Checking Concise Specifications for Multithreaded Software

Stephen N. Freund[1] and Shaz Qadeer[2]

[1] Department of Computer Science, Williams College, Williamstown, MA 01267
[2] Microsoft Research, One Microsoft Way, Redmond, WA 98052

Abstract. Ensuring the reliability of multithreaded software systems is difficult due to the potential for subtle interactions between threads. Unfortunately, checking tools for such systems do not scale to programs with a large number of threads and procedures. To improve this shortcoming, we present a verification technique that uses concise specifications to analyze large multithreaded programs modularly. We achieve thread-modular analysis by annotating each shared variable by an *access predicate* that summarizes the condition under which a thread may access that variable. We achieve procedure-modular analysis by annotating each procedure by its specification, which is related to its implementation by an abstraction relation that combines the notions of *simulation* and *reduction*. We have implemented our analysis in Calvin-R, a static checker for multithreaded Java programs. To validate our methodology, we have used Calvin-R to check a number of important properties for a file system. Our experience shows that requirements for complex multithreaded systems can be stated concisely and verified in our framework.

Presented at the *Workshop on Formal Techniques for Java-like Programs*, 2003

1 Introduction

Software verification is an important and difficult problem. Over the past few decades, a variety of techniques based on dataflow analysis, theorem proving, and model checking have emerged for the analysis of sequential software. However, these techniques have not yet enabled verification of large, multithreaded software systems. Since concurrency is an insidious source of programming errors, multithreaded programs would benefit significantly from automated error-detection tools. The need for such tools will continue to grow as multithreaded software becomes even more widespread, expanding from the domain of low-level systems software (operating systems and databases) to most programs written in high-level languages like Java [2] and C# [12]. In this paper, we present a new modular verification technique for multithreaded Java programs.

Modularity is the key to scaling program analyses to large software systems. For sequential programs, modular analysis is achieved through pre- and post-conditions for procedures. However, due to interaction among the threads, pre- and post-conditions are insufficient for modular verification of multithreaded programs. Jones [22] proposed the first proof rule for modular verification of multithreaded programs. The proof rule of Jones required, in addition to pre- and post-conditions, a rely-guarantee specification for each procedure to capture the interaction among the threads. Both the rely and the guarantee specifications are actions (binary relations on the shared store). While the guarantee specification is a requirement on the updates performed by the thread executing the procedure, the rely specification is a requirement on the updates performed by the other threads.

In previous work [16], we extended Jones' method by generalizing the guarantee of a procedure from a single action to a program constructed from actions. This *guarantee program* has the property that every sequence of atomic updates to shared variables in the implementation is matched by a sequence of atomic updates in the specification. The implementation and guarantee of a procedure must be related via *simulation*. Simulation provides *data abstraction* for multithreaded programs by supporting abstract descriptions of the sequence of actions performed during the execution of a procedure. In particular, simulation allows steps that modify only local variables to be abstracted away. This generalization is crucial for allowing modular specification and verification of a multithreaded library independently from the clients of the library.

Unfortunately, simulation requires every step in a procedure's implementation that updates a shared variable to be matched by a step in its specification. Consider, for example, a multithreaded program in which the shared variable `count` is protected by the mutex `m`. The following procedure increments `count` by one.

```
void increment() { acquire(m); int j = count; j++; count = j; release(m); }
```

A specification that simulates the implementation must have at least three steps corresponding to acquiring `m`, updating `count`, and releasing `m`. Consequently, such a specification is no more concise or intuitive than the implementation. Although the programmer's intuition is that the execution of `increment` by a thread appears to happen in "one step" (rather than three), simulation does not allow such a specification to be proved.

