are reachable through actual parameters, the abstract thread, and the set of current abstract locks determine the context of a method invocation. Despite the cubic time worst case complexity of the subset-based reference analysis, [86] shows that object-sensitivity enables the efficient analysis of even large and complex Java programs.

3.6.3 Related work

Extensive research has been done in the field of reference and points-to analysis for object-oriented languages. A good overview on this complex topic is given by Hind [62, 61], Rinard [99], and Ryder [102]. The discussion here is focused on analyses for multi-threaded programs.

The principle of a data flow analysis is to determine for a given statement, the set of control flow successors; in sequential programs, this set is typically small and hence allows flow-based analyses to be more precise than flow-insensitive analyses. For multi-threaded programs however, the set of successor statements may encompass all statements of concurrent threads, which can significantly reduce the accuracy of the data flow analysis. Three principal strategies have been pursued to handle this problem (from the most conservative to the most precise):

- *Flow-insensitive analysis.* A flow-insensitive analysis is oblivious to the execution order of individual statements and hence the result is independent of the control flow of the program.

- *Thread structure and synchronization based analysis.* This kind of analysis refines the flow-insensitive analysis and rules out certain control flows that are impossible due to the thread start or join relation or explicit synchronization.

- *Coarsened flow-sensitive analysis.* This kind of analysis operates on an approximated set of possible flows of a parallel program. Intra-thread and inter-thread activity are distinguished to reduce to the number of relevant flows that the analysis has to consider.

Flow-insensitive analysis. Conceptually, a flow-insensitive analysis assumes that all intra-thread and inter-thread flows are possible and thus this kind of analysis can be easily adopted to multi-threaded programs. Context-insensitive [15] and context-sensitive variants of this analysis [21, 100] are sufficiently accurate to determine the sharing of objects and to effectively remove unnecessary synchronization. Our approach uses the flow-insensitive and context-sensitive technique in [100] to approximate the call structure and to compute a global model of heap data (Section 3.3). Some escape analyses are sensitive to the data flow among local variables (inclusion-based reference to analysis), however field variables – which may be accessed from multiple threads – are treated in a flow-insensitive manner [24, 128]; hence these analyses are also conservative about the ordering of accesses to shared variables and thus we categorize them as flow-insensitive here.

Thread structure and synchronization based analysis. This kind of analysis is particularly effective for programs that declare the structure of threading explicitly; these programs typically follow fork-join or post-wait style synchronization. Rugina and Rinard [101] developed a flow-sensitive reference analysis that explicitly considers the data flow at thread fork and join points.

Thread structure analysis is however hard to apply to programs with unstructured parallelism where the lifetime and the number of threads is difficult or impossible to assess by a static analysis, and many threads execute the same code; typical Java and C♯ server applications fall into this category. In Ruf's [100] and our work, threads are determined by their type. The model in Choi. et. al. [26, 25] can be more precise because in addition to the type, start site and the context of the start site are used to differentiate threads at compile-time.

Choi et al. [26] use the *inter-thread control flow graph* as program representation; this graph contains one control flow representation of each method and connects thread start sites with the corresponding thread entry methods (thread start relation). The authors develop a context-sensitive points-to analysis where the notion of a context is defined by the control flow path that reaches a statement. The authors note that this path-sensitive approach is very expensive and hence evaluate only a simplified, less precise *all-path* (i.e., context-insensitive) variant of their analysis.

Our work has a different context model: Not the control flow that reaches a method incarnation, but the relevant abstract objects that some method incarnation operates on define the context (Section 3.4). This notion of context – although expensive for larger programs – allows an efficient context-sensitive analysis of the largest benchmark that is checked in [26], which is `mtrt`.

In addition to the thread structure, the analyses provide also information about common synchronization contexts. Choi et al. [26] use must-alias information to determine common synchronization objects along different path contexts. In our analysis, the symbolic execution (Section 3.4) tracks the set of abstract locks that are held at a certain statement during the context-sensitive symbolic execution (Section 3.4).

Coarsened flow-sensitive analysis. A simple flow-sensitive procedure would consider all interleavings of instruction of all threads. This procedure is however not practical and performs a lot of redundant work. There are analysis techniques that do better and exploit the observation that thread interference is limited to certain program points or language constructs; these analyses handle points of potential interference specially and treat program regions between

interaction points (that perform only local computations) like sequential code. The concept of treating code regions instead of individual statements as the unit for the intra-thread analysis is called *coarsening*.

Knoop et al. [72] developed a data flow analysis for parallel programs that processes each thread individually and propagates the findings about potential interactions of the analyzed thread to threads that may execute concurrently. For certain bit-vector analyses the precision of this algorithm is the same as a the precision of a procedure that considered all possible thread-interleavings. The accuracy for a more complicated reference analysis is however hampered because the algorithm overestimates the effects of thread-interaction [99].

For multi-threaded object-oriented programs flow-sensitive analysis techniques have been proposed in [32, 90, 88, 131] that compute a precise models of dynamic data structures and concurrent threads at every program point. A crucial aspect of effective coarsening is to determine as few relevant thread interaction sites as possible. In [32], the interaction sites are narrowed by determining thread-local data (access to those is not an interaction point) and data that are protected through locking (relevant interaction point is the lock access not the data access). While coarsening enables these analyses, the complexity of these analyses is still an important limitation when treating larger programs.

The work on flow-sensitive analyses of multi-threaded programs mostly presents techniques that enable this form of analysis. However, there is yet few work that actually applies these techniques to relevant programs and problems like race detection. Hence it is difficult from our perspective to assess the practical benefit that these techniques have over techniques that are flow-insensitive or variants thereof.

4

Static detection of data races

The identification of data races requires information about three aspects of a program:

- *Program data.* This aspect is approximated by a reference analysis; data that is potentially shared is modeled by abstract objects in the HSG.
- *Access events.* Events that access shared data are approximated by the symbolic execution; the program model underlying the symbolic execution is based on abstract threads, the call graph (CG), and the control flow inside methods (CFG).
- *Synchronization.* Two ways of explicit inter-thread synchronization are determined by the static analysis: First, the symbolic execution tracks locking along monitor boundaries. Second, methods for thread start and join are recognized in the call graph and control flow.

The core of the static data race detection is a data structure, called the *object use graph* (OUG) [124], that combined these three aspects. An OUG can be understood as the lifemap of an abstract object that notes events relevant to that object (creation, access, escape, thread start and join) and approximates the ordering among those events according to intra-thread control flow and inter-thread synchronization.

Section 4.1 defines OUGs and provides an example. Section 4.2 describes how OUGs are computed along the symbolic execution. Sections 4.3 and 4.4 describe the data race detection based on OUGs. Section 4.5 reports on our experience and the experiments we have done with benchmark programs.

4.1 Object use graphs

An OUG refines "escape" information: Instead of attributing a global classification to an abstract object and the sites it is accessed from, the OUG recognizes structural, temporal, and lock-based protection of object access in different contexts. An *object use graph* (OUG) contains information that is specific to one abstract object, hence there is a separate OUG for every node in the HSG. For the purpose of data race detection it is sufficient to build OUGs for those abstract objects that can be subject to a data race, i.e., that have the *shared* property. The nodes in the OUG represent *abstract events*, i.e., a common representative of runtime events that are issued by a specific statement in a specific runtime context. An abstract event e_i is a tuple $\langle k, m, o, t, L \rangle$:

- i is a unique id.
- k specifies the kind of abstract event. Possible kinds are

NEW.	Object allocation.
GET/PUT.	Read or write access to a field of the object.
LOAD/STORE/ESCAPE.	Fetch or deposit a reference to the object from/to another variable. ESCAPE is a variant of STORE and occurs if the reference to the object is leaked into a context or variable where it might be made available to other threads.
START/JOIN.	Start or join of a thread.
ENTRY/EXIT.	Method entry and exit. These events do not correspond to a program action and serve only to record the compiler's analysis.
CALL.	Method invocation site. These nodes are only used during the construction of the graph (Section 4.1); the effect of events that are issued downstream of calls are "inlined" into the graph at the position of the CALL node. For polymorphic call sites, each target method is processed separately. Recursive calls sites are not unfolded, but connected to the surrounding invocation context (Section 4.2).

- m is the accessed member in case of CALL and GET/PUT nodes.
- o is the abstract target object in case of LOAD/STORE/ESCAPE nodes and the root of the controlled thread in case of a START/JOIN event.
- t is the abstract thread issuing the event.
- L is the set of abstract objects that are locked during a GET/PUT event.

There are three relations among abstract events:

- *Control flow relation* γ*:* Control flow edges establish an intra-thread relationship among abstract events. Not all events issued by the same abstract thread are however (transitively) related: For reasons of efficiency, the inter-procedural analysis maps different access contexts of an object separately into the OUG, and hence there is no control flow relation among events from different contexts (details are given in Section 4.2.2). If the access occurs inside a loop or recursion, the control flow relation is cyclic.
- *Reference flow relation* ρ*:* An object cannot be accessed unless a reference to it is available in the current method context. There are two ways of conveying references between contexts: First, through the call stack and parameter passing; this aspect is covered by the control flow relation. Second, through other objects that store the reference in a field variable and make it available in another context; this aspect is addressed by the reference flow relation that relates STORE/ESCAPE events with corresponding LOAD events. Reference flow relates events of the same thread or different abstract threads.

- *Thread control relation* τ*:* A thread t cannot issue any event before t has been started. This fact is modeled by a relation between the thread START event and all abstract events issued by the started thread. Conversely, all events of an abstract thread are related to the corresponding JOIN event. Depending on the context in which a thread is started or joined, START and JOIN events might not be at all or only incompletely be recorded in an OUG. There are three important constraints that have to be met by thread START/JOIN events to yield useful information and hence establish the thread control relation.

  – A START event is only valid if the abstract thread issuing the event is unique. This property of a START event can be easily determined during the assembly of the OUG (Section 4.2.2).

  – A JOIN event is valid if all instances of a given joined abstract thread are actually joined at that event. The automated analysis uses the following heuristic to determine this property: If there is a single JOIN event for a given abstract thread, then it is valid. This heuristic is not sound (i.e., there is no guarantee that the constraint is actually met at runtime) but has been correct for the benchmarks we investigated. This limitation is discussed further in Section 4.7.

  – If the set of START events is incomplete (e.g., an abstract thread is started in two contexts, but only one START event is noted in the OUG), then the START events are invalid.

  Invalid events are ignored by the analysis; while this loss of start and join information can be tolerated by the conflict analysis, it may deteriorate the precision because conservative assumptions must then be made about the concurrency of events, e.g., in the starter and started thread.

Definition. An OUG is a directed graph $OUG[o] = \langle N, E \rangle$:

- o is the abstract object that is characterized by the OUG.
- N is the set of nodes corresponding to abstract events.
- E is the set of edges corresponding to the control flow relation γ.

Example. We continue the example of the Dining Philosopher program (Program 3.1 and Figure 3.1). Figure 4.1 shows the OUG of the `Philo` object (o_7 in the HSG in Figure 3.1).

There are two abstract threads that contribute to the events registered in the OUG: Events (1) to (17) result from the `main` thread (o_1), events (18) to (21) from the `Philo` thread (o_2). Events (1) to (11) are triggered by method `Main::main` and correspond to allocation, constructor of `Philo`, thread start, join, and the invocation of `Philo::report`. Events issued in callees of `Main::main` are noted separately (e.g., events (12) to (14) for the constructor invocation) and inlined at the corresponding CALL node (e.g., node (3)).

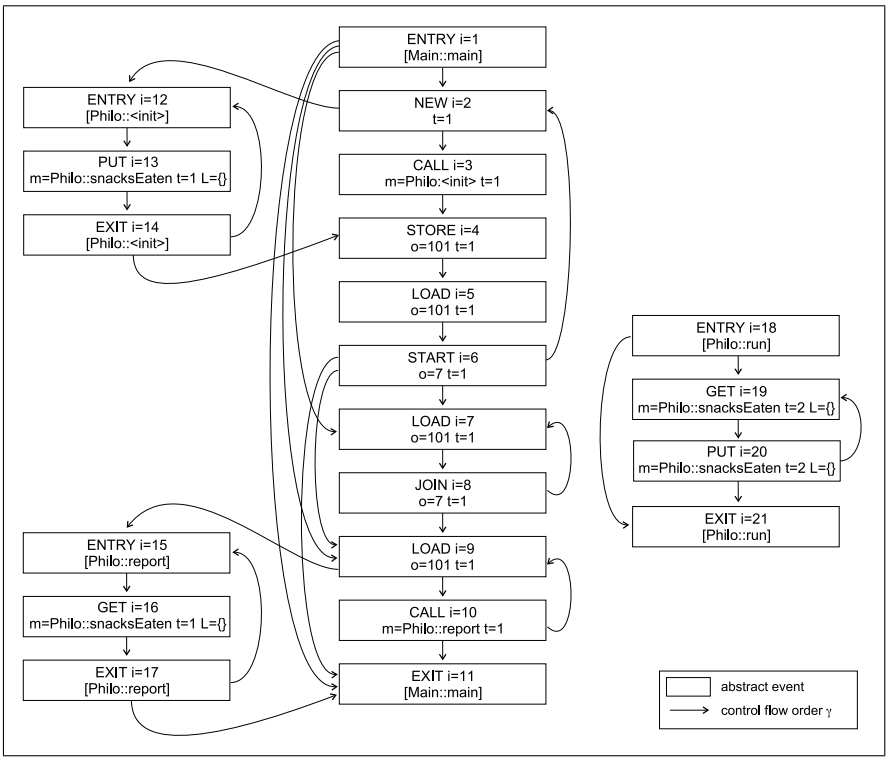

Figure 4.1: OUG for the `Philo` object.

4.2 Building object use graphs

OUGs are associated with abstract objects (nodes in the HSG) and are created gradually during the symbolic execution. The creation of OUGs is conceptually split into an intra-procedural and an inter-procedural analysis. The intra-procedural analysis computes subgraphs that model the treatment of an abstract object inside a specific method, the inter-procedural analysis coordinates the assembly of sub-graphs and connects them to complete OUGs.

4.2.1 Intra-procedural analysis

Similar to an OUG at the whole program level, a so-called *method object use graph* (MOUG) models the relevant events at the level of an individual method. A MOUG can be understood as a control flow graph in which statements that do not result in events for an abstract object of interest are pruned. A MOUG is generic with respect to the thread and locking context and serves as building block that is copied and instanced in specific contexts along the creation of OUGs in the inter-procedural analysis (Section 4.2.2). A MOUG is a connected, directed graph $MOUG[m,R] = \langle N, E \rangle$:

- m is the method underlying the MOUG.
- R is a set of alias sets $o_0, \ldots, o_n$ from the method summary of m that, in a given method context, are aliases' of the object of interest; if the object of interest corresponds to a class, R contains only a single entry representing the class.
- N is a set of nodes that correspond to relevant abstract events.

- E is a set of edges that correspond to the control flow relation among events.

For a specific method *m*, different MOUGs can exist, depending on the object of interest and aliasing that is given in different calling contexts. A MOUG is built in a single flow-sensitive method traversal. The transfer rules for the computation of relevant events are specified in Figure 4.2.

The operation *createEvent* adds an abstract event to the graph. The thread or locking context remains unspecified at this point; this information is furnished when an MOUG is cloned and embedded into a specific calling context during the inter-procedural analysis. The edges among the events are generated according to the control flow (CFG). For *multiply executed* methods (Section 3.2), a control flow edge is added from EXIT to ENTER. This is necessary to approximate the ordering of abstract events inside OUGs correctly (Section 4.3).

Access to class variables, fields, or arrays can have two effects: First, a GET or PUT node is created if the target object corresponds to the object of interest. Second, a LOAD or STORE node is created if the object that is referenced by the handled value corresponds to the object of interest.

`throw` and `catch` statements lead to STORE and LOAD events in the MOUG and are used as hooks for modeling the reference flow of exception objects inside a method and across method boundaries.

An object allocation leads to a NEW event if the allocated object corresponds to the object of interest.

CALL nodes serves as hooks for inlining the MOUG of a callee during the inter-procedural analysis (Section 4.2.2). The following three strategies for introducing CALL nodes in an MOUG are are possible:

The first strategy would insert a CALL node at every call site. This option is however too expensive and lets the size of the MOUGs grow considerably.

The second strategy would omit CALL nodes altogether; while this alternative is efficient and still lead to correct OUGs, essential control flow information across method boundaries is tossed away and hence the results of consequent analyses would be unnecessarily conservative and imprecise.

The third strategy is a balance between the first and the second option and reflects the implementation that we have chosen: At a call site, a node should be created if the callee starts or joins a thread or the callee accesses the object of interest. The occurrence of start or join can be approximated as follows: Invocations of `java.lang.Thread::start` and `java.lang.Thread::join` lead to START and JOIN nodes. If start and join happen in the scope of a callee, a CALL node is created that might later (see Section 4.2.2) be converted to a START or JOIN. *mayStartThread*(n) specifies that n or one of its callees starts a thread on some control flow path, *mustJoinThread*(n) specifies that a thread is joined on all control flow paths; both predicates are computed along an inter-procedural analysis. For JOIN, we must also ensure that the caller m satisfies *mustJoinThread*(m), otherwise there could be a control flow that bypasses the JOIN in m. The effects of the callee to the object of interest are approximated as well: There are three basic cases how a reference to the object of interest can be available to a callee: (1) A reference is passed on the stack as actual parameter; (2) the reference is obtained by name (for classes); (3) the reference is read from the heap. The solution specified in Figure 4.2 recognizes all cases that fall in categories (1) and (2): A CALL node is created if

Variables and domains		**Analysis state**	
$m, n \in \mathcal{M}$	methods	$CAS : \mathcal{C} \to O$	alias set lookup for classes
$c \in \mathcal{C}$	classes	$AS : \mathcal{V} \to O$	alias set lookup for local vars
$f \in \mathcal{F}$	fields		
$v \in \mathcal{V}$	local variables		
$o \in O$	alias sets		

Statement	**Action**

$v = c.f$	if $(CAS(c) \in R)$
	  $createEvent(\langle \text{GET}, f, -, -, - \rangle)$;
	if $(AS(v) \in R)$
	  $createEvent(\langle \text{LOAD}, -, CAS(c), -, - \rangle)$;

$c.f = v$	if $(CAS(c) \in R)$
	  $createEvent(\langle \text{PUT}, f, -, -, - \rangle)$;
	if $(AS(v) \in R)$
	  $createEvent(\langle \text{STORE}, -, CAS(c), -, - \rangle)$;

$v_0 = v_1.f$	if $(AS(v_1) \in R)$
	  $createEvent(\langle \text{GET}, f, -, -, - \rangle)$;
	if $(AS(v_0) \in R)$
	  $createEvent(\langle \text{LOAD}, -, AS(v_1), -, - \rangle)$;

$v_1.f = v_0$	if $(AS(v_1) \in R)$
	  $createEvent(\langle \text{PUT}, f, -, -, - \rangle)$;
	if $(AS(v_0) \in R)$
	  $createEvent(\langle \text{STORE}, -, AS(v_1), -, - \rangle)$;

$v_0 = n(v_1, \ldots, v_k)$	if $(n =$ `java.lang.Thread :: start`$)$
	  $createEvent(\langle \text{START}, -, AS(v_1), -, - \rangle)$;
	else if $(n =$ `java.lang.Thread :: join` $\wedge$ $mustJoinThread(m))$
	  $createEvent(\langle \text{JOIN}, -, AS(v_1), -, - \rangle)$;
	else if $(mayStartThread(n) \vee$
	  $(mustJoinThread(n) \wedge mustJoinThread(m)) \vee$
	  $(\exists i : AS(v_i) \in R) \vee$
	  $(o \in R$ corresponds to a class $\wedge$ n accesses $o))$
	  $createEvent(\langle \text{CALL}, n, -, -, - \rangle)$;

$v =$ `new` c	if $(AS(v) \in R)$ $createEvent(\langle \text{NEW}, -, -, -, - \rangle)$;
`throw` v	if $(AS(v) \in R)$ $createEvent(\langle \text{STORE}, -, -, -, - \rangle)$;
`catch` v	if $(AS(v) \in R)$ $createEvent(\langle \text{LOAD}, -, -, -, - \rangle)$;

Figure 4.2: Transfer rules for the computation of MOUGs. m is the method to be processed, R is the set of relevant alias sets.

a relevant alias set is passed as actual parameter or returned from the method. If the relevant alias set stands for a class, a CALL node is created if the callee reads or writes fields of that class. The former aspect can be directly determined at the call site, the latter information can be obtained from the read and write sets that are associated with the method summary of the callee. If the callee obtains references to the object of interest through alternative (3), the CALL node might not be noted in the OUG. Section 4.2.2 discusses how the inter-procedural analysis accommodates this case and relates abstract events of caller and callee (reference flow relation ρ).

The single-entry-single-exit region of a method is delimited in the MOUG by a BEGIN and an END node. These nodes serve as hooks when the MOUG is copied and inlined at a CALL node during the inter-procedural analysis (Section 4.2.2) .

Example Figure 4.3 shows the CFG for the `Main::main` method. The statements are denoted as pseudo-bytecode instructions (A)–(L) that lead to abstract events in the MOUGs.

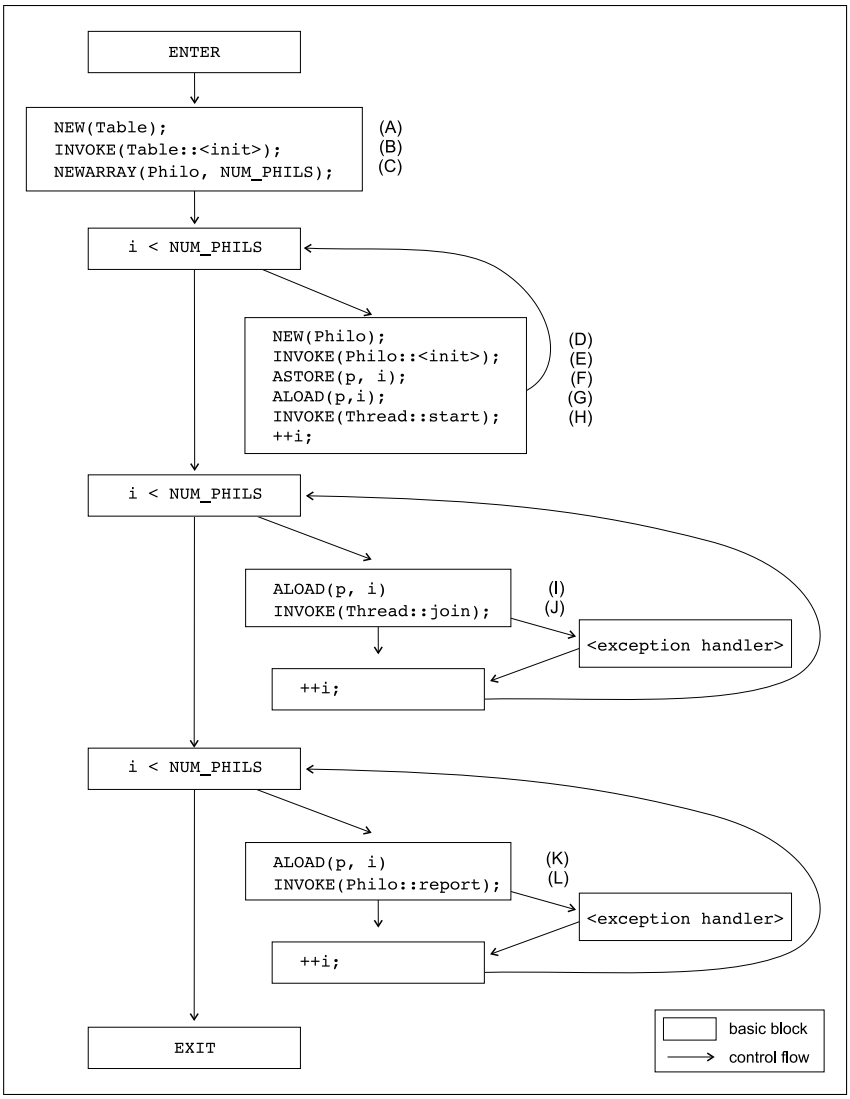

Figure 4.3: CFG of method `Main::main`.

Some possible MOUGs of this method are illustrated in Figure 4.4. The annotations on the abstract events unveil the correspondence between abstract events and program actions. For MOUG (a), the object of interest is the `Philo` instance (o_7); for the MOUG (b), the object of interest is the `Table` instance (o_6); the MOUG (c) models the effects of the method to the array instance holding reference to the thread objects (o_{101}); finally MOUG (c) models the interaction of the method with the string array passed as parameter (o_{100}) – there is no access, hence only the START and JOIN events are noted.

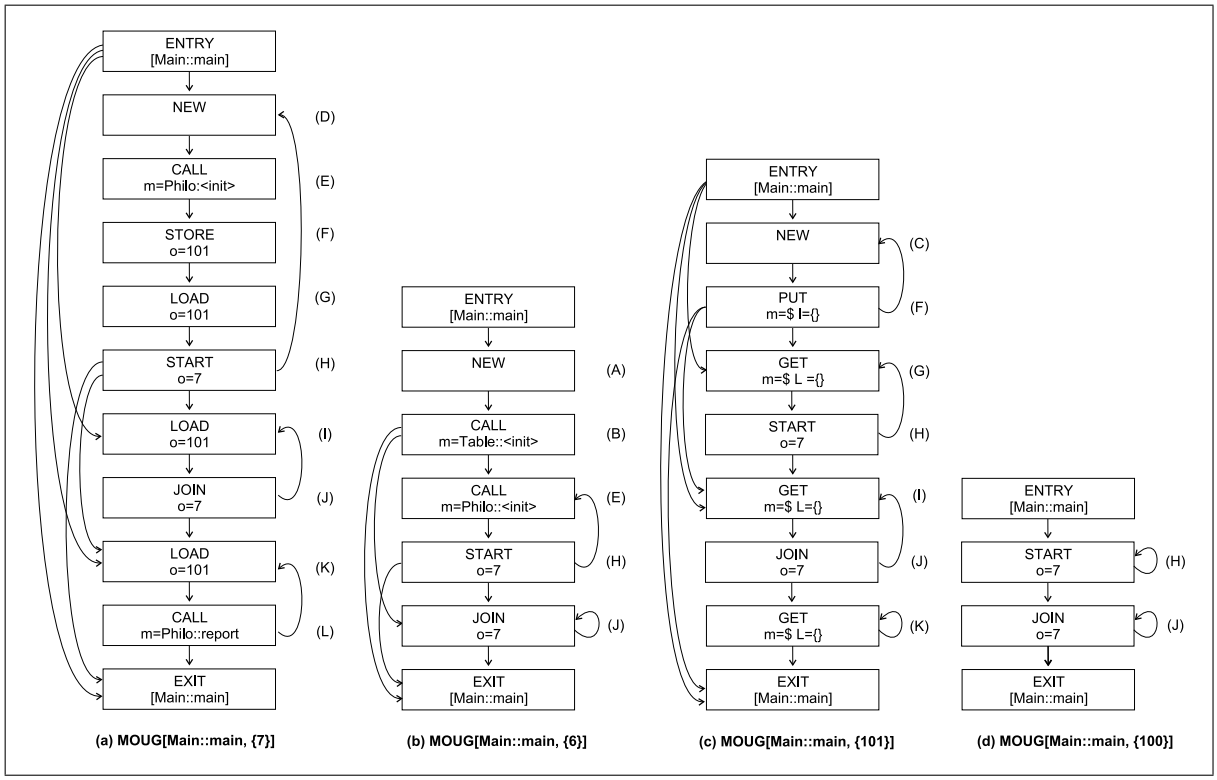

Figure 4.4: Example MOUGs of method `Main::main`.

4.2.2 Inter-procedural analysis

The inter-procedural computation of OUGs proceeds in three phases:

1. *Assembly and control flow relation.* Assemble all MOUGs to an OUG during the symbolic execution. After this phase, all relevant abstract events are noted in the OUG. This phase establishes the control flow relation.

2. *Reference flow relation.* Compute the reference flow relation and determine ESCAPE events.

3. *Thread control relation.* Compute the thread-control relation.

Assembly and control flow relation. Figure 4.5 shows extensions of the transfer rules of the symbolic execution in Figure 3.8 that build the OUGs. The description uses the following auxiliary functions:

Variables and domains

$m \in \mathcal{M}$	methods
$v \in \mathcal{V}$	local variables
$o \in O$	alias sets
$b \in \mathcal{B}$	method context
$c \in \mathcal{C}$	classes
$f \in \mathcal{F}$	fields
$t \in \mathcal{T}$	abstract threads
$s \in \mathcal{S}$	statements
$e \in \mathcal{E}$	abstract events
$p \in \mathcal{P}$	subgraph of OUG

Analysis state

$CAS : \mathcal{C} \to O$	alias set lookup for classes
$AS : \mathcal{V} \to O$	alias set lookup for local vars
$MC : \mathcal{M} \to \mathcal{B}$	method context creation

Statement	**Action**
$v = c.f$ $c.f = v$	`let` $p = getMOUG(CAS(c), b),\ e = getEvent(p, s)$: $initEvent(e, -, t, L)$; `let` $p = getMOUG(AS(v), b),\ e = getEvent(p, s)$: $initEvent(e, CAS(c), t, L)$;
$v_0 = v_1.f$ $v_1.f = v_0$	`let` $p = getMOUG(AS(v_1), b),\ e = getEvent(p, s)$: $initEvent(e, -, t, L)$; `let` $p = getMOUG(AS(v_0), b),\ e = getEvent(p, s)$: $initEvent(e, AS(v_1), t, L)$;
$v_0 = m(v_1, \ldots, v_k)$	`let` $b' = MC(m) = \langle\langle f_1, \ldots, f_k\rangle, \langle l_0, \ldots, l_n\rangle, r, e, A', R', W', E'\rangle$: $\forall i \in \{0, \ldots, l\}$: `let` $p = getMOUG(AS(v_i), b),$ $e = getEvent(p, s)$: $initCall(e, b')$; $\forall o \in R' \cup W'$: `let` $p = getMOUG(o, b), e = getEvent(p, s)$: `if` ($o$ corresponds to a class) $initCall(e, b')$;
$v =$ `new` c `throw` v `catch` v	`let` $p = getMOUG(AS(v), b),\ e = getEvent(p, s)$: $initEvent(e, -, t, L)$;

Figure 4.5: Transfer rules for building the OUGs along the symbolic execution. t is the current abstract thread, s is the current statement, L is the current set of locked abstract objects, b is the current method context. The value '$-$' expresses that no information is provided for the specific field in the event record.

- For each method context in which an abstract object is allocated/accessed/escaped, a copy of the corresponding MOUG is created and mapped into $OUG[o]$. Function $getMOUG(o, b)$ creates such a subgraph in $OUG[o]$ for the method context b, if it is not already available, and returns it.

- Function *getEvent*(p,s) returns the abstract event corresponding to statement s in the subgraph p.[1]
- The operation *initEvent*(e,o,t,L) associates context information with an abstract event e. t is the abstract thread issuing the event, L is the set of locks held during a GET, PUT, LOAD, or STORE event. For LOAD and STORE events, the object o hosting the reference to the object of interest is initialized. In the case where a STORE or LOAD corresponds to a `throw` or respectively `catch` statement, there is no alias set for the hosting object. The computation of the reference flow relation accounts for this specific case (next paragraph).
- The operation *initCall*(e,b) inlines a clone of a MOUG (obtained by function *getMOUG*) into its calling context, i.e., at the CALL node e that corresponds to the call site. Predecessors of the CALL node become predecessors of the ENTRY node of the subgraph, successors of e become successors of the EXIT node. If the context b is encountered at several call sites (see the optimization of call caching in Section 3.4.2), the inlining is done for all corresponding CALL nodes (there is however only a single copy of the MOUG).

The assembly phase establishes the control flow relation: the intra-procedural control flow is taken from the MOUGs; the inter-procedural control flow is established through the inlining of calling contexts in cases where the reference to the object of interest is passed as argument between caller and callee context. Hence, not all events that target the same abstract object are related through the control flow relation; disconnected contexts obtain the reference to the object through the reference flow relation (next paragraph).

Reference flow relation. This phase relates STORE events with *compatible* LOAD events. Two events $\langle \text{STORE}, -, o, t, L\rangle$ and $\langle \text{LOAD}, -, o', t', L'\rangle$ are compatible if the hosting objects are equal, i.e., $o = o'$. STORE events are mutated to ESCAPE if the event might render the reference available to other runtime threads, i.e., if the hosting object is shared. References to classes are obtained by name, hence there are no LOAD and STORE events in the OUG of a class. Nevertheless, an artificial ESCAPE event is introduced that constitutes the root of the OUG. There reference flow ρ relates this ESCAPE event with the ENTRY nodes of all unconnected components.

