JOURNAL OF OBJECT TECHNOLOGY
Online at www.jot.fm. Published by ETH Zurich, Chair of Software Engineering. ©JOT, 2002

Vol. 3, No. 6, June 2004

Checking Concise Specifications For Multithreaded Software

Stephen N. Freund, Department of Computer Science, Williams College, Williamstown, MA 01267
Shaz Qadeer, Microsoft Research, One Microsoft Way, Redmond, WA 98052

Ensuring the reliability of multithreaded software systems is difficult due to the potential for subtle interactions between threads. We present a new modular verification technique to check concise specifications of large multithreaded programs. Our analysis scales to systems with large numbers of procedures and threads. 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 with a specification related to its implementation by an abstraction relation combining the notions of *simulation* and *reduction*. We have implemented our analysis in Calvin-R, a static checker for multithreaded Java programs.

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 and C#. 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 threads, pre- and post-conditions are insufficient for modular verification of multithreaded programs. Jones [26] 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). The guarantee specification is a requirement on the updates performed by

Cite this article as follows: Stephen N. Freund and Shaz Qadeer: "Checking Concise Specifications For Multithreaded Software", in *Journal of Object Technology*, vol. 3, no. 6, June 2004, Special Issue: ECOOP 2003 Workshop of FTfJP, pages 81–101,
http://www.jot.fm/issues/issues_2004_06/article4

the thread executing the procedure, and the rely specification is a requirement on the updates performed by the other threads.

In previous work [19], 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 an abstract description of each action in 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.

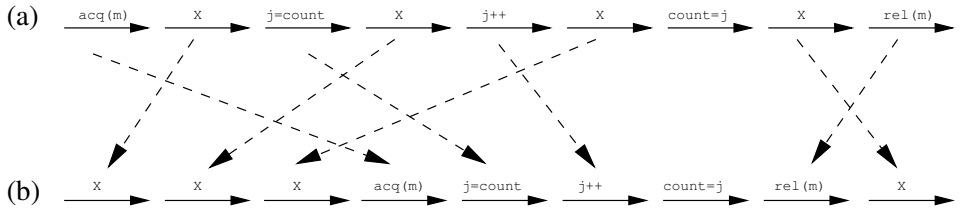
```
void increment() {
    acquire(m);
    int j = count;
    j++;
    count = j;
    release(m);
}
```

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", simulation does not by itself allow us to prove this.

We introduce a new and more expressive criterion for relating the implementation of a procedure to its specification. The new relation augments simulation with the notion of reduction, which was first introduced by Lipton [28]. 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 `count` only while holding the mutex `m`, the read from `count` and write to `count` are both right and left movers 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, as illustrated below. Execution trace (a) shows an execution of `increment` by a thread interleaved with arbitrary actions of other threads. Actions from other threads are labeled `X`, and for simplicity we insert only a single such action between consecutive actions of the thread executing `increment`. Execution (b) shows the reduced execution.



To check that `increment` atomically increments `count` by one, we first apply reduction and then check simulation only on the reduced sequence. Recent work by Flanagan, Freund, and Qadeer [17, 13] suggests that a large majority of methods in multithreaded programs are reducible to a single atomic step. For an atomic method, our new specification is no more complex than the pre- and post-condition that would be written for the method under the assumption that the program is single-threaded. Thus, we expect that for many methods, the intellectual complexity of using our specifications will be comparable to that involved in writing pre- and post-conditions.

The abstraction captured by our specification permits many non-atomic methods to be handled in an equally straightforward way. As an example, consider the following method that models allocation of resources. The method searches for a false bit in the `bits` array and allocates that "resource" by setting that bit to true. The resource `bits[`i`]` is protected by the lock `l[`i`]`, for each i. The lock for the allocated resource is held when the function returns, so that the caller has exclusive access to the resource. For example, in a file system, the resources could be disk I-nodes, and the caller may need to finish initializing the meta-data inside the I-node before releasing the lock on it.

```
public static int alloc() {
    for (int i = 0; i < NBLOCKS; i++) {
        acquire(l[i]);
        if (!bits[i]) {
            bits[i] = true;
            return i;
        }
        release(l[i]);
    }
    return -1;
}
```

