High-level Data Races

Cyrille Artho*, Klaus Havelund†, and Armin Biere*

*Computer Systems Institute, ETH Zurich, Switzerland

†Kestrel Technology, NASA Ames Research Center, Moffett Field, California USA

Keywords: Java, multi-threading, data races, software verification, consistency

Abstract

Data races are a common problem in concurrent and multi-threaded programming. Experience shows that the classical notion of data race is not powerful enough to capture certain types of inconsistencies occurring in practice. This paper investigates data races on a higher abstraction layer. This enables detection of inconsistent uses of shared variables, even if no classical race condition occurs. For example, a data structure representing a coordinate pair may have to be treated atomically. By lifting the meaning of a data race to a higher level, such problems can now be covered. The paper defines the concepts view and view consistency to give a notation for this novel kind of property. It describes what kinds of errors can be detected with this new definition, and where its limitations are. It also gives a formal guideline for using data structures in a multi-threaded environment.

1 Introduction

Multi-threaded, or concurrent, programming is becoming increasingly popular in enterprise applications and information systems [AB01, Sun02]. The Java programming language [AG96] explicitly supports this paradigm [Lea97]. Multi-threaded programming, however, provides a potential for introducing intermittent concurrency errors that are hard to find using traditional testing. The main source of this problem is that a multi-threaded program may execute differently from one run to another due to the apparent randomness in the way threads are scheduled. Since testing typically cannot explore all schedules, some bad schedules may never be discovered. One kind of error that often occurs in multi-threaded programs is a *data race*, as defined below. This paper shall go beyond the traditional notion of what shall be referred to as low-level data races, and introduce high-level data races, together with an algorithm for detecting them. Low-level as well as high-level data races can be characterized as occurring when two or more threads access a shared region simultaneously, the definition of region being dependent on what kind of data race is referred to.

Data races are very hard to detect with traditional testing techniques. Not only does a simultaneous access from two or more threads to a particular region have to occur, but this should additionally result in corrupted data, which violate some user-provided assertion. They are usually harder to detect than deadlocks, which often cause some visible activity to halt. The suggested algorithm, first presented in [AHB03], analyzes a single execution trace obtained by running an instrumented version of the program. In this context, single execution means the program is run once. While running, it emits a series of events which consitute the exection trace.

The analysis algorithm which analyzes the execution trace is, for practical purposes, mostly independent of the thread interleavings during program execution. Hence the program only needs to be run once.

The algorithm requires no user-provided requirement specification, and hence is in line with the Eraser algorithm [SBN+97] for detecting low-level data races. This means that the algorithm is totally automated, requiring no user guidance at all beyond normal input. The algorithm looks for certain patterns in the execution trace, and raises a warning in case such are detected. A data race does not have to occur in the run that generated the execution trace, which is why the algorithm is more powerful than traditional testing techniques. The algorithm is neither sound, nor complete, and may yield false positives (false warnings) and false negatives (missed errors). However, it is our opinion that the increased probability of detecting errors strongly outbalances this, in particular considering that it is fully automated. In addition, practice seems to support that the rates of false positives as well as false negatives are low.

The algorithm has been implemented in the Java PathExplorer (JPaX) tool [HR01, GH03, BH03], which provides a general framework for instrumenting Java programs, and for monitoring and analyzing execution traces. In particular JPaX contains algorithms for detecting problems in multi-threaded programs, such as data races and deadlocks [BH03]. Although JPaX analyzes Java programs, the principles and theory presented here are universal and apply in full to concurrent programs written in languages like C and C++ as well [NBF98].

1.1 Low-level Data Races

The traditional definition of a data race is as follows [SBN+97]:

> A data race can occur when two concurrent threads access a shared variable and when at least one access is a write, and the threads use no explicit mechanism to prevent the accesses from being simultaneous.

Consider for example two threads, that both access a shared object containing a counter variable x, and assume that both threads call an *increase()* method on the object, which increases x by 1. The *increase()* method is compiled into a sequence of bytecode instructions (load x to the operand stack, add 1, write back the result). The Java Virtual Machine (JVM) executes this sequence non-atomically. Suppose the two threads call *increase()* at nearly the same time and that each of the threads execute the *load* instruction first, which loads the value of x to the thread-local operand stack. Then they will both add 1 to the original value, which results in a combined increment of 1 instead of 2. This traditional notion of data race shall be referred to as a *low-level data race*, since it focuses on a single variable.

The standard way to avoid low-level data races on a variable is to protect the variable with a lock: all accessing threads must acquire this lock before accessing the variable, and release it again after. In Java, methods can be defined as `synchronized` which causes a call to such a method to lock the current object instance. Return from the method will release the lock. Java also provides an explicit statement form `synchronized(`*obj*`)` {*stmt*}, for taking a lock on the object *obj*, and executing statement *stmt* protected under that lock. If the above mentioned *increase()* method is declared `synchronized`, the low-level data race cannot occur.

Several algorithms and tools have been developed for analyzing multi-threaded programs for low-level data races. The Eraser algorithm [SBN+97], which has been implemented in the Visual Threads tool [Har00] to analyze C and C++ programs, is an example of a dynamic algorithm that examines a program execution trace

for locking patterns and variable accesses in order to predict potential data races. The Eraser algorithm maintains a *lock set* for each *variable*, which is the set of locks that have been owned by all threads accessing the variable in the past. Each new access causes a refinement of the lock set to the intersection of the lock set with the set of locks currently owned by the accessing thread. The set is initialized to the set of locks owned by the first accessing thread. If the set ever becomes empty, a data race is possible. JPaX implements the Eraser algorithm [HR01, Hav00]. Another commercial tool performing low-level data race analysis is JProbe [Sit00].

1.2 High-level Data Races

A program may contain a potential for concurrency errors, even when it is free of low-level data races and deadlocks. Low-level data races concern unprotected accesses to shared fields. The notion of high-level data races refers to sequences in a program where each access to shared data is protected by a lock, but the program still behaves incorrectly because operations that should be carried out atomically can be interleaved with conflicting operations.

A problem that was detected in NASA's *Remote Agent* space craft controller [PGK+97] shall serve as a realistic example of a high-level data race situation. The problem was originally detected using model checking [HLP01]. The error was very subtle, and was originally regarded hard to find without actually exploring all execution traces as done by a model checker. Because only very particular thread interleavings result in a data race and hence corrupted data, a single execution trace does usually not exhibit this error. As it turns out, it is an example of a high-level data race, and can therefore be detected with the low-complexity algorithm presented in this paper.

The Remote Agent is an artificial-intelligence-based software system for generating and executing plans on board a space craft. A plan essentially specifies a set of tasks to be executed within certain time constraints. The plan execution is performed by the *Executive*. A sub-component of the Executive, the *task manager*, is responsible for managing the execution of tasks, once the tasks have been activated. The data structures used by the task manager are illustrated in Fig. 1.