Thread control relation. This phase mutates CALL events that are not expanded to START or JOIN nodes. Such CALL nodes are added to the MOUG due to the predicates *mayStartThread*() or *mustJoinThread*() (Figure 4.2). The thread control relation relates valid START events with all events of the started abstract threads and conversely all events of that abstract thread with the corresponding valid JOIN event.

Example. Figure 4.6 shows the OUG of the `Table` instance (o_6) in the Dining Philosopher program (Program 3.1 and Figure 3.1). The graph is assembled from 6 MOUGs that

[1] There can be two abstract events for one statement; e.g., a `getfield` bytecode can cause a GET and LOAD event in the same OUG. The given description of the *getEvent* function does not account for this and is simplified for this illustration.

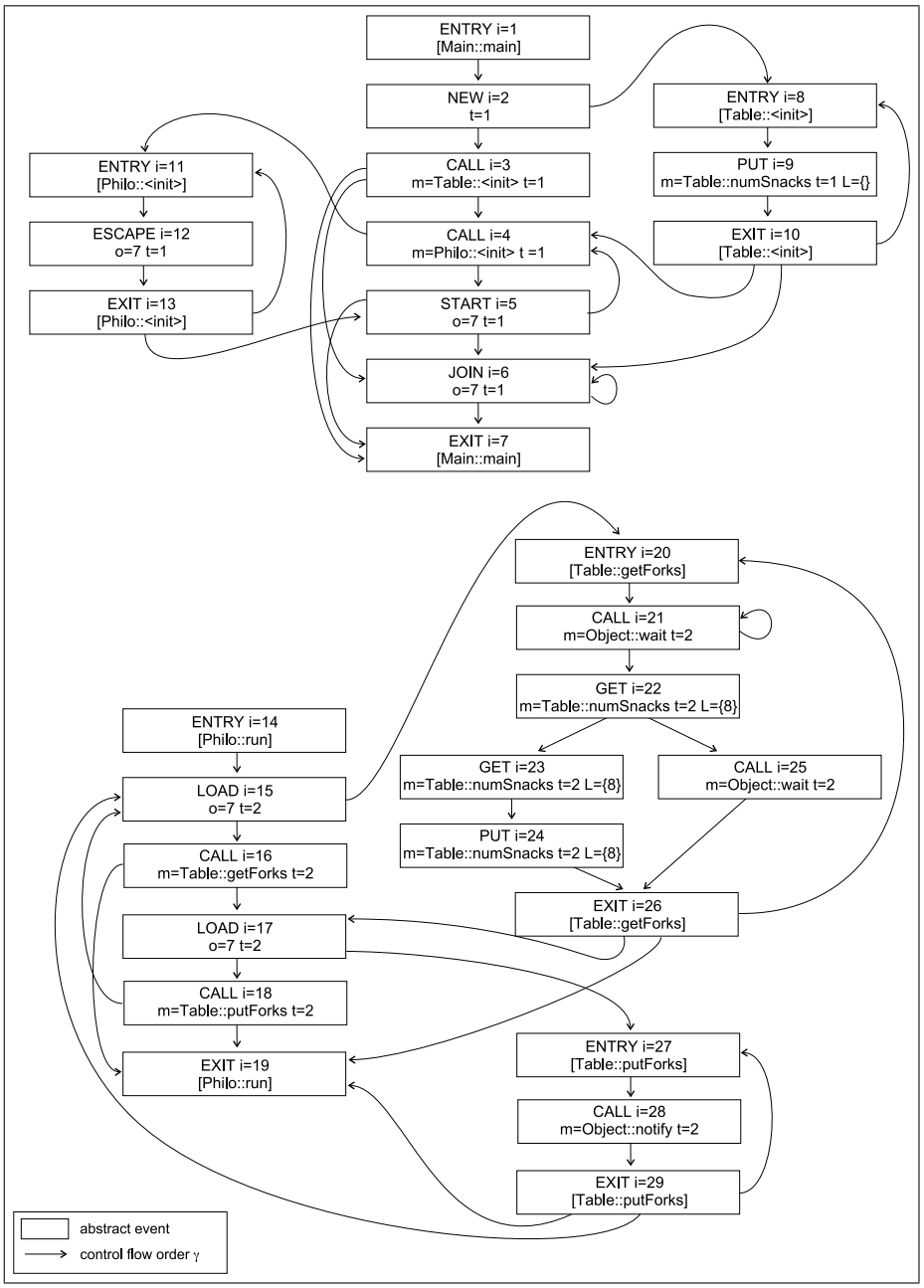

Figure 4.6: OUG for the `Table` object (o_6).

model events relevant to the abstract object (o_6) in different method contexts; in this simple example, all method contexts are from different methods: Events (1)–(7) stem from method `Main::main` (see also Figure 4.4 (b)), events (8)–(10) from `Table::<init>`, events (11)–(13) from `Philo::<init>`, events (14)–(19) from `Philo::run`, events (20)–(26) from `Table::getForks`, and events (27)–(29) from `Table::putForks`. Note that the events (1)–(13) and (14)–(29) are issued by different threads and hence are located in different disconnected components of the OUG.

4.3 Approximating happened-before

This section derives properties of runtime events and their ordering relation from the abstract event model. For the purpose of conflict detection, it is sufficient to consider only the happened-before relationship among events that address/handle the same class or runtime instance, i.e., we focus the investigation on events in the same OUG.

4.3.1 Runtime model

We use the following model to illustrate the correspondence of runtime events and abstract events: Assume that each object has a counter associated with it which is incremented at each occurrence of a runtime event. Possible runtime events of an object o correspond to the homonymous abstract events:

- *NEW.* Allocation of the instance o.
- *PUT/GET.* Field access at object o.
- *LOAD/STORE/ESCAPE.* Handling of the reference to instance o.
- *ENTRY.* Start of a method. If o is an instance, then a reference to o is passed as actual parameter; if o is a class, then some variable of that class is accessed in that method.
- *START/JOIN.* Some runtime thread is started or joined.

Let $r_0, \ldots, r_n$, be a sequence of runtime events that target a certain runtime object (class or instance) and $0 \leq i \leq n$ be the value of the event counter associated with that object. The context and statement issuing a runtime event r_i allow to map r_i onto its abstract counterpart e_i in the OUG. Let $e_0, \ldots, e_n$ be the sequence of abstract events that correspond to the execution $r_0, \ldots, r_n$.

We assume that the runtime order of events is consistent with γ, ρ and τ on the abstract events in the OUG. Consistency means that for all runtime events $r_i, i > 0$, at least one $r_j, j < i$ must exist such that e_j is related to e_i through control flow, reference flow or thread control relation, i.e., $\gamma(e_j, e_i) \vee \rho(e_j, e_i) \vee \tau(e_j, e_i)$. The consistency between the order of the runtime events and corresponding abstract events in the OUG reflects the intuitive notion of the combined relations γ, ρ, and τ. Moreover, we assume that GET events return the value of the most recent PUT to the same variable (or the null value if there is no previous PUT).

This model, which assumes a total order of runtime events and sequentially consistent memory, is a simplification of today's hardware and compiler systems. The reordering of runtime events through the compiler or hardware can lead to incompatibilities of the runtime reality with the abstract event model, i.e., the relations γ, ρ and τ. We discuss this effect and its consequences on the static conflict detection in Section 4.6.

According to the runtime model, an execution associates a set of values with an abstract event; the set expresses the value of the event counter at the time a runtime instance of the event occurred. Let $C(e)$ be the set of counter values for a specific abstract event e in a specific execution.

Definition. A binary relation R over abstract events is *happened-before compatible* if R is empty or $R(e_0, e_1)$ implies that *all* runtime correspondences $\tilde{e}_0$ of e_0 occur before *all* runtime correspondences $\tilde{e}_1$ of e_1. This means that for all possible executions $\forall c_0 \in C(e_0), \forall c_1 \in C(e_1) : c_0 < c_1$.

Relations that are happened-before compatible determine a strict temporal ordering of runtime events corresponding to abstract events. Such information is useful for the static detection of access conflicts. Unfortunately, the relations γ, ρ and τ do not provide that information.

Theorem 4.1 *γ (control flow), ρ (reference flow), τ (thread control) are not happened-before compatible.*

We give an example that violates the happened-before compatibility for each relation:

(γ) Assume the control flow relation relates events inside a loop; these abstract events can have multiple runtime counterparts, e.g., if the loop is traversed multiple times. The corresponding sets of runtime counters would violate the happened-before compatibility.

(ρ) Assume two STORE events s_1, s_2 that are related to the same LOAD event l, i.e., $\rho(s_1, l), \rho(s_2, l)$. The semantics of the program could allow the runtime event sequence $\ldots, \tilde{s}_1, \ldots, \tilde{l}, \ldots, \tilde{s}_2, \ldots$. In this case, $\rho(s_2, l)$ violates the happened-before compatibility.

(τ) Assume a sequence of START runtime events that are issued inside a loop (same abstract event s). A started thread could begin execution (issue some event b) before the sequence of runtime START events is finished. Hence there are executions in which some $\tilde{b}$ occurs before $\tilde{s}$. Hence $\tau(s, b)$ violates the happened-before compatibility. □

Our strategy is as follows: We consider regions of the OUG and relate abstract events in these regions to the remainder of events through a relation that is happened-before compatible. Regions and the resulting relations are defined according to three aspects:

- *Confinement between NEW and ESCAPE.* Abstract events that only occur during the initialization of an object are ordered with respect to abstract events that are issued after a reference to the object is made available to other threads (Section 4.3.2).
- *Ordering before START.* Abstract events that occur before all thread STARTs and are issued by a unique thread are ordered with respect to abstract events that follow some START event (Section 4.3.4).
- *Ordering after JOIN.* Abstract events that occur after all JOINs and are issued by a unique thread are ordered with respect to abstract events that precede some JOIN event. (Section 4.3.5).

4.3.2 Confinement between NEW and ESCAPE

Confinement between NEW and ESCAPE isolates access events that only occur after the allocation of an instance and before the instance is made available to other threads. Let $OUG[o] = \langle N, E \rangle$ be an OUG; o shall not stand for a class. An event in such OUG is recursively classified as *NEW-safe*:

- NEW events are NEW-safe.
- ESCAPE events are not NEW-safe because they might surrender a reference to o to other threads.
- STORE events are not NEW-safe because they make a reference to the object available in a different context in which events are not NEW-safe.
- LOAD events are not NEW-safe because events that follow the LOAD in the control flow might operate on a different instance than the one created at NEW.
- If o stands for a thread root instance then START events that control the corresponding abstract thread are not NEW-safe because the reference to the object is implicitly made available to the started thread (like ESCAPE).
- An event e_1 is NEW-safe if all its predecessors in the OUG are NEW-safe, i.e., e_1 is NEW-safe $\Rightarrow \forall e_0 \wedge \gamma(e_0, e_1) : e_0$ is NEW-safe.

Let $N_{\text{new}} \subseteq N$ be the maximal set of NEW-safe events in an OUG.

Definition. Θ_{new} is a relation among abstract events: $\Theta_{\text{new}} = \{(e_0, e_1) : e_0 \in E_{\text{new}}, e_1 \in E - E_{\text{new}}\}$.

Theorem 4.2 *Θ_{new} is happened-before compatible.*

Proof. We call a runtime event *NEW-safe* if it corresponds to an abstract event that is NEW-safe. Let $r_0, \ldots, r_i, r_{i+1}, \ldots r_n$ be a sequence of runtime events; r_0 must be a NEW event. Θ_{new} is happened-before compatible iff every sequence can be partitioned into events $r_0 \ldots r_i$ that are NEW-safe and $r_{i+1} \ldots r_n$ that are not NEW-safe. Event r_0 is NEW-safe by definition. Subsequent NEW-safe events cannot make the object accessible to threads other than the creator threads (an ESCAPE event would be necessary, but such event and its successors in the control flow are not NEW-safe). Hence events that disrupt the initial sequence of NEW-safe events must stem from the creator thread. This is not possible because (a) the NEW-safe events origin from method contexts that are connected through the control flow relation γ (a STORE/LOAD combination which would render the object accessible to another context is not NEW-safe), and (b) a region of NEW-safe events cannot be entered from a region that is not NEW-safe (all control flow predecessors of NEW-safe events must also be NEW-safe – except for NEW itself). □

4.3.3 Auxiliary acyclic ordering

This section defines an auxiliary relation Γ_{acg} on abstract events that is used to determine events that occur before START (Section 4.3.4) and after JOIN (Section 4.3.5).

An acyclic component graph (ACG) is used to elide loops from the OUG and combine events in strongly connected components (SCC). An ACG $\langle S, E \rangle$ where S is the set of SCCs and E is the set of edges (control flow relation γ) defines a relation γ_{acg} among abstract events as follows: If $s_0, s_1 \in S$ and $s_0 \rightarrow s_1 \in E$, then $\forall e_0 \in s_0 \forall e_1 \in s_1 : \gamma_{\text{acg}}(e_0, e_1)$. Let Γ_{acg} be the transitive closure of γ_{acg}.

Example. Figure 4.7 shows the ACG of the OUG of the `Table` object (o_6). Events that occur in control flow loops are combined in SCCs: SCC (4) aggregates events that occur in the initialization and start loop of the `Philo` threads. SCC (11) aggregates all events that occur in the scope of the `while` loop in method `Philo::run`.

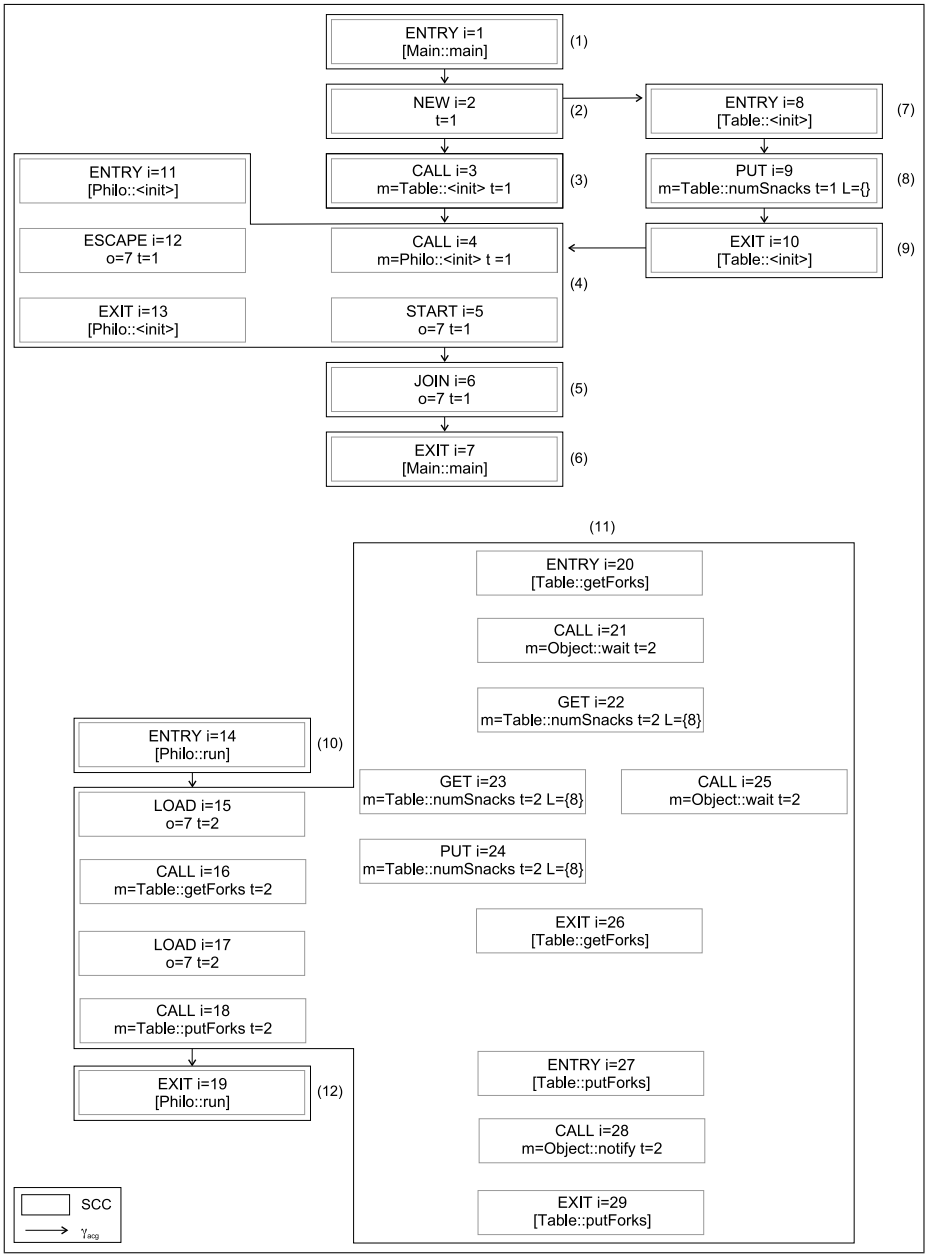

Figure 4.7: SCCs and γ_{acg} relation in the of OUG the `Table` object (o_6).

Effect of call caching. Call caching is an optimization of the symbolic execution that also affects the structure of the OUGs (Section 4.2.2). Figure 4.8 shows parts of an OUG and its partitioning into SCCs. The calls sites corresponding to events (2) and (5) have a recurring context. Figure 4.8 (a) illustrates the OUG built without call caching, in Figure 4.8 (b) call caching caused the reuse of the method subgraph for *callee*. The γ_{acg}-ordering is weakened through call caching due to larger SCCs. Although events (3) and (4) and their intermediaries

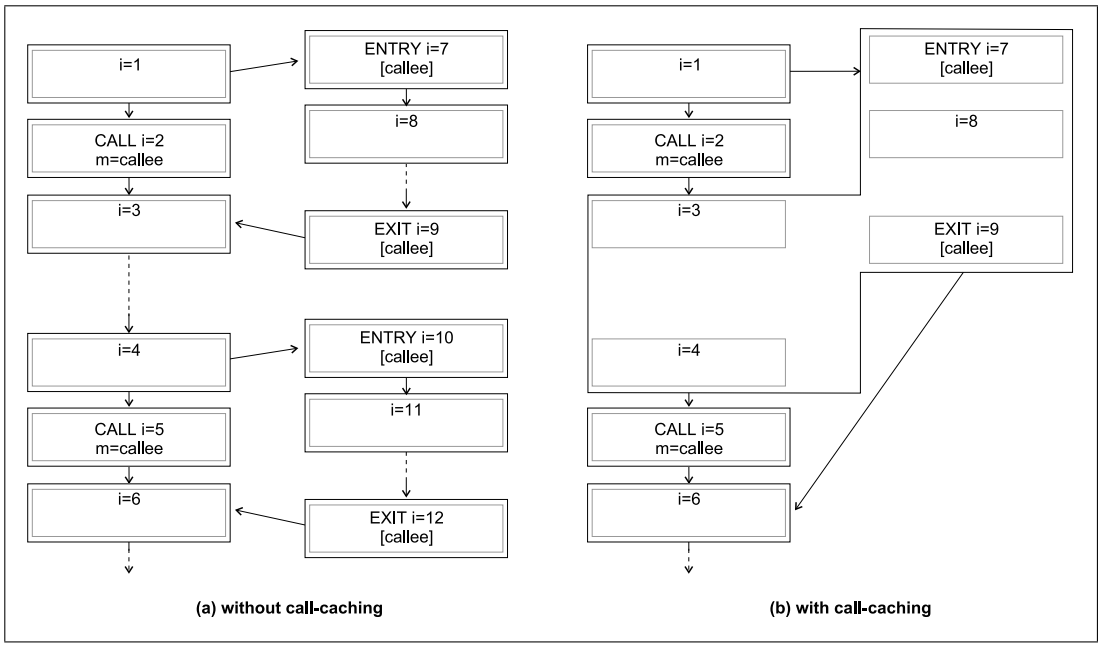

Figure 4.8: Effect of call caching on the structure of the OUG and the γ_{acg}-ordering.

are actually ordered (Figure (a)), call caching blurs this relationship and combines the events into a single SCC (Figure (b)).

Theorem 4.3 Γ_{acg} *is not happened-before compatible.*

Proof. Imagine different runtime instances of some abstract thread that operate on the same runtime object. In the general case, i.e., without explicit synchronization, the runtime events of different thread instances occur in an arbitrary interleaved sequence, although the corresponding abstract events might be related through Γ_{acg}. □

Theorem 4.4 *The restriction of* Γ_{acg} *to events of unique abstract threads is happened-before compatible.*

Proof. Γ_{acg} is an abstraction of the control flow relation γ and hence only relates events that stem from the same abstract thread. The construction of OUG guarantees that γ is cyclic for events that occur in a loop, recursion, or multiply-executed context (Section 4.2.1). Events inside such a cyclic contexts are hence not related by Γ_{acg}. Thus Γ_{acg} may relate events that occur in separate loops, recursions or contexts, i.e., if $\Gamma_{acg}(e_0, e_1)$ then there is some control flow path from e_0 to e_1 which is not cyclic. As e_0, e_1 are issued by a single runtime thread, all runtime occurrences of e_0 precede those of e_1. □

4.3.4 Ordering before START

Let $OUG[o] = \langle N, E \rangle$ be an OUG and $S \subset N$ is the set of START events in that OUG. A prerequisite to determine ordering before START is that S contains the START events of all but one unique abstract thread accessing o. Let t be that unique abstract thread for which no START

event is given (usually, t is the `main` thread). The set of *START-safe* events $N_{\text{start}} \subset N$ shall contain all abstract events e that meet the following criteria:

- e must be ordered with respect to all START events in Γ_{acg}, i.e., $\forall s \in S : \Gamma_{\text{acg}}(e,s)$.
- all events in N_{start} must be issued by the same unique abstract thread t, i.e., $\forall e \in N_{\text{start}} : e.t = t \wedge t$ is *unique*.

Definition. Θ_{start} is a relation among abstract events: $\Theta_{\text{start}} = \{(e_0, e_1) : e_0 \in N_{\text{start}}, e_1 \in N - N_{\text{start}}\}$.

Theorem 4.5 *Θ_{start} is happened-before compatible.*

Proof. If $N_{\text{start}} = \emptyset$ then Θ_{start} is empty and the claim is correct.

Assume $N_{\text{start}} \neq \emptyset$. The happened-before compatibility between events in N_{start} and events $(N - N_{\text{start}})|t$ that are issued by the same unique abstract thread t follows from Theorem 4.4.

Thus consider only the relation among events in N_{start} and $(N - N_{\text{start}})|s, s \neq t$ that are issued by some abstract thread s different from t. All START events, and in particular the START event for thread s, must have been issued by the unique thread t that also issues the events in N_{start}. Hence Theorem 4.4 is applicable and specifies that all runtime events corresponding to abstract events in N_{start} occur before all runtime START events. This ordering can be extended beyond the START event to all events issued by the started threads: According to the runtime model (Section 4.3.1), the runtime ordering of events is compatible with the thread control relation τ, and hence some START event must precede all events of the started threads. This means that all runtime events corresponding to N_{start} also happen-before $(N - N_{\text{start}})|s, s \neq t$. □

4.3.5 Ordering after JOIN

The case of ordering after JOIN is symmetric to the ordering before START. Let $OUG[o] = \langle N, E \rangle$ be an OUG and $J \subset N$ be the set of JOIN events in that OUG. A prerequisite to determine ordering after JOIN is that J contains the JOIN events for all but one unique abstract thread accessing o. Let t be that unique abstract thread for which no JOIN event is given (usually, t is the `main` thread).

In addition to the case for START, ordering after JOIN relies on the fact that at *all* runtime instances of an abstract thread are joined at a specific site (see definition thread control relation τ). We define the set of *JOIN-safe events* $N_{\text{join}} \subseteq N$ as all abstract events e that meet the following criteria:

- e must be ordered with respect to all JOIN events in Γ_{acg}, i.e., $\forall j \in J : \Gamma_{\text{acg}}(j,e)$.
- all events in N_{start} must be issued by the same unique abstract thread, i.e., $\forall e \in N_{\text{start}} : e.t = t$.

Definition. Θ_{join} is a relation among abstract events: $\Theta_{\text{join}} = \{(e_0, e_1) : e_0 \in N - N_{\text{join}}, e_1 \in N_{\text{join}}\}$.

Theorem 4.6 *Θ_{join} is happened-before compatible.*

Proof. The proof is symmetric to the happened-before compatibility of Θ_{start}. □

4.4 Conflict detection

An *access conflict* is a compile-time approximation of the runtime phenomenon *data race*: If two statements can participate in a data race at runtime, then the corresponding events in the OUG are *conflicting*. There reverse is however not true, i.e., there can be access conflicts among abstract events in the OUG that do not become manifest as a data race in any execution. The approximation is hence conservative with some restrictions that are discussed in Section 4.7.

Definition. Two access events (GET or PUT) in an OUG are *conflicting* if all of the following conditions are met:

(1) The events stem from different threads (different or non-unique user threads).

(2) At least one event is a PUT.

(3) The event target the same data (same object, same field).

(4) The static analysis cannot determine temporal ordering among the events.

Events that are not conflicting with any other event are *safe*. An abstract object is subject to an access conflict if there are conflicting events in its OUG. The information for conditions (1) and (2) can be easily obtained from OUG. Strategies for determining conditions (3) and (4) are discussed in Section 4.4.1.

4.4.1 Algorithm

The overall algorithm for detecting access conflicts is shown in Figure 4.9. Each OUG is processed separately. The procedure is structured along four aspects that are discussed in the following paragraphs.

Object exclusion. The general scope of search for access conflicts is limited to specific abstract objects. Abstract objects that are not shared have already been excluded earlier from the inspection (Section 3.3.4). Some of the shared abstract objects cannot have conflicts:

- First, abstract objects are classified as *read-only* if there is no write access from a user thread. This is the case if, e.g., all fields are final or the initialization is done in a static initializer, followed by shared read access. Read-only objects cannot have conflicts.

- Second, abstract objects can be classified as *thread-specific* (Section 3.4.3). Thread-specific objects cannot be conflicting. Abstract objects that are neither read-only nor thread-specific are called *shared r/w* in the following.

Reference confinement. The spread of references to an object determines potential usage contexts; hence confinement to a certain context narrows the scope of potentially conflicting access.

- One variant of reference confinement is the *structural* confinement that addresses, e.g., references to thread root objects: A set of access events to a user thread root object cannot conflict among each other if the accesses are issued by the user thread itself, the events occur through the `this`-reference, and the user thread does not issue a LOAD event to obtain a reference to the abstract thread object (i.e., the only reference through which such fields are accessed is obtained through the implicit argument of the `run` method). We call this form of data encapsulation *intra-thread encapsulation*.
- Another variant of reference confinement is the *temporal* confinement of a reference according to init-escape ordering. This aspect is elaborated in the following paragraph on event ordering.

Event ordering. The set of potentially conflicting access events to the same variable is narrowed in a stepwise process along the criteria for happened-before ordering (Section 4.3):

- *Thread-start.* Access events e_1, e_2 cannot conflict if $\Theta_{\text{start}}(e_1, e_2)$. START-safe events in an OUG are provided by method *start_safe*().
- *Thread-join.* Access events e_1, e_2 cannot conflict if $\Theta_{\text{join}}(e_1, e_2)$. JOIN-safe events in an OUG are provided by method *join_safe*().
- *Init-escape.* NEW-safe events in an OUG are provided by method *new_safe*().
- *Monitor protection.* Given the intersection of locksets $L = e_1.L \cap e_2.L$ and $L \neq \emptyset$; the events e_1 and e_2 cannot conflict if one of the following is true: (1) $\exists l \in L : l$ is unique, (2) e_1, e_2 access the object through the `this`-reference and their issuing methods are synchronized, or (3) the accessed object is object-specific to one of the locked objects. In all cases, locking guarantees that the corresponding runtime events $\tilde{e}_1$ and $\tilde{e}_2$ are ordered through happened-before, i.e., either $\tilde{e}_1 \rightarrow \tilde{e}_2$, or $\tilde{e}_2 \rightarrow \tilde{e}_1$.

  Three auxiliary functions are used in the algorithm in Figure 4.9 to determine monitor protection: *have_unique_lock*() checks if the intersection is not empty and if at least one common locked object is unique. *have_lock*() checks if the intersection of locksets contains a specific object. *host_object*() determines the host of some object-local object.

The conflict analysis reports the following information:

- The abstract objects that have a conflict.
- The symbols of fields that are conflicting.

- The allocation sites of conflicting objects.
- The access sites that participate in conflicts.
- For allocation and access sites: the calling context in which the conflict occurs; this information is used internally by the compiler for method specialization, Section 7.3.

Variable disambiguation. Access events (GET/PUT) can be partitioned according to the fields they target. Access to different fields cannot conflict. Variable disambiguation is complementary to alias information: Access events to the same abstract object and same field may target the same variable at runtime. Hence, all access events of an OUG are partitioned according to the accessed field and conflict analysis is done separately for every partition.

4.4.2 Examples

We continue the example of the abstract `Table` object (o_6, OUG in Figure 4.6, ACG in Figure 4.7). This object is not read-only (e.g., due to e_{24}), and not thread-specific. There are three access events to field `Table::numSnacks`: e_9, e_{23}, e_{24}. The runtime events corresponding to the abstract events are ordered: It holds $\Theta(e_9, e_5)$, hence event e_9 is ordered with respect to the thread start and consequent events issued by the started thread(s). The access events e_{23}, e_{24} are ordered with respect to each other due to monitor protection (access happens in the scope of the same unique lock). All abstract events are safe, hence the abstract object is safe.

Another shared object in the Dining Philosopher program is the `Philo` object (o_7, OUG in Figure 4.1, ACG in Figure 4.10). The abstract object is neither read-only nor thread-specific. There is one shared field `Philo::snacksEaten` that is accessed at e_{13}, e_{16}, e_{19}, and e_{20}. Event e_{13} is safe due to init-escape; hence this event occurs before the reference to the accessed object escapes (or in this case, the thread is started). Event e_{16} occurs after the join, i.e., $\Theta(e_8, e_{16})$, hence it is safe. For the remaining events e_{19} and e_{20}, intra-thread encapsulation applies. All abstract events are safe, hence the abstract object is safe.

The examples demonstrate that the conflict analysis based on OUGs is able to detect common patterns to synchronize threads and protect shared objects from unordered access: Initialization before use, thread start and join, monitor synchronization, and thread-specific data.

4.5 Experience

This section gives an evaluation of the conflict analysis on the benchmarks described in Section 3.5.2. The conflict analysis is applied to abstract objects that correspond to classes and instances. Array instances could be analyzed just like instances of classes, assuming a single abstract field that represents all slots of the array. This model is very conservative and introduces false conflict reports in the common case where different threads operate on disjoint parts of the same array. A precise value analysis for array indices could mitigate this source of imprecision. In this section, the conflict reporting focuses on classes and instances and omits arrays.

```
O = ⟨set of all shared abstract objects in the program⟩;
R = ∅;

conflict_analysis()
   ∀o ∈ O:
      if (o is thread-specific)
         // nothing to do
      elseif (OUG[o].N does not contain a PUT event)
         mark o as shared read-only;
      else
         analyze(o);
   report_conflicts(R);

analyze(o)
   acg = compute_acg(OUG[o]);
   events = ⟨all abstract events in OUG[o].N that are issued by user threads⟩;
   if (o does not stand for a class)
      events = events−new_safe(OUG[o]);
   if (events contains a complete set of START events)
      events = events−start_safe(acg, OUG[o]);
   if (events contains a complete set of JOIN events)
      events = events−join_safe(acg, OUG[o]);
   fields = ⟨set of all fields accessed in events⟩;
   ∀ f ∈ fields:
      f_events = ⟨set of all GET/PUT events in events that target f⟩;
      if (o is root of a user thread ∧ intra-thread encapsulation applies for f_events)
         // nothing to do
      elseif (f_events contains events from a non-unique or several abstract threads ∧
               f_events contains a PUT event)
         is_conflict = have_unique_lock(f_events);
         if (is_conflict ∧ all access events in f_events happen through this)
            is_conflict = have_lock(f_events, o);
         if (is_conflict ∧ o is object-specific)
            host = host_object(o);
            is_conflict = have_lock(f_events, host);
         if (is_conflict)
            R = R ∪ {new Report(o, f, f_events)};
```

Figure 4.9: Algorithm for the conflict analysis.