Clearly, this method does not reduce to one atomic sequence because it may acquire and release a series of locks before finding a false bit. However, by taking advantage of both reduction and simulation, we are able to specify the functional behavior of

`alloc` as a single operation that selects and sets a false bit from the array (or returns -1 if no free resource is found). We develop such a specification and show how to check it in Section 2. 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, a variety of other mechanisms are used in practice [33, 14]. To capture these idioms, we introduce *access predicates*, a general mechanism for describing a wide variety of synchronization mechanisms including mutexes, readers-writer locks, data-dependent locking, etc. The access predicate for a variable expresses the condition under which a thread may access it. 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, 19]. 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 this program does not go wrong using existing verification techniques for sequential programs. Specifically, we employ verification conditions [9, 20] and the Simplify automatic theorem prover [30].

As a first case study, we used Calvin-R to check properties of Daisy, a simple multithreaded NFS file system designed to test 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. 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 for the most complex procedures in Daisy.

We present an overview of our verification technique in Section 2 through several examples. Section 3 introduces many of the details of our analysis. A more complete and formal presentation may be found in our companion technical report [21]. Section 4 discusses related work and Section 5 summarizes the contributions of this work and describes possible future directions.

2 OVERVIEW

We demonstrate our specification and verification system with two example classes. The first implements a counter, and the second 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.

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. We use `acquire` and `release` in place of the built-in Java synchronization operations only to simplify the technical development of our analysis.

The `performs` annotation specifies method behavior. According to `increment`'s specification, the 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. Since the value of the mutex variable `m` is 0 both at the beginning and the end of the atomic action, this variable does not appear in the modifies clause. The values `\old(m)` and `\old(count)` refer to the variable values in the pre-state of this action. Note that `\old(m)` and `\old(count)` do not refer to the variable values at the beginning of the procedure, as they might in pre- and post-conditions.

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 [17], absence of data races is neither necessary nor sufficient for atomicity.

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 [26, 14, 19].

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 for the method, 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*).

Suppose that immediately after executing an operation of thread t, no other thread can access a variable accessed by that operation. Then this operation can be commuted to the right of any operation by another thread and therefore it is a right mover. 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 $E_{\mathtt{m}}(t)$ must be shown in the post-store:

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

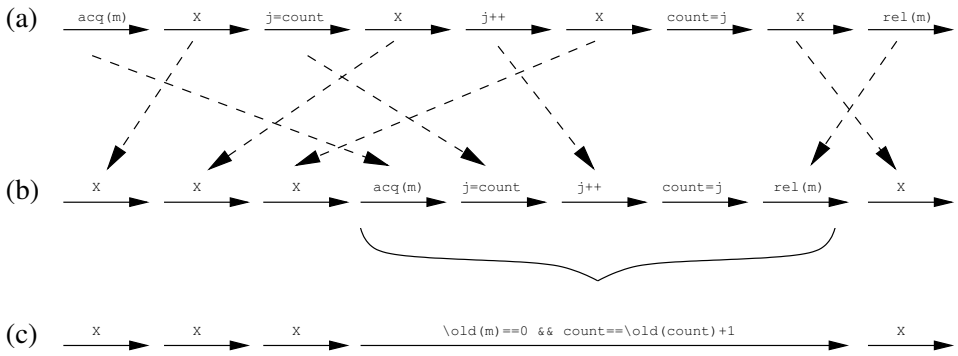
Thus, to prove that an operation by thread t accessing `m` is a right mover, we must show that thread t holds `m` in the post-store of the operation. Intuitively, the predicate $E_{\mathtt{m}}(t)$ is the condition under which thread t has *exclusive access* to `m`.[1] The condition for an operation accessing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

Figure 2(a) shows an execution of `increment` by thread t interleaved with arbitrary actions `X` of other threads, and execution (b) shows the reduced execution. The simulation check is straight-forward once the execution has been reduced, as shown in (c).