The state of the spacecraft (at any particular point) can be considered as consisting of a set of properties, each being an assignment of a value to a variable corresponding to a component on board the space craft. The values of variables are continuously read by sensors and recorded in a *system state*. A task running on board the space craft may require that specific properties hold during its execution, and it notifies the task manager about such requirements before start. That is, upon the start of the task, it first tries to lock those properties it requires in a *lock table*, telling the task manager that its execution is only safe if they hold throughout the execution. For example, a task may require B to be *ON* (assuming that B is some system variable). Now other threads cannot request B to be *OFF* as long as the property is locked in the lock table. Next, the task tries to achieve this property (changing the state of the space craft, and thereby the system state through sensor readings), and when it is achieved, the task sets a flag *achieved*, associated with that variable, to *true* in the lock table. This flag is true whenever the system state is expected to contain the property. These flags are initially set to false at system initialization, are set to true in the just described scenario, and are set back to false once the task no longer requires the property to hold.

A *Daemon* constantly monitors that the system state is consistent with the lock table in the sense that properties that are required to hold, as stated in the lock table, are actually also contained in the system state. That is, it checks that: *if a task has locked a value to some variable, and the corresponding flag* achieved *is*

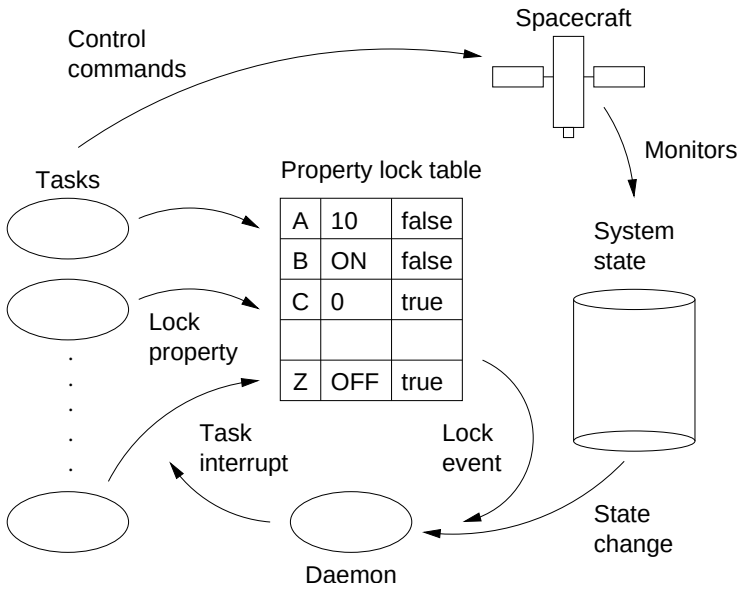

Figure 1: The Remote Agent Executive

true, then it must be a true property of the space craft, and hence true in the system state. Violations of this property may occur by unexpected events on board the space craft, which cause the system state to be updated. The daemon wakes up whenever events occur, such as when the lock table or the system state are modified. In case an inconsistency is detected, the involved tasks are interrupted.

Task

```
synchronized(table) {
  table[N].value = V;
}

/* achieve property */

synchronized(table) {
  table[N].achieved = true;
}
```

Daemon

```
synchronized(table) {
  if (table[N].achieved &&
    system_state[N] != table[N].value) {
    issueWarning();
  }
}
```

Figure 2: The synchronization inconsistency between a task and the daemon.

The relevant code from the task and the daemon is illustrated in Fig. 2, using Java syntax (The Remote Agent was coded in LISP). The task contains two separate accesses to the lock table, one where it updates the value and one where it updates flag *achieved*. The daemon on the other hand accesses all these fields in one atomic block. This can be described as an inconsistency in lock views, as described below, and actually presents an error potential.

The error scenario is as follows: suppose the task has just achieved the property, and is about to execute the second synchronized block, setting flag *achieved* to true. Suppose now however, that suddenly, due to

unpredicted events, the property is destroyed on board the space craft, and hence in the system state, and that the daemon wakes up, and performs all checks. Since flag *achieved* is false, the daemon reasons incorrectly that the property is not supposed to hold in the system state, and hence it does not detect any inconsistency with the lock table (although conceptually now there is one). Only then the task continues, and sets flag *achieved* to true. The result is that the violation has been missed by the daemon. The nature of this error can be described as follows:

> The daemon accesses the value and flag *achieved* in one atomic block, while the task accesses them in two different blocks. These two different ways of accessing tuple $\{value, achieved\}$, atomically and as a compound operation, is a inconsistency called *view inconsistency*.

The seriousness of the error scenario depends on the frequency with which the daemon gets activated. If events updating the lock table or the system state occur often, the daemon will just detect the problem later, and hopefully soon enough. However, if such events are far apart, the task may execute for a while without its required property holding. In the above example the view inconsistency is in itself not an error, but a symptom that if pointed out may direct the programmer's attention to the real problem, that property achievement and setting the flag *achieved* are not done in one atomic block. More formal and generic definitions of view inconsistency are presented in Sec. 2 and 3. Note that repairing this situation is non-trivial since achieving properties may take several clock cycles, and it is therefore not desirable to hold the lock on the table during this process.

Detecting this error using normal testing is very hard since it requires not only to execute the just described interleaving (or a similar one), but it also requires the formulation of a correctness property that can be tested for, and which is violated in the above scenario. However, regarding this as a view inconsistency problem makes it possible to find the error without actually executing this particular interleaving, and without a requirement specification.

Both aspects are very important. The success of the Eraser algorithm [SBN$^+$97] confirms this. Data races, both low-level or high-level ones, occur only rarely in a concrete run, because they only appear under a certain schedule. This makes them very hard to observe using traditional testing. Therefore one tries to observe the locking behavior of a program and infer potential errors from that. The locking behavior of each thread does not change across different schedules (it is only dependent on the input, which can be automated), and therefore it is a reliable base for fault-finding algorithms.

Requiring no annotations is not only important when it comes to the usability of a tool. Tools like ESC/Java suffered from having a high initial overhead due to the required annotations in a program [DLNS98]. Algorithms that do not require annotations are also more interesting from a research point of view: It is attempted to extract as much information as possible from the program itself rather than having the user specify (possibly incorrect or redundant) information. Of course such approaches are sometimes inherently more limited than ones requiring annotations. The goal is to find algorithms that still capture the underlying problems with sufficient precision.

The algorithm presented in this paper achieves this to a high degree. In the Eraser algorithm [SBN$^+$97] for detecting low-level data races, the set of locks protecting a single variable, referred to as the *lock set*, is considered. In this paper this idea is turned upside down. The *variable set* associated to a *lock* is now of interest. This notion makes it possible to detect what shall be referred to as *high-level data races*. The inspiration for this problem was originally due to a small example provided by Doug Lea [Lea00]. It is presented in modified form in Sec. 2. It defines a simple class representing a coordinate pair with two components x and y. All accesses are protected by synchronization on `this`, using `synchronized` methods. Therefore, data race conditions

on a low level are not possible. As this example will illustrate, there can, however, still be data races on a higher level, and this can be detected as inconsistencies in the granularity of *variable sets* associated to locks in different threads. The algorithm for detecting high-level data races is a dynamic execution trace analysis algorithm like the Eraser algorithm [SBN+97].

1.3 Outline

The paper is organized as follows. Sec. 2 introduces the problem of high-level data races. A formal definition of high-level data races is given in Sec. 3. Sec. 4 describes the JPaX framework for analyzing Java programs. Experiments carried out are described in Sec. 5. Sec. 6 gives an overview of related work. Sec. 7 outlines future work and Sec. 8 concludes the paper.