.

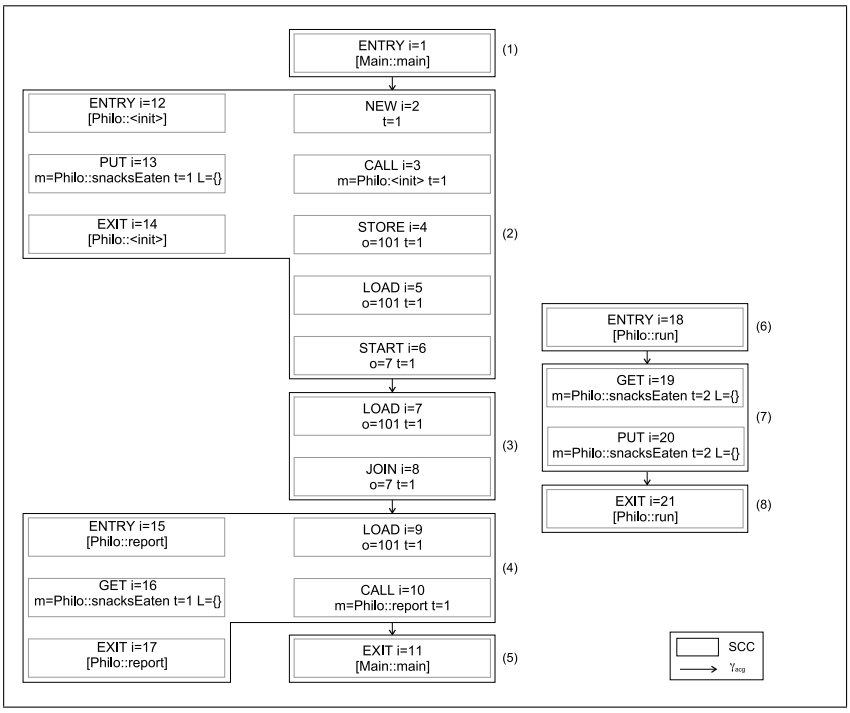

Figure 4.10: Acyclic component graph with γ_{acg} relation for the OUG of the `Philo` object (o_7).

4.5.1 Static aspects

Table 4.1 reports the duration and memory requirements of the analysis on a Pentium IV 1.4 GHz multiprocessor system. The symbolic execution takes longer than reported in Table 3.5 since OUGs are created along the processing of individual statements; similarly, the use of memory is moderately increased. The overall resource demands are acceptable, although hedc, specjbb, and jigsaw are again relatively more expensive for the reasons explained in Section 3.5.4.

Complexity The cost of the conflict analysis is related to the number and size of OUGs. For each $OUG[o] = \langle N, E \rangle$, the cost of computing the ACG is $O(|N| + |E|)$. Similarly, the relations Θ_{new}, Θ_{start}, and Θ_{join} can be obtained in $O(|N| + |E|)$ time [33] and hence the overall complexity of the conflict analysis is linear in the size of an OUG.

The actual complexity encountered during the computation of OUGs is consistent with the conceptual result: The runtime of the conflict analysis for different benchmarks follows the numbers and size of OUGs reported in Table 4.2. Column *conflict detection [k(|N|+|E|)/s]* in Table 4.1 specifies the duration of the conflict analysis relative to the overall number of nodes and edges in all OUGs. This number is almost uniform over all benchmarks. The speed of the conflict analysis for mtrt is above-average because a high proportion of OUGs can be classified as read-only early during the conflict analysis.

The number of OUGs (first column of Table 4.2) corresponds basically to the number of shared abstract objects representing classes or instances (Table 3.4); OUGs for instances of `java.lang.String` and objects for which no access events are registered, e.g., classes that have only final or volatile fields are not analyzed and not included in the statistics.

	symb exe	*conflict detection*		*mem*
	[s]	*[s]*	*[k(|N|+|E|)/s]*	*[MB]*
philo	1.4	0.5	3.4	3.5
elevator	2.2	0.8	4.7	5.0
mtrt	7.6	1.9	18.2	20.2
sor	2.0	0.5	2.7	3.6
tsp	2.5	0.8	4.7	5.8
hedc	176.6	423.3	2.1	206.2
mold	13.8	0.9	5.8	25.5
ray	2.5	0.8	7.0	6.0
monte	2.2	0.9	6.1	6.4
specjbb	77.4	168.7	2.2	133.9
jigsaw	286.6	125.3	8.5	286.5

Table 4.1: Runtime characterization of the computation of OUGs and conflict analysis.

The size of individual OUGs is generally moderate; the benchmarks hedc, specjbb, and jigsaw exhibit few very large OUGs (columns *max*) that also lead to an elevated average value of nodes and edges. Large OUGs correspond either to classes with frequently accessed fields or abstract objects that suffer from false aliasing. False aliasing occurs if the reference analysis combines unrelated alias sets that origin from different allocation sites into a single abstraction, e.g., due to conservative assumptions or the loss of context sensitivity inside a recursion. As a consequence, several objects that are actually thread or stack-local are classified as shared and their access events are registered in the corresponding OUG. Column *allocs/obj - max* in Table 4.6 shows this effect for hedc, specjbb, and jigsaw: Some abstract objects correspond to up to hundred or more allocation sites.

The cost of the conflict analysis for hedc, specjbb, and jigsaw is, similarly to the cost of the symbolic execution, relatively more expensive due to the higher number of OUGs and the very large size of some OUGs (Table 4.1).

		nodes			*edges*		
	OUGs	*max*	*avg*	*median*	*max*	*avg*	*median*
philo	10	217	66.1	41	326	104.7	55
elevator	13	327	107.7	95	505	181.7	168
mtrt	90	803	131.0	74	2074	253.3	116
sor	3	277	167.7	115	394	274.7	219
tsp	13	311	106.1	98	483	184.5	152
hedc	175	87883	1551.5	276	186542	3588.4	435
mold	16	537	129.5	87	598	197.8	148
ray	30	302	72.3	64	456	113.6	90
monte	29	229	71.4	50	532	118.4	78
specjbb	77	63329	1621.8	211	108806	3242.6	372
jigsaw	186	127698	1556.2	91	431930	4179.3	142

Table 4.2: Characteristics of OUGs

Table 4.3 shows the characteristics of OUGs for abstract objects that are *global* respectively *shared r/w*. The columns report the number of OUGs for classes and instances, the total number of allocation sites that correspond to the NEW events and the total number of access sites that correspond to GET and PUT events. The numbers serve as a point of reference to illustrate the performance of the conflict analysis in reducing the number of allocations and access sites that have to be regarded as candidates for conflicts.

	global				*shared r/w*			
	class	*inst*	*allocs*	*accs*	*class*	*inst*	*allocs*	*accs*
philo	4	5	10	93	1	5	10	88
elevator	7	27	33	317	0	9	16	142
mtrt	16	64	90	915	4	51	75	876
sor	4	5	11	218	1	2	3	146
tsp	9	17	30	401	2	5	10	278
hedc	30	131	338	1869	9	77	282	1736
mold	6	12	17	890	2	10	17	871
ray	6	25	42	364	1	19	38	315
monte	18	24	25	276	3	8	13	145
specjbb	49	94	358	2736	10	19	248	2169
jigsaw	33	128	276	1929	7	83	192	1816

Table 4.3: Characteristics of OUGs for *global* and *shared r/w* abstract objects (classes and instances, no arrays).

The conflict analysis is only done for abstract objects that are classified as *shared r/w*: Table 4.4 reports on the effectiveness of the different criteria for event ordering (Section 4.3) to prune events from the OUG that cannot participate in conflicts. The table specifies the maximum (*max*) and average (*avg*) percentage of nodes in an OUG that are ordered due to NEW, START or JOIN-safety. The exclusion of events between NEW and ESCAPE is most effective and covers in many cases all initialization events (PUT) of variables; in such cases only read events (GET) occur after the ESCAPE and the absence of a conflict can be guaranteed at compile-time.

In some benchmarks, e.g., tsp, several initialization events are identified as START-safe such that consequent reads can be considered as non-conflicting. In all OUGs of hedc, START-safety cannot be determined because *not all* START events of the accessing threads are noted in the OUGs.

The yield of JOIN-safety is the least which is partly due to the selection of benchmarks. For the scientific codes mold, ray, and monte, a fork-join pattern is recognized and some events can be identified as JOIN-safe. This is useful to determine the absence of conflicts in a scenario where worker threads update a result data structure that is read by the main thread after joining the workers. Conceptually, hedc also uses such a pattern but implements the join through Java's thread interrupt mechanism (worker threads are terminated after timeout) and hence the pattern is not recognized. philo, elevator, and specjbb do not use join, hence JOIN-safety is not an issue.

Table 4.5 details on the characteristics of those OUGs that are finally classified as *conflicting*. The columns correspond to the categories in Table 4.3. Each column specifies two values, first the absolute number of abstract objects, second the reduction of this number relative to column *shared r/w* in Table 4.3.

The results show that for most benchmarks only a small fraction of the *shared r/w* allocation and access sites are *conflicting*. Hence the heuristics used by the conflict analysis to identify ordering and to prune events and abstract objects from the set of potential conflicts are quite effective. For philo, elevator, mtrt, sor, mold and monte, type and heap shape information is precise and the used synchronization idioms match our heuristics. tsp uses a specific pattern to ensure the absence of critical interference on instances of `TourElement` that represent route information: Such objects are pre-allocated in a pool; worker threads temporarily lend objects from the pool and update them. Eventually, instances of `TourElement` are returned into the

	NEW-safe		*START-safe*		*JOIN-safe*	
	max	*avg*	*max*	*avg*	*max*	*avg*
philo	63.4	27.8	22.2	2.7	-	-
elevator	31.8	15.8	18.9	2.4	-	-
mtrt	76.9	21.3	64.8	2.0	1.9	< 0.1
sor	16.6	13.9	14.4	9.3	0.4	0.1
tsp	33.3	14.5	32.6	14.7	1.2	0.1
hedc	84.2	20.1	0.0	0.0	-	-
mold	21.1	10.3	47.2	6.1	14.9	1.9
ray	48.6	22.6	26.9	3.0	25.8	2.1
monte	38.1	17.4	40.9	3.9	33.3	2.0
specjbb	91.7	17.1	44.3	1.4	-	-
jigsaw	82.4	23.7	35.7	0.4	0.0	0.0

Table 4.4: Effectiveness of the conflict analysis in excluding abstract events from conflicts. The numbers are given as percentage of the total number of nodes in an OUG.

pool and consequently taken by other workers etc.. This form of higher-level synchronization is not recognized by the conflict detection, hence the corresponding abstract object (instance of `TourElement`), its allocation and some of its access sites are flagged as conflicting. In ray, some of the conflicting objects are actually associated with a specific instance of a non-unique thread but the static analysis cannot unravel their thread-specificness. For hedc, specjbb, and jigsaw, very large OUGs with false aliasing are classified as conflicting and hence the reduction of allocations and access events relative to *shared r/w* is not as good as for the other benchmarks.

	conflicting							
	class		*inst*		*allocs*		*accs*	
	total	*red [%]*	*total*	*red [%]*	*total*	*red [%]*	*total*	*red [%]*
philo	0	100.0	0	100.0	0	100.0	0	100.0
elevator	0	0.0	3	66.7	5	68.8	28	80.3
mtrt	0	100.0	1	98.1	1	98.7	2	99.8
sor	0	100.0	0	100.0	0	100.0	0	100.0
tsp	1	50.0	1	80.0	1	90.0	32	88.5
hedc	7	22.2	19	75.3	196	30.5	730	57.9
mold	1	50.0	0	100.0	0	100.0	2	99.8
ray	1	0.0	6	68.4	14	63.2	95	69.8
monte	1	66.6	0	100.0	0	100.0	2	98.6
specjbb	8	20.0	5	73.7	201	19.0	1670	23.0
jigsaw	6	14.3	23	72.3	101	47.4	1149	36.7

Table 4.5: Characteristics of OUGs for *conflicting* abstract objects. The columns show absolute numbers and the reduction relative to the corresponding values in column *shared r/w* in Table 4.3.

Table 4.6 refines the reports and classifies the conflicting fields, the access sites per conflicting field and the number of allocation sites per conflicting abstract object. Conflicting fields fall into one of the following three categories:

- *nr - no read:* All conflicting field access events in this category are writes (i.e., there are no reads). There are two cases: First, fields that are initialized and not used in the context of a specific benchmark; in specjbb, e.g., the private field `spec/jbb/Order::allLocal` and all (write) access events to it could be safely removed;

	fields				*accs/field*		*allocs/obj*	
	nr	*lnu*	*awl*	*other*	*max*	*avg*	*max*	*avg*
philo	0	0	0	0	0	0.0	0	0.0
elevator	0	4	1	0	8	6.0	2	1.7
mtrt	0	0	0	1	2	2.0	1	1.0
sor	0	0	0	0	0	0	0	0.0
tsp	0	0	3	0	14	10.7	1	1.0
hedc	7	15	9	96	33	5.8	161	10.6
mold	0	0	0	1	2	2.0	0	0.0
ray	0	1	0	9	13	5.5	4	2.3
monte	0	0	0	1	2	2.0	0	0.0
specjbb	19	27	42	125	79	7.9	188	40.4
jigsaw	5	35	42	85	24	6.8	62	4.5

Table 4.6: Classification of conflicts and reporting details (*nr* = no read, *lnu* = lock not unique, *awl* = all writes locked).

similarly field `w3c.jigsaw.http.Client::tstart` in jigsaw. However, this optimization is not possible in all cases because there could be uses in other contexts outside the specific benchmark (e.g., for fields of library classes). Second, there could be reads that are not conflicting, e.g., issued by a unique thread after a join; we have not encountered this case in the benchmarks.

- *lnu - lock not unique:* In this case, there exists a common abstract lock among access events to a particular field, however the analysis fails to determine the uniqueness of the locked object. In elevator, e.g., the locks protecting the data structures of individual floors are initialized in a loop and their references are stored in an array. Despite the non-uniqueness of the lock in the compile-time view, the same lock instance consistently protects data for a certain floor at runtime. One report in ray refers to an update of a global variable `JGFRayTracerBench::checksum1` in a synchronized block. The runtime object that is used for synchronization is however specific to each thread (the same abstract object at compile-time, but different instances at runtime), hence this report reflects a synchronization error in the program. Several reports for hedc, specjbb, and jigsaw are due to situations in which the analysis fails to determine the uniqueness or object-specificness (Section 4.4.1). E.g., the runtime instance reachable through the field `System::properties` is unique; the initialization happens once; the static analysis, however, determines that the call site of the initialization method is multiply executed and hence conservatively assumes that the (non-final) field `System::properties` might refer to different instances along the program execution.

- *awl - all writes locked:* In this case, all writes are lock-protected and some read is not. Two reports for elevator correspond to a potential problem in the implementation of `java.lang.Vector`: updates of field `elementData` are done under common lock protection during the initialization and resizing of the vector; the field is however read without lock protection by the implementation of an iterator associated with the vector instance. Higher-level synchronization is used to ensure that a `java.util.ConcurrentModificationException` is thrown in the critical case where `elementData` is updated during an iteration.[2] One of the reports in tsp corresponds to

[2]This feature is not implemented in the version of libgcj [49] that we used.

a global variable for the minimal tour length found so far. The updates are monotone and double checked, and concurrent reads of outdated information are tolerated by the algorithm. Hence this actual race is benign. Another report corresponds to objects that represent route information (class `TourElement`). In the actual execution, writes are ordered with respect to reads due to higher level synchronization, hence there is no actual race.

In hedc, specjbb, and jigsaw, most reports are due to a variety of patterns for lazy initialization resembling the double-checked locking [105]. The correctness of such patterns depends on type modifiers of the initialized fields (final or volatile); these modifiers control the implementation and the actual ordering of memory access at runtime [79]. Several instances of double-checked initialization in the GNU library are indeed incorrectly implemented (field variables are not declared final or volatile), such that partly initialized objects could become accessible to threads other than the creator thread.

Another report addresses the field `nextEntry` in class `java.util.HashtableEntry`; the problem is again related to an unsynchronized read through an iterator, while updates to this field through the implementation of the `java.util.Hashtable` class are synchronized.

- *other:* This case reports conflicts that do not fall into any other category, i.e., cases in which no common abstract lock object can be determined for reads and writes. In mtrt, a conflict is found on the variable `RayTracer::threadCount`, which is however not relevant to the execution of the program (this variable presumably served for debugging purpose). In mold, the conflict report on the global variable `md::interactions` is benign because the conflicting access is done only by one of multiple runtime threads with a specific id (control flow depends on thread id). In ray, some reports correspond to fields of objects that are initialized by the `main` thread and consequently associated with a specific instance of a worker thread that issues reads and writes (the objects are not however not recognized as thread-specific). The conflict analysis recognizes that events of the `main` thread do not participate in a conflict; however, the worker thread is not unique and hence the analysis conservatively assumes that the read and write events by the worker threads conflict.

  In monte, a conflict is found on a variable that flags if debug information should be printed. The conflict is benign, because the variable is always set to the same value. In hedc most reports are spurious for two reasons: Some critical objects are accessed from a non-unique user thread; at runtime however, each instance is affiliated with only one actual thread (but not recognized as thread-specific by the compiler). Moreover, some critical objects are accessed by different abstract threads without lock protection. At runtime, an ordering of these access events is guaranteed through thread start and join, however the corresponding thread control events are not safely recognized by the conflict analysis. A true report points to an unsynchronized assignment of `null` to a shared variable `Task::thread_`, which could be read by another thread and lead to a `NullPointerException`. Figure 4.1 illustrates this scenario. Note that the conflicting access statements to variable `thread_` (lines 10, 18, 19) are in different classes and hence difficult to spot through manual inspection.

Program 4.1: Harmful data race in class `Task` and subclasses (benchmark hedc).

```
1  abstract class Task implements Runnable {
2      protected Thread thread_ = null;
3      public abstract void cancel();
4      public abstract void runImpl();
5
6      public void run() {
7          try {
8              runImpl();
9          } catch(Exception e) { ... }
10         thread_ = null; // data race!
11     }
12 }
13
14
15 class TaskImpl extends Task {
16
17     public synchronized void cancel() {
18         if (thread_ != null)
19             thread_.interrupt(); // may cause NullPointerException
20     }
21
22     public void runImpl() {
23         ...
24     }
25 }
```

4.5.2 Dynamic aspects

We do not report numbers for specjbb and jigsaw because these benchmarks instantiate objects through reflection, i.e., through method `Class::newInstance`. Although recognized as as allocation sites by the static analysis, the current implementation of the program instrumentation for tracking objects and their properties at runtime does not handle such object creation sites.

First, we investigate the discrepancy between the compile-time classification of allocated objects and their runtime properties. Table 4.7 lists the number of allocated instances (no arrays and classes) that are *shared* and *conflict*ing. The compile-time classification as *shared* is done according to the algorithm in Section 3.3.4, hence *thread-specific* or *shared read-only* objects are included in the numbers. At runtime, an object becomes shared as soon as a thread different from the allocating thread accesses the object. We use a mechanism for object ownership tracking that is described in [122]. The compile-time classification of *conflicting* corresponds to the results of the conflict analysis (an object is conflicting if it has at least one conflicting field). Column *conflict - runtime* reports the number of objects that are subject to an *object race*. Object-race detection is described in Chapter 7.

The numbers illustrate that the correspondence between the static approximation and actual situation at runtime can widely differ among the benchmarks. mtrt, sor, and monte show a good match, while the compile-time classification in tsp, ray, and hedc is far off the runtime reality. There are two main reasons for this discrepancy: (1) unnecessary conservatism of the static analysis and (2) the runtime classification depends on the control flow and input of the program. While aspect (1) could be improved through a more precise

	shared		*conflict*	
	compile-time	*runtime*	*compile-time*	*runtime*
philo	7	4	0	0
elevator	41	36	31	0
mtrt	23	12	1	1
sor	3	3	2	0
tsp	10006	371	5000	162
hedc	781	143	510	28
mold	4109	6	0	0
ray	2104354	679	2103799	0
monte	20008	20005	0	0

Table 4.7: Allocation of instances with compile-time classification and actual situation at runtime (no arrays and classes).

static analysis or programmer annotations, aspect (2) is an inherent limitation of static analysis.

In ray and hedc, the discrepancy is mainly due to the first aspect (unnecessary conservatism). The thread-specific analysis (Section 3.4.3) is not successful to recognize several actually thread-specific objects in ray; in hedc, another source of imprecision is the loss of context sensitivity in the HSG due to in a large recursion that classifies a number of objects as *shared* that are actually thread-local.

In tsp, the discrepancy is due to the second aspect: A certain number of objects that represent route information are pre-allocated by the main thread. Worker threads acquire and release such objects from a pool and depending on the problem size, only few of objects in the pool may be used by some worker thread (i.e., become shared or conflicting). Hence the relation between compile-time and runtime classification as a metric for the accuracy of the shared/conflict analysis has to be taken with care.

	shared r/w	*conflict*	*conflict site*
philo	96.7	0.0	0.0
elevator	72.7	25.5	25.0
mtrt	91.4	0.0	0.0
sor	99.7	99.7	99.7
tsp	74.9	18.1	18.1
hedc	78.1	68.6	48.3
mold	99.0	0.5	0.5
ray	71.7	64.3	64.0
monte	31.7	5.4	5.4

Table 4.8: Field access according to the static classification of target objects (classes and instances, no arrays). The numbers are reported relative to access events to *global* objects (100%).

Table 4.8 reports on the number of field access events to *shared r/w* and *conflict* objects relative to the access events to *global* objects; the classification of objects is done at compile-time (Table 4.7). For benchmarks that are well amenable to the conflict analysis (philo, mtrt, sor, mold, and monte), the number of access events to *conflict* objects is only a small fraction of the access events to *shared r/w* objects. This reduction enables to implement a sparse program instrumentation for efficient, dynamic conflict detection (Chapter 7). Not all access events to conflict objects can participate in access conflicts. Column conflict site reports the number of

Program 4.2: Violation of the init-escape ordering illustrated on the example of an erroneous implementation of the Double Checked Locking idiom.

```
class DoubleChecked {
    static Data data = null;

    static void getUnique() {  // executed by "reader" threads
        if (data == null) {
            synchronized (this) {
                if (data == null)  // "double check"
                    data = new Data();
            }
        }
        use(data);
    }
}
```

executed access statements that are neither NEW-safe, nor START-safe, nor JOIN-safe. For most benchmarks, the reduction compared to conflict is not significant.

4.6 Extensions for weak memory models

The runtime model in Section 4.3 assumes a total order of events and abstracts from the runtime reality of modern compiler- and hardware systems that reorder memory accesses. This section discusses the consequences of that simplification on the validity of the three ordering relationships derived from OUGs. The conflict analysis is extended such that situations where the reordering of memory accesses could invalidate the init-escape ordering are identified.

4.6.1 Confinement between NEW and ESCAPE

The confinement of events between NEW and ESCAPE relies on the assumption that memory access happens in a total order that is consistent with the program order of individual threads More specifically, PUT events that occur before an ESCAPE according to the control flow γ are assumed to happen-before subsequent LOAD and GET events. Although such ordering between ESCAPE and preceding PUTs could be achieved with the compiler's help through a write barrier before the ESCAPE [97, Section 3.1.1], Java's memory model does not prescribe this ordering for ordinary, i.e., non-volatile or non-final, variables [79].

The Double-Checked Locking idiom [105] illustrates the problem: The idiom is commonly used to provide a one-time initialization and subsequent lock-free read access to a variable. Unfortunately, many implementations of this idiom are incorrect [97] such as the example in Program 4.2. The OUG of the `Data` instance specifies that all access events issued inside the constructor occur between NEW and ESCAPE (target variable is `data`). The conclusion that these access events are ordered with respect to subsequent reads of other threads is incorrect: Java's memory model [79] does not guarantee that the update of `data` is visible only after all updates inside the constructor of `Data` are visible to other threads. Hence some thread could obtain the a reference to the `Data` instance and come across a partly initialized object. The problem is that the variable that conveys the reference to other threads (`data`) is itself subject

of a data race. In the given implementation, (Program 4.2), two reports should be issued: one for variable `data` and another report for ordinary field variables of `Data` that are initialized in the constructor and read by other threads.

There is however a simple fix to the implementation in Program 4.2: If variable `data` was declared as volatile it would not be conflicting and subsequently init-escape protection could be assumed for access events inside the constructor of `Data`. In such case, the conflict analysis should not generate a report.

More general, ordering across an ESCAPE event is only guaranteed if the variable through which the reference escapes is *not* subject to a conflict. Abstract objects for which all ESCAPE events target non conflicting variables are *orderly escaping*. We have extended the conflict analysis to determine the *orderly escape property* of an object before confinement between NEW and ESCAPE is inferred: The conflict analysis is invoked recursively on all abstract objects that are target of an ESCAPE; cyclic dependencies in the analysis are resolved conservatively (all objects are not orderly escaping).

Note that there is a chain of logical implication among conflicts: A conflict c_1 on a reference variable v might lead to a conflict c_2 on the object which reference escapes through that variable v (conflict c_2 is due to the loss of init-escape ordering). In the original version of the conflict analysis, such "second order" conflicts c_2 might be overlooked, while c_1 would still be reported.

As an aside, note that most implementations of lock-based data race checkers that are described in literature, e.g., [103], also assume strict ordering between escape and preceding variables accesses: To account for an initialization phase during which unsynchronized access is permitted, the lockset checking is usually delayed until the first access of a thread that is different from the allocating thread. Preceding access events are not recorded in the access history and hence errors due to reordering through an unsafe ESCAPE may not be recognized.

4.6.2 Ordering before START

Java's memory model [79] defines that a START event happens-before all events of the started thread [79]. Hence the ordering of runtime events in all executions are consistent with the thread control relation τ among the corresponding abstract events and "ordering before START" protection is valid, independently of the ordering constraints for ordinary memory accesses.

4.6.3 Ordering after JOIN

Java's memory model [79] defines that all events of a thread happen before the JOIN event of that thread; the argument is symmetric to "ordering before START".

4.6.4 Experience

Table 4.9 specifies the conflict reports of the enhanced conflict analysis; we compare the results to the reporting of the standard conflict analysis in Tables 4.4, 4.5, and 4.6.

The first two columns illustrate the number of events protected through init-escape ordering relative to the values in Table 4.4. For some benchmarks, the condition of an orderly escape means a severe setback: In elevator the average number of NEW-safe events is only 27.2%

of those determined by the standard analysis. The reduction is mainly due to events in abstract objects that are reachable from the field `java/lang/Vector::elementData`. The field is subject to a conflict (an iterator reads it without synchronization) and hence init-escape confinement cannot be assumed for all objects that are reachable through that field. Consequently, the number of conflicting instances, alloc and access sites also increase for this benchmark. Although the ESCAPE is not considered as *safe* by the analysis, the implementation ensures that the deposit of a reference in a Vector and a subsequent retrieval of it through an iterator in another thread is always separated through an intervening synchronization in both threads on the `Vector` instance.

A similar situation occurs in moldyn where the number of events that are NEW-safe is reduced to 37.9% of the original value. The reason is however different: The analysis runs into a cyclic dependency (determine safety of ESCAPEs for $o_1 \rightarrow$ conflict analysis for some $o_2 \rightarrow$ relies on conflict property of o_1), such that thread-locality cannot be determined for four abstract objects which are then conservatively classified as conflicting. These objects are frequently accessed (static access sites and dynamic access counts).

For hedc, specjbb, and jigsaw, the effectiveness of the conflict analysis is also significantly reduced and there are two main reasons: First, all benchmarks exhibit large recursions in their CG which leads to a loss of context sensitivity during the construction of the HSG; consequently several nodes in the HSG are merged (false aliasing), leading to spurious cycles in the shape-graph. This introduces cyclic dependencies in the conflict analysis when the orderly escape property is determined; such dependency is handled conservatively leading to a loss of init-escape confinement for all OUGs in a cycle. Second, large OUGs that subsume multiple allocation and escape sites are more likely to loose init-escape confinement due to a number of different targets objects of ESCAPE events; the classification of a single ESCAPE as unordered leads to a loss of init-escape confinement in the overall OUG such that a large number of allocation and access sites is classified as conflicting.

	NEW-safe		*inst*		*allocs*		*accs*		*fields*	
	max (%)	*avg (%)*	*total*	*%*	*total*	*%*	*total*	*%*	*total*	*%*
philo	100.0	95.3	0	100.0	0	100.0	0	100.0	0	100.0
elevator	64.8	27.2	8	266.7	10	200.0	76	271.4	9	180.0
mtrt	100.0	76.1	1	100.0	1	100.0	2	100.0	1	100.0
sor	100.0	100.0	0	100.0	0	100.0	0	100.0	0	100.0
tsp	100.0	95.2	1	100.0	1	100.0	32	100.0	3	100.0
hedc	91.3	34.3	36	189.5	236	120.4	1070	146.6	211	166.1
mold	66.4	37.9	3	100.0	4	100.0	683	34150.0	64	6400.0
ray	100.0	79.2	7	116.7	16	114.3	98	103.2	10	100.0
monte	100.0	85.1	0	100.0	0	100.0	2	100.0	1	100.0
specjbb	65.9	15.2	16	320.0	239	118.9	1989	119.1	254	119.3
jigsaw	100.0	21.5	54	234.8	153	151.5	1636	142.4	246	147.3

Table 4.9: Conflict characteristics if confinement between NEW and ESCAPE is abandoned for objects that do not orderly escape. The percentages given are relative to the results of the standard conflict analysis in Tables 4.4, 4.5, and 4.6).

4.7 Discussion

The computation and analysis of OUGs requires whole-program knowledge and hence Java features like reflection and dynamic class loading are not accommodated.

4.7.1 Sources of unsoundness

The conflict analysis considers only object access from user threads (Section 3.2). However access from init threads can also participate in conflicts, and those conflicts are not detected by our procedure.

Our model does not consider the execution of `finalize` methods that are typically invoked by a separate *finalizer thread*.

The occurrence of a JOIN event in the OUG means that all runtime instances of the respective abstract thread are joined. The analysis uses a heuristic for determining the validity of a join but this heuristic is not sound (Section 4.1). The predicate $mustJoinThread(m)$, which determines if method m executes a join on every control flow path is defined such that an invocations of `java.lang.Thread::join` in a loop is still regarded as "must join", although the control flow may allow to bypass the loop body.

4.7.2 Sources of incompleteness

There are two main sources of incompleteness: (1) the inaccuracy of alias and type information that lead to unnecessary conservatism; (2) parallel programming patterns that are not recognized in our analysis. The first aspect has been discussed in Section 3.6.2, the second aspect is discussed in this section.

Java provides language features for synchronization, i.e., monitor-style synchronization (`synchronized` keyword, wait, notify) and thread management primitives (start, join). Our analysis is tuned to recognize the protection scope of these features in the most common cases. In some cases, however, the analysis is unnecessarily conservative and reports a potential conflict event though there is no actual synchronization defect. We give some examples for such programming patterns in the following:

Lock-free access to shared data. Program 4.3 is an efficient (lock-free) implementation of a shared variable that is frequently read and seldomly updated. The issue of this synchronization pattern is that threads that read the `data` field (line 7) never see a value that is "too old". In particular, it is never the case that `available` is seen as `true` but `data` is seen as `null`. The writer thread may exchange the object reached through field `data` and readers eventually see the new object. The correctness of this synchronization pattern (which cannot normally be automatically deduced) relies on the ordering guarantees for `volatile` variables given by the Java memory model [79]. Moreover, in a real program, class "Object" should be replaced by a class with all-final fiends. The example is taken from [70] and has been given by Doug Lea. Our conflict analysis reports a conflict on the variable `data` due to concurrent access and the absence of a unique enclosing lock.