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

Figure 2: Checking abstraction for `increment`



Note that the specification of `increment` shown above is comparable in complexity to the post-condition 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; }
```

Our recent work suggests that a large fraction of methods in multithreaded systems are written to be free of interference and may be considered to execute atomically [13]. In these cases, Calvin-R's abstraction relation enables us to specify a method's behavior in a fashion no more complex than specifying a method's sequential behavior.

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 a simple file system implemented as an initial case study for Calvin-R.

The `alloc` method 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.) The `alloc` method returns -1 if it fails to find a free block.

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`. Locking at such a fine granularity is a standard technique for improving throughput in commercial file systems. However, it is also a major source of errors and demands substantial debugging effort.

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]) {
                0 <= \result && \result < NBLOCKS &&
                 \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";
  }
}
```

The `free` method takes a block index `i` as an argument and requires that the the mutex `l[i]` be held on entry to the method. It frees block `i` by setting `bits[i]` to false and returns after releasing `l[i]`.

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 method. 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. In general, it is difficult to infer the correspondence between implementation steps and specification steps automatically, but it is trivial for the programmer to specify the correspondence. The specification for the `free` method 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` method 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 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 *acquire-test-release* subsequence 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.[2] 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 VERIFICATION TECHNIQUE

Calvin-R modularly checks that each method satisfies (1) the access predicates and (2) its `performs` annotation. To check a method, Calvin-R first replaces method calls in its body with the desugaring of the called methods' `performs` annotations. This desugaring and inlining is routine and is described in [15]. As an example, the statement `x = alloc()` is desugared into the following code, written in Java extended with non-deterministic choice and `assert` e and `assume` e statements. The statement `assert` e goes wrong if it is executed when the boolean expression e is false. The statement `assume` e blocks indefinitely if executed when e is false.

```
x = random();
{ assume x == -1; } [] { assume 0 <= x && x < NBLOCKS &&
                         l[x] == 0 && !bits[x];
                       l[x] = tid; bits[x] = true;      }
```

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

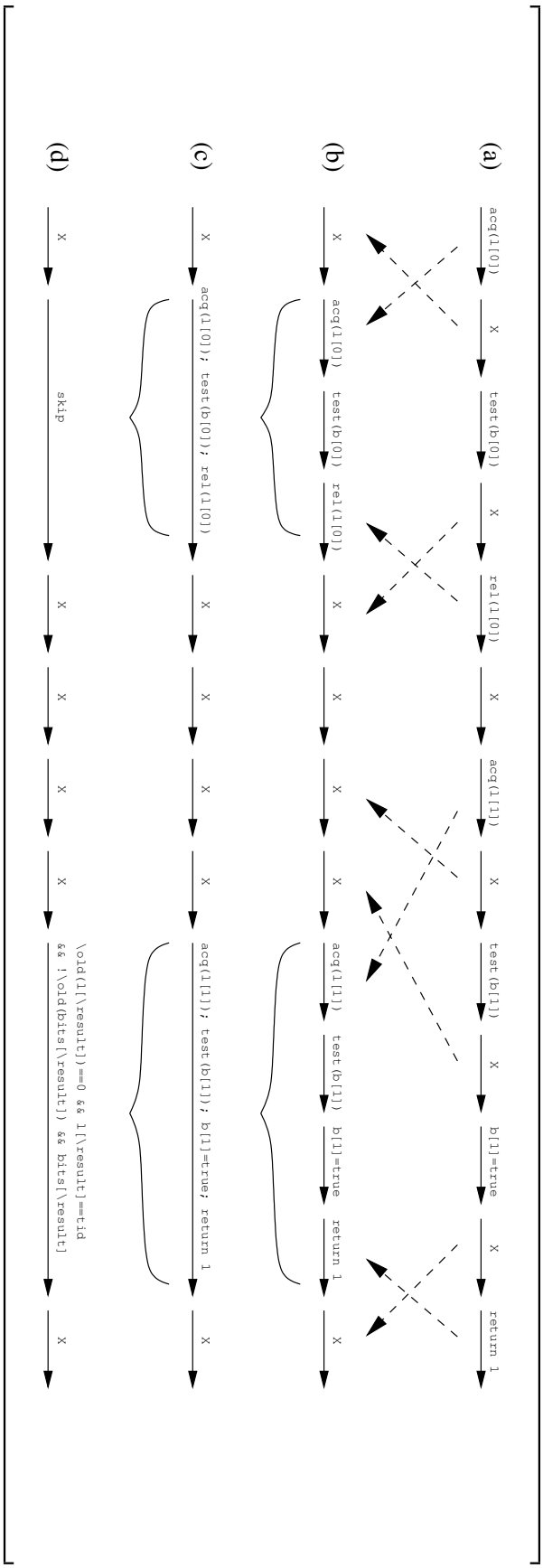
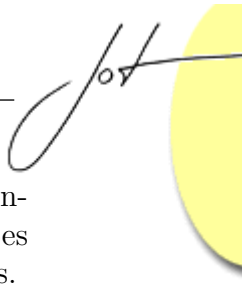