2 Informal Definition of High-level Data Races

Consistent lock protection for a shared field ensures that no concurrent modification is possible. However, this only refers to low-level access to the fields, not their entire use or their use in conjunction with other fields. The remainder of this paper assumes detection of low-level data races is covered by the Eraser algorithm [SBN+97], which can be applied in conjunction with the analysis described in this paper. This section introduces a more precise definition of high-level data races. First a very basic example is presented, followed by more pathological ones, which at first sight may appear as high-level data races, but which shall not be classified as such.

2.1 Basic Definition

Fig. 3 shows a class implementing a two-dimensional coordinate pair with two fields x, y, which are guarded by a single lock.

```
class Coord {
  double x, y;
  public Coord(double px, double py) { x = px; y = py; }
  synchronized double getX() { return x; }
  synchronized double getY() { return y; }
  synchronized Coord getXY() { return new Coord(x, y); }
  synchronized void setX(double px) { x = px; }
  synchronized void setY(double py) { y = py; }
  synchronized void setXY(Coord c) { x = c.x; y = c.y; }
}
```

Figure 3: The `Coord` class encapsulating points with x and y coordinates.

If only `getXY`, `setXY`, and the constructor are used by any thread, the pair is treated atomically by all accessing threads. However, the versatility offered by the other accessor (`get`/`set`) methods is dangerous: if one thread only uses `getXY` and `setXY` and relies on complete atomicity of these operations, other threads using the other accessor methods (`getX`, `setX`, `getY`, `setY`) may falsify this assumption.

Imagine for example a case where one thread, t_r, reads both coordinates while another thread, t_w, sets them to zero. Assume that t_r reads the variables with `getXY`, but that t_w's write operation occurs in two phases, `setX` and `setY`. The thread t_r may then read an *intermediate result* which contains the value of x already

set to zero by t_w but still the original y value (not zeroed yet by t_w). This is clearly an undesired and often unexpected behavior. In this paper, the term *high-level data race* will describe this kind of scenario:

> A high-level data race can occur when two concurrent threads access a set V of shared variables, which should be accessed atomically, but at least one of the threads does not access the variables in V atomically.

In the coordinate pair example above, the set V is $\{x, y\}$, and thread t_w violates the atomicity requirement. Of course, a main question is how one determines whether a set of variables are to be treated atomically, assuming that the user does not specify that explicitly. For now it is assumed that an oracle determines this. In Sec. 3 an approximation to this oracle will be suggested, which does not require any kind of specification to be provided by the user. Of course it is an undecidable problem in practice, and furthermore requires a specification of the expected behavior of the program. For instance, in the above coordinate pair example, atomicity might not be required at all if the reader only wants to sample an x value and a y value without them being related.

2.2 Refinement of Basic Definition

Although the definition above may be useful, it yields false positives (false warnings). Using the coordinate example, any use of the methods `getX`, `setX`, `getY`, and `setY` will cause a high-level data race. However, there exist scenarios where some of these access methods are allowed without the occurence of high-level data races. Hence the notion of high-level data race needs to be refined. This is analog to the refinement in Eraser [SBN+97] of the notion of low-level data races in order to reduce the number of false positives.

The refinement of the definition will be motivated with the example in Fig. 4, which shows four threads working in parallel on a shared coordinate pair c. Thread t_1 writes to c (and is similar to t_w from Subsec. 2.1) while the other threads t_2, t_3 and t_4 read from c (t_3 is similar to t_r from Subsec. 2.1). The threads use local variables `xi` and `yi` of type `double` and `di` of type `Coord`, where i identifies the thread.

Thread t_1	Thread t_2	Thread t_3	Thread t_4
			`x4 = c.getX();`
			`use(x4);`
`d1 = new Coord(1,2);`	`x2 = c.getX();`	`x3 = c.getX();`	`d4 = c.getXY();`
`c.setXY(d1);`	`use(x2);`	`y3 = c.getY();`	`x4 = d4.getX();`
		`use(x3,y3);`	`y4 = d4.getY();`
			`use(x4,y4);`

Figure 4: One thread updating a pair of fields and three other threads reading fields individually.

Initially, only threads t_1 and t_3 are considered, the situation already described in Sec. 2.1. Inconsistencies might arise with thread t_3, which reads x in one operation and y in another operation, releasing the lock in between. Hence, thread t_1 may write to x and y in between, and t_3 may therefore obtain inconsistent values corresponding to two *different* global states.

Now consider the two threads t_1 and t_2. It is not trivial to see whether an access conflict occurs or not. However, this situation is safe. As long as t_2 does not use y as well, it does not violate the first thread's assumption that the coordinates are treated atomically. Even though t_1 accesses the entire pair $\{x, y\}$ atomically

and t_2 does not, the access to x alone in t_2 can be seen as a partial read. That is, the read access to x may be interpreted as reading $\{x, y\}$ and discarding y. So both threads t_1 and t_2 behave in a consistent manner. Each thread is allowed to use only a part of the coordinates, as long as that use is consistent.

The difficulty in analyzing such inconsistencies lies in the wish to still allow such partial accesses to sets of fields, like the access to x of thread t_2. The situation between t_1 and t_4 serves as another, more complicated, example of a situation which at first sight appears to provide a conflict, but which shall be regarded as safe. Regard thread t_4 as consisting of two operations: the first consisting of the first two statements, including `use(x4)`, and the second operation consisiting of the remaining four statements. The second operation is completely self-contained, and accesses in addition to y everything the first operation accesses (namely x). Consequently, the first operation in t_4 likely represents an operation that does not need y. Therefore, the two operations are unrelated and can be interleaved with the atomic update statement in t_1 without interfering with the operations of t_4 on x and y. On a more formal basis, t_4 is safe because the set of variables accessed in the first operation of t_4 is a subset of the set of variables accessed in its second operation; the variable sets form a *chain*. When they do not form a chain, they *diverge*. On the basis of this example, the definition of high-level data races can be refined as follows:

> A high-level data race can occur when two concurrent threads access a set V of shared variables, which should be accessed atomically, but at least one of the threads accesses V partially several times such that those partial accesses diverge.

This definition is adopted for the remainder of the paper. It can, however, still lead to false positives and false negatives as is described in Sec. 3.4.

The algorithm presented in the remainder of this paper does not distinguish between read and write accesses. This abstraction is sufficiently precise because view consistency is independent of the whether the access is a read or a write: A non-atomic read access may result in inconsistent values among the tuple read, because other threads may update the tuple between reads. A write access that is carried out non-atomically allows other threads to read partial updates between write operations. Note that it is assumed that at least one write access occurs; constant values can be ignored in this analysis.

3 Formal Definition of High-Level Data Races

This section defines *view consistency.* It lifts the common notion of a data race on a single shared variable to a higher level, covering sets of shared variables and their uses. This definition assumes that the specification of what fields have to be treated atomically is not provided by the user. It is instead extracted by program analysis. This analysis turns the problem of high-level data races into a testable property, using view consistency. The definition of this analysis is precise but allows for false positives and false negatives. This is discussed at the end of this section.