Program 4.3: Example for synchronization through a volatile variable.

```
1  class VolatileSync {
2      volatile boolean available = false;
3      Object data = null;
4
5      void f() { // executed by "reader" threads
6          if (available)
7              use(data);
8      }
9
10     void g() { // executed by a "writer" thread
11         while (true) {
12             if (occasionallyTrue) {
13                 data = new Data();
14                 available = true;
15             }
16         }
17     }
18 }
```

Program 4.4: Example for synchronization based on a semaphore.

```
1  class Semaphore {
2      boolean occupied;
3
4      synchronized void p() {
5          while (occupied)
6              wait();
7          occupied = true;
8      }
9
10     synchronized v() {
11         occupied = false;
12         notify();
13     }
14 }
15
16
17 class Example2 extends Thread {
18     static Semaphore sem;
19     static Data data;
20
21     static main(String args[]) {
22         sem = new Semaphore();
23         data = new Data();
24         for (int i = 0; i < 2; ++i)
25             new Example2().start();
26     }
27
28     void run() {
29         sem.p();
30         data. ...; // operate on data
31         sem.v();
32     }
33 }
```

Synchronization with semaphores. Program 4.4 shows the use of a semaphore for mutual exclusive access to the shared data reachable through field `data`. According to the definition of an *access conflict* (Section 4.4), the conflict analysis classifies the access statement in line 30 as conflicting (no lock held during access).

Shared object pools. Another pattern that is not recognized by our concurrency analysis are shared object pools: Threads temporarily borrow objects from the pool; references to such objects are guaranteed to be available only to one thread at a time. This pattern is used in `tsp` and `hedc`; both cases are handled conservatively by our analysis such that the respective pooled objects are considered and reported as conflicting.

Summarizing, we found that synchronization patterns like the above examples that deviate significantly from Java's monitor-style `synchronized` mechanism are sparsely used in Java applications. Their correct implementation requires most diligent programming expertise and an in-depth knowledge of the Java memory model. Some effort to make efficient synchronization patterns and thread-safe data structures available in the standard library is underway [74]; specific library classes and their correct uses could be incorporated in the static analysis presented here to reduce spurious reporting for such data structures.

4.7.3 Related work

We discuss related work in the field of the parallel program analysis only as far as it relates to determining the structure of concurrency, synchronization, and the interference on shared data. Extensive research is done on the analysis and optimization of parallel (shared memory) programs that assumes that such information is known or can be easily determined from the program (structured multi-threading); this research is not described here.

Bristow et al. [17] use an inter-process precedence graph for determining anomalies in programs with post-wait synchronization. Taylor [118] and Duesterwald and Soffa [37] extend this work and define a model for parallel tasks in Ada programs with rendez-vous synchronization. The program representation in [37] is modular and allows to efficiently analyze programs with procedures and recursion based on a data flow framework. Masticola and Ryder [81] generalize and improve the approach of [37] and provide experimental evidence of the effectiveness of their technique. OUGs borrow ideas from the program representations proposed in [17, 37], but OUGs implicitly focus on those parts of the program that access the abstract object modeled by the OUG. Hence, OUGs account for an object-oriented model of memory, threads, and locks and allow to tailor the analysis of methods to different object contexts (Section 4.1).

Naumovich et al. [89] present the MHP (may happen in parallel) analysis that computes potential concurrency among statements in multi-threaded Java programs. The authors show that the precision of their data flow algorithm is optimal for most of the small applications that have been evaluated; medium to large sized benchmark programs have not been studied. The approach requires that the number of real threads in the system is specified as input to the analysis; the handling of recursion is not described in the paper. MHP analysis does not distinguish statements according to their execution context and the accessed data. The combination of MHP information with a model of program data (heap shape and reference information) could be used to determine conflicting data accesses. This approach is discussed by Midkiff et al. [84] but no compiler implementation results are yet available. OUGs naturally provide such an integration

of control and data flow information in context-sensitive manner.

Mellor-Crummey [83] developed an inter-procedural data dependence analysis for Fortran programs with parallel loops. The analysis determines protection guarantees for certain data that are accessed by one thread (loop iteration) at a time. The results of this analysis are used to reduce the instrumentation for a dynamic race checker.

The static analysis of Emrath and Padua [40] and similarly the analysis of Balasundaram and Kennedy [11] determine the interaction of program statements or blocks with shared data in the presence of post-wait synchronization. The novel aspect of this work is that the program model addresses program statements and data. Both analyses can determine the ordering of accesses from different threads to the same data even for complex parallel loop constructs with nested synchronization. The approaches assume a static data model of variables and arrays; the thread structure corresponds to parallel loops or is explicitly given. Our approach targets object-oriented programs that commonly have unstructured threading and dynamic data structures that do not easily map into the abstractions used by [40, 11].

In object-oriented programs, access to shared data is typically done indirectly through references. In such an environment, program analysis faces more difficulties to determine the relation between threads and the data they access than in languages with limited pointer usage (e.g., Fortran). Flow-sensitive pointer analysis has been extended to multi-threaded programs by Rugina and Rinard [101]. Their algorithm for Cilk programs explicitly models the interference between parallel sections in the program and the additional aliasing created through this interference. Unlike Cilk, other object-oriented languages model concurrent activities themselves as objects, and hence the scope of concurrency is not limited to a static program scope (unstructured concurrency). A starting point for the optimization of such programs is to determine the locality of objects with respect to allocating threads. Based on different variants of pointer analyses, escape analyses have been developed for Java, e.g., [13, 15, 24, 128]. OUGs extend this work and model locking, threads- and their interaction with objects explicitly.

Choi et. al. [25] use an inter-thread call graph, which extends an ordinary call graph to include edges for thread start (but not for thread join). Synchronized blocks are modeled explicitly by approximating locks held at block boundaries. The inter-thread call graph contains one node per method and hence, unlike OUGs, does not distinguish method executions in different thread and heap contexts. In [26], the authors describe a more sophisticated, path-sensitive static data race analysis. Potential data races are determined along a complex data flow analysis that approximates lock protection and statement ordering. While the goals of their analysis and our static data race detection in Chapter 4 are the same, the algorithms and data structures are different. One of the important differences between [26] and our analysis is the notion of context in the context-sensitive analysis (see Section 3.6.3).

Other representations of parallel programs have been designed with specific optimizations in mind. Diniz and Rinard [35], e.g., focuses on the movement and elimination of lock operations in automatically parallelized object-based programs. Their analysis is based on an inter-procedural control flow graph of all threads. This graph models the structure of locking and protected program scopes. Data are modeled as read and write sets that are attributed to the individual nodes of the graph. The transformations require that the program is free of data races, a condition met by automatically parallelized programs. Unlike OUGs, protection of data against concurrent access is only determined according to lock protection and a known relation between locks and protected code regions. Polymorphism and different execution contexts of

methods make it difficult to infer such a relation in typical object-oriented programs. OUGs, in contrast, compute a relation between data and their protecting locks.

Warlock is a tool [112] that implements a purely static approach to determine lock-based data races in multi-threaded C programs. The tool basically traces the execution of every path through a program and notes which locks are held each time a variable is accessed. Warlock is an interactive tool that is guided by program annotations and user input to reduce the number of false reports.

All aforementioned approaches to relate threads and accessed data, including OUGs, are based on data and control flow information. Recent work on type systems has shown that data protection and locking policies can be codified in data and method declarations that are checked statically. The main advantage of this approach is its modularity, which makes it, in contrast to a whole program analysis, well amenable to treat incomplete or large programs. Unlike OUGs, the application of these approaches to existing programs is not without difficulties: The type systems have either been proposed as extensions to existing programming languages [9, 16], or as annotations [41]. Flanagan and Freund [41] present a type system that is able to specify and check lock-protection of individual variables. In combination with an annotation [42] generator, they applied the type checker to Java programs of up to 450 KLOC. The annotation generator is able to recognize common locking patterns and further uses heuristics to classify as benign certain accesses without lock protection. The heuristics are effective in reducing the number of spurious warnings; some are however unsound (but this property has not been a problem for the benchmarks investigated in [42]). OUGs model the reachability of objects explicitly and recognize cases of isolation beyond lock-protection that are covered by the heuristics automatically. The number of clustered warnings generated per KLOC is of the same magnitude as the reports we obtain for our benchmarks.

5

Static detection of atomicity violations

This chapter investigates situations where access to shared variables is ordered through critical sections, i.e., there are no data races. However, critical sections can be too fine grain and may not cover at once an entire sequence of actions that a programmer intended to happen without interference. The technique described in this chapter presumes that regions of activity without interference correspond to methods. We present an algorithm that detects those methods that could violate this presumption, i.e., methods that can appear to execute non-atomically. The algorithm is based on the abstractions and the whole program analysis that are described in Chapter 3.

Example Consider the example of a bank account in Figure 5.1: method `update` is invoked by several concurrent `Update` threads. The shared variable `balance` is accessed under common lock protection and hence there is no data race. The structure of locking specifies that the lock associated with the `Account` instance protects *either* a read *or* a write of field `balance`. Method `update` applies this synchronization discipline, however it performs a read *and* a write and hence cannot be atomic.

The example illustrates a common pattern of software defects where one thread first queries the state of some shared data structure and then relies on this information during the further execution while being oblivious to the fact that other concurrent threads could have changed the data in the meantime. More generally, the interaction sequence of the thread with the data structure is not atomic, hence a *violation of atomicity* occurred.

Our strategy to determine violations of atomicity is as follows: First an object access discipline is defined (Section 5.1) that, if violated, allows to conclude on common patterns of atomicity violations at the method level. Then, an algorithm is presented (Section 5.2) that determines violations of this access discipline and hence potential violations of atomicity at compile-time.

5.1 Method consistency

Method consistency (MC) specifies an access discipline for shared variables. The access discipline is determined from the access behavior of methods and the usage of locks. MC adopts concepts from *view consistency* [8] and extends it to accommodate the scope of methods as consistency criterion. A violation of MC indicates that the corresponding method may not execute atomically, i.e., there there could be interference by other threads on shared data that are accessed in the scope of the method. The rationale of MC is to conjecture atomic treatment for

Program 5.1: Example of an `Account` class with non-atomic `update` method.

```
class Account {
    int balance;

    synchronized int read() {
        return balance;
    }

    void update(int a) {
        int tmp = read();
        synchronized(this) {
            balance = tmp + a;
        }
    }
}


class Update extends Thread {
    static Account acc;

    static void main(String args[]) {
        acc = new Account();
        new Update().start();
        new Update().start();
    }

    void run() {
        acc.update(123);
    }
}
```

a set of shared variables that are accessed in the dynamic scope of a method (*method view*). The execution of a method is atomic if there are no concurrent updates of variables in its method view.

The activities of threads are modeled by *lock views*. A *lock view* is a set of $\langle$*variable*, *access*$\rangle$ pairs that correspond to variable access events of a thread t in the dynamic scope of a lock. The *access* component specifies if the variable is read (r) or updated (u), i.e., written, or read and written. There is at most one entry per variable in a lock view. The set of lock views of a thread t is specified as $L_t = \{l_0, \ldots, l_n\}$ where each l_i stands for an access $\langle v_i, a_i \rangle$.

Example For the program in Figure 5.1, lock views correspond to the fields accessed inside the synchronized method `read` and the synchronized block in method `update`: $L_{\texttt{Update}} = \{l_{\texttt{read}}, l_{\langle syncblock \rangle}\} = \{\{\langle \texttt{balance}, \text{r} \rangle\}, \{\langle \texttt{balance}, \text{u} \rangle\}\}$.

A *method view* models the conjecture about sets of variables that should be treated atomically. There is one method view m_i for each method i; this method view contains two entries per accessed variable, namely a read and an update entry (there are always both entries, irrespective the kind of access performed by the method). The set of method views of a thread t is $M_t = \{m_0, \ldots, m_n\}$.

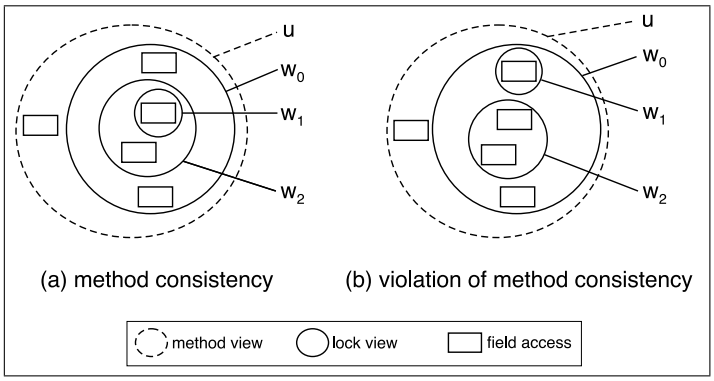

Figure 5.1: Illustration of method consistency.

Example The method views for the `Update` threads in the example are[1] $M_{\texttt{Update}} = \{m_{\texttt{run}}, m_{\texttt{update}}\}$ where $m_{\texttt{run}} = \{\langle \texttt{balance}, \text{r}\rangle, \langle \texttt{balance}, \text{u}\rangle, \langle \texttt{acc}, \text{r}\rangle, \langle \texttt{acc}, \text{u}\rangle\}$ and $m_{\texttt{update}} = \{\langle \texttt{balance}, \text{r}\rangle, \langle \texttt{balance}, \text{u}\rangle\}$.

Definition We need two concepts to define method consistency (MC):

- *View overlap.* Two views u_i and u_j *overlap* if their intersection is not empty, i.e., $u_i \cap u_j \neq \emptyset$.
- *Chain property.* A set of views $\{u_0, \ldots, u_n\}$ *forms a chain* with respect to a view u if the set contains only a single element or for all pairs of non-empty views $w_i = u \cap u_i$, $w_j = u \cap u_j$, where at least one $u_{i,j}$ originates from a thread that is concurrent to the originating thread of u, it holds $(w_i \subseteq w_j) \vee (w_j \subseteq w_i)$.

MC is given if for all methods i, the lock views that overlap the method view m_i form a chain. The concept of overlap serves to filter out irrelevant variables. The chain property detects lock usage scenarios that are susceptible to atomicity violations: e.g., a lock protects *either* reads *or* updates of one variable or a lock protects different but overlapping sets of variables (see high-level data races [8]). If the chain property does not hold for a given method i and a set of overlapping lock views, MC is violated in method i and a potential violation of atomicity in i is detected.

Figure 5.1 illustrates the definition of method consistency using a method view u, lock views w_0, w_1, w_2, and a number of events that stand for runtime occurrences of read and write access to shared objects. In part (a) of the figure the lock views form a chain, i.e., they are nested. Part (b) of Figure 5.1 shows a scenario where the lock views do not form a chain (w_1 and w_2 are not nested), hence method consistency is violated.

Example In the example, there are concurrent threads $t_{\texttt{Update}}$ with method views $m_{\texttt{run}}, m_{\texttt{update}}$, and $m_{\texttt{read}}$. All lock views in $L_{\texttt{Update}} = \{l_{\texttt{read}}, l_{<\texttt{syncblock}>}\}$ overlap with $m_{\texttt{run}}$ and $m_{\texttt{update}}$ but they do not form a chain; hence MC is violated for the methods `run` and `update`. Figure 5.2 illustrates this situation.

[1] We omit the method view for `read` because this method does not execute subordinate locking and does not call `java.lang.Object::wait`; provided that shared data access is synchronized through monitors, this method is not a candidate for an atomicity violation (details in Section 5.2.2).

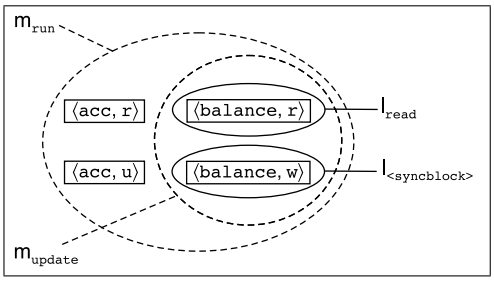

Figure 5.2: Method view and lock views for Program 5.1.

Program 5.2: Class `Account` with atomic `update` method.

```
class Account {
    int balance;

    synchronized int read() {
        return balance;
    }

    synchronized void update(int a) {
        int tmp = read();
        balance = tmp + a;
    }
}
```

Program 5.2 shows the corrected implementation of class `Account` where method `update` is atomic. If this implementation is used in the context of the `Update` threads in Program 5.1, method consistency exists: There is a single lock view in this program corresponding to method `update`, i.e., $L_{\texttt{Update}} = \{l_{\texttt{update}}\} = \{\{\texttt{balance/u}\}\}$; method `read` is only called in the scope of `update`, hence locking is reentrant in this invocation context of `read` and the analysis does not register a lock view. The lock view $l_{\texttt{update}}$ overlaps with the method view $m_{\texttt{update}}$ and, as there is only a single overlapping lock view, trivially forms a chain. Figure 5.3 illustrates this situation.

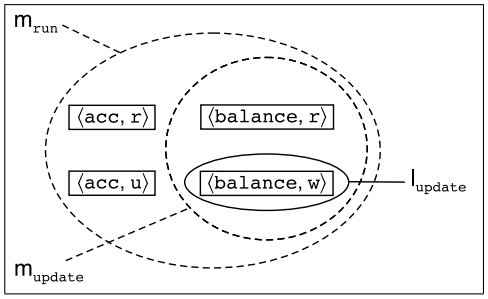

Figure 5.3: Method view and lock views for the `Account` class with atomic `update` method (Program 5.2).

5.2 Algorithm

MC is a property of a program execution. In this section, an algorithm is presented that determines *potential* violations of MC through static program analysis. The algorithm is based on abstract domains of access events and views, like the static conflict detection presented in Chapter 4. When there is no risk of confusion, we just refer to "MC violations" and "views", instead of "potential MC violations" and "abstract views". Violations of MC are determined in two steps: First, abstract views are computed (Section 5.2.1); then violations of method consistency are determined and reported (Section 5.2.2).

5.2.1 Computation of views

Views are computed along the symbolic execution according to the transfer rules in Figure 5.4. The execution maintains a stack of lock views (*LV*) and a stack of method views (*MV*) that keep track of the method and lock scopes that are *active*, i.e., on the simulated runtime stack at a certain point during the symbolic execution.

Field access to shared abstract objects is recorded in the views: Lock views contain *either* a read entry *or* an update entry (update overrides read), depending on whether the variable is read or updated in the scope of the corresponding lock. A read *and* update entry is added to all active method views regardless of the kind of access. This means that method views always contain pairs of entries for each variable (one for access type read and one for update) such that overlap with a lock view can be determined through intersection (the intersection is not empty if the lock view operates on a variable that is common with an entry in the method view).

Access events that occur during the initialization of a shared object cannot participate in inter-thread interference that leads to a violation of atomicity. Hence we chose that views should not account for access through the `this`-reference in the scope of a constructor and for access in the scope of initializer methods. This convention is practical to reduce the number of spurious reports but entails the potential of underreporting (Section 5.4.1).[2] Access to final and volatile fields is omitted from the views.

The elements of the view set specify variables in terms of fields. Hence these transfer rules do not exploit the availability of precise heap context information during the symbolic execution: The same field in different abstract objects is considered as the same variable (*field-based* analysis [127]). A more precise variable disambiguation would be possible, however there are several reasons that justify the simpler variant we present here:

- The abstraction errs on the conservative side because access events that *may* target the same instance at runtime target the same variable abstraction in the static analysis (one abstract object per class).
- In the benchmarks at hand, most of the shared abstract objects are accessed by the same code in equivalent contexts, i.e., the lock and method views would be the same for different instances. Hence the overreporting introduced through the omission of the heap context information is minor.

[2] This heuristic approximates the NEW-safe property (Section 4.3.2) of access statements; as access events are collected during the symbolic execution, OUGs and the Θ_{new} relation are not available at the stage of the whole program analysis.

Variables and domains		**Analysis state**	
$m \in \mathcal{M}$	methods	$CAS : \mathcal{C} \to O$	alias set lookup for classes
$c \in \mathcal{C}$	classes	$AS : \mathcal{V} \to O$	alias set lookup for local vars
$f \in \mathcal{F}$	fields	$MV : 2^{\mathcal{U}}$	stack of current method views
$v \in \mathcal{V}$	local variables	$LV : 2^{\mathcal{U}}$	stack of current lock views
$o \in O$	alias sets		
$t \in \mathcal{T}$	abstract threads		
$u, w \in \mathcal{U}$	views		

Statement	**Action**
$v = c.f$	`if` $(CAS(c).p$ is *shared*$)$ $\quad \forall u \in MV : u = u \cup \{\langle f,\mathrm{r}\rangle, \langle f,\mathrm{u}\rangle\};$ $\quad \forall u \in LV :$ `if` $(\langle f,\mathrm{u}\rangle \notin u)$ $\quad\quad u = u \cup \{\langle f,\mathrm{r}\rangle\};$
$v_0 = v_1.f$	`if` $(AS(v_1).p$ is *shared*$)$ $\quad \forall u \in MV : u = u \cup \{\langle f,\mathrm{r}\rangle, \langle f,\mathrm{u}\rangle\};$ $\quad \forall u \in LV :$ `if` $(\langle f,\mathrm{u}\rangle \notin u)$ $\quad\quad u = u \cup \{\langle f,\mathrm{r}\rangle\};$
$c.f = v$	`if` $(CAS(c).p$ is *shared*$)$ $\quad \forall u \in MV : u = u \cup \{\langle f,\mathrm{r}\rangle, \langle f,\mathrm{u}\rangle\};$ $\quad \forall u \in LV : u = (u \cup \{\langle f,\mathrm{u}\rangle\}) - \{\langle f,\mathrm{r}\rangle\};$
$v_1.f = v_0$	`if` $(AS(v_1).p$ is *shared*$)$ $\quad \forall u \in MV : u = u \cup \{\langle f,\mathrm{r}\rangle, \langle f,\mathrm{u}\rangle\};$ $\quad \forall u \in LV : u = (u \cup \{\langle f,\mathrm{u}\rangle\}) - \{\langle f,\mathrm{r}\rangle\};$
$v = m(v_0, \ldots, v_n)$	$MV.push($`new` $MethodView(m,t))$ `if` $(m$ is `synchronized`$)$ $\quad LV.push($`new` $LockView(t));$ $process(m);$ `if` $(m$ is `synchronized`$)$ $\quad LV.pop();$ $MV.pop();$
`monitorenter` v	$LV.push($`new` $LockView(t))$
`monitorexit` v	$LV.pop()$

Figure 5.4: Transfer rules for the computation of method and lock views along the symbolic execution; t is the current abstract thread.

- The implementation of the consistency checker and the reporting are simplified.

The analysis does not register lock views for lock operations that are found to be reentrant, i.e., an acquire operation on a lock that is already taken is ignored.

```
M = ⟨all method views⟩;
L = ⟨all lock views⟩;
R = ∅;

method_consistency_analysis()

  /* phase 1: narrow views */
  readonly_fields = { f : ∄ u ∈ L : ⟨f, u⟩ ∈ u };
  ∀u ∈ L ∪ M:
    if (⟨f, _⟩ ∈ u ∧ f ∈ readonly_fields)
      u = u − {⟨f, u⟩, ⟨f, r⟩};
  ∀u ∈ M:
    if (meth(u) does not have subordinate locking)
      M = M − {u};
  partition_views();

  /* phase 2: assess method consistency */
  ∀u ∈ M:
    olv = overlap(u, L);
    ilv = check_chain(olv, thread(u));
    if (ilv ≠ ∅)
      R = R ∪ {new Report(meth(u), class(u), u, ilv)};

  /* phase 3: aggregate reporting */
  ∀r ∈ R:
    if (∃s ∈ R − {r} : s.fields = r.fields ∧
                       s.meth is above r.meth in the caller hierarchy)
      R = R − {s};
```

Figure 5.5: Algorithm for determining violations of MC.

.

5.2.2 Violations of method consistency

Figure 5.5 shows the algorithm that determines potential violations of MC. Input to the algorithm are sets of method views and lock views M and L that have been determined during the symbolic execution. The result of the algorithm is the set of violation reports R. The auxiliary functions *meth*(), *class*(), and *thread*() take a view as argument and return the method (if view corresponds to a method), the class (views are partitioned according the affiliation of fields with classes, see below), and the thread that exhibits the view.

The algorithm proceeds in three phases:

1. Views are reduced and partitioned to limit the scope of search for subsequent phases: Fields that are *shared read-only* (heuristic used here: fields that are only assigned through `this` in the constructor) cannot bear interference in the form of atomicity violations. In addition, views of methods that do not execute synchronization actions in their dynamic scope are pruned from M because they are atomic, provided that the program is free from data races. The operation *partition_views*() partitions views according to the affiliation of fields with classes. This means that the overlap and chain properties are determined only among field variables that belong to the same class. This strategy is justified because object-oriented design typically imposes consistency constraints on variables of the same class; moreover, spurious reports due to a violations of the chain property for unrelated variables are omitted. We discuss the effect of this partitioning of views on the reporting further in Section 5.4.3.

2. Method consistency is assessed among the method and lock views according to the overlap and chain criteria. Method *check_chain*() returns a set of potentially interfering lock views $ilv \subseteq L$ that violate the chain property with respect to a method view u.

3. The reporting is aggregated in the third phase of the algorithm. Assume some method m is found to violate MC; naturally, all callers of m will also violate MC. Hence, for a specific violation only the lowermost method in the caller hierarchy is reported.

A report of a potential violation of MC specifies the following information:

- *meth*: the method that exhibits this view.
- *fields*: the field variables in this view.
- *class*: the class to which the fields in this view belong to.
- *olv*: set of overlapping lock views (locked object or method and the set of field variables that cause interference).

5.3 Experience

First, we verify if the analysis is capable do detect known violations of atomicity that correspond to synchronization defects. Our system determines atomicity violations that correspond to the scenarios of the `Account`, `java.lang.StringBuffer`, `java.util.Vector` and `java.io.PrintWriter` classes in [45]. Moreover, non atomicity in the use of iterators for common collection classes like `java.util.Vector` and `java.util.Hashtable` that are discussed in [126] are detected. These classes provide explicit means to determine actual violations of atomicity at runtime (`java.util.ConcurrentModificationException`). We have also successfully checked several scenarios with high-level data races, e.g., the `Coordinate` example in [8]. Second, we look at several benchmark programs (Section 3.5.2) and determine potential violations of atomicity at the application scope.

I/O facilities are commonly shared among threads and interaction sequences of individual threads are usually not atomic. We omit the reporting of violations of method consistency

	time	*mem*	*reports*		*methods*			
			app	*lib*	*app*		*lib*	
	[s]	*[MB]*	*false–spurious–benign–harmful*		*reports*	*total*	*reports*	*total*
philo	0.1	1	0	0	0	4	0	24
elevator	0.2	3	0–0–1–0	0–1–0–0	4	16	2	32
mtrt	1.1	5	0	0–3–0–0	0	43	3	207
sor	0.1	1	0	0	0	5	0	2
tsp	0.2	1	0–1–0–0	0	1	14	0	27
hedc	19.9	58	0–2–3–0	3–5–2–0	10	141	3	366
mold	0.3	2	0–1–0–0	0–2–0–0	6	28	0	24
ray	0.3	3	0–1–0–0	0–2–0–0	3	45	0	26
monte	0.3	3	0	0–1–0–0	2	71	0	38
specjbb	23.9	30	0–17–3–0	1–3–0–0	19	472	5	317
jigsaw	18.4	34	0–19–2–2	1–3–0–0	17	474	3	276

Table 5.1: Analysis characteristics and reports of atomicity violations.

related to I/O library classes because non-atomicity is the natural behavior of these classes that users expect. Moreover, our current implementation does not account for array access and hence atomicity violations that are due to thread interference on shared arrays may not be reported.

The first columns in Table 5.1 show the execution times of the MC analysis and the memory requirements of the static analysis on a Pentium IV 1.4 GHz multiprocessor system. Overall, the analysis is practical for the reported programs and the resource requirements are moderate.

Further columns in Table 5.1 characterize violations of MC that we found. Column *reports* specifies the number of method views that are found to be inconsistent with lock views. We report only the smallest method views that still exhibit violations; method views that are a supersets of those reported would exhibit the same violations but would make it more difficult to identify the cause of the report. If interference is due to field variables that belong to the library classes, numbers are reported in category *lib*, otherwise in category *app*.

The assessment of individual reports can be difficult and requires precise information about the synchronization discipline for the affected shared data structures. We use the following classification schema as a guidance (column *reports*: *false–spurious–benign–harmful*):

- *False reports* are due to the imprecision of the static analysis (e.g., if data is not shared but actually thread-local).
- *Spurious reports* specify that violations of atomicity do not occur at runtime in the given usage context of a data structure due to higher level synchronization (e.g., through a protected encapsulating object or thread start/join; see also Program 5.5).
- *Benign reports* refer to situations where an atomicity violation at the method level is possible. Such situations are not uncommon and do not necessarily represent a synchronization fault. This is especially true for methods that are invoked at a high level in the caller hierarchy of a multi-threaded application with shared data. An exemplary situation where non-atomicity is desirable are methods that call `java.lang.Object::wait`: The execution of this method suspends the current thread, expecting that other threads change a shared (condition) variable and signal the state change such the current thread can continue execution. More general, any explicit inter-thread communication through shared variables will lead to a violation of atomicity.

- *Harmful reports* mean that a violation of atomicity can occur that can lead to unintended runtime behavior.

Columns *methods* in Table 5.1 specifies the number of methods reported by the checker (*reports*) and the overall number of methods for which a view is registered (*total*). The reports contain only methods that (1) access variables in at least one subordinate lock view (view v_1 is subordinate to v_2 if v_1 occurs in the scope of v_2), and (2) that are at the lowest levels of the caller hierarchy. Aspect (1) suppresses reports of methods that do not use synchronization during their execution but exhibit a method view that is conflicting with lock views. For those methods, we report their callers (one of those will make use of synchronization because we assume that there are no data races). Aspect (2) excludes the reporting of all callers of a method for which a potential violation is determined (a method that calls a non-atomic method is not atomic either). If a method belongs to a library class, it is reported in category *lib*, else *app*.

Most of the smaller benchmarks share data in arrays, hence there are few or no classes that we consider for reporting. In elevator, there is one benign report for a shared data structure that represents the state of the simulated system and is repeatedly accessed by the top-level methods of the simulator threads. A spurious report concerns an instance of class `java.util.Vector` that is however used such that no concurrent modification can occur.

mtrt exhibits three spurious reports that concern some instances of `java.util.Vector` and `java.util.Hashtable` used in the library; these data structures are initialized once and then read (the scenario is similar to Program 5.5). tsp has one spurious report due to a lock scope that violates the chain property but actually executes without concurrency during the initialization of the program.

In hedc, three reports are false and correspond to execution scenarios that the compiler conservatively assumed due to imprecise type information. Similar to mtrt, several reports are spurious on shared collection classes where initialization and subsequent shared read are ordered. Some reports are benign, e.g., for variables that are used to communicate information between worker and controller thread; another benign report addresses methods that perform subsequent access to a shared thread pool.

In specjbb, 10 reports correspond to instances that represent database records, where fields are accessed independently and atomicity is only necessary at the level of individual fields, or explicitly ensured by the transaction logic that is implemented at the application level. Depending on the correctness criteria at the application level, these reports can be classified as spurious or benign. Three reports are benign and concern shared data containers that hold database records.

We discuss two interesting reports for jigsaw. The first report addresses class `w3c.jigsaw.http.ClientState` that represents an element of a linked list of client connections. Its fields `prev` and `next` link the structure and are accessed independently from fields `idle` and `client` (lock views are disjoint). All fields are cleared when a connection is removed from the pool and hence the fields are combined to a method view, leading to a report that does not reflect a problem in the program. The second report concerns class `w3c.tools.store.ResourceStoreManager` shown in Program 5.3. Method `shutdown` intends to remove all entries from the store (map referenced through field `entries`) and prevent further insertions by setting the latch `closed`. Atomicity is violated for method `loadResourceStore` (the sequence `checkClosed`

Program 5.3: Violation of atomicity in class `w3c.tools.store.ResourceStoreManager` in jigsaw.

```
class ResourceStoreManager {
    boolean closed = false;
    Map entries = new HashMap();

    synchronized void checkClosed() {
        if (closed)
            throw new RuntimeException();
    }

    ResourceStore loadResourceStore(..) {
        checkClosed();
        StoreEntry se = lookupEntry(...);
        return se.getStore();
    }

    synchronized Entry lookupEntry(...) {
        Entry e = (Entry) entries.get(..);
        if (e == null) {
            e = new Entry();
            entries.put(..., e);
        }
        return e;
    }

    synchronized void shutdown() {
        while (...) {
            // remove all entries
        }
        closed = true;
    }
}
```

and `lookupEntry` is not atomic). An unfortunate schedule can lead to the situation that entries are added to a resource manager after method `shutdown` has executed.

The benchmarks mold and ray share part of the code and both report violation for lock views with disjoint variable sets on class `jgfutil.JGFTimer`. There is indeed a notion of consistency among the variables that could be violated if methods would be interleaved in a particular sequence. There is however an explicit runtime check that detects this situation and issues a warning.