We introduce a new and more expressive criterion for relating the implementation of a procedure to its specification in this paper. The new relation augments simulation with the notion of reduction, which was first introduced by Lipton [24]. The notion of reduction is based on commuting operations performed by different threads when they do not interfere with each other. An operation that commutes to the right of a succeeding operation by a different thread is a *right mover*, and an operation that commutes to the left of a preceding operation by a different thread is a *left mover*. For example, the operation `acquire(m)` is a right mover, and the operation `release(m)` is a left mover. Moreover, since all threads access `x` only while holding the mutex `m`, the read (and write) operation to `count` is both a right and a left mover since no other thread can concurrently access `count`. Any execution sequence in which a thread performs a sequence of right movers followed by a single atomic operation followed by a sequence of left movers can be viewed as occurring "in one step". The execution of `increment` by a thread has this property. To check that `increment` atomically increments `x` by one, we first apply reduction and then check simulation only on the reduced sequence. Thus, reduction allows control abstraction for multithreaded programs akin to pre- and post-conditions for sequential programs.

Our experience with multithreaded software checking indicates that the most intuitive and concise specifications for procedures in multithreaded programs are obtained by combining simulation and reduction. In fact, the specification of a procedure in our framework is often no more complex than its specification would be under the assumption that the program is single-threaded. We also show in Section 2.2 that the ability to use both simulation and reduction yields verifiable specifications that can not be expressed with just reduction or just simulation. We are not aware of any other automated checking tool that uses a combination of simulation and reduction to check abstraction.

In order to apply reduction to code sequences that access shared variables, the locking discipline for these shared variables must be specified by the programmer. Although mutexes are the most common synchronization discipline, there are a variety of other mechanisms used in practice [28, 14]. To capture these idioms, we introduce access predicates, a novel and general mechanism for specifying a wide variety of synchronization mechanisms including mutexes, readers-writer locks, data-dependent locking, etc. The access predicate expresses the condition under which a thread may access that variable. Our verification technique checks the code of each thread assuming that the environment (containing other threads) behaves according to the access predicates.

We have implemented our analysis in the Calvin-R checker, an extension of the Calvin checker for multithreaded Java programs [14, 16]. Our tool modularly checks that each method in a program satisfies the access predicates and is abstracted by its specification. For each check, Calvin-R constructs a sequential program capturing the necessary correctness requirements and verifies that it does not go wrong using existing verification techniques for sequential programs. Specifically, we employ verification conditions [10, 17] and the Simplify automatic theorem prover [26].

To validate our approach, we have used Calvin-R to check many properties of Daisy, a simple NFS file server we designed as our first case study. Daisy uses synchronization mechanisms similar in complexity to those found in other file systems. We have verified that all procedures in Daisy satisfy the access predicates, showing that the code adheres to the specified synchronization discipline. In addition, we specified and checked functional requirements on a number of the most complex procedures in Daisy. Our notion of abstraction invariably led to intuitive and concise specifications, and we uncovered several unknown bugs, primarily in the code for handling errors.

We present an overview of our verification technique in Section 2 through several examples. Due to space limitations, we omit the full formal presentation of our modular analysis and soundness proof. The full details may be found in our companion technical report [18]. We present a discussion of the related work in Section 3 and conclude in Section 4.

2 Verification Technique

To demonstrate our specification and verification system, we present two example programs— a class that implements a counter, and a class that implements block allocation in a simple file system. In both cases, we state concise specifications for the code and describe how our analysis checks them.

2.1 Counter

The following class implements a counter:

Figure 1: Counter

```
class Counter {
  int m /*@ accessible_if m == 0 || m == tid */;
  int count /*@ accessible_if m == 0 || m == tid */;

  /*@ performs action (count) { \old(m) == 0 && count == \old(count) + 1 } */
  void increment() {
      acquire(m); int j = count; j++; count = j; release(m);
  }
}
```

The `increment` method adds one to `count`. Concurrent calls to `increment` are serialized using the mutex lock `m`, which is acquired at the beginning and released at the end of `increment`. We model the mutex as an integer variable whose value is the identifier of the thread holding the mutex, or 0 if it is not held. The atomic operation `acquire(m)` blocks until `m` is 0 and then sets `m` to the identifier of the currently executing thread (`tid`), and the atomic operation `release(m)` sets `m` back to 0.