3.1 Views

A lock *guards* a shared field if it is held during an access to that field. The same lock may guard several shared fields. Views express what fields are guarded by a lock. Let I be the set of object instances generated by a particular run of a Java program. Then F is the set of all fields of all instances in I.

A *view* $v \in \mathbb{P}(F)$ is a subset of F. Let l be a lock, t a thread, and $B(t, l)$ the set of all `synchronized` blocks using lock l executed by thread t. For $b \in B(t, l)$, a view *generated by* t with respect to l, is defined as the set of fields accessed in b by t. The *set of generated views* $V(t) \subseteq \mathbb{P}(F)$ of a thread t is the set of all views v generated by t. In the previous example in Fig. 4, thread t_1 using both coordinates atomically generates view $v_1 = \{x, y\}$ under lock $l =$ `c`. Thread t_2 only accesses x alone under l, having view $v_2 = \{x\}$. Thread t_3 generates two views: $V(t_3) = \{\{x\}, \{y\}\}$. Thread t_4 also generates two views: $V(t_4) = \{\{x\}, \{x, y\}\}$.

3.2 Views in Different Threads

A view v_m generated by a thread t is a *maximal view*, iff it is maximal with respect to set inclusion in $V(t)$:

$$\forall v \in V(t)\ [v_m \subseteq v \rightarrow v_m = v]$$

Let $M(t)$ denote the set of all maximal views of thread t. Only two views which have fields in common can be responsible for a conflict. This observation is the motivation for the following definition. Given a set of views $V(t)$ generated by t and a view v' generated by another thread, the *overlapping views* of t with v' are all non-empty intersections of views in $V(t)$ with v':

$$\text{overlap}(t, v') \equiv \{v' \cap v \mid (v \in V(t)) \wedge (v \cap v' \neq \emptyset)\}$$

A set of views $V(t)$ is *compatible* with the maximal view v_m of another thread iff all overlapping views of t with v_m form a chain:

$$\text{compatible}(t, v_m) \quad \text{iff} \quad \forall v_1, v_2 \in \text{overlap}(t, v_m)\ [v_1 \subseteq v_2 \vee v_2 \subseteq v_1]$$

View consistency is defined as mutual compatibility between all threads: A thread is only allowed to use views that are compatible with the maximal views of all other threads.

$$\forall t_1 \neq t_2,\ v_m \in M(t_1)\ [\text{compatible}(t_2, v_m)]$$

In the example in Fig. 4, the views were

$$\begin{aligned} V(t_1) &= M(t_1) = \{\{x, y\}\} \\ V(t_2) &= M(t_2) = \{\{x\}\} \\ V(t_3) &= M(t_3) = \{\{x\}, \{y\}\} \\ V(t_4) &= \{\{x\}, \{x, y\}\} \\ M(t_4) &= \{\{x, y\}\} \end{aligned}$$

There is a conflict between $t_1$and t_3 as stated, since $\{x, y\} \in M(t_1)$ intersects with the elements in $V(t_3)$ to $\{x\}$ and $\{y\}$, which do not form a chain. A similar conflict exists between t_3 and t_4.

The above definition of *view consistency* uses three concepts: the notion of *maximal views*, the notion of *overlaps*, and finally the *compatible* notion, also referred to as the *chain* property. The chain property is the core concept. Maximal views do not really contribute to the solution other than to make it more efficient to calculate

#		Thread t_a	Thread t_b	Incompatible views
1	Views $V(t)$ Maximal views $M(t)$	$\{x\},\{y\}$ $\{x\},\{y\}$	$\{x\},\{y\}$ $\{x\},\{y\}$	none
2	Views $V(t)$ Maximal views $M(t)$	$\{x,y\}$ $\{x,y\}$	$\{x\},\{y\}$ $\{x\},\{y\}$	$\{x\} = \{x,y\} \cap \{x\} \in M(t_a) \cap V(t_b)$ $\{y\} = \{x,y\} \cap \{y\} \in M(t_a) \cap V(t_b)$
3	Views $V(t)$ Maximal views $M(t)$	$\{x,y\},\{x\},\{y\}$ $\{x,y\}$	$\{x\},\{y\}$ $\{x\},\{y\}$	$\{x\} = \{x,y\} \cap \{x\} \in M(t_a) \cap V(t_b)$ $\{y\} = \{x,y\} \cap \{y\} \in M(t_a) \cap V(t_b)$
4	Views $V(t)$ Maximal views $M(t)$	$\{x,y,z\}$ $\{x,y,z\}$	$\{x,y\},\{x\}$ $\{x,y\}$	none

Table 1: Examples with *two* threads illustrating the principle of view consistency.

#		Thread t_c	Thread t_d	Thread t_e	Incompatible views
5	$V(t)$ $M(t)$	$\{x,y\}$ $\{x,y\}$	$\{x\}$ $\{x\}$	$\{x\},\{y\}$ $\{x\},\{y\}$	$\{x\} = \{x,y\} \cap \{x\} \in M(t_c) \cap V(t_e)$ $\{y\} = \{x,y\} \cap \{y\} \in M(t_c) \cap V(t_e)$
6	$V(t)$ $M(t)$	$\{x,y\}$ $\{x,y\}$	$\{x\}$ $\{x\}$	$\{y\}$ $\{y\}$	none
7	$V(t)$ $M(t)$	$\{x,y\},\{x\},\{y\}$ $\{x,y\}$	$\{y,z\},\{y\},\{z\}$ $\{y,z\}$	$\{z,x\},\{z\},\{x\}$ $\{z,x\}$	none
8	$V(t)$ $M(t)$	$\{x,y\},\{x\},\{y,z\}$ $\{x,y\},\{y,z\}$	$\{y,z\},\{y\},\{z\}$ $\{y,z\}$	$\{z,x\},\{z\},\{x\}$ $\{z,x\}$	$\{y\} = \{y,z\} \cap \{y\} \in M(t_c) \cap V(t_d)$ $\{z\} = \{y,z\} \cap \{z\} \in M(t_c) \cap V(t_d)$

Table 2: Examples with *three* threads illustrating the principle of view consistency.

and reduce the number of warnings if a violation is found. The notion of overlaps is used to filter out irrelevant variables.

3.3 Examples

A few examples help to illustrate the concept. Table 1 contains examples involving two threads. Note that the outermost brackets for the set of sets are omitted for better readability. Example 1 is the trivial case where no thread treats the two fields $\{x\}$ and $\{y\}$ atomically. Therefore there is no inconsistency. However, if thread t_a treats $\{x,y\}$ as a pair, and thread t_b does not, there is a conflict as shown in example 2. This even holds if the first thread itself uses partial accesses on $\{x\}$ or $\{y\}$, since this does not change its maximal view. Example 3 shows that case. Finally, example 4 illustrates the case where thread t_a uses a three-dimensional coordinate set atomically and thread t_b reads or updates different subsets of it. Since the subsets are compatible as defined in Sec. 3.2, there is no inconsistency.