So far, views are restricted to shared variables. We have experimented with a further restriction: reads are only entered into lock views if the value is exposed outside the lock scope or method (i.e, the value is returned from a synchronized method or assigned to a stack escaping object). This modification reduces the number of reports by around 30-50%, however some cases of high-level data races are not recognized any more.

Program 5.4: Example of underreporting.

```
class Counter {
    int val;

    synchronized int inc(int a) {
        val = val + a;
        return val;
    }
}


class Main extends Thread {
    static Counter ctr;

    static void main(String args[]) {
        ctr = new Counter();
        new Main().start();
        new Main().start();
    }

    void run() {
        int i = ctr.inc(0);
        ctr.inc(i);
    }
}
```

5.4 Discussion

The algorithm for detecting method-level violations of atomicity is effective and detects several important scenarios that can lead to actual violations at runtime. Our experience shows that non-atomicity can also be a natural and desirable property of a method (Section 5.3, benign reports) and hence does not necessarily correspond to a synchronization defect.

The reporting of the algorithm can err in two directions: First, the algorithm is *unsound*, i.e., it might overlook cases where violations of atomicity are possible at runtime; however, the algorithm detects all cases where one thread reads a shared variable under lock protection that may consequently be modified by concurrent threads (hence the result of the read might become stale). Second, the algorithm is *incomplete*, i.e., it might report incidents that do not lead to an atomicity violation in any execution.

5.4.1 Sources of unsoundness

Program 5.4 demonstrates the unsoundness of our algorithm, i.e., it illustrates a violation of method-level atomicity that is not detected. There are two threads that each perform two consequent updates of a shared counter variable `val`. If method `run` would execute atomically, i.e., without interference, the final value of the counter would be doubled; n threads would increment the counter by 2^n. However the access sequences of multiple threads could interfere and the update of some thread might get lost. Hence `run` is not atomic. Note that there is only a single lock view $l_{\texttt{inc}}$ that overlaps with the method view $m_{\texttt{run}}$ (see Figure 5.6) and trivially satisfies the chain property. Hence method consistency is not violated.

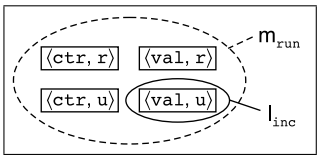

Figure 5.6: Method view and lock view for Program 5.4.

The focus of our method is to detect violations of atomicity due to interference of field variables. Violations that involve array variables are not detected.

5.4.2 Sources of incompleteness

Program 5.5 illustrates an example for the incompleteness of our algorithm, i.e., a report of a method inconsistency that does not correspond to a violation of atomicity: The lock views that overlap with the view of method `run` do not form a chain (lock protects reads *or* updates, see Figure 5.7); hence MC is violated. However, the initialization of the map happens only once and the effect of method `run` is the same regardless of the thread interleaving. The general reason for this imprecision is that our analysis assumes that all control flow paths are feasible for all threads (which is does not hold for the example at hand).

So far, the conceptual capabilities of MC have been discussed. Additional imprecision is added through the fact that static analysis relies in many cases on a conservative approximation of the runtime situation. Abstract views might specify an approximated set of variable access events that may differ from the actual runtime views due to infeasible control flows or the inability of the static analysis to differentiate access to field variables in different object instances. The approximation of the static analysis to distinguish different object instances and to determine thread interference and data sharing can lead to reports that do not correspond to real violations of atomicity – hence a potential source of overreporting.

5.4.3 Effect of view partitioning

The algorithm in Section 5.2.2 partitions views according to the affiliation of fields with classes (operation *partition_views*()). This strategy reduces the size of lock and method views and we argue that it avoids spurious violations of MC that involve unrelated variables and also makes the reporting of harmful incidents more concise (large method views do not easily unveil the variable(s) that caused the violation). One concern of view partitioning is that it might lead to underreporting. We discuss in the following that this concern is immaterial.

Consider Program 5.6: Two threads operate on the shared objects of class `A` and `B` that protect their fields from unordered access through their instance locks; hence, there is no data race. Method `update` operates on these data structures that are individually protected through their own locks; hence this method is a candidate for an atomicity violation. In the following, we discuss how this potential problem is discovered in an approach without and with view partitioning.

Table 5.2 shows the method and lock views of Program 5.6; read-only fields are not removed. The views are specified without and with partitioning. There are no views for method `put` because this method is not in the call graph of the program.

Program 5.5: Example of overreporting.

```
class Map {
    Object[] keys;
    Object[] values;
    boolean volatile init_done = false;

    void init() {
        if (!init_done)
            synchronized (this) {
                if (!init_done)  {
                    init_done = true;
                    // update keys and values
                }
            }
    }

    synchronized Object get(...) {
        // read keys and values
        return ...;
    }
}

class MapClient extends Thread {
    static Map map;

    static void main(String args[]) {
        map = new Map();
        new MapClient().start();
        new MapClient().start();
    }

    void run() {
        // lazy initialization
        m.init();
        o = m.get(...);
        ...;
    }
}
```

The views in Table 5.2 illustrate that a partitioning can indeed lead to the loss of reports: The three lock views l_0, l_1, l_2 overlap with m_1. These lock views do however not form a chain because $l_0 \cap l_2 = \emptyset$. Hence a violation of method consistency is reported for `mayinc`. If the views are partitioned according to classes, the situation is different: the lock views that overlap $m'_{1.1}$ and $m'_{1.2}$ form a chain and hence there is no report.

This difference in the reporting is however misleading and does not reflect the impact of view partitioning in practice: Program 5.6 does *not* suffer from a potential atomicity violation because there are no updates of `a` in the program, which is necessary for method `mayinc` to observe a violation of atomicity. If there was an update of `a` (e.g., a call to `inc`), method `mayinc` could indeed observe an atomicity violation and *both* strategies (with and without partitioning of views) would issue a report.

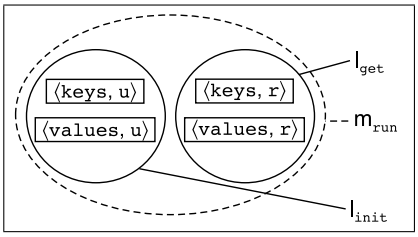

Figure 5.7: Method view and lock views for Program 5.5.

Program 5.6: Example of underreporting due to the partitioning of views.

```
class A {
    int a;

    synchronized int get() {
        return a;
    }

    synchronized void put(int i) {
        a = i;
    }
}


class B {
    int b;

    synchronized void mayinc(A a) {
        if (b == a.get())
            inc();
        // assert(a.a == b-1) ?
    }

    synchronized void inc() {
        b++;
    }
}


class Main extends Thread {
    static A aobj;
    static B bobj;

    static void main(String[] args) {
        aobj = new A();
        bobj = new B();
        new Main().start();
        new Main().start();
    }

    void run() {
        bobj.mayinc(aobj);
    }
}
```

method	*method views*		*lock views*	
not partitioned				
`get`	$m_0 =$	$\{\langle \mathtt{a},\mathtt{r}\rangle, \langle \mathtt{a},\mathtt{u}\rangle\}$	$l_0 =$	$\{\langle \mathtt{a},\mathtt{r}\rangle\}$
`mayinc`	$m_1 =$	$\{\langle \mathtt{a},\mathtt{r}\rangle, \langle \mathtt{a},\mathtt{u}\rangle,$ $\langle \mathtt{b},\mathtt{r}\rangle, \langle \mathtt{b},\mathtt{u}\rangle\}$	$l_1 =$	$\{\langle \mathtt{a},\mathtt{r}\rangle, \langle \mathtt{b},\mathtt{u}\rangle\}$
`inc`	$m_2 =$	$\{\langle \mathtt{b},\mathtt{r}\rangle, \langle \mathtt{b},\mathtt{u}\rangle\}$	$l_2 =$	$\{\langle \mathtt{b},\mathtt{u}\rangle\}$
partitioned				
`get`	$m'_0 =$	$\{\langle \mathtt{a},\mathtt{r}\rangle, \langle \mathtt{a},\mathtt{u}\rangle\}$	$l'_0 =$	$\{\langle \mathtt{a},\mathtt{r}\rangle\}$
`mayinc`	$m'_{1.0} =$	$\{\langle \mathtt{a},\mathtt{r}\rangle, \langle \mathtt{a},\mathtt{u}\rangle\}$	$l'_{1.0} =$	$\{\langle \mathtt{a},\mathtt{r}\rangle\}$
	$m'_{1.1} =$	$\{\langle \mathtt{b},\mathtt{r}\rangle, \langle \mathtt{b},\mathtt{u}\rangle\}$	$l'_{1.1} =$	$\{\langle \mathtt{b},\mathtt{u}\rangle\}$
`inc`	$m'_2 =$	$\{\langle \mathtt{b},\mathtt{r}\rangle, \langle \mathtt{b},\mathtt{u}\rangle\}$	$l'_2 =$	$\{\langle \mathtt{b},\mathtt{u}\rangle\}$

Table 5.2: Method and lock views of Program 5.6.

More general, our algorithm has, independent of the strategy that is used for view partitioning, the following property: All incidents where one thread reads a shared variable under lock protection and other threads may update the same variable under lock protection are reported as potential violations of atomicity.

5.4.4 Related work

Method consistency is motivated by previous work of Artho et. al. [8], which analyzes the structure of locking in a program and infers consistency constraints for sets of shared variables (*view consistency*). Violations of their consistency model correspond to potential synchronization defects called *high-level data races*. The notion of high-level data races is similar to violations of atomicity although both concepts are incomparable and several important scenarios of atomicity violations are not covered by the definition of high-level data races [126].

Burrows and Leino [18] identify local variables that hold copies of shared data; a potential error occurs if during the program execution, the local copy becomes inconsistent with the original shared variable but is still assumed to hold the up-to-date value. The kind of errors that are detected (stale-value errors) resemble violations of method consistency. Their analysis is complementary to our work, because it detects errors that are not found by our work (our procedure does not inspect uses of local variables; also the error in Program 5.4 would be found by the stale-value analysis), and vice versa (the error in Program 5.3 would not be found by the stale-value analysis).

Flanagan and Qadeer [46, 45] follow a different approach to detect potential violations of atomicity and focus on the structure of statements and the possible interleavings of statements through multi-threading. They have developed a type system that verifies atomicity. Unlike our technique that regards methods as the unit of atomic execution, Flanagan and Qadeer conjecture atomicity only for synchronized methods and blocks (the error in Program 5.3 would not be found). In their work, the type checker associates atomicities at the level of statements and combines these atomicities based on Lipton's theory of left and right movers [78] to obtain atomicity information for statement groups and methods. This approach is modular and requires explicit information about the synchronization discipline and lock protection of shared variables. This information is pro-

vided through annotations and hence Flanagan and Qadeer's technique is not fully automated.

Wang and Stoller [126] propose dynamic techniques to detect atomicity violations that is based on [46, 45] but improves on the precision. The key observation is that there can be groups of transactions that the reduction-based algorithm in [45] reports as non-atomic that still behave atomically in all schedules and hence should not be reported. Instead of individual transactions, the algorithm of Wang and Stoller groups transactions and searches for specific non-serializable access patterns in groups of transactions. So far, the authors have not applied their algorithms to large programs.

Based on their earlier work in [46, 45], Flanagan and Freund [43] present also a dynamic technique to determine violations of atomicity. Atomicity is checked for regions of code that are synchronized and, similar to our approach, for all methods that are public or package protected. The checker determines atomicity for over 90% of these methods, confirming that atomicity at the method level is a reasonable expectation of the programmer and common design principle for multi-threaded object-oriented programs. The checker infers the relation between data and protecting locks dynamically (using an object ownership model that is similar to the model we present in [122] and Section 7.1.2) and hence, unlike in their type-based detection of non-atomicity [46], this relation need not be specified explicitly. Moreover, the runtime technique is able to handle programs with data races.

6

Static deadlock detection

This chapter presents a program analysis that detects potential deadlock in programs with monitor-style synchronization. The algorithm is based on lock-usage information that are gathered during the symbolic execution (Section 3.4).

6.1 Resource deadlock

There are two cases that cause a thread to block upon access to an exclusive resource, i.e., a resource that can be used by a single thread at a time: First, the invocation of a synchronized method or the entry of a synchronized block is delayed until the accessing thread obtains the lock associated with the target argument. Second, a call to method `java.lang.Object::wait` (or variants of it) causes a thread to release the lock associated with the target argument and to block until method `java.lang.Object::notify` (or some variant of it) is called by another thread.

Definition. A *resource deadlock* occurs if one or more threads move into a wait situation that cannot not be resolved in the subsequent execution of the program. There are three factors that can lead to such a situation:

- *Critical wait:* A thread invokes `wait` while holding locks other than the target of the call.
- *Indefinite wait:* A thread invokes `wait` but there is no corresponding call to `notify`.
- *Mutual wait:* Several threads want to acquire the same locks in different orders and become entrapped in a cyclic wait condition.

More general definitions of deadlock exist that also consider wait situations due to external communication; our deadlock detection does not account for such wait situations.

6.2 Algorithm

The static deadlock detection algorithm proceeds in two steps: First lock, unlock, and wait events and their execution contexts are determined. Second, a so-called *lock-order graph* is created and mutual wait dependencies are determined (Section 6.2.3). An overview of the algorithm is given in Figure 6.1 and is discussed in more detail in Sections 6.2.1 to 6.2.3.

$Locked = \langle$all abstract objects that are used as locks$\rangle$;
$Waits = \langle$all contexts in which `wait` is called$\rangle$;
$Notifys = \langle$all contexts in which `notify` is called$\rangle$;
$LockStrings = \langle$all lock-strings$\rangle$;
$R = \emptyset$;

deadlock_detection()

/* phase 1: critical and indefinite waits */
$\forall\, w \in Waits$:
  `let` $o = \langle$target object of $w\rangle$, $L = \langle$lockset at $w\rangle$:
    `if` $(\exists\, l \in L : l \neq o)$
      $R = R \cup \{$`new` *Report*$(w,$ CRITICAL$)\}$;
    `if` $(\nexists\, n \in Notifys : o = \langle$target object of $n\rangle)$
      $R = R \cup \{$`new` *Report*$(w,$ INDEFINITE$)\}$;

/* phase 2: mutual waits */
$log = \langle$empty graph$\rangle$;
$\forall\, o \in Locked$:
  log.addNode(o);
$\forall\, s = \langle l_0, \ldots, l_n \rangle \in LockStrings$:
  `for` $(i \in \{0, \ldots, n-1\})$
    log.addEdge(l_i, l_{i+1});
$\langle N_{acg}, E_{acg} \rangle$ = *compute_acg*(log);
$\forall\, n \in N_{acg}$:
  `if` (n corresponds to multiple locks in log)
    $R = R \cup \{$`new` *Report*$(w,$ MUTUAL$)\}$;

Figure 6.1: Algorithm for determining potential deadlock.

.

6.2.1 Computation of lock usage information

Lock usage information is recorded along the symbolic execution (Section 3.4) according to the rules in Figure 6.2. The rules reflect situations that may cause a thread to block a runtime. Variable L refers to the ordered collection of abstract objects that are locked at the current statement (not including the lock acquired by the statement itself). This collection is maintained by the operations *acquireLock*() and *releaseLock*(); these operations are already indicated in the corresponding rules in Figure 3.8 but we repeat them here (Figure 6.2) for clarification. Although not specified explicitly in the rules, it is only necessary to record lock usage information for objects that are *shared* (Section 3.3.4). Operation *prefix*() converts the lockstack L to an ordered set of abstract objects LS, called *lock-string* such that the locks acquired earliest in the scoped hierarchy are retained and abstract objects that correspond to lock reentrance are elided.

Variables and domains		**Analysis state**	
$m \in \mathcal{M}$	methods	$AS : \mathcal{V} \to O$	alias set lookup for local vars
$c \in \mathcal{V}$	local variables		
$o \in O$	alias sets		
$LS \in 2^O$	ordered set of alias sets (lock-string).		

Statement	**Action**

v.`wait`()	`let` $LS = prefix(L)$:
	  if ($LS.length() > 1$)
	  $warn$("critical wait");
	$registerWait(AS(v))$;

v.`notify`()	$registerNotify(AS(v))$;

$v = m(v_0, \ldots, v_n)$	`let` $LS = prefix(L)$:
	  if (m is `synchronized` $\wedge \neg\, LS.contains(AS(v_0))$
	  $LS.append(AS(v_0))$;
	  $registerLockString(l)$;
	$acquireLock(AS(v_0))$;

`monitorenter` v	`let` $l = prefix(L)$:
	  `if` ($\neg\, LS.contains(AS(v_0))$
	  $LS.append(AS(v_0))$;
	  $registerLockString(l)$;
	$acquireLock(AS(v))$;

`monitorexit` v	$releaseLock(AS(v))$;

Figure 6.2: Transfer rules for the computation of lock-strings along the symbolic execution; t is the current abstract thread; L is the stack of currently locked objects.

The operations *contains*() and *append*() operate on this ordered set and have the obvious semantics. Operation *registerLockString*() memorizes lock-strings and makes them available for the computation of the lock-order graph (Section 6.2.3). Similarly, operations *registerWait*() and *registerNotify*() record information for the detection of critical and indefinite waits (Section 6.2.2). Lock usage information is the input to the algorithm in Figure 6.1 and is provided by the sets *Locked*, *Waits*, *Notifys*, and *LockStrings*.

6.2.2 Identifying critical and indefinite wait

Critical and indefinite waits are identified in a straightforward manner through iteration over all calling contexts of `wait`: An invocation is critical if the lockset specified by the context contains locked objects other than the target of the call. An invocation of `wait` (without timeout) is indefinite if there are no invocations of `notify` on the same object.

6.2.3 Lock-order graph and cycle detection

In our model exclusive resources correspond to locks. A mutual wait condition cannot occur if there exists a total order of locks and all threads adhere to this order when acquiring locks. The *lock-order graph* serves to determine an ordering among locks from a program: Nodes correspond to locks; edges correspond to the order in which some thread acquired the lock, i.e., there is an edge from $l_1 \rightarrow l_2$ if some thread acquires l_2 while holding lock l_1. While the lock-order graph can be built at runtime and used for dynamic deadlock detection (e.g., [69]), the algorithm presented in this section computes a static approximation of the lock-order graph over all program paths on the abstract domain of objects and threads defined in Chapter 3.

Definition. A *lock-order graph* is a directed graph $LOG = \langle N, E \rangle$:

- N is the set of nodes corresponding to abstract objects that are used as locks.
- E is the set of directed edges that reflect the order in which each thread acquires locks.

The lock-graph is built in the second phase of the algorithm in Figure 6.2 (operations *addNode*() and *addEdge*()); nodes correspond to nodes in the HSG, edges are created from lock-strings. Cycles in the lock-order graph are determined through the computation of an acyclic component graph (ACG): If a node in the ACG subsumes multiple nodes in the original graph, a mutual wait situation is detected and reported.

6.3 Experience

We have detected all deadlocks in the small example programs that are distributed with the Rivet virtual machine and model checking tool [19] (benchmark sections 'Deadlock' and 'Deadlock-Wait'). Moreover, we apply the deadlock checker to the benchmarks introduced in Section 3.5.2 and report the results in Table 6.1.

	critical	*indefinite*	*mutual*
philo	0	0	0
elevator	0	0	0
mtrt	0	0	0
sor	0	0	0
tsp	0	0	0
hedc	0	0	1
mold	0	0	0
ray	0	0	0
monte	0	0	0
specjbb	0	0	1
jigsaw	0	0	1

Table 6.1: Results of the deadlock detection.

Most benchmarks have a simple structure of locking and do not allow for resource deadlock. The results of the deadlock analysis reflect this situation such that there are no reports for all but three benchmarks.

The report for hedc is spurious and does not reflect a genuine synchronization error. Three locks are involved: The locks associated with the classes `java.util.TimeZone` and `java.util.Locale` that are used to provide lazy one-time initialization of default values, and the lock corresponding to the default instance of `java.util.Properties`. The lock of the time zone and the locale classes are accessed independently, the lock of the properties instance is accessed in the locking scope of time zone and locale (i.e, 'after' in the lock-order). The properties instance is however aliased with other objects that are locked 'before' time zone and locale in the lock-order; hence the lock-order graph contains edges in both directions between the properties and the time zone respectively locale nodes but each direction is due to a different object instance. The resulting cycle leads to the spurious report. Two other spurious reports in specjbb and jigsaw follow a similar pattern of three abstract locks, one of which obliterates the lock-order due to false aliasing.

6.4 Discussion

The algorithm is effective in determining important scenarios where inappropriate lock usage can lead to deadlock at runtime. Apart from reporting true positives, the algorithm can err: The algorithm is *unsound*, i.e., it might overlook cases where resource deadlock might actually occur at runtime. The algorithm is also *incomplete*, i.e., it might report incidents that do not lead to deadlock in any execution.

6.4.1 Sources of unsoundness

The analysis is flow-insensitive and assumes that some `notify` is alway executed after a corresponding `wait` – an assumption that might not hold in all program executions. Hence a deadlock situation involving threads that wait on monitors on which no signal will be forthcoming may not be recognized by the static analysis. Tools that are based on model checking like Bandera [31] or the Java Path Finder [120] can remedy this problem (they cannot provide a precise solution either because the problem of determining feasible execution paths is equivalent to the halting problem). For the given benchmark programs, we could not determine false negatives of indefinite waits.

Moreover, the analysis treats abstract objects as if they correspond to unique runtime instances: Lock sets are contracted to lock-strings such that an acquire operation on an abstract lock that is already held is considered as reentrant (each abstract objects occurs at most once in a lock-string). Multiple runtime lock instances that the static analysis cannot disambiguate are represented as a single node in the lock-order graph, and consequently cyclic dependencies among different runtime instances of aliased locks are not apparent from the graph. Hence mutual wait conditions between aliased locks may not be reported. We have not encountered this phenomenon in the benchmarks.

6.4.2 Sources of incompleteness

First, the flow-insensitivity of the analysis could lead to a report of a mutual wait situation which actually cannot occur at runtime: Imagine that a parallel program that executes in two

phases, e.g., with intervening barrier synchronization. Threads in the first and second phase could assume different orders among the locks as long as all threads obey a common ordering in each of the phases.

Second, false aliasing may lead to cycles in the lock-order graph that would not exist if the lock-order graph was based on runtime instances. According to our experience, false aliasing is one of the major sources of overreporting. Our analysis compares favorable to other tools like Jlint [7] that disambiguates locks only based on declaration information (type-based) and does not omit uncontended locks (abstract objects that are not shared).

6.4.3 Related work

A common technique for static deadlock detection is to translate a program into a state-based model where states represent control- and data information and transitions correspond to statement execution or inter-thread communication. The number of states in such a model can grow exponentially if the effects of individual threads are combined. Hence a major difficulty in applying static state-based deadlock detection to larger programs is the *state-explosion problem*, and different techniques have been proposed to reduce and control the number of states that have to be considered by the static checker [30, 131]. This static technique of verification is very general, i.e., it can also be used to determine communication deadlock, and precise, i.e., it detects all potential deadlock situations and reports few false positives. However, the very large resource demands due to the state-explosion are still a limiting factor if this approach is applied to larger programs.

Masticola and Ryder [80] developed a static deadlock detection that operates on a flow-oriented program representation; the so-called *sync graph* captures synchronization interactions between tasks that use the barrier rendez-vous model of Ada. The analysis is more efficient in time and space than state-based approaches that rely on program verification techniques. However it can be less precise due to conservatism that leads to spurious reports. Naumovich et. al. [88] adopt and extend this technique for Java programs and describe constraints for checking misuses of Java specific synchronization features, not general synchronization defects like deadlock however.

Other than previous techniques, our approach to deadlock detection is basically flow-insensitive but context-sensitive; this is possible because we focus on the detection of resource deadlock in programs with a monitor-style lock and unlock discipline that follows control flow scopes; this is the default synchronization style of the Java language and is hence frequently encountered in practice. While the synchronization structure in this class of programs is scoped and hence may be simpler than the more general cases considered in previous work, the issue of our work relates to the disambiguation of locks and the tracking of contexts in which these locks are active; both aspects are addressed in our static analysis framework for concurrent-object oriented programs (Chapter 3).

7

Dynamic checking

This chapter discusses two low-overhead techniques for the dynamic detection of synchronization faults. Both techniques are implemented as a combination of sparse program instrumentation and runtime library support.

7.1 Object race detection

This section describes the design and implementation of a dynamic checker for object races [122]. The intuition of object races is to coarsen the granularity of runtime checks from individual variables to objects. This decision is motivated by the observation that all fields of mutable shared object are typically protected by the same synchronization discipline, e.g., the same lock instance is held during the access of any field. This assumption helps to reduce the checking overhead and retains the accuracy of reporting for programs that adhere to this typical synchronization discipline.

7.1.1 Locking discipline

An object race is a variant of unique-lock data race (Section 2.2.2). The definition of an object race in a program execution is given in Section 2.2.3: Every access to some mutable field of a shared object must be protected by a unique lock, i.e., the lock must be held during the access. This definition prescribes a *locking discipline* which – if obeyed by the a certain execution – implies the absence of data races in that execution.

An online checker can validate this locking discipline as follows [103]: Each runtime object or class o has an associated lockset $o.L$; initially, $o.L$ contains all possible locks. At each access (GET/PUT event) to o during the program execution, the associated lockset is refined, i.e., $o.L$ is intersected with the set of locks held by the accessing thread. Like this, the protecting lock of an object is gradually inferred during the execution. The locking discipline is violated if $o.L$ becomes empty and o is accessed by more than one thread.

This simple locking discipline is however impractical for determining object races in real program executions because the following correct programming patterns would lead to spurious reports [103]:

- *Initialization:* The creator thread of a shared object can initialize and access the object without lock protection as long as no reference to the object is available to other threads.

- *Shared-read objects:* Objects that are initialized by the allocating thread, then made available to other threads that perform concurrent reads, do not need lock protection.

These patterns occur frequently and hence a practical checker should account for them and separate the initialization phase of an object from its shared use. This separation is accomplished by the object ownership model.

7.1.2 Object ownership model

The locking discipline is complemented by an *ownership model* that distinguishes different life phases of an object and adjusts the checking policy accordingly. A design goal for our detection system is to determine the initialization phase of an object and to carry out expensive lockset operations only for those objects that are actually shared. The ownership model is illustrated by the transition graph in Figure 7.1 where nodes correspond to the ownership states of an object. The following states are possible:

- *Init:* Initial state after object allocation and during initialization. The owner of the object is the allocating thread. This state requires no actions with regard to the race checking. An action that may make the reference to the instance available to other threads (escape) will transit the object to the *owned* state.

- *Owned:* The owner remains the same as in state *init*. In this state, the locks held at object access are *not* tracked in the lockset associated with the accessed object. Access by a non-owner thread must be recognized and leads the object to a *handoff* state (*handoff read* if the access is a read, *handoff modified* if the access is a write).

- *Handoff read:* In this state, the object is accessed by a single thread other than the initial owner. This thread is called the *second owner* [122]. Only read access is allowed. Write access by the second owner leads to the state *handoff modified*. In contrast to state *owned*, the locks held at object access are tracked in the lockset associated with the accessed object. An empty lockset does not yet lead to a report. Access by threads other than the second owner must be recognized and lead the object to a *shared* state (*shared read* if the access is a read, *shared modified* if the access is a write).

- *Handoff modified:* Same as state *handoff read*, however, the second owner may write to the object. Access by a thread other than the second owner leads to *shared modified*.

- *Shared read:* The object may experience concurrent read access. Access is tracked by updating the lockset associated with the object. No report is given, even if the lockset becomes empty.

- *Shared modified:* The object may experience concurrent read and write access, hence it must thus be consistently protected by at least one common lock: the lockset associated with the object is updated at every access, and a conflict is reported if the set becomes empty.

- *Conflict:* An access conflict has been observed for this object; it is not subject to further access checks.

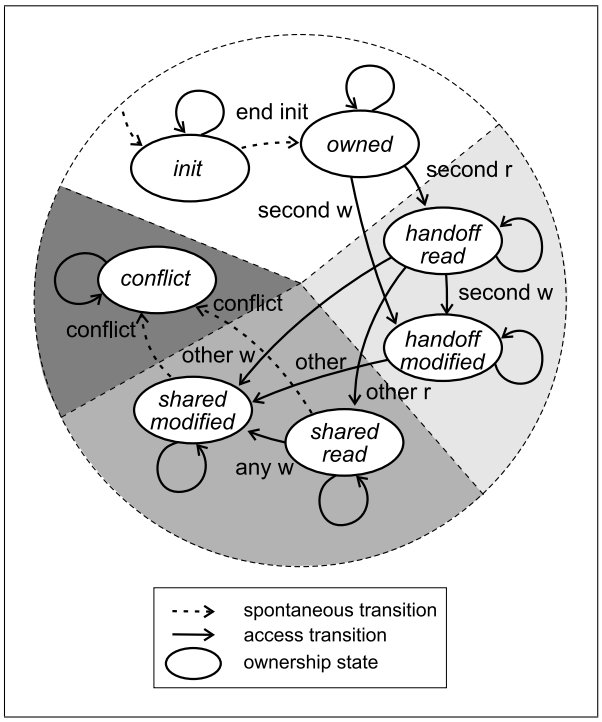

Figure 7.1: Ownership model.

There are two kinds of transitions:

1. *Access transitions* happen in response to read and write access of different threads.

2. *Spontaneous transitions* happen due to object creation, escape, and the findings of the access checker.

Objects may not be advanced through all states in the ownership model. Our evaluation in Section 7.1.6 shows that it is common that object remain in state *owned*, or one of the *handoff* states throughout their lifetime. The ownership model is tuned for two aspects: First, *thread-local* objects are handled efficiently, because they prevail in state *owned* and locks are not tracked in this state. Second, object-handoff is accommodated though state *handoff*: If first the creator and then some other threads access an object without common lock, no report is given. This heuristic is useful to avoid false reports in scenarios where one thread creates and initializes an objects that is passed to and then exclusively used by a second thread.

7.1.3 Object access protocol

The online object race checker is implemented as an *object access protocol*, i.e., a convention of access that threads obey when accessing objects that are candidates for object races. There are two tasks that the protocol must perform: (1) *ownership tracking* and (2) *validation of the access discipline*. The following auxiliary data structures are used in the definition of the object access protocol:

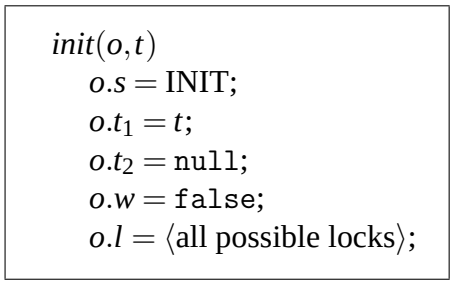

Figure 7.2: Object initialization in the object access protocol.

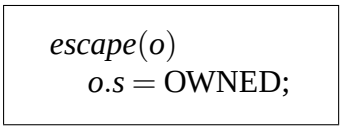

Figure 7.3: Escape event of object access protocol.

- $t.L$ set of locks held by thread t.
- $o.l$ the lockset associated with object o.
- $o.s$ the ownership state of object o (one of {INIT, OWNED, HANDOFF, SHARED, CONFLICT}).
- $o.w$ flag that specifies if there has been a write in state HANDOFF or SHARED.
- $o.t_1$ the first owner of o.
- $o.t_2$ the second owner of o.

Object initialization

The initial owner of an object is the allocating thread, the initial ownership state is OWNED. In this state, the object cannot be accessed concurrently from different threads because the reference to it is confined inside the allocating thread. No provision needs to be taken to detect concurrent access or to update the lockset associated with the object. The procedure *init*() summarizes the actions that immediately follow the allocation of object o by thread t (Figure 7.2).

Object escape

If the owner thread t assigns the reference of an object o to a variable that may communicate the reference to another thread, *escape*() is triggered (Figure 7.3). The escape event terminates the INIT phase. Our implementation determines the point of escape conservatively at compile-time: The allocation of an object is followed by a series of NEW-safe access events (Section 4.3.2). The first access that is not NEW-safe triggers the escape event that transits the object to the OWNED state.

```
/* prologue */
if (o.t1 ≠ t)
   atomic:
      if (o.s = OWNED)
         o.s = HANDOFF;
         o.t1 = null;
         o.t2 = t;
      else
         o.w = o.w ∧ is_write(e);
         if (o.s = HANDOFF ∧ o.t2 ≠ t)
            o.s = SHARED;
            o.t2 = null;

/* access */
⟨thread t issues access event e to object o⟩;

/* epilogue */
if (o.t1 ≠ t ∧ o.s ≠ CONFLICT)
   atomic:
      o.l = o.l ∩ t.L;
   if (o.l = ∅ ∧ o.s = SHARED ∧ o.w)
      new Report(e,o,t);
      o.s = CONFLICT;
```

Figure 7.4: Prologue and epilogue of the object access protocol.