The `performs` annotation specifies method behavior. According to its specification, the `increment` method behaves as if at some point during its execution, it performs a single atomic action that (1) modifies only the variable `count`, which is indicated with the modifies clause `(count)`; and

(2) blocks until the value of `m` is 0 and then increments `count` by 1. The values `\old(m)` and `\old(count)` refer to the variable values in the pre-state of this action.

The `accessible_if` annotations indicate to our checker the access predicates for the variables. The access predicates for `m` and `count`, denoted $A_{\texttt{m}}$ and $A_{\texttt{count}}$, express the requirement that a thread t may access `m` and `count` only if $\texttt{m} = 0$ or $\texttt{m} = t$. It is worth noting that the access predicates for the shared variables do not preclude data races. As we have pointed out earlier [15], absence of data races is neither necessary nor sufficient for atomicity. Hence, our analysis allows general `accessible_if` predicates for shared variables.

The method `increment` can be called (possibly concurrently) by any number of threads. To ensure the specification of `increment` is valid for any calling thread, Calvin-R checks the method for an arbitrary thread t with the assumption that other threads operating concurrently with thread t access `m` and `count` according to the access predicates. For each execution trace of the method in such an environment, the tool checks both that thread t satisfies the access predicates and that the execution is abstracted by an execution of the specification of `increment` in the same environment. The first check can be easily performed using the technique of thread-modular verification [22, 14, 16].

To check abstraction, Calvin-R *reduces* the given execution to another execution that ends in the same final store and in which the operations performed by thread t (in `increment`) happen atomically without any interleaved actions by the environment. After reduction, the tool checks that the single atomic action in the specification of `increment` *simulates* the composition of the consecutively occurring actions of thread t.

To reduce an execution trace, the tool shows that the operations by thread t in the execution form a sequence of zero or more right-commuting operations (*right movers*) followed by a single operation followed by a sequence of zero or more left-commuting operations (*left movers*).

An operation of t is a right mover if, immediately after its execution, no other thread can access a variable accessed by the operation. Since the environment operations behave according to the access predicates, we can derive the condition to verify that an operation is right-mover from the access predicates. For example, if the operation by thread t accesses variable `m`, the condition $E_{\texttt{m}}(t)$ must be shown in the post-store:

$$\begin{aligned} E_{\texttt{m}}(t) &= \forall j \in \mathit{Tid}.\ j \neq t \Rightarrow \neg A_{\texttt{m}}(j) \\ &= \forall j \in \mathit{Tid}.\ j \neq t \Rightarrow \neg(\texttt{m} = 0 \vee \texttt{m} = j) \\ &= (\texttt{m} = t) \end{aligned}$$

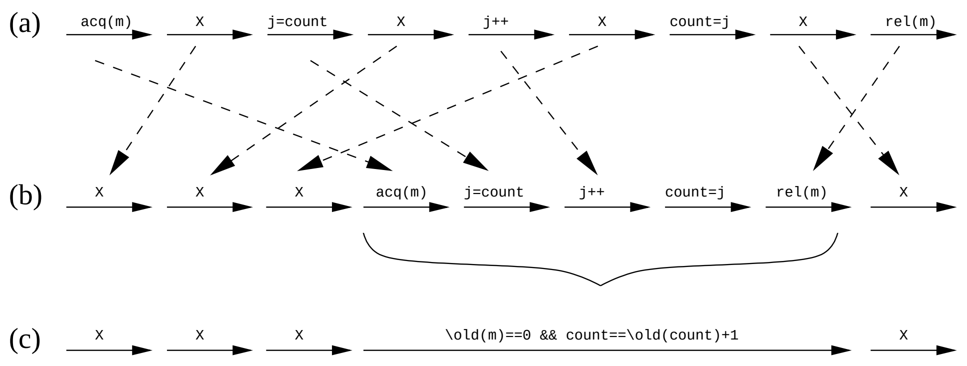
Thus, to prove that an operation by thread t accessing `m` is a right mover, we must show that thread t holds `m` in the post-store of the operation. Intuitively, the predicate $E_{\texttt{m}}(t)$ is the condition under which thread t has *exclusive access* to `m`.[1] The condition for an operation accessing `count` is identical. We may commute a right mover with that operation following it because we are guaranteed that the two operations access disjoint sets of variables.