Figure 4: Checking abstraction for alloc



The first statement assigns an arbitrary value to `x`, and the second statement constrains that value in accordance with the specification of `alloc`. If a method does not have a `performs` annotation, Calvin-R inlines the method's body at call sites.

For each method, Calvin-R performs two checks: (1) the method conforms to the access predicates, and (2) the method satisfies its `performs` annotation. Each of these checks is performed in a two-step process. The first step translates the method's body into a sequential program that contains statements of the method, statements that model the effect of operations performed by other threads, and assertions that encode the check being performed. The second step verifies that none of these assertions fail by transforming the sequential program into a verification condition whose validity is verified by the Simplify theorem prover [30]. If the verification condition is valid, then the assertions hold and the check is satisfied.

We describe the sequential programs embodying the two checks for access predicates and `performs` annotations in this section. For simplicity, we present the transformations to sequential programs at the level of the Java language. The Calvin-R tool actually performs the transformations on methods after they have been converted into Plato, a simple guarded command language for multithreaded programs [21]. For a description of the translation of a sequential program into a verification condition, we refer the reader elsewhere [9, 20].

Checking access predicates

Before proceeding, we first introduce a new statement form to unify our treatment of both non-blocking and blocking operations (such as `acquire`). The statement "`when P do S`" blocks until the boolean predicate `P` is true and then atomically executes the statement `S`. For non-blocking operations, `P` is `true`. The operation `acquire(m)` may be written as "`when m == 0 do m = tid`". In the rest of this section, we assume that only statements written in this form access shared data. We introduce assignments to auxiliary local variables to handle expressions accessing shared data that are embedded in control structures. For example, "`if (e) S else T`" may be rewritten as "`boolean t = e; if (t) S else T`" if `e` accesses shared variables.

To check that a method violates no access predicates, we must check that each statement accessing shared variables in the body obeys the access predicates. We embed this check in our translation to a sequential program. We check that the statement "`when P do S`" from the original code violates no access predicates when executed by a thread `tid` running in parallel with other threads by translating it according to the translation function $\Longrightarrow_{\text{acc}}$ defined below. In this translation, V is the set of variables accessed by `P` and `S`.

$$\texttt{when P do S;} \quad \Longrightarrow_{\text{acc}} \quad EA^{*}\texttt{; assume P; assert } A_V\texttt{; S;}$$

The statement EA^{*} represents zero or more actions by any possible environment of

thread `tid`. Each action of the environment is represented by the statement EA, which changes the value of all shared variables to arbitrary values, except those for which `tid` has exclusive access. For the `increment` example, EA is the following:

$$EA \stackrel{\text{def}}{=} \begin{array}{l} \texttt{if (m != tid) then m = random();} \\ \texttt{if (m != tid) then count = random();} \end{array}$$

Note that if thread `tid` does not have exclusive access to a shared variable, we may make no assumption about the value stored in it.

Since "`when P do S`" is enabled only when `P` is true, we need only check that the statement conforms to the access predicates in states where `P` holds. Thus, we assert A_V only after assuming P is true. Note that the translation behaves exactly as "`when P do S`" when the access predicates are respected. Using the translation rule from above, we check access predicates for a method by translating every statement in the method, as demonstrated for `increment`:

Figure 5: Checking access predicates for `increment`

```
void increment() {
    acquire(m);        ⟹acc   EA*; assume m == 0; assert Am;m = tid;
    int j = count;     ⟹acc   EA*; assume true; assert Acount; int j = count;
    j++;               ⟹acc   EA*; assume true; assert true; j++;
    count = j;         ⟹acc   EA*; assume true; assert Acount; count = j;
    release(m);        ⟹acc   EA*; assume true; assert Am; m = 0;
}
```

If no assertion in the translated program fails when it is verified by a sequential checker, then the original method does not violate the access predicates.