Table 2 shows four cases with three threads. The first entry, example 5, corresponds to the first three threads in Fig. 4. There, thread t_e violates the assumption of t_c about the atomicity of $\{x,y\}$. Example 6 shows a "fixed" version, where t_e does not access $\{x\}$. More complex circular dependencies can occur with three threads. Such a case is shown in example 7. Out of three fields, each thread only uses two, but these two fields are used atomically. Because the accesses of any thread only overlap in one field with each other thread, there is no inconsistency. This example only requires a minor change, shown in example 8, to make it faulty: Assume the third view of t_c were $\{y,z\}$ instead of $\{y\}$. This would contribute another maximal view $\{y,z\}$, which conflicts with the views $\{y\}$ and $\{z\}$ of t_d.

3.4 Soundness and Completeness

Essentially, this approach tries to infer what the developer intended when writing the multi-threaded code, by collecting views. The sets of shared fields which must be accessed atomically are not available in the program code. Therefore views are used to detect the most likely candidates, or maximal views. View consistency is used to detect violations of accesses to these sets. The underlying assumption behind the algorithm is that an atomic access to a set of shared fields is an indication of atomicity. Under this assumption, a view consistency violation indicates a high-level data race.

However, sets of fields may be used atomically even if there is no requirement for atomic access. Therefore, an inconsistency may not automatically imply a fault in the software. An inconsistency that does not correspond to a fault is referred to as a *false positive* (spurious warning). Similarly, lack of a reported inconsistency does not automatically imply lack of a fault. Such a missing inconsistency report for an existing fault is referred to as a *false negative* (missed fault).

False positives are possible if a thread uses a coarser locking than actually required by operation semantics. This may be used to make the code shorter or faster, since locking and unlocking can be expensive. Releasing the lock between two independent operations requires splitting one `synchronized` block into two blocks.

False negatives are possible if all views are consistent, but locking is still insufficient. Assume a set of fields that must be accessed atomically, but is only accessed one element at a time by every thread. Then no view of any thread includes all variables as one set, and the view consistency approach cannot find the problem. Another source of false negatives is the fact that a particular (random) run through the program may not reveal the inconsistent views, if the corresponding code sections are not executed even once.

The fact that false positives are possible means that the solution is not sound. Similarly, the possibility of false negatives means that the solution neither is complete. This may seem surprising, but actually also characterizes the Eraser low-level data race detection algorithm [SBN+97] implemented in the commercial Visual Threads tool [Har00], as well as the deadlock detection algorithm also implemented in the same tool. The same holds for the similar algorithms implemented in JPaX. For Eraser, it is very hard to determine automatically whether a warning is a false positive or a false negative [Bur00]. Furthermore, it is an unsolved problem to prove soundness and completeness properties about the Eraser algorithm. In real software programs, there are always situations where having program threads use inconsistent values is acceptable. For example, a monitoring thread may just "sample" a value at a given time; it is not crucial that this value is obtained with proper synchronization, because it does not have to be up-to-date.

The reason for the usefulness of such algorithms is that they still have a much higher chance of detecting an error than if one relies on actually executing the particular interleaving that leads to an error, without requiring much computational resource. These algorithms are essentially based on turning the property to be verified (in this case: no high-level data races) into a more testable property (view consistency). This aspect is discussed in more detail in [BH03] in relation to deadlock detection.

4 Implementation

For detecting high-level data races in software, a program analysis algorithm that extracts the views generated by all threads has to be implemented. This analysis can be static, analyzing the code, or dynamic, analyzing an execution trace. The implementation used in the experiments is a dynamic analysis. The experiments were

carried out with JPaX [HR01, GH03], a run-time verification tool consisting of two parts: an instrumentation module and an observer module. The instrumentation module produces an instrumented version of the program, which when executed, generates an event log with the information for the observer to determine the correctness of examined properties. Fig. 5 illustrates the situation.

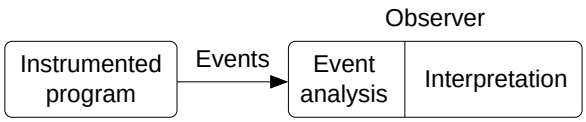

Figure 5: Structure of the run-time analysis.

The observation of events generated by the instrumented program is divided into two stages: an event analysis and an interpretation of events. The former reconstructs the context required for the event interpretation, while the latter contains the actual observation algorithms. The observer used here only checks for high-level data races. For these experiments, a new and yet totally un-optimized version of JPaX was used. It instruments every field access, regardless of whether it can be statically proven to be thread-safe. This is the reason why some data-intensive applications created log files which grew prohibitively large (> 0.5 GB) and could not be analyzed.

4.1 Java Bytecode Instrumentation

Part of JPaX is a very general and powerful instrumentation package for instrumenting Java bytecode [GH03]. The requirements of the instrumentation package include power, flexibility, ease of use, portability, and efficiency. Alternative approaches were rejected, such as instrumenting Java source code, using the debugging interface, and modifying the Java Virtual Machine because they violated one or another of these requirements.

It is essential to minimize the impact of the instrumentation on program execution. This is especially the case for real time applications, which may particularly benefit from this approach. Low-impact instrumentation may require careful trade-offs between the local computation of the instrumentation and the amount of data transmitted to the observer. The instrumentation package allows such trades to be made by allowing seamless insertion of Java code at any program point.

Code is instrumented based on an *instrument specification* consisting of a collection of predicate-action rules. A predicate is a filter on source code statements. These predicates are conjunctions of atomic predicates including predicates that distinguish statement types, presence of method invocations, pattern-matched references to fields and local variables, etc. The actions are specifications describing the inserted instrumentation code. Actions are inserted where predicates evaluate to true. The actions include reporting the program point (method, and source statement number), a time stamp, the executing thread, the statement type, the value of variables or an expression, and invocation of auxiliary methods. Values of primitive types are recorded in the event log. If the value is an object, a unique integer descriptor of the object is recorded.

The instrumentation has been implemented using Jtrek [Coh99], a Java API that provides lower-level instrumentation functionality. In general, use of bytecode instrumentation, and use of Jtrek in particular, has worked out well, but there are some remaining challenges with respect to instrumenting the concurrency aspects of program execution.

4.2 Event Stream Format

All operations in the instrumented application which write to the event log have to be as fast as possible. Among other factors, light-weight locking, incurring as little lock contention as possible, helps achieving this goal. When several pieces of information are logged by the instrumentation, they are therefore recorded separately, not atomically. As a result of this, one event can generate several log entries. Log entries of different threads may therefore be interleaved.

In order to allow a faithful reconstruction of the events, each log entry includes the hash code of the active thread creating the log entry. Therefore the events can all be assigned to the original threads. The contextual information in events includes thread names, code locations, and reentrant acquisitions of locks (lock counts). The event analysis package maintains a database with the full context of the event log.

4.3 Observer Architecture

As described above, run-time analysis is divided into two parts: instrumenting and running the instrumented program, which produces a series of events, and observing these events. The second part, event observation, can be split into two stages: event analysis, which reads the events and reconstructs the run-time context, and event interpretation (see Fig. 6). Note that there may be many event interpreters.