Similarly, an operation is a left mover if we can prove that `m` is held by thread t in the pre-store. For `increment`, we can show that `acquire(m)` and all subsequent operations until `release(m)` are right movers and `release(m)` is a left mover. Therefore, the code in `increment` is reducible to a single action.

[1] Alternatively, we could have specified the exclusive access predicate and derived the access predicate from it.

Figure 2(a) shows an execution of `increment` with arbitrary actions of other threads and interleaved between the steps of t. Actions from other threads are labeled `X`, and for simplicity we insert only a single such action between consecutive actions of t. Execution (b) shows the reduced execution.

Figure 2: Checking abstraction for `increment`



The simulation check is straight-forward once the execution has been reduced, as shown in (c).

Note that the specification of `increment` shown above is comparable in complexity to the postcondition specification for `seq_increment`, a version of `increment` designed for sequential programs:

```
/*@ modifies count; ensures count == \old(count) + 1 */
void seq_increment() { int t = count; t++; count = t; }
```

2.2 Block allocation in Daisy

To further illustrate the importance of using both reduction and simulation for proving succinct procedure specifications, we present in Figure 3 the code for block allocation from the Daisy file system. Daisy is a simple NFS file system we designed as our first case study for Calvin-R. The file system synchronization mechanisms are similar in complexity to those found in other file systems. However, the data structures and algorithms in Daisy are relatively simple, allowing us to implement it in approximately 1200 lines of Java code.

The `alloc` function searches for a free file system block by finding a false bit in the `bits` array. The flag `bits[j]` indicates whether the j-th disk block is currently in use. When `alloc` identifies a free block, it allocates the block by setting the appropriate bit to true and returns the index of the block with the lock corresponding to it still held. (The caller will release the lock after it has finished initializing the data structures for the new block.) If `alloc` fails to find a free block, it returns -1. The `free` function takes a block index `i` as an argument and requires that the the mutex `l[i]` be held on entry to the function. It frees block `i` by setting `bits[i]` to false and returns after releasing `l[i]`.

The mutex `l[j]` guards the bit `bits[j]`. The `accessible_if` annotation on `bits` is parameterized by `j` to indicate this relationship for all `j`. A program acquires and releases lock `l[j]` by calling `acquire(l[j])` and `release(l[j])`, respectively. Locking at such a fine granularity is a standard

Figure 3: Allocator

```
class Allocator {
  static int l[NBLOCKS] /*@ accessible_if[j] l[j] == 0 || l[j] == tid */;
  static boolean bits[NBLOCKS] /*@ accessible_if[j] l[j] == 0 || l[j] == tid */;

  /*@ performs action "NoBlocks": () { \result == -1 }
           [] action "Allocated": (bits[\result], l[\result]) {
                \old(l[\result]) == 0 && l[\result] == tid
                !\old(bits[\result]) && bits[\result]
        }
  */
  public static int alloc() {
      for (int i = 0; i < NBLOCKS; i++) {
          acquire(l[i]);
          if (!bits[i]) {
              bits[i] = true;
              //@ witness "Allocated";
              return i;
          }
          release(l[i]);
      }
      //@ witness "NoBlocks";
      return -1;
  }

  /*@ requires l[i] == tid
      performs action "Free": (bits[i], l[i]) { l[i] == 0 && !bits[i] }
  */
  public static void free(int i) {
      bits[i] = false;
      release(l[i]);
      //@ witness "Free";
  }
}
```

technique for improving throughput in commercial file systems. However, it is also a major source of errors and demands substantial debugging effort. We are not aware of any other static tool for checking synchronization at such fine granularity.