Object access

Object access statements that are not NEW-safe are encased within an access prologue and epilogue (Figure 7.4): Let e be the access event, o the accessed object, and t the accessing thread.

The prologue serves for ownership tracking: A thread that accesses an object that it does not own in state OWNED or HANDOFF must advance the object to HANDOFF respectively SHARED. The state change must happen atomically because several threads may perform the update concurrently and updates of the auxiliary header fields s, w, t_2 could be lost. The epilogue serves to validate the locking policy. If a report is generated, the ownership state of o is advanced to CONFLICT such that subsequent access checking of o is bypassed.

Call sites of synchronized methods are not instrumented.[1] Instead, the equivalent of the prologue is done inside the method after the lock is acquired; symmetrically, the epilogue is performed immediately before the lock is released.

The auxiliary fields for the owning thread and the ownership state in each object are themselves accessed concurrently and may be subject to data races. The specification of *prologue*() and *epilogue*() in Figure 7.4 assumes a sequentially consistent memory model for these auxil-

[1] All call sites are "de-virtualized" in programs that are instrumented for race detection. Details are described in Section 7.3.

```
acquire(l,t)
    t.L.add(l);
    ⟨perform lock operation on l⟩;

release(l,t)
    if (¬⟨release corresponds to a reentrant acquire⟩)
        t.L.remove(l);
    ⟨perform lock operation on l⟩;
```

Figure 7.5: Lock operations in the object access protocol.

iary fields; code regions that are required to execute without interference are specified in `atomic` sections. The implementation of *prologue*() and *epilogue*() is platform dependent and may require that ordering or atomicity are ensured through memory fences, atomic instructions or locking. Our implementation and the issue of memory ordering are discussed further in Section 7.1.4.

Locking

The operations for lock *acquire*() and *release*() are complemented and maintain the lockset $t.L$ of locks held by the current thread (Figure 7.5).

7.1.4 Implementation

Instrumentation

The access prologue and epilogue are implemented through inline code at object access sites; only those access sites are instrumented that participate in an access conflict, i.e., correspond to a conflicting event in some OUG (Section 4.4).

Optimization

The dynamic frequency of prologue/epilogue executions can be reduced through a compile-time optimization: The principal observations is that for the validation of a locking discipline, an access sequence to the same object in the same locking context can be subsumed under a single check. Hence it sufficient to execute the prologue before the first access in the sequence and the epilogue after the last access. We have implemented the omission of redundant epilogue/prologue sequences in two scenarios:

First, the instrumentation corresponding to critical access statements inside a method through the `this`-pointer can be combined: The instrumenter treats the invocation of an instance method that may execute conflicting access through the `this`-reference like an object access, i.e., the invocation site is surrounded by a corresponding prologue/epilogue pair. Consequently, access through the `this`-reference inside the method need no longer be instrumented. The execution of a method is considered as write access if the method may write to a field of

its target object. Class methods can be treated similar to instance methods such that accesses to global variables of the same class need not be instrumented. To simplify the presentation, we refer to both cases (class and instance) as "access to `self`", where `self` refers to the object context provided by the enclosing method and means the `this`-reference in case of an instance method and the defining class in the case of a class method. Synchronized methods or methods with block monitors are treated specially such that the execution of the prologue/epilogue takes into account the change in the locking context caused by the lock operation. This optimization is effective in reducing the number of runtime checks if access to `self` occurs frequently (this aspect is evaluated and discussed in more detail in Section 7.1.6). The optimization can however lead to overreporting, e.g., if a method call signals a write access to the target object but the flow inside the method incarnation does not actually execute the write.

Second, we prune redundant occurrences of prologue and epilogue for field access to objects other than `self`. The SSA property of the intermediate representation can be exploited to identify object access to the same instance inside a method. The effectiveness of disambiguating local references variables is increased through preparative program transformations like inlining, loop unrolling, copy propagation, and PRE-based load elimination [125]; these transformations promote heap variables to local variables (or registers) and hence support the scope of SSA-based value disambiguation. The rationale of the optimization is the following: If multiple access statements that target the same object occur in the same basic block, a prologue at the beginning and an epilogue at the end of the block is sufficient. For access statements in different basic blocks, the prologues can be combined and hoisted to a common dominator block; similarly, the epilogue can be sunk to a common post-dominator. Note that prologue and epilogue sequences must not be moved across block synchronization statements or calls that may invoke `java.lang.Object::wait`.

Code generation

This section discusses issues of code generation for the object access protocol on the Intel IA-32 platform [67]. All instruction of the prologue and the common execution path of the epilogue are inlined at critical object access sites. The lockset operation in the epilogue and the extension of the locking are implemented in the runtime library [49].

The frequent use of the thread identification information in the instrumentation lead us to change the compiler (register allocator) and runtime system to make the unique id of the currently executing thread available in a callee-saved register (`%ebx`).

We have enhanced the IA-32 code generator and specified code templates corresponding to inline code. The instrumentation of the access prologue requires five instructions. In state OWNED, one read and one conditional jump are executed; in states HANDOFF and SHARED, two reads and two conditional jumps are executed. The access epilogue requires 8 assembly instructions. In the states HANDOFF and SHARED, a branch into the library is executed to register the current lockset; in state OWNED/CONFLICT, one/two reads and one/two conditional jumps are executed.

	init	*owned*	*handoff-0*	*handoff-1*	*shared*	*conflict*
field access	1.0	1.1	6.5	19.1	20.4	1.5
static call	1.0	1.3	3.7	9.1	9.6	1.4
virtual call	5.7	5.9	8.2	13.1	13.6	5.9
monitor (enter+exit)			3.8			

Table 7.1: Cost of instrumented object and lock access relative to the un-instrumented variants. The columns distinguish the ownership state of the accessed object; column *handoff-0* refers to the execution in state HANDOFF where no lock is held during the access; for *handoff-1*, one lock is held during the access.

Micro benchmarks

We use a micro benchmark that executes object access statements inside a loop of 10^7 iterations; all (but the first) memory accesses are served from the L1 processor cache. Execution time is measured on a Pentium III 933 system with 512 MB of main memory. Table 7.1 illustrates the relative cost of the object access protocol. The numbers show the relative execution time of an instrumented access (access that is not instrumented for race detection corresponds to 1.0). We report data dependent on the ownership state of the accessed object.

Row *field access* considers read and write access and reports the average. There is no additional cost for variable and static method access in state INIT. The significant overhead for *virtual call* stems from our mechanism for code specialization where polymorphic calls are not dispatched through the ordinary vtbl-mechanism but through a chain of `instanceof` invocations (details in Section 7.3). The number given for state INIT is the baseline overhead due to this mechanism. Additional overhead for *virtual call* in other states stems from the instrumentation.

The overhead in state OWNED stems from the comparison of the owner thread with the current accessing thread in the prologue and epilogue. This overhead is small.

In states HANDOFF and SHARED, the overhead of object access stems from the execution of the prologue and epilogue. The cost of the ownership tracking in the prologue which requires an additional check of the object header is largely hidden by the processor cache (the header field and the accessed variable are in the same cache line) and the processor-level instruction level parallelism (ILP). The cost of the epilogue is more significant because a library call is executed that updates the lockset associated with the accessed object if necessary and reports a violation (in state SHARED). The predominant factor is the cost of the lockset operations that depends on the number of locks being held (the table shows numbers where no or one lock is held during access in state HANDOFF). The cost in state SHARED is even higher due to the lockset intersection. The implementation of the lockset data structure is based on the STL [111] `set` data type; concurrent intersection is prevented by spin locks.

Once an empty lockset is encountered in state SHARED, the conflict is reported and the object is advanced to state CONFLICT. In this state, the epilogue bypasses the library call and the cost of the instrumentation becomes negligible.

The overhead of monitor access (*monitor (enter+exit)*) stems from maintaining a lock-nest-counter, and adding and removing the target lock from the set of locks currently held by a thread. This overhead is independent of the ownership state.

Memory ordering

In our implementation, the race detection is performed inline by the application threads. Hence auxiliary variables and data structures that support the race checker constitute themselves shared data such that concurrent access must be ordered. Two kind of data are relevant: The additional header field that is used for ownership tracking and the lockset associated with an object that records the locks held during object access.

An aligned word (4 bytes) variable is reserved in each object header to record the owner thread identification and the ownership state. All operations on this header field operate on the entire word and hence the IA-32 architecture guarantees that all values that are read have been written previously (no word-tearing and out-of-thin-air values can occur). Threads might however observe stale values that can lead to the situation that the checker overlooks an actual violation of the locking policy or issue more than one report for a specific object. The effect of stale values is equivalent to the scheduling dependence of the detection mechanism, which is discussed in Section 7.1.6.

There is an auxiliary structure per object that implements the monitor lock. This structure is allocated lazily when the lock associated with an object is used for the first time. Our implementation extends this object specific auxiliary structure to hold a reference to the lockset and the flag that specifies if a write access has been seen. The lazy allocation of this structure now also occurs if the object transits to a handoff state.

7.1.5 Memory overhead

Four additional bytes in the object header increase the total amount of memory by at most 25% (the smallest object in the original implementation is three words in size). The memory necessary for the extension of the auxiliary per object structure and the locksets is very low because most benchmarks advance a rather small fraction of the allocated objects and arrays to a handoff state and beyond (see Section 7.1.6, Table 7.2).

7.1.6 Experience

Ownership model

Table 7.2 shows the fraction of arrays and instances that are advanced to the states *handoff* (read or write), *shared-r*ead, *shared-w*rite, and *conflict*. Each column includes the objects reported in the subsequent category (right column next to it). Benchmarks that operate mostly on local data, e.g., `mtrt` and `ray`, advance only very few objects to state *handoff* and beyond. In other benchmarks, a larger fraction of instances of arrays are actually accessed by more than a single thread. In *sor*, the majority of objects are rows of two matrices (that are implemented as jagged arrays in Java) that are accessed concurrently. In *tsp*, the proportion of *shared* objects seems low; the majority of these objects stem from a pool of pre-allocated objects from which only a small fraction is used for the tsp-problem size in this run.

	handoff	*shared-r*	*shared-w*	*conflict*
philo	1.70	1.09	0.97	0.00
elevator	21.14	8.88	5.71	1.27
mtrt	0.47	< 0.01	< 0.01	< 0.01
sor	94.56	94.47	93.90	93.81
tsp	3.80	3.32	3.21	2.01
hedc	23.78	4.53	0.76	0.32
mold	1.59	0.52	0.31	0.26
ray	0.01	< 0.01	< 0.01	< 0.01
monte	27.88	3.49	3.48	< 0.01
specjbb	5.22	0.01	0.01	< 0.01
jigsaw	19.23	1.61	0.41	0.18

Table 7.2: Ownership states reached by instances and arrays as percentage of the overall number of allocations (state *init*).

Access characteristic

Table 7.3 summarizes the object access characteristics of the benchmark programs. The first six columns of the table partition access events into *field* or *array* access, *method* invocation, and *monitor* lock access. The numbers report the relative frequency of each category, the sum is 100%. Access to fields distinguishes between references to variables that belong to the same instance or class as the currently executed method (*self*), or not (*other*). Method invocations are split up according to *static* and *virtual*[2] dispatch. The number of method invocations is in most cases clearly lower than the number of field accesses. For most benchmarks, the number of access events in category *self* is larger than in category *other*. Both observations support the design rationale of the dynamic checker that focuses on objects rather than individual variables and render checks through `self` redundant. Some benchmarks have a significant number of array accesses which, in the case of sor and mold, mostly target shared data.

	field		*array*	*method*		*monitor*	*rc (incl array)*	*rc (no array)*	
	self	*other*		*static*	*virtual*				*optimized*
philo	39.0	9.9	12.6	29.5	3.8	5.1	0.5	0.0	0.0
elevator	40.3	6.6	15.9	17.1	9.9	10.2	14.3	4.6	4.6
mtrt	48.1	< 0.1	11.6	2.5	37.7	0.2	< 0.1	< 0.1	< 0.1
sor	37.5	< 0.1	62.5	< 0.1	< 0.1	< 0.1	31.2	0.0	0.0
tsp	39.2	29.8	30.0	< 0.1	1.0	< 0.1	13.7	6.9	5.6
hedc	41.6	4.2	30.4	17.0	4.0	2.8	12.9	2.0	1.7
mold	40.9	37.7	21.3	0.1	< 0.1	< 0.1	1.4	< 0.1	< 0.1
ray	31.2	54.7	2.7	8.7	2.7	< 0.1	48.1	48.1	18.7
monte	61.4	< 0.1	22.4	4.3	6.8	5.0	< 0.1	< 0.1	< 0.1
specjbb	41.3	5.6	25.1	9.2	14.4	4.3	17.8	8.9	4.4
jigsaw	45.2	3.4	18.1	22.3	6.3	4.6	21.2	11.5	11.4
average							14.8	7.4	4.2

Table 7.3: Access characteristics and dynamic frequency of access checks for object race detection. The number of access events are normalized such that the sum is 100%. The number of checks are relative to the total number of access events.

[2]Polymorphism is reduced through type information that is computed along the variable-type analysis (VTA, [115]).

Table 7.3 also specifies the number of runtime checks for object race detection: with checks for array access (column *rc (incl arrays)*) and without (columns *rc (no arrays)*). Details of the *optimized* version in the latter variant are discussed below.

The frequency of access checks for object race detection is investigated in more detail in Table 7.4. Two principal settings are distinguished: checking that includes (*rc (incl array)*) and excludes (*rc (no array)*) array access. Several categories stand for different degrees of instrumentation:

- *all:* In this category, all field and array access statements (for *rc (incl array)*) and respectively field access statements (for *rc (no array)*) are instrumented. These variants are not explicitly stated in the table, but the numbers of other categories are reported relative to it, i.e., the reports for *all - rc (incl array)* and *all - rc (no array)* would be 100%. The instrumentation strategy of this category does not exploit the information provided by the static conflict analysis in reducing the amount of instrumentation.

- *conflict:* This instrumentation variant applies checking only to the objects that are determined as conflicting by the static conflict analysis (Section 4.4). It is the standard configuration of the object race checker where method invocations are instrumented if the call targets a conflicting object and one of the invoked methods reads or write the target object; additionally, field respectively array access to objects other than `self` is instrumented.

- *conflict-self:* This variant is similar to *conflict*, however access to `self` is instrumented but calls are not instrumented. This configuration, compared to *conflict*, expresses the runtime effect of grouping access checks to `self` at method boundaries.

- *optimized:* Similar to *conflict*, however the program as well as the instrumentation are optimized. Some optimizations do not affect the variable access frequency (inlining and loop peeling), others may reduce the number of access events (copy propagation and partial redundancy elimination). The optimized program variant also incorporates transformations of the program instrumentation, i.e., the reduction of redundant checks discussed in Section 7.1.4. Both columns indicate that the number of runtime checks is very low relative to the overall access frequency.

The first number in each column of Table 7.4 refers to the fraction of inline checks relative to category *all*, the second entry specifies the percentage of checks that necessitate a library call (which is only necessary for objects that reach the ownership states *handoff* or *shared*). The fraction of inline checks that also descend into the library is also high, at least for the benchmarks that exercise a significant number of checks. This confirms that the static analysis is able to accurately determine the sharing of objects (for objects that remain in state *owned*, the static analysis erred (on the conservative side) in assuming that the object would eventually be advanced to state *shared* or *conflict*). Note that the number of inline access checks and library descents can vary in different program runs. We did however not experience large variations in the measurements reported here.

The numbers show that the static conflict analysis is very effective (categories reported in Table 7.4 vs. category *all*) in reducing the amount of dynamic checking, although most benchmarks operate intensively on shared data. For some benchmarks however, the static analysis is

	rc (incl array)		*rc (no array)*					
	conflict		*conflict-self*		*conflict*		*optimized*	
	inline	*lib [%]*	*inline*	*lib [%]*	*inline*	*lib [%]*	*inline*	*lib [%]*
philo	1.4	96.4	0.0	-	0.0	-	0.0	-
elevator	27.6	36.3	11.7	99.6	11.6	100.0	11.6	100.0
mtrt	3.7	8.6	< 0.1	75.0	< 0.1	75.0	< 0.1	100.0
sor	31.2	99.2	0.0	-	0.0	-	0.0	-
tsp	14.0	10.4	10.0	1.9	10.0	0.9	8.2	2.3
hedc	20.1	91.3	12.4	37.2	6.5	39.8	5.5	35.5
mold	1.4	20.6	< 0.1	100.0	< 0.1	100.0	< 0.1	0.0
ray	54.4	77.5	63.1	79.4	56.1	77.5	21.8	80.5
monte	< 0.1	1.1	< 0.1	0.7	< 0.1	1.1	< 0.1	1.6
specjbb	29.7	87.7	37.5	49.3	17.4	73.7	8.6	57.9
jigsaw	27.6	53.1	27.5	11.1	20.6	20.3	20.4	12.8

Table 7.4: Dynamic frequency of access checks for object race detection. The numbers are normalized to a program variant where all fields and array (*rc (incl array)*) respectively field access statements (*rc (no array)*) are instrumented (100%).

not able to identify the absence of conflicts on core data structures; this is the case for some configurations that include arrays and also the benchmark ray, resulting in a relatively high frequency of checks.

The grouping of access checks to `self` at method boundaries (categories *conflict-self* vs. *conflict*) can reduce the number of checks, e.g., for the benchmarks hedc and ray. As both benchmarks execute a relatively high number of runtime checks, the optimization is especially profitable. The grouping of checks may however also lead to an increased number of checks and overreporting: if a check at a method boundary is done although no actual object access/update is performed inside the method. We observed that this contingency is not a significant problem in practice.

Runtime performance

	original			*rc (incl array)*		*rc (no array)*			
	SUN jvm [s]	*[s]*	*optimized [s]*	*all*	*conflict*	*all*	*conflict-self*	*conflict*	*optimized*
mtrt	6.4	20.0	19.6	1.33	1.26	1.30	1.24	1.25	1.24
sor	2.6	2.8	2.4	6.96	3.89	1.25	1.00	1.00	1.00
tsp	5.6	7.3	6.0	5.68	1.78	2.92	1.14	1.15	1.10
mold	12.9	18.3	23.8	4.01	1.95	2.73	1.96	1.87	1.17
ray	43.3	44.4	45.8	6.01	3.64	5.85	3.98	3.65	2.27
monte	12.3	21.6	22.8	2.33	1.91	1.97	1.82	1.81	1.78
average				4.39	2.41	2.68	1.86	1.79	1.44

Table 7.5: Runtime of original and instrumented programs. The execution times are normalized to the runtime of *original*, which is specified in seconds; the runtime of *rc optimized* is relative to *original optimized*

.

Table 7.5 shows the execution times of different instrumentation variants. The measurements are done on a Pentium III 933 MHz system with 512 MB of main memory, which is sufficient for each program to execute in main memory. We report the average execution time

of three runs; unless noted explicitly, the duration of different runs varied insignificantly. philo, elevator, hedc, jigsaw, and specjbb are not reported because these benchmarks are not CPU-bound or have a fixed execution time. The runtime of the *original*, i.e., uninstrumented program variants (without the extension of the object header and the auxiliary per object structure) are specified in seconds, the runtime of the instrumented variants are given as a factor relative to the performance of the uninstrumented counterpart, e.g., 1.33 means a 33% overhead.

Column *original - SUN jvm* specifies the execution time on SUN's HotSpot virtual machine [69]. The numbers demonstrate that the quality of the code generated by our static compiler is comparable to a commercial optimizing JIT compiler. The performance for mtrt is hampered by the inefficient implementation of locks in the runtime system we use (GNU libgcj [49]). monte is floating-point intensive and our register allocator and code-generator does not handle floating-point arithmetic efficiently; hence, also the performance of this benchmark is impaired relative to SUN's HotSpot. It is important that the performance of our compiler is comparable to commercial environments, because we report the overhead of the race detection relative to the uninstrumented program variant that is also generated with our compiler; an inefficient baseline variant could hide the actual overhead of the race detection that would be encountered in highly optimizing compile- and runtime environments.

For some of the scientific benchmarks (mold, ray, and monte), the program transformations in mode *original - optimized* and *rc (no array) - optimized*, do not improve but even degrade the execution performance compared to mode *original*. As some of the optimizations (inlining, loop-unrolling and partial redundancy elimination) introduce temporary variables, the effectiveness of the optimizations ultimately depends on the capability of the compiler and architecture to allocate these temporaries in registers or conceive a cache-efficient layout of local variables on the stack. The optimizations in the compiler we use for the experiments are not tuned with respect to this issue.

The overhead in the execution time is determined by two factors: (1) The compile- and runtime provisions that allow to execute specialized code in different contexts incur some runtime overhead (this issue is discussed in Section 7.3). For benchmarks that execute no or very few runtime checks, e.g., sor and mtrt in variant *rc (no array)*, the overhead is entirely due to the method specialization. A more detailed evaluation of the overhead due to method specialization is given in Section 7.3. (2) Additional inline code and library functionality cause a runtime overhead that is roughly proportional to the number of dynamic access checks and lock events reported in Table 7.3 and 7.4: the lower the share of runtime checks, the lower the execution overhead. The sparse instrumentation of conflicting access statements in the case of *conflict*, *conflict-self*, and *optimized* lead to a significant reduction of the runtime overhead vs. *all*. For benchmarks with frequent access to conflicting arrays tsp, and mold, the omission of checks at array access reduces the total overhead significantly.

The runtime overhead of the checker is significantly lower than that of previously reported systems. In the variant where the instrumentation of array access is omitted (*rc (no array) - conflict*), the overhead is between 25% for mtrt and 265% for ray (average 79%). For the optimized variant, the relative is even lower, i.e., between 24% and 127% (average 44%). The benchmark specjbb is not reported because it has a fixed runtime and reports the rate (operations/s) at which the program progressed. We have observed that race detection halves the execution rate of this benchmark in both the optimized and the non-optimized variant.

Accuracy

Table 7.6 shows the number of reports issued during a specific execution. There is one report for every class or runtime instance on which a conflict is detected. The columns *incl array* and *no array* report the results of two program variants, one where arrays are treated like objects, one where arrays are not considered. For the latter variant, the reports are classified as follows (column reports *spurious–benign–harmful*):

- *Spurious reports* can reflect spurious object races, e.g., if object access is actually ordered but not through unique lock protection. Spurious reports can also correspond to true object races that are however no data races: e.g., a report due to access of different variables on the same object (and access to each individual variable is without conflict), or a report due to a static classification of a method as write that actually performed only reads in the checked execution.

- *Benign reports* correspond to true data races that are intended or tolerable in the program.

- *Harmful reports* reflect true and unintentional data races that occurred due to incorrect synchronization. Harmful reports correspond to program defects.

The inclusion of arrays into the race detection generally adds few additional reports that, in most cases, can be related to other reports of objects that these arrays are affiliated with. In tsp, there is a large number of reports for instances of class `TourElement`. Each instance has an `int`-array associated that also appears in the reports if the checking of array access is enabled. Note that the results in mode *rc (incl array)* and *rc (no arrays)* stem from different program runs. The reporting in some benchmarks, and in particular the reports given for tsp depend on the thread scheduling. The number of reports is generally manageable. The reports correspond to incidents that are detected by the static conflict analysis and are discussed in Section 4.5.1.

The report for mtrt refers to a real data race on variable `RayTracer::threadCount` which is however not relevant for the correctness of the program.

In sor, 1000 reports occur for arrays that represent the rows (Java implements jagged arrays) of two shared 500 matrices.

In tsp, 160 reports correspond to instances of `TourElement` that are managed in a pool and reused by different threads. Individual instances are accessed without lock protection, accesses from different threads is however ordered because a reference to each instance is available only to one thread at a time. One of the reports in tsp corresponds to the global variable `TspSolver.MinTourLen` for the minimal tour length found so far. The updates are monotone and double checked, and concurrent reads of outdated information are tolerated by the algorithm.

In hedc, there are 15 spurious races on instances of `java.lang.String`; this is amazing, since objects of that type are actually immutable. However, the static analysis classifies the effect of a polymorphic call to `java.Object::toString` in method `java.util.Hashtable::toString` conservatively as a write of the target object because *some* variant of the callee may modify the target object; there is however no write if the target of the call is `java.lang.String::toString`. Moreover, four spurious reports concern objects that are created through method `clone`; the

	rc (incl array) *total*	*rc (no arrays)* *spurious–benign–harmful*
philo	0	0
elevator	0	0
mtrt	2	0–1–0
sor	1000	0
tsp	287	160–1–0
hedc	34	21–0–10
mold	11	0
ray	2	0–0–1
monte	1	0–1–0
specjbb	3	1–0–2
jigsaw	14	7–1–1

Table 7.6: Conflict reports (each conflicting object instance, array, or class is reported once).

cloning also conveys the header fields used for race detection from the original to the cloned object and this perturbs the race checker. Two further spurious reports are raised due to the invocation of methods `java.net.InetAddress::getByName` and `ethz.util.SystemProperties::getProperty`. Both methods actually access the variables of the underlying class under lock protection through a separate synchronized method. The instrumentation does however not account for this subtlety and classifies both methods (which are not synchronized themselves) as writes in the object access protocol. Ten real race are detected that correspond to errors in the program and the library: Two reports address instances of `java.text.DecimalFormat`; the implementation of this class in [49] is not thread-safe, but the instances are used in a multi-threaded context and hence may function incorrectly. Moreover, there is one report for each of the library classes `gnu.gcj.convert.UnicodeToBytes` and `nu.gcj.convert.BytesToUnicode` that initialize the default instance of the de-/encoder through an incorrect implementation of the double-check locking idiom (i.e., the variable that holds the reference to the unique instance is not volatile). Another five reports point to an unsynchronized assignment of `null` to a shared variable `Task::thread_`, which could be read by another thread and lead to a `NullPointerException`. The details and order of reports for the `Task` objects depend on the specific execution; note however that there can be other benign or spurious reports on the same object such that the checking might be passed in ownership state conflict and the critical report (referring to variable `Task::thread_`) might not appear in the execution log.

The report in ray relates to the update of the global variable `checksum1` in class `JGFRayTracerBench` in a synchronized block. The runtime object that is used for synchronization is however different for each thread; hence this report reflects a synchronization error in the program.

The report in monte is due to the global variable `UNIVERSAL_DEBUG` in class `montecarlo.Universal` which is read and written by different threads without synchronization. In the specific program, no inconsistencies can occur, because all threads write the same value; hence this report is classified as benign.

There is one spurious report in specjbb that refers to an object of instance `spec.jbb.SaveOutput` with unsynchronized methods. The instance acts as a wrapper of an underlying synchronized implementation of a print stream; the report is due to a call to a method of class `spec.jbb.SaveOutput` that is classified as write, although the actual write occurs in

the scope of a downstream synchronized method. Two reports correspond to errors: There is a data race on field `mode` of an instance of class `spec.jbb.Company`. This variable is used to communicate the current phase of the benchmark between the main thread and the workers; the field should be declared as volatile. Another error is found in the library related to an erroneous implementation of the double checked locking idiom for the default instance of class `java.util.TimeZone`.

Two reports in jigsaw refer to instances of the thread class `w3c.jigsaw.http.Client`. The thread is started inside the constructor such that the `run` method may start to execute before the constructor finishes. Although this report points to a censurable programming pattern, the report is benign because the start happens as the last action in the constructor. Two spurious reports occur on instances of class `w3c.jigsaw.http.ClientState` which are artifacts due to race checking at the level of objects: Different fields of the instance are protected by different locking contexts; hence the lockset associated with the object becomes empty, although each individual field is actually protected by a unique lock. There is also a violation of atomicity related to this object (Section 5.3). Two other reports for instances of class `w3c.jigsaw.http.CommonLogger` and `w3c.jigsaw.resources.DirectoryResource` are spurious due to the invocation of methods that are not synchronized, that are classified as write access and effectively that access fields of the underlying object through the invocation of private synchronized methods. A benign race is found on variable `w3c.tools.timers.EventManager::done` which is used to communicate a thread stop notification to a running thread. The variable is set under lock protection but read outside a protecting lock scope. The reading thread does regularly acquire and release the lock that is used when the flag is set, and hence the visibility of the update of the flag is guaranteed. One report corresponds to a genuine error, namely a data race on the global variable `java.lang.System::secman` that refers to the default instance of the security manager. This error in the implementation of the library might unintentionally cause the creation of several security manager instances.

Sources of unsoundness The reporting of the checker can depend on the thread interleaving in a particular program execution, i.e., an actual race that occurred might be hidden in a specific interleaving of access events. This is due to the fact that the checker delays reporting until an object is recognized as *shared modified*. Imagine an execution where the last update of the creator thread occurs just before the access by the second owner thread. Although access may not be ordered through explicit synchronization, the checker transits the object to state *handoff* and does not report a conflict.

Sources of incompleteness Not all reports issued by the runtime checker correspond to genuine synchronization errors. Some of the inaccuracy is due to the general approach of checking (unique) lock-based races and has been discussed in Section 2.2.2. Here we only report additional inaccuracy introduced through object races.

First, there is one lockset per object, assuming that all field variables of a shared object are protected by the same synchronization discipline. This is common for object-oriented programs but not mandatory. Hence our checker would report spuriously a case where, e.g., one field variable of an object is initialized then shared read (without lock), and another field variable is read and written under unique lock protection. Our experiments have shown that spurious reports due to the detection at the object granular-

ity are rare in practice. This observation has also been made by other authors [25, 93].

A second concern is the aggregation of checks for `self`-access at method boundaries. The static analysis classifies the call to a method conservatively as a write if there is some path in the method where a write access to `self` occurs. This classification could mislead the checker in a scenario where the first execution initializes some variable and subsequent executions only read from it.

7.1.7 Related work

Savage at al. [103] developed the Eraser system which is the original implementation of lockset-based data race detection. Since then, the 'Eraser principle' has been adopted by other dynamic race checkers and optimized along two dimensions: First, to reduce the runtime overhead of the checking; second, to increase the accuracy of the reporting. The focus of our work is on the first aspect.

Overhead. Static program analysis is useful to avoid the monitoring of data for which races can be excluded already at compile-time. While this idea has been applied earlier to Fortran programs [83], object-oriented programs are different and open new challenges and opportunities for static analysis. A simple approach determines thread-local data through escape analysis [122], while more elaborate static analyses also approximate access ordering on shared data like [25] and the analysis presented in Chapter 4.

There are two principal directions to reduce the residual checking for access to shared data that are candidates for data races according to the approximation of the static analysis: First, checks to related variables or memory regions can be combined, e.g., at the level of objects [122] or minipages [96]. Second, some checks are redundant and need not to be performed. A check is redundant if some previous access to the same data has been checked in the same thread and the same or a weaker locking context. Choi et al. [25] implement a system that allows to efficiently recognize and avoid redundant checking. The architecture of their system is different from ours: The checking is done by a separate thread which polls access events from access event caches that are filled by and associated with individual threads. This architecture is advantageous for dynamic optimizations in two ways: First most of the core data structures that support race checking can be accessed without synchronization from the unique checker thread. Second, the event caches are used to filter redundant events, hence reducing the work of the checker thread. [25] reports that this technique eliminates almost all access checks. The runtime overhead of this race checker is only 13%, 20%, and 42% for the benchmarks `sor`, `mtrt`, and `tsp` (the overhead in our optimized implementation is 4%, 24%, and 10% (*array access is not checked*). Some of the redundant checking can already be identified and removed at compile-time ([25] and Section 7.1.4).

Accuracy. The original lockset algorithm [103] is not effective in recognizing synchronization through mechanisms other than locks. For situations like object-migration and barrier synchronization, the algorithm reports false positives. Pozniansky and Schuster [96] extend the original lockset algorithm [103] to accommodate barrier synchronization; the key idea is to re-initialize the locksets to hold all possible locks once a barrier is executed.

In addition, [96] and also [93] combine happened-before race detection with the lock-based technique (this combination was originally proposed by Dinning [36]). In these so-called *hybrid* data race checkers, a race is reported if two accesses violate the lockset criterion *and* there is no happened-before ordering.

7.1.8 Discussion

Low-overhead dynamic data race detection is an important mechanism to support the debugging of parallel programs. We have designed an automated race detection system that is integrated with the compiler and exploits structural properties of object-oriented programs.