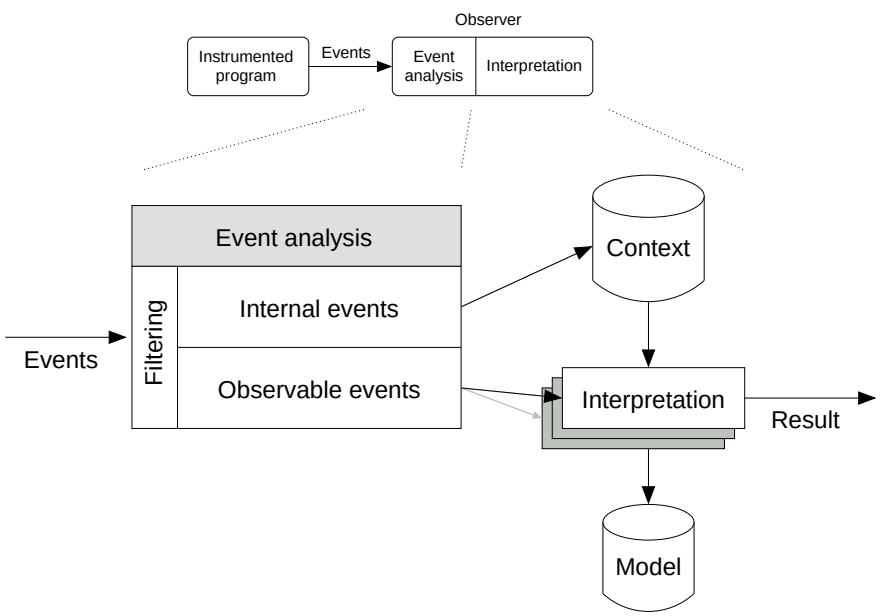

Figure 6: The observer architecture.

Reusing the context reconstruction module allows for writing simpler event interpreters, which can subscribe to particular event types made accessible through an observer interface [GHJV95] and which are completely decoupled from each other.

Each event interpreter builds its own model of the event trace, which may consist of dependency graphs or other data structures. It is up to the event interpreter to record all relevant information for keeping a history of events, since the context maintained by the event analysis changes dynamically with the event evaluation. Any

information that needs to be kept for the final output, in addition to context information, needs to be stored by the event interpreter. If an analysis detects violations of its rules in the model, it can then show the results using stored data.

Besides clearly separating two aspects of event evaluation, this approach has other advantages: Many algorithms dealing with multi-threading problems require very similar information, namely lock and field accesses. If a log generated by an instrumented program includes at least this information, then several analysis algorithms can share the same events. Furthermore, splitting event observation into two steps also allows writing an event analysis front-end for event logs generated by tools other than JPaX, reusing the back-end, event interpretation.

5 Experiments

Before analyzing applications, the implementation of the algorithm was tested using ten hand-crafted programs exhibiting different combinations of tuple accesses, such as the ones shown in Sec. 3.3. The test set included applications which contain high-level data races and others that do not. The primary purpose of this test set was to test the implementation of the view consistency algorithm. Furthermore, the tests served to fine-tune the output so it is presented in an easily readable manner. This makes evaluation of warnings fairly easy, as long as the semantics of fields used in conflicting views is known and it can be inferred whether these fields have to be used as an atomic tuple or not.

Once these tests ran successfully, four real applications were analyzed. Those applications include a discrete-event elevator simulator, and two task-parallel applications: SOR (Successive Over-Relaxation over a 2D grid), and a Travelling Salesman Problem (TSP) application [vPG01]. The latter two use worker threads [Lea97] to solve the global problem. In addition, a Java model of a NASA planetary rover controller, named K9, was analyzed. The original code is written in C++ and contains about 35,000 lines of code, while the Java model is a heavily abstracted version with 7,000 lines. Nevertheless, it still includes the original, very complex, synchronization patterns. Note that all Java foundation classes were excluded from the analysis. This would have increased the overlapping sets to a point where the analysis would have produced too many warnings: Every print statement using `System.out` would have resulted in overlapping sets with any other view containing an access to `System.out`.

Application	Size [LOC]	Number of classes	Run time [s], uninstrumented	Run time [s], instrumented	Log size [MB]	Warnings issued
Elevator	500	5	16.7	17.5	1.9	2
SOR	250	3	0.8	343.2	123.5	0
TSP, very small run (4 cities)	700	4	0.6	1.8	0.2	0
TSP, larger run (10 cities)			0.6	28.1	2.3	0
NASA's K9 Rover controller	7000	82	–	–	–	1

Table 3: Analysis results for the given example applications.

Table 3 summarizes the results of the experiments. All experiments were run on a Pentium III with a clock frequency of 750 MHz using Sun's Java 1.4 Virtual Machine, given 1 GB of memory. Only applications which could complete without running out of memory were considered. It should be noted that the overhead of the built-in Just-In-Time (JIT) compiler amounts to 0.4 s, so a run time of 0.6 s actually means only about 0.2 s

were used for executing the Java application. The Rover application could not be executed on the same machine where the other tests were run, so no time is given there.

It is obvious that certain applications using large data sets incurred a disproportionately high overhead in their instrumented version. Many examples passed the view consistency checks without any warnings reported. For the elevator example, two false warnings referred to two symmetrical cases. In both cases, three fields were involved in the conflict. In thread t_1, the views $V(t_1) = \{\{1,3\},\{3\},\{2,3\}\}$ were inconsistent with the maximal view $v_m = \{1,2,3\}$ of t_2. While this looks like a simple case, the interesting aspect is that one method in t_1 included a *conditional* access to field 1. If that branch had been executed, the view $\{2,3\}$ would actually have been $\{1,2,3\}$, and there would have been no inconsistency reported. Since not executing the branch corresponds to reading data and discarding the result, both warnings are false positives.

One warning was also reported for the NASA K9 rover code. It concerned six fields which were accessed by two threads in three methods. The responsible developer explained the large scope of the maximal view with six fields as an optimization, and hence it was not considered an error. The Remote Agent space craft controller was only available in LISP, so it could not be directly tested. However, the tool used was successfully applied to test cases reflecting different constellations including that particular high-level data race.

So far, experiments indicate that experienced programmers intuitively adhere to the principle of view consistency. Violations can be found, but are not very common, as shown in the experiments. Some optimizations produce warnings that constitute no error. Finally, the two false positives from the elevator example show that the definition of view consistency still needs some refinement.

6 Related Work

As described in Sec. 1, view consistency was partially inspired by the Eraser algorithm [SBN$^+$97]. Beyond this algorithm, related work does not only exist in software analysis, but also in database and hardware concurrency theory.

6.1 Static Analysis and Model Checking

Beyond Eraser, several static analysis tools exist that examine a program for low-level data races. The Jlint tool [AB01] is such an example. The ESC [DLNS98] tool is also based on static analysis, or more generally on theorem proving. It, however, requires annotation of the program, and does not appear to be as efficient as the Eraser algorithm in finding low-level data races. Dynamic tools have the advantage of having more precise information available in the execution trace. More heavyweight dynamic approaches include model checking, which explores all possible schedules in a program. Recently, model checkers have been developed that apply directly to programs (instead of just on models thereof). For example, the Java PathFinder system (JPF) developed by NASA [HP00, VHBP00], and similar systems [God97, CDH$^+$00, HS99, BPR01, Sto00]. Such systems, however, suffer from the state space explosion problem. An extension of Java PathFinder performs low-level data race analysis (and deadlock analysis) in simulation mode, whereafter the model checker is used to demonstrate whether the data race (deadlock) warnings are real or not [Hav00]. However, a data race, low-level as well as high-level, can be hard to find with model checking since it typically needs to cause a violation of some explicitly stated property.