The `performs` annotation for `alloc` is intuitive and mirrors the two possible outcomes of executing `alloc`. The specification is a choice between two atomic actions. In the first action, no free block is found and -1 is returned. The special variable `\result` refers to the value returned by a function. In the second action, the return value is the index of an unused block. This action blocks until the mutex protecting the allocation bit of the block is zero, and it then updates the bit from false to true. The `witness` annotations in the code indicate program points where simulation steps in the specification may occur. We use the explicit witness to guide the simulation check by indicating the "commit points" in the implementation of the atomic actions in the specification.[2] The specification for the `free` function is similar.

As in the `increment` example, Calvin-R checks that the implementation of `alloc` is abstracted by its specification for an arbitrary thread t in an environment that respects the access predicates. However, the checking of `alloc` is significantly more complicated than that of `increment`. The `alloc` function has an unbounded number of execution sequences, each consisting of 0 or more *acquire-test-release* subsequences, followed by either a return of -1 or an *acquire-test-set* and a return of

[2] In general, finding the correspondence between implementation steps and specification steps is a hard problem.

a non-negative index. Such executions are not reducible to a single atomic action. Therefore, our checker decomposes the sequence of actions performed by thread t into subsequences that are each reducible to a single action, as shown in Figure 4(a) and (b) for one possible execution.

Calvin-R deduces that each of the *acquire-test-release* subsequences is reducible to a single atomic action, and further checks that each of these actions is simulated by `skip`, an action that leaves every variable unchanged. If there is no final *acquire-test-set* sequence, then Calvin-R further deduces that the last implementation action returns `-1` and is therefore simulated by the action `"NoBlocks"` of the specification. If there is a final *acquire-test-set* sequence followed by the return of a non-negative index, Calvin-R reduces it to a single atomic action and checks that it is simulated by the action `"Allocated"` of the specification. In both cases, by first using reduction and then simulation, Calvin-R abstracts the execution to a (possibly empty) sequence of `skip` action followed by an action from the specification.[3] Figure 4 illustrates one possible execution of `alloc`. Execution (b) shows the reduced execution of (a), and (c) and (d) demonstrate the simulation. We divide the simulation into two steps to show that simulation involves composing a sequence of actions into a single action, as well as generalizing an action.

Although `alloc` uses fine-grained synchronization, our method allows us to prove a concise and intuitive specification that is similar in complexity to the specification of `alloc` assuming single-threaded execution.