While the majority of reports obtained by the object race checker are spurious, the source of the report and the classification could be done with without difficulty. The claim that "object race checking leads to the reporting of many false races" [93] can be rebutted as follows:

- The overall number of reports is manageable and is lower than in a previous version of our online object race checker [122], which was not guided by detailed static conflict analysis. E.g., the system in [122] instrumented and reported potential race conditions in a number of situations where the static conflict analysis (Section 4.4) recognizes ordering through thread-start, thread-join, or init-escape.
- Many of the false race reports, e.g., for tsp, are not due to particularities of object race checking, but due to the underlying lock-based race checking principle that accounts only for synchronization based on monitors.
- Some of the reports, even if spurious or benign, point to weaknesses in the application and synchronization design, e.g., for jigsaw. Though spurious, these reports can be useful to the programmer!
- Two sources of spurious reports in the object race checker could be eliminated through a simple extension of the implementation: (1) The read/write classification at polymorphic call sites could be resolved separately for each target method; this would remove the reporting of the `java.lang.String` instances in hedc. (2) Access to `self` that occurs in the scope of a synchronized block or method should not be attributed to the encapsulating respectively the calling method. This procedure would remove some spurious reports in the benchmarks hedc and jigsaw.

7.2 Detecting violations of object consistency

This section defines *object consistency* (OC) [58, 4, 95, 23, 121, 123] which is a high-level memory model that guarantees the absence of thread interference on objects at runtime. We have designed a compiler and runtime system that enable object consistency through a program instrumentation and detect violations due to insufficient thread synchronization at runtime. In contrast to mechanisms for dynamic data race detection, the system does not aim at finding all synchronization errors in an execution but only those situations where the impact of insufficient synchronization becomes apparent in a violation of the high-level memory consistency model. This philosophy is similar to a system for the detection of violations of sequential consistency

proposed by Gharachorloo and Gibbons [51]: Either a specific execution complies with the memory model (object consistency in our case, sequential consistency in [51]), or the system issues a report. Our system is implemented in software and requires, apart from memory fences, no specific hardware support.

7.2.1 Object consistency

With sequential consistency, no thread ever sees an inconsistent state of a variable (it always observes the value of the most recent write). OC extends this model to objects. The central idea is that the intermediate object state during method execution must not become visible to concurrent threads. Hence this model guarantees *atomicity* of object access (method or field) to the programmer. Object access is *sequentially consistent*, i.e., the execution behaves as if there was a global access order that is compatible with the program order of individual threads.

OC has originally been introduced [4, 23] as a programming model that users should follow by applying appropriate synchronization to a parallel object-oriented program. Our system checks if a specific execution follows this model. Hence, the checker is charged with verifying an *object access discipline*.

In the OC model, there is no notion of consistency for arrays and hence array access is not subject to a specific access discipline. Moreover, objects that are used as monitors (actually condition variables) have to be treated specially because atomicity of method execution can sometimes be violated on purpose (synchronized methods, invocation of `wait`).

7.2.2 Object access discipline

A simple approach to achieve OC is to require that object access is serialized and naturally leaves the object in a consistent state. This discipline can be unnecessarily restrictive, because according to the definition of OC, only object access for which different execution orders leave the object in different final states [23] must be serialized.

We pursue a pragmatic approach and check that mutual exclusion is fulfilled for field and method accesses that read or write the internal state of the object; we allow read-read overlap and also concurrent unordered access to variables that the programmer declared as volatile. With this access discipline, all actual violations of OC are detected; in addition, some object access with benign overlap (e.g., same object but different variables) also violates the access discipline. Hence there is no *underreporting*, but there might be some *overreporting*. Overreporting is moderate and has not been a problem in our experience (Section 7.2.6). To simplify the presentation, we use the terms ”violation of OC” instead of ”violation of the object access discipline” if the distinction between both is apparent or irrelevant.

7.2.3 Object access protocol

The compliance of an execution to the OC object access discipline is verified through an object access protocol. The following auxiliary data structures are associated with each object to facilitate the checking for the access discipline:

- $o.t$ the thread that is currently active on the object o.

```
init(o)
    o.t = null;
    o.w = false;
```

Figure 7.6: Object initialization.

- $o.w$ flags if the current thread writes to the object.

Additionally, four local variables t_{before}, t_{after}, w_{before}, and w_{after} are reserved in each method to temporarily store thread and write information about an accessed object.

Object initialization

At object allocation, there is no active thread on the allocated object and the write flag is not set. The procedure *init*() summarizes the actions that immediately follow the allocation of an object o (Figure 7.6).

Object access

The object access discipline is then checked at field access statements and method access. The checking is split into an access prologue and epilogue (Figure 7.7). Let e be the access event, o the accessed object, and t the accessing thread.

The access prologue stores the existing thread and write flag of the accessed object and installs the current thread and write flag corresponding to the access. The checking for a violation is done in the epilogue. To avoid that the actual access and the actions of the access protocol are reordered, a memory fence (`mfence`) is inserted after the prologue and before the epilogue.

In the epilogue, two checks are done: (1) the active thread on o that is determined after the access (variable t_{after}) must be the same as the thread executing the epilogue (t). (2) The active thread before the epilogue (variable t_{before}) must be `null` or t; in the latter case the access e is reentrant. A report is issued if one of the test fails and at least one access that overlapped is a write.

Treatment of monitors

Objects that are used as monitors are treated specially by the checker because in such case, overlapped object access can be intentional and should not be reported as a violation of OC. Consider, e.g., some thread t_1 that invokes a synchronized method and temporarily leaves the monitor through a call to `wait`. Another thread t_2 can enter the monitor, alter the state of the object, call `notify`, and finally leave the monitor. An object access protocol that is ignorant of monitors would report an OC violation. Hence, the following adaptations are made to accommodate the special case of monitor objects:

- There is no instrumentation at the call site of a synchronized method. Instead, the prologue is executed immediately after the monitor lock is acquired; the epilogue is executed immediately before the method returns.

```
/* prologue */
atomic:
   t_before = o.t;
   w_before = o.w;
atomic:
   o.t = t;
   if (is_write(e))
      o.w = true;
mfence;

/* access */
⟨thread t issues access event e to object o⟩;

/* epilogue */
mfence;
atomic:
   t_after = o.t;
   w_after = o.w;
atomic:
   o.t = t_before;
   o.w = w_before;
if (t_after ≠ t ∨ (t_before ≠ null ∧ t_before ≠ t))
   if (is_write(e) ∨ w_before ∨ w_after)
      new Report(e,o,t);
```

Figure 7.7: Object access prologue and epilogue.

- Invocations of `java.lang.Object::wait` are instrumented specially, namely in the *inverse* way of Figure 7.7: *Before* the call, the "epilogue" restores the header fields of the object prior to entering the monitor; *after* the invocation, the "prologue" re-acquires the monitor object and the thread continues with its activities on the monitor object.

- The start of a block monitor is treated like the entry of a synchronized method; the end of a block monitor is treated like the return from a synchronized method.

7.2.4 Verification

We have used the SPIN model checker [65] to validate that the protocol correctly detects overlapped object access, irrespective of the thread interleaving. The model in Program 7.1. is configured with 4 concurrent threads; three global variables represent the header of the accessed object (`header`), the accessed field variable (`ctr`), and a flag that marks if a report is issued (`report`). All threads execute the prologue, the access itself, then the epilogue. The double colon (`::`) means selection, i.e., one of the statements inside an `if/fi`-block is executed. In the model, the statements following the choice is guarded, i.e., the second part (after the $->$) is executed after and if the guard (before the $->$) evaluates to `true`. Finally, the assertion de-

Program 7.1: Simplified model of OC protocol as input for the SPIN model checker.

```
#define NUM_THREADS 4

byte header = -1;
byte ctr = 0;
bool report = 0;

active [NUM_THREADS] proctype user()
{
       byte before;
       byte after;

        /* prologue */
        before = header;
        header = _pid;

        /* access */
        ctr++;

        /* epilogue */
        after = header;
        header = -1;
        if
        :: (after != _pid) -> report = 1;
        fi;
        if
        :: (before != -1 && before != _pid) -> report = 1;
        fi;

        assert (report == 1 || ctr == NUM_THREADS);
}
```

mands that not lost update occurs (`ctr == NUM_THREADS`) or a report is issued (`report == 1`). Overreporting is allowed, i.e., both conditions may become `true`.

The given model passes the checker without violating the assertion. This means that any sequentially consistent execution of the program maintains the invariant. The model in Program 7.1 is simplified for the purpose of presentation. The actual model that we used to validate the protocol is more complex and also accounts for reentrant object access.

7.2.5 Implementation

Instrumentation

The instrumentation is *sparse* and only targets those object access statements that participate in an access conflict, i.e., correspond to a conflicting event in some OUG (Section 4.4). Access to `self` needs no instrumentation.

Code generation

This section discusses issues of code generation for the object access protocol on the Intel IA-32 platform [67]. The inline code sequences are designed such that the most frequent cases are

	OC
field access	4.8
field access (shared read)	7.2
static call	2.3
virtual call	4.0

Table 7.7: Execution time of object access with instrumentation for OC checking relative to the un-instrumented access.

handled most efficiently and do not require a library call. We enhanced the IA-32 code generator and specified code templates corresponding to inline code. The protocol requires thread identification information to enable threads to distinguish their own activity from the activities of other threads. We use high-order bits of the stack-pointer (`%esp`) as a unique identification.

The instrumentation of the access prologue requires 3 respectively 5 instructions for read respectively write access; there is one read, one write access to memory and a memory fence. The epilogue is 15 assembly instructions long. The common cases (no violation of OC) requires a memory fence, one write, one read memory access, and 5 integer arithmetic instructions. Overlapping reads are resolved in a library call and hence are slightly more expensive (see micro benchmarks below).

Memory ordering

On modern multi-processor architectures the program actions and the actions of the object access protocol might appear out of order from the perspective of different threads. This can impact the detection accuracy such that harmful thread interference (e.g., lost update) might remain undetected. As the object access protocol should reliably detect any actual interference, ordering between prologue, actual object access, end epilogue must be enforced. This is done through a memory fence (Figure 7.7). On the IA-32 architecture [67], we use the `lock` instruction prefix to achieve the effect of a bidirectional memory barrier.

Micro benchmarks

We use micro benchmarks to assess the execution overhead due to the program instrumentation. The methodology is the same as in Section 7.1.4. Table 7.7 reports the execution times of an instrumented access relative to its un-instrumented version. For field access, the most significant portion of the overhead is due to the memory fences in the inlined code sequence. On the Pentium III architecture that we used for the experiment, the cost of a memory fence is several times the cost of an ordinary memory access. Concurrent reads are relatively more expensive because this case is not handled by the inline code such that a branch into the runtime library is necessary. The cost of an access that leads to a report is in the same order; only one report is issued per object. The absolute overhead of a static call is the same as for a field access (as a call is generally more expensive than a field access, the relative overhead is lower for the call). Polymorphic calls are again relatively more expensive than static calls due to the call resolution through `instanceof`-chains (details in Section 7.3).

7.2.6 Experience

Performance

We investigate three different program variants that differ in the degree of instrumentation:

- *all:* In this category, all sites that do not access `self` are instrumented. The instrumentation does not account for different access contexts or access classifications by the static conflict analysis.
- *conflict:* This variant limits the instrumentation to sites that that do not access `self` and participate in access conflicts.
- *optimized:* Similar to *conflict*, however the program as well as the instrumentation are optimized (Section 7.1.4).

	oc (no array)	
	conflict	*optimized*
philo	0.0	0.0
elevator	20.4	20.4
mtrt	< 0.1	< 0.1
sor	0.0	0.0
tsp	23.0	18.7
hedc	9.7	2.4
mold	< 0.1	< 0.1
ray	72.6	28.2
monte	0.2	0.2
specjbb	34.9	27.2
jigsaw	12.9	12.8

Table 7.8: Frequency of access checks for different instrumentation variants.

Table 7.8 shows the frequency of dynamic access checks relative to variant *all*. Apart from ray, the static analysis reduces the numbers significantly. Optimization is especially beneficial for ray, where partial redundancy elimination eliminates a large number of the field access events and their associated checks.

	original		*oc (no array)*		
		optimized	all	conflict	*optimized*
mtrt	20.0	19.6	1.09	1.05	1.09
sor	2.8	2.4	1.00	1.00	1.00
tsp	7.3	6.0	2.74	1.41	1.33
mold	18.3	23.8	2.69	(0.93)	(0.64)
ray	44.4	45.8	3.99	3.12	1.91
monte	21.6	22.8	1.44	1.38	1.29
average			2.15	1.59	1.25

Table 7.9: Runtime of original and instrumented programs.

Table 7.9 shows the runtime of different instrumentation variants. The time for instrumented programs in columns *oc (no array)* is given as a factor of the execution time of the uninstrumented program (*original*); *all* and *conflict* are relative to the un-optimized, *oc (no array)* -

optimized is relative to the optimized program variant. The measurements are done on a Pentium III 933 MHz system with 512 MB of main memory, which is sufficient for each program to execute in main memory. We report the average execution time of three runs.

The cost of OC varies across the benchmarks, depending on their access characteristics. In sor, e.g., most access event target arrays or `self` hence the cost of OC checking is negligible; the static analysis determines no conflicts on classes or object instances and hence not a single check is executed in the variants *conflict* and *optimized*. The situation for ray is the opposite: A large number of field accesses to objects that are not `self` result in very frequent checking. The static analysis is not able to determine the absence of conflicts for some core data structures and hence the overhead remains significant (212% in conflict and 91% with *optimization*).

The performance of moldyn is very sensitive to cache performance and in particular the number and layout of local variables in one of its very large methods (`moldyn.mdRunner::run`). The number and order of local variables varies with the degree of instrumentation and optimization; hence the execution times of the instrumented variants are improved due to the fortunate coincidence of a cache compliant variable layout. Our compiler is not tuned to address this issue.

Reporting

Table 7.10 classifies the OC violations reported for a specific program run. The reports are classified as follows (column reports *spurious–benign–harmful*):

- *Spurious reports* do not correspond to actual violations of OC; such reports occur, e.g., due to the static classification of a method as write that actually performed only reads in the execution.
- *Benign reports* correspond to actual violations of OC that are intended or tolerable in the program.
- *Harmful reports* correspond to actual violations of OC that reflect synchronization defects.

	total	*spurious–benign–harmful*
philo	0	0
elevator	0	0
mtrt	0	0
sor	0	0
tsp	0	0
hedc	9	0–1–8
mold	0	0
ray	0	0
monte	0	0
specjbb	2	0–1–1
jigsaw	14	2–12–0

Table 7.10: Violations of the object access discipline.

There are generally very few reports, which confirms that object consistency is mostly followed in practice. The reporting may slightly vary between different program runs due to scheduling dependence.

8 reports in hedc concern thread objects. These threads provide a method `Task::cancel` that allows other threads to issue a termination signal. The `run` and `cancel` methods may hence be executed with overlap on the same thread object by different threads and hence a violation of OC is reported. This is also the context where the critical unsynchronized assignment of `null` to a shared variable `Task::thread_` occurs that could be read by another thread and lead to a `NullPointerException`. Hence these reports are classified as errors. Another report is spurious and addresses class `java.net.InetAddress` where the unsynchronized method `getByName` is accessed concurrently. The implementation of this method is, however, based on other synchronized methods and is safe.

There are two reports for specjbb: There is an error report that addresses an instance of class `spec.jbb.Company` where field `mode` is used to communicate the current phase of operation to the transaction managers of individual threads. The variable is updated and read concurrently and should be declared as volatile. There is another report for the global variable `myCompany` in class `spec.jbb.JBBmain`. The variable is read by different threads while the `main` method is active. The initialization occurs before the start of the threads and hence this report is benign.

12 benign reports in jigsaw occur in a situation where a thread is started, hence the `run` method starts to execute while the constructor of the thread has not terminated. All situations do not represent an actual problem, because no critical access occurs after the start. One spurious report concerns an instance of class `ClientState` where disjoint set of fields are accessed concurrently under different locking contexts. Another spurious report is given for an instance of class `DirectoryResource` where an unsynchronized method is classified as a write access, the actual variable access is however done in the scope of a private synchronized method.

7.2.7 Related work

This section discusses fundamental approaches at different layers of a system to compensate and hide the programmer-visible effects of low-level memory access optimizations. Hardware-centric SC systems [51, 60] are not included since, due to the combined effects of compiler and runtime system, hardware-based solutions can simplify but not entirely resolve the aspect of memory consistency at the programming level.

Weak memory systems. Weak memory systems [2, 52, 12, 66, 48] constitute a *programmer-centric* [50] approach to hiding the effects of buffering and reordering of memory accesses from the user. Weak memory models are defined from the hardware perspective and thus are not specific about programming language aspects. For a certain class of programs where synchronization is present, e.g., according to the conventions imposed by Release Consistency [52], the memory abstraction for the user is SC. Restricting the scope of programs for which guarantees are made is however a limitation of weak memory systems: The property of compliance to the programming model is undecidable and cannot be efficiently determined from program executions either.

Our approach of OC differs from Scope Consistency [66] and Entry Consistency [12] in that programming conventions for these memory models define memory scopes based on critical regions (i.e., program segments) that are specified by the programmer, whereas our approach uses a partitioning based on the object space that is determined automatically by the compiler.

Concurrent object systems. Various systems (including Amber [20], Orca [10], SAM [104], Concert [22, 95, 23]) have exploited the idea of user-defined objects as the basis for sharing. These systems have been successful in mapping parallel programs onto memory systems that are far more complicated than the shared memory systems that are the target of our work and have even targeted distributed memory machines (where explicit copy or transfer operations are necessary to maintain a shared object space). These systems define tailored programming languages and require system specific annotations by the user, e.g., about available parallelism and how data will be accessed.

Compiler-based memory consistency. Lee et. al. [76] investigate constraints on basic compiler optimizations in the presence of access conflicts. In [75, 84] a compile-time algorithm for the sparse placement of memory fence instructions is described. This procedure guarantees a specific memory behavior in explicitly parallel programs with access conflicts. The work proposes the design of an optimizing compiler that exploits information about variable sharing and the memory model of the underlying hardware. [84] does not quantify the cost of the analysis and the runtime impact of the program instrumentation. Our work is complementary to the work of Lee et. al.. We focus on the identification of shared data and structural protection guarantees. Our goal is to provide a pragmatic memory model with access checking (OC) for Java while limiting the impact of the instrumentation on the execution performance.

7.2.8 Discussion

Object consistency is a *high-level memory model* that addresses not only interference on individual variables but also the aspect of access atomicity on individual objects. This notion of atomicity is however different from the method-level atomicity discussed in Chapter 5: the scope of object-level atomicity extends only to specific objects, whereas method-level atomicity requires atomic treatment of all data that are accessed in the dynamic scope of a method. This narrowing the scope of atomicity to individual objects opens new opportunities for the efficient implementation of a runtime checker.

We have developed a runtime systems that checks for the compliance of an execution to OC. The system is fully automated and assumes OC as execution model for all objects. The reporting is *safe*, i.e., every violation of OC is recognized and reported. The number of reports is generally low for the Java programs we studied. This confirms our conjecture that OC also can be understood as a programming model that is commonly followed when arranging the synchronization in a multi-threaded object-oriented program. While only few reports pointed out genuine error conditions, most of the reports unveiled flaws in the design of the application that should be improved.

Despite the compiler support, optimization, and efficient inline instrumentation, the runtime overhead of the checker is still significant (9–91%, avg. 25%). This overhead is mainly due to the software-centric implementation. Memory fences are the major factor contributing to the overhead; more selective hardware mechanisms such as non blocking synchronization primitives (e.g., the load-linked (LL) and store-conditional (SC) instructions [71, 57]) would allow a more efficient implementation of OC checking. The rationale of these hardware extensions is to expose information about cache coherence actions to a program for the purpose of interference detection.

7.3 Method specialization

This section discusses a code specialization technique that enables to execute different variants of a method in different calling context. The technique is general with respect to the notion of context, i.e., different criteria could be used to distinguish invocation contexts.

We use the technique here to implement the object access protocols discussed in Sections 7.1.3 and 7.2.3. The criterion for context disambiguation is the conflict property that the analysis in Chapter 4 associates with the abstractions of heap allocated data. This procedure allows to limit the execution of the object access protocol to objects and access sites that are classified as conflicting.

The effectiveness of the method specialization in reducing the execution frequency of the object access protocol depends on the the contexts in which a certain data type is used: If the same type of data structure is used in different contexts and access conflicts are only found in few contexts that are rarely executed, the method specialization is highly effective in reducing the number of access checks. Otherwise, if, e.g., all instances of a certain type are conflicting, most or all access to that type should execute the object access protocol and hence method specialization is without effect; in this case, all access sites of a certain object type could be instrumented.

7.3.1 Example

Program 7.2 illustrates context sensitivity and motivates the necessity of context-sensitive method specialization. The program creates two instances of class `Flag` and accesses them in different contexts. The instance allocated at line 22 is thread-local and is accessed only by the main thread. The instance allocated at line 23 is shared. The static conflict analysis (Chapter 4) reveals that there is a conflict on field `Flag::b` between the read access at line 35 and the update in the scope of the call at line 27.

A simple, context-insensitive instrumenter would insert the prologue and epilogue of the access protocol at all access sites of field `Flag::b`. This is possible but might entail unnecessary checking (the protocol is also applied to object access that is thread-local) or spurious reporting (access statements that occur before the escape would also be instrumented and this may mislead the access protocol).

An ideal instrumenter would rearrange the program such that the object access protocol is executed at runtime only for conflicting objects at conflicting access sites. The context-sensitive method specialization approximates this goal by generating specializations of method respective their invocation contexts and by adapting the calling structure of the program accordingly; the procedure is not perfect though and some unnecessary instrumentation may be executed under certain circumstances where the static program analysis looses context sensitivity, e.g., inside in a recursion.

7.3.2 Implementation

The classification of object access statements depends on the heap context within which the method that performs the access is called. Specializations are created in 4 steps:

Program 7.2: Example of a program where instances of class `Flag` are used in different contexts.

```
1  class Flag {
2      boolean b;
3
4      Flag() {
5          b = false;
6      }
7
8      void set() {
9          b = true;
10     }
11
12     void unset() {
13         b = false;
14     }
15 }
16
17
18 class Specialization extends Thread {
19     Flag s;
20
21     static void main(String args[]) {
22         Flag local = new Flag();
23         Flag shared = new Flag();
24         Thread t = new Specialization(shared);
25         t.start();
26         local.set();
27         shared.set();
28     }
29
30     Specialization(Flag s) {
31         this.s = s;
32     }
33
34     void run() {
35         if (s.b)
36             System.out.println("main␣is␣done");
37     }
38 }
```

1. First, all heap contexts are identified within which a method operates. This information is recorded during the symbolic execution (Section 3.4). Every heap context provides a *specialization candidate* of that method.

2. Then, the conflict analysis determines the conflict properties of abstract objects and individual access sites. (Chapter 4.4).

3. Then specialization candidates are classified and grouped according to the properties determined by the conflict analysis.

4. Finally, method specializations are determined by combining specialization candidates that operate in equivalent abstract contexts; call sites are adjusted to invoke specializations if necessary. In specialized methods, the resolution of polymorphic calls must consider not only the type/compile-time properties of the target object (as done with different vari-

ants of vtables), but also the calling context. Our implementation unfolds polymorphic call sites to cascades of `instanceof` checks.

The implementation does not specialize thread root methods (`main` and `run`) because they are directly called from the runtime system and hence their call sites cannot be changed; only for thread root methods, the calls and instrumentation are adjusted directly in the original version of the method. All other methods that have instrumented access statements or call methods with instrumented access statements are specialized themselves. Their original versions are not instrumented in our current implementation; consequently call-backs from native methods or the runtime system branch into un-instrumented code. Although this shortcoming of our current implementation may affect the detection accuracy of the object access protocol, we have not found it to be a critical issue during our experiments.

Example In the simple example of Figure 7.2, the context-sensitive method specialization would apply the following code transformations:

- Ultimately, the conflicting access to the shared instance of `Flag` happens in methods `Flag::set` and `Specialization::run`. The former method is specialized and hence a variant `Flag::set.1` is created that contains the access instrumentation; the original method remains unaffected. The latter method cannot be specialized (it is a thread root method) and hence the access instrumentation is inserted directly.
- Call sites that invoke `Flag::set` on the shared instance of `Flag` must be updated. This concerns the call at line 27 of method `Specialization::main`. As `Specialization::main` cannot be specialized itself (it is a thread root method), the site is updated directly in the original version of the method to invoke `Flag::set.1` instead of `Flag::set`.

7.3.3 Experience

Table 7.11 specifies the number of methods and specializations in the benchmarks. Column *methods total* specifies the number of methods in the call graph. A large fraction of those methods require to be specialized (column *methods spec*) because they perform access that needs to be instrumented, or they invoke methods that are specialized. Column *specializations* gives the total number of specializations, as well as the maximum and average number of specialization per specialized method. In hedc, specjbb and jigsaw, the imprecision of the compiler in assessing the call structure and bounding the polymorphism may lead the specializer to tailor method specializations to specific execution contexts that never occur at runtime. Hence in these benchmarks, a number of specialized methods will never actually execute at runtime.

The programs philo and sor show another inefficiency of our current implementation of the specializer: Although the static analysis shows that there are no access conflicts and and hence no checking and code specialization would be necessary, method specializations for the case "no instrumentation" are generated and executed. This leads to some (unnecessary) overhead that is an artifact of the current implementation.

Column *code bloat* specifies the growth of the executable file with context-sensitive method specializations relative to the size of the original program. Note that the executable does not

	methods		*specializations*			*code bloat*	*runtime (rc)*		*runtime (oc)*	
	total	*spec*	*total*	*max*	*avg*	[%]		*frac* [%]		*frac* [%]
philo	192	52	65	3	1.25	3	-	-	-	-
elevator	311	76	105	5	1.38	5	-	-	-	-
mtrt	719	345	504	10	1.48	21	1.26	> 99	1.05	90
sor	206	14	14	1	1.00	1	1.04	< 1	1.00	< 1
tsp	299	68	93	4	1.37	5	1.12	20	1.03	7
hedc	1021	720	3891	84	5.40	175	-	-	-	-
mold	224	83	110	4	1.33	6	1.83	93	1.64	n/a
ray	270	111	151	4	1.36	7	1.44	18	1.26	12
monte	441	165	196	3	1.19	8	1.47	56	1.37	98
specjbb	1398	978	2270	46	2.12	77	-	-	-	-
jigsaw	1396	972	1841	53	1.89	58	-	-	-	-
average						33	1.36		1.22	

Table 7.11: Benchmark characteristics with context-sensitive method specializations.

only contain the methods specified in the call graph (column *methods total*), but the complete set of methods of all standard library classes (libgcj version 2.96 [49]) that are used by the application program or runtime system. The code bloat is generally low and slightly increased for benchmarks with deep call hierarchies and an allegedly high degree of polymorphism (hedc, specjbb, and jigsaw).

Columns *runtime* quantify the overhead due to method specialization. We report the program variants *rc* and *oc* separately: The variants correspond to the programs evaluated in Sections 7.1.6 and 7.2.6, the inline code instrumentation at critical access sites is however omitted (to isolate the overhead due to method specialization). While both program variants (*rc* and *oc*) have the same structure of method specializations, *rc* reserves one register for the id of the current thread. Hence variant *rc* is generally slower than *oc*, and the difference can be attributed to the reserved register that is not available to standard register allocator.

Each of the columns *runtime (xx)* specifies two values: The first value is the execution time relative to the original benchmarks (Table 7.5 and 7.9). The second value (columns *frac* [%]) specifies the fraction of the overhead of the race checker (*rc*) respectively object consistency (*oc*) that is due to the method specialization.

The runtime cost of context-sensitive method specialization is due to cascades of `instanceof` checks at polymorphic call sites. Benchmarks with frequent dynamic method dispatch (Table 7.3) are specifically impaired, e.g., mtrt and monte. For mtrt, the net overhead of object race checking can almost entirely be attributed the method specialization. There are several directions for improvement: (1) The static call resolution could be refined; our current technique is context-insensitive (the variable-type analysis, VTA [115]) and has only a coarse approximation of alias information. (2) The current implementation of dynamic sub-type checking is not tuned.

Despite only few dynamic method resolutions (Table 7.3), the performance of moldyn seems particularly hampered through the code specialization. The performance degradation is however due to another reason: moldyn is very sensitive to cache performance and in particular the number and layout of local variables in one of its very large methods (`moldyn.mdRunner::run`). The layout is different for the original and the program variant with specialized methods. We do not report the fraction of the specialization overhead for variant *oc* of benchmark moldyn because we observed that the in-

strumented variant executes faster than the original variant due to cache effects (Table 7.9).

The runtime of the benchmarks philo, elevator, hedc, jigsaw, and specjbb is not reported in Table 7.11 because these benchmarks are not CPU-bound or have a fixed execution time.

7.3.4 Related work

There are several techniques to distinguish different method execution contexts in object-oriented programs at runtime. The implementation strongly depends on the notion of context used by the static analysis. Hence not all variants that we describe are applicable and equally effective for different analyses.

Choi et. al. [24] develop a solution to suppress unnecessary synchronization on thread-local objects. The notion of context is given by object allocation sites; the compiler classifies sites in a conservative manner as local or global; this classification is denoted in a header flag when an instance is allocated at runtime. On a synchronization access, the flag is checked and a real synchronization action is only executed if the flag specifies a global object. In this approach, a small runtime cost remains for synchronization access to local objects.

For the same purpose, Bogda and Hölzle [15] propose the creation of different class variants for local and global instances. Unlike [24], this technique does not require to modify the runtime environment or virtual machine and eliminates any overhead associated with synchronization. Each class is subclassed such that the synchronization related to instances is eliminated in the subclass implementations. At object allocation sites that are local, the subclass instead of the original class is allocated and dynamic method dispatch ensures that methods without locking are invoked on that instance. Several subtleties of the Java language, like final classes, final methods, and private fields, complicate the implementation of this approach but can be overcome [15].

Whaley and Rinard [128] implement their context-sensitive synchronization elimination and stack allocation with method specialization – a technique that is similar to our implementation. In their case, the notion of context is determined through access paths from the allocation site of an object to access sites of synchronized methods (or synchronized blocks). Methods along these paths are specialized by the compiler such that synchronization on the object is eliminated and call sites of methods along these paths are adjusted to invoke specialized method variants (the corresponding procedure is applied for stack allocation).

7.3.5 Discussion

Context-sensitive method specialization is useful to tailor code to specific object contexts. We use the technique to differentiate between execution contexts that operate on shared conflicting and other data. Access to shared conflicting data can be furnished with an access protocol that determines actual conflicts or memory consistency. This flexibility comes however at a certain cost: In our implementation, which is not tuned, the overhead relative to the original program is on average 22% (variant *oc*) in time and 33% in code space.

The two object access protocols that are presented in Section 7.1 and 7.2 do not depend on the availability of the code specialization: A simple approach would furnish every poten-

tially conflicting access site with an object access protocol, irrespective the object contexts in which this site executes. While this approach saves the cost of method specializations, there could be redundant executions of the object access protocol, namely in contexts where the compiler would have already deemed the access protocol as unnecessary. Hence, if performance is critical, the use of code specialization has to trade off between these two cost factors.

For the benchmarks at hand, we used the context-sensitive code specialization mainly as a research vehicle to precisely assesses the effect of actual data sharing and access conflicts. Access conflicts are mainly found on application data that tend to be used in uniform contexts (i.e., either in a shared conflicting or some non-conflicting context). In this scenario, the cost of the code specialization might not have been offset by the benefit of avoiding redundant checks and hence the simpler lump-sum instrumentation approach might have resulted in even less overhead for the object access protocols.

8

Conclusions

The way to correctly synchronized parallel programs is thorny and full of pitfalls: Too much synchronization can lead to liveness problems (deadlock), too few synchronization can lead to safety problems (data races, violations of atomicity). The problem of correct synchronization is more relevant than ever in the context of recent developments in hardware architectures that foster multi-processing capabilities [39, 119] and programming languages enable explicit multi-threading at the software level (Java and C♯). Hence it is a legitimate concern that a new class of software error not known in sequential programming, namely synchronization defects, spreads and puts software quality at a risk. This concern is confirmed by the study in this work and related work, e.g., [25, 54, 93, 45], that finds a number of synchronization defects in common application and benchmark programs.