High-level data races cover inconsistencies in value accesses. Another kind of fault that is closely related to high-level data races is the idea of *atomicity* of sequences of operations, such as an entire method [FQ03]. High-level data races do not cover such atomicity violations, although it is possible that an atomicity violation can lead to a high-level data race. Figure 7 shows a possible scenario where reading the value, performing an operation using it, and writing the result back are not carried out atomically. The result will be based on a possibly outdated value, because other threads may have updated the shared field, x, in the meantime. Because view consistency deals with sets of values, it cannot capture this kind of error, as shown by Wang and Stoller [WS03]. Only full knowledge about the desired atomicity can achieve this. A static analysis algorithm checking an implementation against an atomicity specification is presented in [FQ03]. Recently von Praun has shown that an extension of view consistency can be used to detect similar faults [vPG03]. It uses a *method view* which assumes that all shared fields accessed within the scope of a method should be used atomically.

```
synchronized(lock) {
  tmp = x.getValue();
}
tmp++;
synchronized(lock) {
  x.setValue(tmp);
}
```

Figure 7: A non-atomic operation that does not violate view consistency.

6.2 Database Concurrency

In database theory, shared data is stored in a database and accessed by different processes. Each process performs *transactions*, sequences of read and write operations, on the data. A sequence of these operations corresponding to several transaction is called a *history*. Based on this history, it can be inferred whether each transaction is *serializable*, i.e., whether its outcome corresponds to having run that transaction in isolation [Pap79, BHG87]. Database accesses try to avoid conflicts by construction, by structuring operations into transactions. The view consistency approach attempts to analyze behavior patterns in multi-threaded programs and to verify a similar behavior in an existing program.

There are several parallels to multi-threaded programs, which share their data in memory instead of in a database. Data races on shared fields in a multi-threaded program can be be mapped to database access conflicts on shared records. Lock protection in a multi-threaded program corresponds to an encapsulation of read and write accesses in a transaction. The key problem addressed by this paper, having intermediate states accessible when writing non-atomically a set of fields, maps to the *inconsistent retrieval* problem in databases. In such a history, one transaction reads some data items in between another transaction's updates on these items. A correct *transaction scheduler* will prevent such an access conflict, as long as the accesses of each process are correctly encapsulated in transactions.

High-level data races concern accesses to sets of fields, where different accesses use different sets. Similar problems may be seen in databases, if the programmer incorrectly defines transactions which are too fine-grained. For example, assume a system consists of a global database and an application using reading and writing threads. The writing threads use two transactions to update the database, the reading threads access

everything in a single transaction. Here, the reader's view is inconsistent, since it may read an intermediate state of the system. If the writer uses a single transaction, the fault is corrected. It is likely that the abstraction provided by database query languages such as SQL [CB76] prevents some of these problems occurring.

Furthermore, concurrency theory as used for databases and transaction systems is moving towards richer semantics and more general operations, called *activities* [SABS02]. Activities are atomic events in such a system. Like in classical transactions, low-level access conflicts are prevented by a scheduler which orders these operations.

Finally, database theory also uses the term *view* under different meanings. Specifically, the two terms *view equivalence* and *view serializability* are used [BHG87]. These two terms are independent of view consistency as defined in this paper.

So far, only single database systems have been covered. In *distributed databases*, the *virtual partitioning* algorithm exhibits a problem very similar to the view consistency problem presented here: Each transaction on an item operates on a *set of entries*, the set of all database entries for a single item, which is distributed on different sites. A *view* in this context is the set of sites with which a transaction is able to communicate [BHG87]. Ideally, a transaction has a view including all sites, so all updates are "atomic" on a global scale. However, communication delays and failures prevent this from being a practical solution. The virtual partitioning protocol [ASC85] ensures that all transactions have the same view of the copies of data that are functioning and those that are unavailable. Whereas a view in a distributed database corresponds to one data item which should be accessed atomically, a view as described in this paper encompasses sets of distinct data items. The applicability of ideas from this protocol to the view consistency model in the multi-threading domain looks promising.

6.3 Hardware Concurrency

In hardware design and compiler construction, Lamport has made a major step towards correct shared memory architectures for multiprocessors [Lam79]. He uses *sequential consistency* as a criterion for ensuring correctness of interleaved operations. It requires all data operations to appear to have executed atomically. The order in which these operations execute has to be consistent with the order seen by individual processes.

Herlihy and Wing use a different correctness condition called *linearizability* [HW90]. It provides the illusion that each operation applied by concurrent processes takes effect instantaneously at some point between its invocation and response. Linearizability is a stronger property than sequential consistency and has the advantage that it preserves real-time ordering of operations. Although the theory is very general, it is geared towards hardware and compiler construction because it allows exploiting special properties of concurrent objects where transactions would be too restrictive. However, it is not directly applicable to multi-valued objects and seems to be incapable of capturing such high-level problems.

Lamport's notion of sequential consistency is rather restrictive and can be relaxed such that processors are allowed to read older copies of data as long as the observed behavior is indistinguishable from a conventional shared memory system [ABM93]. Mittal and Garg extended this work and Herlihy's linearizability [HW90] to multi-object operations, such as double-register compare and swap operations [MG98]. Problems occurring with such multi-object operations are very much alike to high-level data races. Unlike the approach shown in this paper, which deals with access patterns, their approach is concerned with the interleaving of operations and based on histories as known in database literature.

7 Future Work

The most urgent problem is to make the run-time analysis more scalable. The instrumentation in the run-time analysis tool JPaX has to be optimized with respect to statically provable thread-safety. For instance, read-only or thread-local variables do not have to be monitored. Of course such an optimization has to be conservative and take possible aliasing of references into account. Another optimization is to execute logging instructions only a few times, instead of every time they are reached. A few executions of each instruction (one by each involved thread) are often enough to detect a problem.

Apart from that, the observer analysis could run on-the-fly without event logging. This would certainly eliminate most scalability problems. In addition to that, the current version reports the same conflict for different instances of the same object class. This results in a large error log, which contains redundant information.

On the theoretical side, it is not yet fully understood how to properly deal with nested locks. The views of the inner locks cause conflicts with the larger views of outer locks. These conflicts are spurious. The elevator case study has shown that a control-flow independent definition of view consistency is needed. Furthermore, data flow between field accesses has to be considered to refine the notion of high-level data races. A combination of static and dynamic analysis may be better suited to check such a revised definition. Finally, there is a need to study the relationship to database concurrency and hardware concurrency theory in more detail.

8 Conclusions

Data races denote concurrent access to shared variables with insufficient lock protection, leading to a corrupted program state. Classical, or low-level, data races concern accesses to single fields. The notion of high-level data races deals with accesses to sets of related fields which should be accessed atomically.

View consistency is a novel concept considering the association of variable sets to locks. This permits detecting high-level data races that can lead to an inconsistent program state, similar to classical low-level data races. Experiments on a small set of applications have shown that developers seem to follow the guideline of view consistency to a surprisingly large extent. Thus view consistency captures an important idea in multi-threading design.