3 Related work

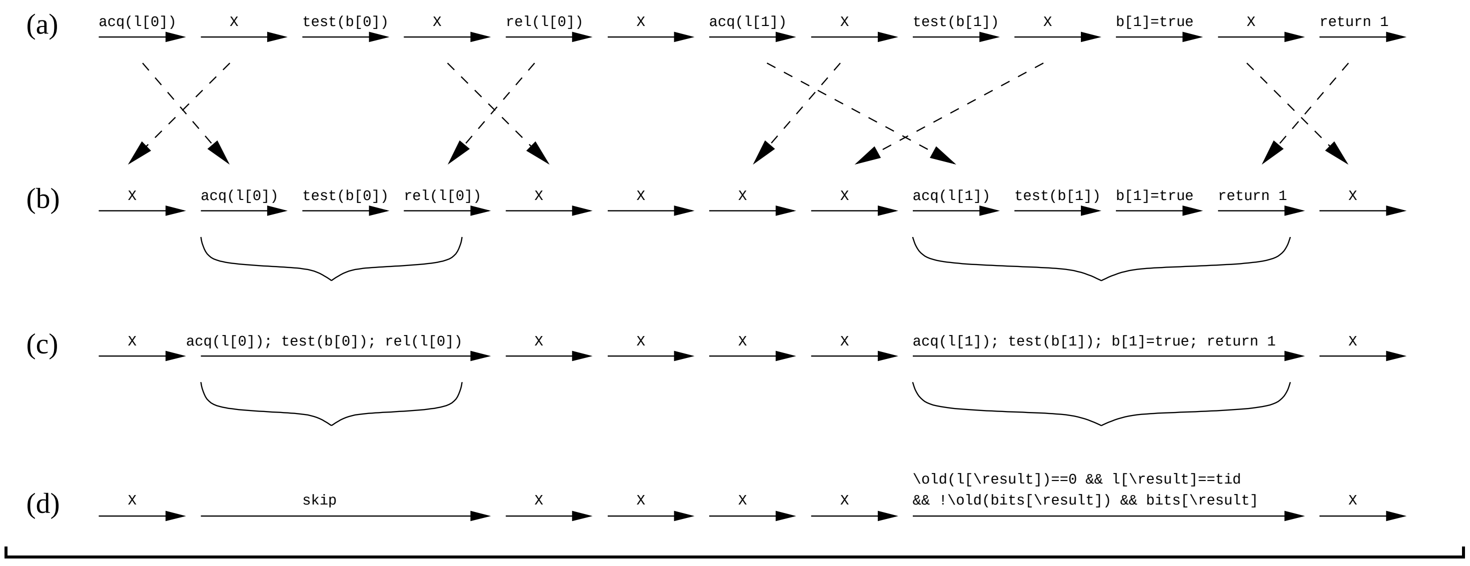
Lipton [24] first proposed reduction as a way to reason about concurrent programs without considering all possible interleavings. He focused primarily on checking deadlock freedom. Doeppner [11], Back [3], and Lamport and Schneider [23] extended this work to allow proofs of general safety properties. Misra [25] has proposed a reduction theorem for programs built with monitors [21] communicating via procedure calls. Cohen and Lamport [8] have extended reduction to allow proofs of liveness properties. All of these papers have focused on the theory of reduction. However, they do not describe a methodology for verifying programs. We go beyond their work by developing a specification and verification methodology for a widely used programming language.

Partial-order methods [20, 27] have been used to limit state-space explosion while model checking concurrent programs. These methods identify sequences of interleaved steps for which the property being checked is insensitive to the exact ordering. A single representative interleaving of the operations is then explored. These methods have mostly been applied to systems built from processes communicating through message passing. Verisoft [19] is an example of such a tool. While these methods are typically unable to reorder accesses to shared variables, Calvin-R uses access predicates to determine when it is safe to reorder accesses to shared variables as well.

Using ideas from reduction and partial-order methods, Bruening [5] has built an assertion checker based on state-space exploration for multithreaded Java programs. His tool requires another checker to ensure the absence of races. This assumption allows `synchronized` code blocks to be treated as atomic. Stoller [31] provides a generalization of Verisoft and Bruening's method to allow model checking of programs with either message-passing or shared-memory communication. Both of these approaches are restricted to mutex-based synchronization and operate on the concrete program without any abstraction. In our work, access predicates provide a more general mechanism for

[3] The `performs` specification implicitly allows arbitrary number of `skip` actions at any control point.

Figure 4: Checking abstraction for `alloc`



specifying synchronization. More recently, Stoller and Cohen have adopted access predicates in order to capture a richer set of synchronization idioms and to perform reduction during model checking [32].

Flanagan and Qadeer use reduction in a type system to identify procedures in multithreaded programs that may be considered to execute atomically [15]. Their syntactic type-based analysis is more scalable because it requires fewer and less complex program annotations, but it is limited to checking this single atomicity property. In contrast, our paper presents a modular semantic analysis based on verification conditions and theorem proving that can check many complex properties of multithreaded programs.