Ideally, the quest for synchronization defects should be done statically and the checker should have have two properties: It should be sound, i.e., all errors are detected, and complete, i.e., all reports correspond to genuine errors. There are however three major impediments that make it difficult to achieve these goals in practice:

- The presence of synchronization errors is defined on the basis of a program execution. Hence, tools for static program analysis are challenged to deduce the potential of a defective execution from the program text. While it is already difficult for sequential programs to determine the existence of certain program executions (input dependence), the situation is even more difficult for executions of multi-thread programs that are additionally characterized by interleaving of individual threads (scheduling dependence).

- Synchronization patterns are manifold and a precise static analysis has to recognize such patterns. Popular programming languages like Java and C♯ are extensions of sequential languages and offer low-level synchronization primitives (locks, volatile variables) that do not reflect the high-level synchronization discipline to a static analysis tool.

- Two important classes of synchronization defects (data races and violations of atomicity) target the unwanted interference of threads on common shared variables; hence the precise disambiguation data and threads at compile-time is important. In object-oriented programs, this disambiguation cannot be easily established: Shared memory is commonly accessed through indirection, such that aliasing can lead to imprecision in the static approximation of a runtime situation. Moreover, threads are commonly modeled as objects (active objects) and hence similar problems arise in assessing the lifetime, structure, and multiplicity of threads.

Despite these impediments, this dissertation shows on the example of an existing programming language (Java) that a fully automated detection of synchronization defects is possible in practice, i.e., at an acceptable cost and a useful accuracy. There are two essential factors for success: First, the flexible static analysis framework that allows to determine possible program executions and their essential properties. Second, the close integration of the static analysis with a sparse program instrumentation that enables efficient runtime monitoring and dynamic checking.

8.1 Summary and contributions

Static analysis framework. We have designed a static analysis framework for multi-threaded programs that approximates the effect of all possible program flows and thread interleavings: The program is simulated along a symbolic execution on an abstract domain of data and threads. The criteria for defective synchronization in an execution can be mapped onto the abstract domain such that synchronization defects can be recognized during the symbolic execution. The design of the abstract domain for data and threads is thereby a critical factor to the efficiency and accuracy of the static analysis: First, we found that precise points-to and alias information is of key importance. Second, it is highly beneficial to the efficiency of the analysis to narrow the focus of the analysis to certain data of interest.

Framework instantiation. The static analysis framework is used to implement three algorithms for the detection of (1) data races, (2) atomicity violations, and (3) deadlock in concurrent object-oriented programs. The novel aspect is that all algorithms have been originally designed to operate at runtime and are now adopted for static analysis: (1) the data race detection is based on the lockset algorithm that has been initially used by an online race checker called Eraser [103]; (2) the detection of atomicity violations is motivated by the principles of an online checker for high-level data races [8]; (3) the detection of deadlocks is based on the cycle detection in a resource allocation graph, which is a common procedure of online deadlock detection [106].

Evaluation. The accuracy of the static analysis is evaluated and related to observations in actual program executions. As a result, a detailed study of several programs (up to 30 KLOC) and their access behavior to local and shared data is done. This study sheds light on the sources of inaccuracy and the intrinsic limits of static (concurrency) analysis.

The design of the checkers for the three types of synchronization defects is a trade-off between accuracy and efficiency (none of the procedures is sound or complete). The runtime of the static analyses is comparable to common static program analyses done in compilers. Underreporting has not been an issue for the benchmark programs, and the number of spurious reports is typically low and amenable to manual inspection.

Two methods are useful to assess the reports and filter out false positives: First, some reports are due to false aliasing and refined alias and type information can help to screen out reports manually. Second, a dynamic checker can assess the runtime situation in critical cases. Our experience shows that the overhead for the residual dynamic checking is very low, i.e., far below the overhead of previously reported dynamic checkers.

Runtime mechanisms. As part of the compiler framework, a code specialization technique has been designed that allows to execute different code in the context of objects with different compile-time classification. Based on this code specialization mechanism, two techniques are presented that make information about concurrency and locking available at runtime: An online race checker (object races) and a checker that assesses the compliance of an execution with a high-level memory model (object consistency). The novel aspect is that relevant information about concurrency and sharing is determined at the granularity of objects, not individual variables. We show that the object-level view and the tight integration with the compile-time analysis enables the compiler to reduce the runtime overhead of both runtime techniques without affecting the accuracy of the tracked information in practice.

8.2 Trends and future work

8.2.1 Interaction with hardware

The programming interface of future systems and micro-architectures might provide abstractions known from distributed systems and may expose features that determine the absence or presence of concurrency or atomicity. Such features have been proposed in the literature (e.g., transactional memories [57]) and allow for optimistic techniques to dynamic optimization and concurrency control. This proposed hardware functionality should be leveraged by compilers and dynamic analysis tools to implement higher-level synchronization primitives (Section 8.2.2) or runtime monitoring. In general, the increasing availability of computational resources will allow a runtime system or compiler to schedule cooperative actions for runtime monitoring without intrusion at a low cost for the execution to be monitored.

An initial step towards a concurrency-aware compiler and runtime system has been developed in this dissertation. Our model of object consistency (Section 7.2.1) and other runtime systems that assesses concurrency (e.g., approaches to optimistic concurrency control [114, 56]) have demonstrated the potential and benefit of concurrency awareness at runtime. One important limiting factor of today's implementations is the overhead of the software layer on which these systems are based.

8.2.2 Language aspects

The introduction of this chapter describes tree major impediments that hinder the precise static detection of synchronization errors in current programming languages. These aspects lead to the following recommendations for the design of languages that are amenable to synchronization analysis:

Synchronization discipline. Current models for parallel programming and synchronization are diverse and hardly amenable to automated static or dynamic analysis. It is desirable that the structure and boundaries of synchronization are more explicit and follow certain patterns, even if this implies restrictions on their use.

Most static and dynamic checkers for synchronization defects verify that an execution or program meets a certain synchronization discipline. Compliance with the discipline implies

the absence of synchronization errors. The checker's task of re-engineering and unraveling the structure of synchronization from the program text or an execution is hard and often afflicted with unnecessary conservatism. A more elegant approach would be to codify the synchronization discipline statically. The role of the checker in such a system would be different: Instead of giving proof of compliance, it may only have to validate compliance in occasional critical cases. There are several approaches that are based on type systems and guarantee the absence of data races altogether [9, 41, 16]. These systems have been discussed under the language-centric approach to correct synchronization in Section 2.5.

High-level synchronization primitives. Commonly used synchronization primitives are *operational*, i.e., they describe an action that the issuing thread performs with respect to the behavior of other threads. Examples of such synchronization mechanisms are locks, monitors, barriers and rendez-vous constructs. In contrast to operational constructs, transactions provide a *declarative* description of synchronization, e.g., in the form of 'atomic regions' [56] that guarantee the absence of conflicting interference on any shared data that is accessed during the execution of the region. If programming languages adopt such a declarative model of synchronization, the ground rules for synchronization analysis and compiler may change and compilers may find themselves in a new role of orchestrating and synthesizing operational synchronization actions. So far, features for transactional execution have been hardly incorporated into programming languages because their implementation is considered to be expensive. Novel trends in hardware (Section 8.2.1) may however change this picture.

Alias control. Most object-oriented programming languages provides a simple and flexible object model: dynamic allocation and object access through references. This model implicates the phenomenon of aliasing which aggravates static program analysis significantly (Chapters 3 to 6). It is generally desirable to limit the spread of references (which serve as access keys to objects) such that the usage scope of an object can be confined to a specific thread or component. Yet, practical mechanism for *alias control* [64] need to be developed. The purpose of alias control is to restrict the occurrence of aliases to certain parts of a program. Research in this field addresses, e.g., type systems to control the handling of references to objects and object access, e.g., [5, 87, 16]). As such techniques provide a more precise notion of aliasing at compile-time, they could improve the accuracy of the static synchronization analyses presented in Chapters 4 to 6.

Modular program analysis. The static analysis in Chapter 3 assumes the availability of the whole program and explicit models for the use of reflection and native methods. This is unrealistic in some environments with dynamic compilation or mobile code. There are two aspects that necessitate whole-program information:

First, the flow of references in Java is not bounded through the lexical scope or the type system, and hence a precise reference analysis is required to operate across class and component boundaries. Mechanisms for alias control that are discussed in the previous paragraph declare the bounds of reference flow explicitly and enable modular reasoning over object structures.

The second aspect that gets in the way of a modular synchronization analysis in Java is related to the synchronization mechanisms themselves: The potential of thread interference during the execution of a certain methods depends on the context in which the method is invoked.

This is due to the synchronization (e.g., the monitor) that could prevent critical interference on a module *m* is supplied by the caller of *m*, while *m* itself is oblivious to the protection properties provided by its usage contexts.

Any analysis can be modular if it is conservative with respect to effects outside the analysis scope. For synchronization analysis of Java programs, this procedure is however not practical because the precision of analysis would deteriorate dramatically.

Program optimization. Today's compilers typically transform and optimize programs according to a sequential execution model, accounting for data and control dependence. The transformation of parallel programs, however, has to be done with care: The existence of access conflicts may introduce additional constraints that inhibit reordering beyond intra-thread control- and data dependence [85, 76]. A naive optimizer for parallel programs, i.e., an optimizer that performs code transformation irrespective of concurrent conflicting access, encounters the following limitations: First, the resulting programs may behave incorrectly with respect to a sequentially consistent execution model (subset correctness, [76]). Second, such a naive optimizer might not exploit the full optimization potential [125]. There are two strategies that address these limitations: First, memory consistency could be relaxed such that transformations for sequential programs are also "correct" for parallel programs; this strategy is common practice, e.g., [79]. Alternatively, compilers for parallel programs could adopt concurrency and synchronization analysis (as presented in this dissertation); the results of these analyses could guide the program transformation, enabling aggressive optimization while at the same time preserving correctness with respect to a sequentially consistent memory model [125].

Bibliography

[1] S. Adve and K. Gharachorloo. Shared memory consistency models: A tutorial. *IEEE Computer*, 29:66–76, 1996.

[2] S. Adve and M. Hill. Weak ordering — A new definition. In *Proceedings of the Annual International Symposium on Computer Architecture (ISCA'90)*, pages 2–14, June 1990.

[3] S. Adve and M. Hill. A unified formalization of four shared-memory models. *IEEE Transactions on Parallel and Distributed Systems*, 4(6):613–624, June 1993.

[4] G. Agha. Concurrent object-oriented programming. *Communications of the ACM*, 22(9):125–141, Sept. 1990.

[5] P. S. Almeida. Balloon types: Controlling sharing of state in data types. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP'97)*, pages 32–59, June 1997.

[6] L. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, May 1994. (DIKU Report 94/19).

[7] C. Artho. Finding faults in multi-threaded programs. Master's thesis, Swiss Federal Institute of Technology, Zürich, 2001.

[8] C. Artho, A. Biere, and K. Havelund. High-level data races. In *Proceedings of the Workshop on Verification and Validation of Enterprise Information Systems (VVEIS'03)*, Apr. 2003.

[9] D. Bacon, R. Strom, and A. Tarafdar. Guava: A dialect of Java without data races. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'00)*, pages 382–400, Oct. 2000.

[10] H. Bal, F. Kaashoek, and A. Tanenbaum. Orca: A language for parallel programming of distributed systems. *IEEE Transactions on Software Engineering*, 18(3):190–205, 1992.

[11] V. Balasundaram and K. Kennedy. Compile-time detection of race conditions in a parallel program. In *Proceedings of the International Conference on Supercomputing (ISC'89)*, pages 175–185, 1989.

[12] B. Bershad, M. Zekauskas, and W. Sawdon. The Midway distributed shared memory system. In *Proc. of the 38th IEEE Int'l Computer Conf. (COMPCON Spring'93)*, pages 528–537, Feb. 1993.

[13] B. Blanchet. Escape analysis for object-oriented languages - Application to Java. In *Proc. Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'99)*, pages 20–34, Nov. 1999.

[14] H.-J. Boehm. Destructors, finalizers, and synchronization. In *Proceedings of the Symposium on Principles of Programming Languages (POPL'03)*, pages 262–272, 2003.

[15] J. Bogda and U. Hölzle. Removing unnecessary synchronization in Java. In *Proc. Conf. Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'99)*, pages 35–46, Nov. 1999.

[16] C. Boyapati, R. Lee, and M. Rinard. Ownership types for safe programming: preventing data races and deadlocks. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'02)*, pages 211–230, Nov. 2002.

[17] G. Bristow, C. Dreay, B. Edwards, and W. Riddle. Anomaly detection in concurrent programs. In *Proceedings of the International Conference on Software Engineering (ICSE'79)*, pages 265–273, 1979.

[18] M. Burrows and K. R. M. Leino. Finding stale value errors in concurrent programs. Research Report 2002-004, Compaq SRC, 2002.

[19] J. Chapin et al. The Rivet project. `http://sdg.lcs.mit.edu/rivet.html`, 1998.

[20] J. Chase, F. Amador, E. Lazowska, H. Levy, and R. Littlefield. The amber system: Parallel programming on a network of multiprocessors. In *Proceedings of the Symposium on Operating Systems Principles (SOSP'89)*, pages 147–158, 1989.

[21] R. Chatterjee, B. G. Ryder, and W. Landi. Relevant context inference. In *Proceedings of the Symposium on Principles of Programming Languages (POPL'99)*, pages 133–146, 1999.

[22] A. Chien and J. Dolby. The Illinois Concert System: A problem-solving environment for irregular applications. In *Proceedings of the Symposium on Parallel Computing and Problem Solving Environments (DAGS'94)*, 1994.

[23] A. Chien, U. Reddy, J. Plevyak, and J. Dolby. ICC++: A C++ dialect for high performance parallel computing. In *Proceedings of the International Symposium on Object Technologies for Advanced Software (ISOTAS'96)*, pages 190–205, Mar. 1996.

[24] J. Choi, M. Gupta, M. Serrano, V. Sreedhar, and S. Midkiff. Escape analysis for Java. In *Proc. Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'99)*, pages 1–19. ACM Press, Nov. 1999.

[25] J.-D. Choi, K. Lee, A. Loginov, R. O'Callahan, V. Sarkar, and M. Sridharan. Efficient and precise datarace detection for multithreaded object-oriented programs. In *Conference Programming Language Design and Implementation (PLDI'02)*, pages 258–269, June 2002.

[26] J.-D. Choi, A. Loginov, and V. Sarkar. Static datarace analysis for multithreaded object-oriented programs. Technical Report RC22146, IBM Research, Aug. 2001.

[27] M. Christiaens and K. de Bosschere. TRaDe: A topological approach to on-the-fly race detection in Java programs. In USENIX Association, editor, *Proceedings of the Java Virtual Machine Research and Technology Symposium (JVM'01)*, pages 105–116, Apr. 2001.

[28] E. Coffman, M. Elphick, and A. Shoshani. System deadlocks. *ACM Computing Surveys*, (3)2:67–78, June 1971.

[29] S. A. Cook. Soundness and completeness of an axiom system for program verification. *SIAM Journal of Computing*, 7(1), 1978.

[30] J. Corbett. Evaluating deadlock detection methods for concurrent software. *IEEE Transactions on Software Engineering*, 22(3), 1996.

[31] J. Corbett, M. Dwyer, J. Hatcliff, and S. Laubach. Bandera: Extracting finite-state models from Java source code. In *Proceedings of the International Conference on Software Engineering (ICSE'00)*, pages 439–448, June 2000.

[32] J. C. Corbett. Constructing compact models of concurrent Java programs. In *Proceedings of the ACM International Symposium on Software Testing and Analysis (ISSTA '98)*, volume 23(2) of *ACM Software Engineering Notes*, pages 1–10. ACM Press, Mar. 1998.

[33] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. The MIT Press, 1990.

[34] E. Dijkstra. Hierarchical ordering of sequential processes. *Acta Informatica*, 1(2):115–138, 1971.

[35] P. Diniz and M. Rinard. Synchronization transformation for parallel computing. In *Proc. Symp. Principles of Programming Languages (POPL'97)*, pages 187–200, Jan. 1997.

[36] A. Dinning and E. Schonberg. Detecting access anomalies in programs with critical sections. In *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 85–96, Dec. 1991.

[37] E. Duesterwald and M. Soffa. Concurrency analysis in the presence of procedures using a data-flow framework. In *Proceedings of the Symposium on Testing, Analysis, and Verification (TAV4)*, pages 36–48, 1993.

[38] ECMA. Standard ecma-334: C# language specification. `http://www.ecma-international.org/publications/files/ecma-st/Ecma-334.pdf`, Dec. 2002.

[39] S. Eggers, J. Emer, H. Levy, J. Lo, R. Stamm, and D. Tullsen. Simultaneous multithreading: A platform for next-generation processors. In *Proceedings of IEEE Micro*, pages 12–18, Sept. 1997.

[40] P. A. Emrath and D. A. Padua. Automatic detection of nondeterminacy in parallel programs. In *Proc. of the ACM Workshop on Parallel and Distributed Debugging*, pages 89–99, Madison, Wisconsin, Jan. 1989.

[41] C. Flanagan and S. Freund. Type-based race detection for Java. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI'00)*, pages 219–229, June 2000.

[42] C. Flanagan and S. Freund. Detecting race conditions in large programs. In *Proceedings of the Workshop on Program Analysis for Software Tools and Engineering (PASTE'01)*, pages 90–96, June 2001.

[43] C. Flanagan and S. N. Freund. Atomizer: A dynamic atomicity checker for multithreaded programs. In *Proceedings of the Symposium on Principles of Programming Languages (POPL'04)*, pages 256–267, Jan. 2004.

[44] C. Flanagan, R. Leino, M. Lillibridge, G. Nelson, J. Saxe, and R. Stata. Extended static checking for Java. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI'02)*, pages 234–245, June 2002.

[45] C. Flanagan and S. Qadeer. A type and effect system for atomicity. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI'03)*, pages 338–349, June 2003.

[46] C. Flanagan and S. Qadeer. Types for atomicity. In *Proceedings of the International Workshop on Types in Language Design and Implementation (TLDI'03)*, pages 1–12, Jan. 2003.

[47] I. Foster and C. Kesselman. *The Grid: Blueprint of a New Computing Infrastructure*. Morgan Kaufmann, 1998.

[48] G. Gao and V. Sarkar. Location consistency — A new memory model and cache consistency protocol. CAPSL Technical Memo 16, University of Delaware, Department of Electrical and Computer Engineering, Feb. 1998.

[49] GCJ: The GNU Compiler for the Java Programming Language. `http://gcc.gnu.org/java`.

[50] K. Gharachorloo. Retrospective: Memory consistency and event ordering in scalable shared-memory multiprocessors. In *25 Years of the International Symposia on Computer Architecture (ISCA), Selected Papers*, pages 67–70, 1998.

[51] K. Gharachorloo and P. Gibbons. Detecting violations of sequential consistency. In *Proceedings of the Symposium on Parallel Algorithms and Architectures, (SPAA'91)*, pages 316–326, July 1991.

[52] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proceedings of the International Symposium on Computer Architecture (ISCA'90)*, pages 15–26, June 1990.

[53] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification, 2nd Edition*. Addison-Wesley, 2000.

[54] A. Greenhouse and W. Scherlis. Assuring and evolving concurrent programs: Annotations and policy. In *Proceedings of the International Conference on Software Engineering (ICSE'02)*, pages 453–463, May 2002.

[55] D. Grove, G. DeFouw, J. Dean, and C. Chambers. Call graph construction in object-oriented languages. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'97)*, pages 108–124, Oct. 1997.

[56] T. Harris and K. Fraser. Language support for lightweight transactions. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'03)*, pages 388–402, Oct. 2003.

[57] M. Herlihy and E. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the International Symposium on Computer Architecture (ISCA'93)*, pages 289–301, May 1993.

[58] M. P. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12:463–492, July 1990.

[59] A. Heydon, R. Levin, and Y. Yu. Caching function calls using precise dependencies. In *Proceedings on the Conference on Programming Language Design and Implementation (PLDI'00)*, pages 311–320, June 2000.

[60] M. Hill. Multiprocessors should support simple memory-consistency models. *IEEE Computer*, 31(8):28–34, Aug. 1998.

[61] M. Hind. Pointer analysis: Haven't we solved this problem yet? In *Proceedings of the Workshop Program Analysis for Software Tools and Engineering (PASTE'01)*, pages 54–61, June 2001.

[62] M. Hind and A. Pioli. Which pointer analysis should i use? In *International Symposium on Software Testing and Analysis*, pages 113–123, Aug. 2000.

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

[64] J. Hogg, D. Lea, A. Wills, D. deChampeaux, and R. Holt. The geneva convention on the treatment of object aliasing. *OOPS Messenger*, 3(2), Apr. 1992.

[65] G. J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, 1997.

[66] L. Iftode, J. Singh, and K. Li. Scope consistency: A bridge between release consistency and entry consistency. In *Proc. of the 8th ACM Annual Symp. on Parallel Algorithms and Architectures (SPAA'96)*, pages 277–287, June 1996.

[67] Intel. IA-32 Intel architecture optimization reference manual. `http://developer.intel.com/design/PentiumIII/manuals`, 2001.

[68] Java Grande Forum multi-threaded benchmark suite. `http://www.epcc.ed.ac.uk/javagrande/`.

[69] Java HotSpot Virtual Machine (VM). `http://java.sun.com/products/hotspot`.

[70] Java memory model mailing list. `http://www.cs.umd.edu/~pugh/java/memoryModel`.

[71] E. Jensen, G. Hagensen, and J. Broughton. A new approach to exclusive data access in shared memory multi-processors. Technical Report UCRL-97663, Lawrence Livermore National Laboratory, Nov. 1987.

[72] J. Knoop, B. Steffen, and J. Vollmer. Parallelism for free: Efficient and optimal bitvector analyses for parallel programs. *ACM Transactions on Programming Languages and Systems*, 18(3):268–299, 1996. Extended and revised version of KnSV95b-C.

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

[74] D. Lea et al. Java Specification Request #166: Concurrency Utilities. `http://gee.cs.oswego.edu/dl/concurrency-interest`.

[75] J. Lee and D. Padua. Hiding relaxed memory consistency with compilers. In *Proc. of The IEEE Intl. Conf. on Parallel Architectures and Compilation Techniques (PACT'00)*, pages 111–122, Oct. 2000.

[76] J. Lee, D. Padua, and S. Midkiff. Basic compiler algorithms for parallel programs. In *Proceedings of the Symposium on Principles and Practice of Parallel Programming (PPoPP'99)*, pages 1–12, May 1999.

[77] D. Liang, M. Pennings, and M. J. Harrold. Extending and evaluating flow-insensitive and context-insensitive points-to analyses for Java. In *Proceedings of the Workshop on Program Analysis for Software Tools and Engineering (PASTE'01)*, pages 73–79, June 2001.

[78] R. Lipton. A method of proving properties of parallel programs. *Communications of the ACM*, 18(12):717–721, Dec. 1975.

[79] J. Manson, W. Pugh, et al. Java Specification Request #133: Java Memory Model and Thread Specification. `http://www.cs.umd.edu/~pugh/java/memoryModel`.

[80] S. Masticola and B. Ryder. Static infinite wait anomaly detection in polynomial time. In *Proceedings of the International Conference on Parallel Processing (ICPP'90)*, pages 78–87, 1990.

[81] S. Masticola and B. Ryder. Non-concurrency analysis. In *Proceedings of the Symposium on Principles and Practice of Parallel Programming (PPoPP'93)*, pages 129–138, 1993.

[82] J. Mellor-Crummey. On-the-fly detection of data races for programs with nested fork-join parallelism. In *Proceedings of the Supercomputer Debugging Workshop*, pages 24–33, Nov. 1991.

[83] J. Mellor-Crummey. Compile-time support for efficient data race detection in shared-memory parallel programs. In *Proceedings of the Workshop on Parallel and Distributed Debugging*, pages 129–139, May 1993.

[84] S. Midkiff, J. Lee, and D. Padua. A compiler for multiple memory models. In *Rec. Workshop Compilers for Parallel Computers (CPC'01)*, June 2001.

[85] S. Midkiff and D. Padua. Issues in the optimization of parallel programs. In *Proceedings of the International Conference on Parallel Processing*, pages 105–113, Aug. 1990.

[86] A. Milanova, A. Rountev, and B. Ryder. Parametrized object-sensitivity for poits-to and side-effect analyses foe Java. In *Proceedings of the ACM International Symposium on Software Testing and Analysis (ISSTA '02)*, pages 1–11, July 2002.

[87] P. Müller and A. Poetzsch-Heffter. Universes: A type system for alias and dependency control. Technical Report 279, Fernuniversität Hagen, 2001.

[88] G. Naumovich, G. Avrunin, and L. Clarke. Data flow analysis for checking properties of concurrent java programs. In *Proceedings of the International Conference on Software Engineering (ICSE'99)*, pages 399–410, May 1999.

[89] G. Naumovich, G. Avrunin, and L. Clarke. An efficient algorithm for computing MHP information for concurrent Java programs. In *Proceedings of the European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 338–354, Sept. 1999.

[90] G. Naumovich, L. A. Clarke, and J. M. Cobleigh. Using partial order techniques to improve performance of data flow analysis based verification. In *Proceedings of the Workshop Program Analysis for Software Tools and Engineering (PASTE'99)*, pages 57–65, Sept. 1999.

[91] R. Netzer and B. Miller. Detecting data races in parallel program executions. Technical Report TR90-894, University of Wisconsin, Madison, Department of Computer Science, Aug. 1990.

[92] R. Netzer and B. Miller. What are race conditions? Some issues and formalizations. *ACM Letters on Programming Languages and Systems*, 1(1):74–88, Mar. 1992.

[93] R. O'Callahan and J.-D. Choi. Hybrid dynamic data race detection. In *Proceedings of the Symposium Principles and Practice of Parallel Programming (PPoPP'03)*, pages 167–178, June 2003.

[94] J. Oplinger, D. Heine, and M. Lam. In search of speculative thread-level parallelism. In *Proceedings of the Conference on Parallel Architectures and Compilation Techniques (PACT'99)*, pages 303–313, Oct. 1999.

[95] J. Plevyak, X. Zhang, and A. Chien. Obtaining sequential efficiency for concurrent object-oriented languages. In *Proceedings of the Symposium on Principles of Programming Languages (POPL'95)*, pages 311–321, 1995.

[96] E. Pozniansky and A. Schuster. Efficient on-the-fly data race detection in multithreaded c++ programs. In *Proceedings of the Symposium Principles and Practice of Parallel Programming (PPoPP'03)*, pages 179–190, June 2003.

[97] W. Pugh. The Java memory model is fatally flawed. *Concurrency: Practice and Experience*, 12(6):445–455, 2000.

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

[99] M. Rinard. Analysis of multithreaded programs. In *Proceedings of the Static Analysis Symposium (SAS'01)*, July 2001.

[100] E. Ruf. Effective synchronization removal for Java. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI'00)*, pages 208–218, June 2000.

[101] R. Rugina and M. Rinard. Pointer analysis for multithreaded programs. In *Proc. Conf. Programming Language Design and Implementation (PLDI'99)*, pages 77–90, May 1999.

[102] B. Ryder. Dimensions of precision in reference analysis of object-oriented languages. In *Proceedings of the Conference on Compiler Construction (CC'03)*, pages 126–137, Apr. 2003.

[103] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A dynamic data race detector for multi-threaded programs. In *Proceedings of the Symposium on Operating Systems Principles (SOSP'97)*, pages 27–37, Oct. 1997.

[104] D. Scales and M. Lam. The design and evaluation of a shared object system for distributed memory machines. In *Proceedings of the Symposium on Operating Systems Design and Implementation (OSDI'94)*, pages 101–114, 1994.

[105] D. Schmidt and T. Harrison. Double-checked locking: An optimization pattern for efficiently initializing and accessing thread-safe objects. In *Pattern Languages of Program Design (PLoP) 3*, pages 363–375, 1998.

[106] M. Singhal. Deadlock detection in distributed systems. *IEEE Computer*, 22(11):37–48, Nov. 1989.

[107] SPEC JBB2000 Java Business Benchmark. `http://www.specbench.org/osg/jbb2000`.

[108] SPEC JVM98 Benchmarks. `http://www.spec.org/osg/jvm98`.

[109] M. Sridharan. Dynamic datarace detection for object-oriented programs. Master's thesis, Massachusetts Institute of Technology, 2002.

[110] B. Steensgaard. Points-to analysis in almost linear time. In *Proceedings of the Symposium on Principles of Programming Languages (POPL'96)*, pages 32–41, Jan. 1996.

[111] A. Stepanov and M. Lee. The Standard Template Library. Technical report, Hewlett-Packard Company, 1994.

[112] N. Sterling. WARLOCK: A static data race analysis tool. In USENIX Association, editor, *Proceedings of the USENIX Winter 1993 Conference*, pages 97–106, Jan. 1993.

[113] E. Stolte, C. von Praun, G. Alonso, and T. Gross. Scientific data repositories – designing for a moving target. In *Proceedings on the International Conference on Management of Data and Symposium on Principles of Database Systems (SIGMOD/PODS'03)*, pages 349–360, June 2003.

[114] R. Strom and J. Auerbach. The optimistic readers transformation. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP'01)*, pages 275–301, June 2001.

[115] V. Sundaresan, L. J. Hendren, C. Razafimahefa, R. Vallée-Rai, P. Lam, E. Gagnon, and C. Godin. Practical virtual method call resolution for java. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'00)*, pages 264–280, Oct. 2000.

[116] D. Sutherland, A. Greenhouse, and W. Scherlis. The code of many colors: Relating threads to code and shared state. In *Proceedings of the Workshop Program Analysis for Software Tools and Engineering (PASTE'02)*, pages 77–83, June 2002.

[117] R. Taylor. Complexity of analyzing the synchronization structure of concurrent programs. *Acta Informatica*, 19:57–84, 1983.

[118] R. Taylor. A general purpose algorithm for analyzing concurrent programs. *Communications of the ACM*, 26(5):362–376, May 1983.

[119] T. Ungerer, B. Robic, and J. Silc. A survey of processors with explicit multithreading. *ACM Computing Surveys*, 35:29–63, Jan. 2003.

[120] W. Visser, K. Havelund, G. Brat, and S. Park. Model checking programs. In *Proceedings of the International Conference on Automated Software Engineering*, Sept. 2000.

[121] C. von Praun and T. Gross. Compiler-based object consistency. In *Workshop on Caching, Coherence, and Consistency (WC3'01)*, June 2001.

[122] C. von Praun and T. Gross. Object race detection. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'01)*, pages 70–82, Oct. 2001.

[123] C. von Praun and T. Gross. Compiling multi-threaded object-oriented programs. In *Workshop on Compilers for Parallel Computers (CPC'03)*, Jan. 2003.

[124] C. von Praun and T. Gross. Static conflict analysis for multi-threaded object-oriented programs. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI'03)*, pages 115–129, June 2003.

[125] C. von Praun, F. Schneider, and T. Gross. Load elimination in the presence of side effects concurrency and precise exceptions. In *Proceedings of the International Workshop on Compilers for Parallel Computing (LCPC'03)*, Oct. 2003.

[126] L. Wang and S. Stoller. Run-time analysis for atomicity. In *Workshop on Runtime Verification (RV'03)*, July 2003.

[127] J. Whaley and M. Lam. An efficient inclusion-based points-to analysis for strictly-typed languages. In *Proceedings of the Static Analysis Symposium (SAS'02)*, Sept. 2002.

[128] J. Whaley and M. Rinard. Compositional pointer and escape analysis for Java programs. In *Proc. Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'99)*, pages 187–206, Nov. 1999.

[129] R. Wilson and M. Lam. Efficient context-sensitive pointer analysis for C programs. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI'95)*, pages 1–12, June 1995.

[130] World Wide Web Consortium. Jigsaw: Open Source web server. `http://www.w3.org/Jigsaw`.

[131] E. Yahav. Verifying safety properties of concurrent Java programs using 3-valued logic. In *Proceedings of the Symposium on Principles of Programming Languages (POPL'01)*, pages 27–40, Jan. 2001.

Curriculum Vitae

Christoph von Praun

January 20, 1972	Born in Munich, Germany
1978 – 1982	Primary school, Munich
1982 – 1991	Ludwigs-Gymnasium, Munich
1991	Abitur
1991 – 1996	Studies in Computer Science, TU-Munich
1994	Internship at IXOS Software AG, Grasbrunn
1996	Diploma in Computer Science, TU-Munich
1996 – 1998	Fellowship at CERN, Geneva
1998	Visiting Scientist at the California Institute of Technology
since 1998	Research and Teaching Assistant Laboratory for Software Technology, ETH Zurich