Acknowledgements

Many thanks go to Christoph von Praun for kindly providing most of the example applications.

References

[AB01] C. Artho and A. Biere. Applying Static Analysis to Large-Scale, Multi-threaded Java Programs. In D. Grant, editor, *Proc. 13th ASWEC*, pages 68–75, Australia, 2001. IEEE Computer Society.

[ABM93] Y. Afek, G. Brown, and M. Merritt. Lazy Caching. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 15(1):182–205, 1993.

[AG96] K. Arnold and J. Gosling. *The Java Programming Language*. Addison-Wesley, 1996.

[AHB03] C. Artho, K. Havelund, and A. Biere. High-Level Data Races. In *VVEIS'03: The First International Workshop on Verification and Validation of Enterprise Information Systems*, France, 2003.

[ASC85] Amr El Abbadi, Dale Skeen, and Flaviu Cristian. An Efficient, Fault-Tolerant Protocol for Replicated Data Management. In *Proc. of the Fourth ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, pages 215–229, USA, 1985. ACM.

[BH03] S. Bensalem and K. Havelund. Reducing False Positives in Runtime Analysis of Deadlocks. Submitted for publication, 2003.

[BHG87] P. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[BPR01] T. Ball, A. Podelski, and S. Rajamani. Boolean and Cartesian Abstractions for Model Checking C Programs. In *Proc. TACAS'01: Tools and Algorithms for the Construction and Analysis of Systems*, LNCS, Italy, 2001.

[Bur00] M. Burrows. Personal communication, 2000.

[CB76] D. Chamberlin and R. Boyce. SEQUEL: A structured English query language. In *Proc. of the 1976 ACM SIGFIDET (now SIGMOD) workshop on Data description, access and control*, pages 249–264, 1976.

[CDH+00] J. Corbett, M. Dwyer, J. Hatcliff, C. Pasareanu, Robby, S. Laubach, and H. Zheng. Bandera: Extracting Finite-state Models from Java Source Code. In *Proc. 22nd International Conference on Software Engineering*, Ireland, 2000. ACM Press.

[Coh99] S. Cohen. Jtrek, 1999. Compaq,`http://www.compaq.com/java/download/jtrek`.

[DLNS98] D. Detlefs, R. Leino, G. Nelson, and J. Saxe. Extended Static Checking. Technical Report 159, Compaq Systems Research Center, Palo Alto, USA, 1998.

[FQ03] C. Flanagan and S. Qadeer. Types for Atomicity. In *Proc. Workshop on Types in Language Design and Implementation (TLDI)*, USA, 2003. ACM Press.

[GH03] A. Goldberg and K. Havelund. Instrumentation of Java Bytecode for Runtime Analysis. In *Proc. Formal Techniques for Java-like Programs*, volume 408 of *Technical Reports from ETH Zurich*, Switzerland, 2003. ETH Zurich.

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns – Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[God97] P. Godefroid. Model Checking for Programming Languages using VeriSoft. In *Proc. 24th ACM Symposium on Principles of Programming Languages*, pages 174–186, France, 1997.

[Har00] J. Harrow. Runtime Checking of Multithreaded Applications with Visual Threads. In *7th SPIN Workshop*, volume 1885 of *LNCS*, pages 331–342. Springer, 2000.

[Hav00] K. Havelund. Using Runtime Analysis to Guide Model Checking of Java Programs. In *SPIN Model Checking and Software Verification*, volume 1885 of *LNCS*, pages 245–264, USA, 2000. Springer.

[HLP01] K. Havelund, M. Lowry, and J. Penix. Formal Analysis of a Space Craft Controller using SPIN. *IEEE Transactions on Software Engineering*, 27(8):749–765, 2001. An earlier version occurred in the Proc. 4th SPIN workshop, 1998, Paris, France.

[HP00] K. Havelund and T. Pressburger. Model Checking Java Programs using Java PathFinder. *International Journal on Software Tools for Technology Transfer*, 2(4):366–381, 2000.

[HR01] K. Havelund and G. Roşu. Monitoring Java Programs with Java PathExplorer. In *Proc. First International Workshop on Runtime Verification (RV'01)*, volume 55 of *ENTCS*, pages 97–114, France, 2001. Elsevier.

[HS99] G. Holzmann and M. Smith. A Practical Method for Verifying Event-Driven Software. In *Proc. ICSE'99, International Conference on Software Engineering*, USA, 1999. IEEE/ACM.

[HW90] M. Herlihy and J. Wing. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3):463–492, 1990.

[Lam79] L. Lamport. How to Make a Multiprocessor that Correctly Executes Multiprocess Programs. *IEEE Trans. Comput.*, 9:690–691, 1979.

[Lea97] D. Lea. *Concurrent Programming in Java*. Addison-Wesley, 1997.

[Lea00] D. Lea. Personal e-mail communication, 2000.

[MG98] N. Mittal and V. Garg. Consistency Conditions for Multi-Object Distributed Operations. In *International Conference on Distributed Computing Systems*, pages 582–599, 1998.

[NBF98] B. Nichols, D. Buttlar, and J. Farrell. *Pthreads Programming*. O'Reilly, 1998.

[Pap79] C. Papadimitriou. The Serializability of Concurrent Database Updates. *Journal of the ACM (JACM)*, 26(4):631–653, 1979.

[PGK+97] B. Pell, E. Gat, R. Keesing, N. Muscettola, and B. Smith. Plan Execution for Autonomous Spacecrafts. In *Proc. of the International Joint Conference on Artificial Intelligence*, pages 1234–1239, Japan, 1997.

[SABS02] H. Schuldt, G. Alonso, C. Beeri, and H.-J. Schek. Atomicity and Isolation for Transactional Processes. *ACM Transactions on Database Systems (TODS)*, 27(1):63–116, 2002.

[SBN+97] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A Dynamic Data Race Detector for Multithreaded Programs. *ACM Transactions on Computer Systems*, 15(4):391–411, 1997.

[Sit00] Sitraka. JProbe, 2000. `http://www.sitraka.com/software/jprobe`.

[Sto00] S. Stoller. Model-Checking Multi-threaded Distributed Java Programs. In *SPIN Model Checking and Software Verification*, volume 1885 of *LNCS*, pages 224–244. Springer, 2000.

[Sun02] Sun Microsystems. *Java 2 Platform Enterprise Edition Specification*, 2002. `http://java.sun.com/j2ee`.

[VHBP00] W. Visser, K. Havelund, G. Brat, and S. Park. Model Checking Programs. In *Proc. ASE'2000: The 15th IEEE International Conference on Automated Software Engineering*. IEEE CS Press, 2000.

[vPG01] C. von Praun and T. Gross. Object-Race Detection. In *OOPSLA*, pages 70–82. ACM, 2001.

[vPG03] C. von Praun and T. Gross. Static Detection of Atomicity Violations in Object-Oriented Programs. In *Proc. Formal Techniques for Java-like Programs*, volume 408 of *Technical Reports from ETH Zurich*, Switzerland, 2003. ETH Zurich.

[WS03] L. Wang and S. Stoller. Run-Time Analysis for Atomicity. In *Proc. Third Workshop on Runtime Verification (RV)*, volume 89(2) of *ENTCS*, USA, 2003. Elsevier.