A number of static tools have been designed to detect data races. These include several type systems [13, 4], Warlock [30], and a race detector for SPMD programs [1]. Dynamic race detection tools [29, 6] require very few annotations, if any, but may fail to detect some errors due to insufficient coverage. Several tools combining dynamic and static analyses have recently been proposed [33, 7]. Access predicates provide a simple, general method for specifying and verifying a wider variety of synchronization mechanisms than allowed by these tools.

Several tools [9, 34] verify safety properties using a combination of data abstraction and model checking. These tools consider all possible thread interleavings while performing state exploration. The approach in this paper can be used to abstract a program, thereby reducing the possible interleavings. Invariant checking can then be performed on the abstract program thus improving the efficiency of these techniques.

4 Conclusions

Enforcing program specifications statically can greatly improve software quality. However, the inability to express complex properties in a simple, intuitive way is a major impediment to adopting many specification-based program checking tools, especially those targeting multithreaded code.

We outline in this paper a checking methodology that simplifies the task of specifying and verifying multithreaded programs. We allow thread synchronization mechanisms to be concisely expressed as access predicates, and we also allow concise, yet expressive specifications for procedures. The specification of a procedure is related to its implementation by a powerful abstraction relation that allows both data and control abstraction by combining the theories of simulation and reduction. Our access predicates and procedure specifications enable program analysis that is both thread-modular and procedure-modular. We have implemented our analysis in the Calvin-R checker for multithreaded Java programs and checked important properties of multithreaded systems with this tool.

The next step is to validate this methodology further by showing that it can scale to larger programs. Some issues to address are how to best map errors in the generated sequential program back to errors in the original, multithreaded program; how to reduce annotation overhead by automatically inferring some annotations, such as simple access predicates for variables guarded by mutual exclusion locks; and how to ensure that our translation does not produce sequential programs too complex for the underlying theorem prover to handle.

Acknowledgments: We thank Cormac Flanagan for the initial design of Daisy and for useful discussions and feedback on this paper. We also thank Sanjit Seshia for useful discussions, and Mark Lillibridge for comments on a draft of this paper.

References

1. A. Aiken and D. Gay. Barrier inference. In *Proceedings of the 25th Symposium on Principles of Programming Languages*, pages 243–354, 1998.
2. K. Arnold and J. Gosling. *The Java Programming Language*. Addison-Wesley, 1996.
3. R.-J. Back. A method for refining atomicity in parallel algorithms. In *PARLE 89: Parallel Architectures and Languages Europe*, Lecture Notes in Computer Science 366, pages 199–216. Springer-Verlag, 1989.
4. C. Boyapati and M. Rinard. A parameterized type system for race-free Java programs. In *Proceedings of 16th Annual Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 56–69, October 2001.
5. D. Bruening. Systematic testing of multithreaded java programs. Master's thesis, Massachusetts Institute of Technology, 1999.
6. G. Cheng, M. Feng, C. Leiserson, K. Randall, and A. Stark. Detecting data races in Cilk programs that use locks. In *Proceedings of the 10th annual ACM Symposium on Parallel Algorithms and Architectures*, pages 298–309, 1998.
7. 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 *Proceedings of the Conference on Programming Language Design and Implementation*, pages 258–269, 2002.
8. E. Cohen and L. Lamport. Reduction in TLA. In *International Conference on Concurrency Theory*, pages 317–331, 1998.
9. J. Corbett, M. Dwyer, J. Hatcliff, S. Laubach, C. Pasareanu, Robby, and H. Zheng. Bandera: extracting finite-state models from java source code. In *International Conference on Software Engineering*, pages 439–448, 2000.
10. E. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.
11. T. Doeppner, Jr. Parallel program correctness through refinement. In *Proceedings of the 4th Symposium on Principles of Programming Languages*, pages 155–169, 1977.
12. ECMA. Standard ECMA-334: C# Language Specification, 2002. Available on the web as `http://www.ecma-international.org/publications/files/ecma-st/Ecma-334.pdf`.
13. C. Flanagan and S. N. Freund. Type-based race detection for Java. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, pages 219–232, 2000.
14. C. Flanagan, S. N. Freund, and S. Qadeer. Thread-modular verification for shared-memory programs. In *Proceedings of European Symposium on Programming*, pages 262–277, 2002.
15. C. Flanagan and S. Qadeer. A type and effect system for atomicity. In *Proceedings of the ACM SIGPLAN '03 Conference on Programming Language Design and Implementation*, 2003.
16. C. Flanagan, S. Qadeer, and S. Seshia. A modular checker for multithreaded programs. In *CAV 02: Computer Aided Verification*, pages 180–194, 2002.
17. C. Flanagan and J. Saxe. Avoiding exponential explosion: Generating compact verification conditions. In *Proceedings of the 28th Symposium on Principles of Programming Languages*, pages 193–205. ACM, 2001.
18. S. N. Freund and S. Qadeer. Checking concise specifications of multithreaded software. Williams College Technical Note 01-2002 (available from `http://www.cs.williams.edu/~freund`), 2002.
19. P. Godefroid. Model checking for programming languages using VeriSoft. In *Proceedings of the 24th Symposium on Principles of Programming Languages*, pages 174–186, 1997.
20. P. Godefroid and P. Wolper. A partial approach to model checking. In *Proceedings of the Sixth Annual Symposium on Logic in Computer Science*, pages 406–415. IEEE Computer Society Press, 1991.
21. C. Hoare. Monitors: an operating systems structuring concept. *Communications of the ACM*, 17(10):549–557, 1974.
22. C. Jones. Specification and design of (parallel) programs. In R. Mason, editor, *Information Processing*, pages 321–332. Elsevier Science Publishers B. V. (North-Holland), 1983.
23. L. Lamport and F. Schneider. Pretending atomicity. Research Report 44, DEC Systems Research Center, 1989.
24. R. Lipton. Reduction: A method of proving properties of parallel programs. In *Communications of the ACM*, volume 18:12, pages 717–721, 1975.
25. J. Misra. *A Discipline of Multiprogramming: Programming Theory for Distributed Applications*. Springer-Verlag, 2001.
26. C. Nelson. Techniques for program verification. Technical Report CSL-81-10, Xerox Palo Alto Research Center, 1981.
27. D. Peled. Combining partial order reductions with on-the-fly model checking. In D. Dill, editor, *CAV 94: Computer Aided Verification*, Lecture Notes in Computer Science 818, pages 377–390. Springer-Verlag, 1994.
28. M. Rinard. Analysis of multithreaded programs. In *SAS 01: Static Analysis Symposium*, Lecture Notes in Computer Science 2126, pages 1–19. Springer-Verlag, 2001.
29. S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. E. Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems*, 15(4):391–411, 1997.
30. N. Sterling. WARLOCK — a static data race analysis tool. In *USENIX Technical Conference Proceedings*, pages 97–106, Winter 1993.
31. S. D. Stoller. Model-checking multi-threaded distributed java programs. In *Proceedings of the 7th International SPIN Workshop on Model Checking and Software Verification*, Lecture Notes in Computer Science 1885, pages 224–244. Springer-Verlag, 2000.
32. S. D. Stoller and E. Cohen. Optimistic synchronization-based state-space reduction. In H. Garavel and J. Hatcliff, editors, *Proceedings of the 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 2619 of *Lecture Notes in Computer Science*, pages 489–504. Springer-Verlag, 2003.
33. C. von Praun and T. Gross. Object-race detection. In *16th Annual Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 78–82, 2001.
34. E. Yahav. Verifying safety properties of concurrent Java programs using 3-valued logic. In *Proceedings of the 28th Symposium on Principles of Programming Languages*, pages 27–40, 2001.