Checking atomic `performs` annotations

We present the algorithm to check `performs` annotations in two stages. In the first stage, we show how to check abstraction for a method whose `performs` annotation is a single atomic action. In this setting, every execution path through the method must reduce to a single sequence abstracted by this atomic action. This property simplifies the checking problem, and we show a generalized translation to handle more expressive `performs` annotations in the next subsection.

To illustrate the algorithm, we consider the `increment` method. To check `increment`, we introduce an auxiliary variable `phase` that can take two values—`MatchingRight` and `MatchingLeft`. The `phase` variable tracks whether the method is executing in the right mover phase or the left mover phase. This variable is initialized to `MatchingRight` and is then set to `MatchingLeft` upon encountering the first action that is not a right mover. After that, every action must be a left mover.

An action is a right mover if the thread has exclusive access to all variables accessed by the action in the post-state. An action is a left mover if it does not

block and the thread has exclusive access to all variables accessed by the action in the pre-state. We require that the left movers do not block to ensure that the method eventually finishes executing the reducible sequence. In order to track the phase variable and ensure the entire method body is reducible to a single atomic block, we translate each statement that accesses shared variables "`when P do S`" according to the translation function $\Longrightarrow_{\text{abs}}$.

Figure 6: Checking abstraction for simple `performs` annotations

`when P do S;` $\Longrightarrow_{\text{abs}}$

```
if (phase == MatchingRight) {
    assume P; S;                          execute statement
    if (!E_V) phase = MatchingLeft;       if no exclusive access, then at commit point
} else {
    assert E_V && P;                      ensure exclusive access and non-blocking
    S;                                    execute statement
}
```

If a statement only accesses data local to the executing thread, no translation is necessary.

In addition to checking that a method body is atomic, we must also ensure that the pre-state and post-state of the method matches the `performs` annotation. For each program variable `x`, we introduce a variable `\old(x)` to store the value of `x` at the beginning of the procedure. At the end of the procedure, we check that the old and current values of the program variables are related according to the action in the `performs` annotation. The translation for `increment` is shown in Figure 7.

The assertions in this sequential program will succeed only if the steps of the original method are a sequence of right-movers followed by a sequence of left-movers, and if the overall effect of the method is to increment `count` by one.

Checking arbitrary `performs` annotations

We now generalize the checking algorithm to handle `performs` annotations constructed from atomic actions and the sequential composition (`;`), choice (`[]`), and iteration (`*`) operators.

The checking algorithm verifies that every execution path through the method is a sequence of disjoint subsequences, each of which is reducible to either `skip` or an action in the `performs` annotation. Moreover, the sequence of actions obtained in this way must be comprise a valid path through the stutter-closed program in the `performs` annotation. For example, if a method's performs annotation is `A ; (B [] C)`, every execution through the method's body must reduce to a sequence of atomic steps that are simulated by `skip*;A;skip*;B;skip*`, or `skip*;A;skip*;C;skip*`.

In order to check this requirement, the checker tracks the correspondence between the flow of control in the method body with the flow of control through the program embedded in the `performs` annotation with the variable `pc`. The value of `pc` is

Figure 7: Checking abstraction for `increment`

```
/*@ performs action (count) { \old(m) == 0 && count == \old(count) + 1 } */
void increment() {
                                phase = MatchingRight;
                                \old(m) = m; \old(count) = count;
                              { if (phase == MatchingRight) {
                                    assume m == 0; m = tid; if (!E_m) phase = MatchingLeft;
    acquire(m);     ⟹abs     } else {
                                    assert E_m && m == 0; m = tid;
                                }
                              { if (phase == MatchingRight) {
                                    assume true; j = count; if (!E_count) phase = MatchingLeft;
    j = count;      ⟹abs     } else {
                                    assert E_count && true; j = count;
                                }
    j++;            ⟹abs       j++;
                              { if (phase == MatchingRight) {
                                    assume true; count = j; if (!E_count) phase = MatchingLeft;
    count = j;      ⟹abs     } else {
                                    assert E_count && true; count = j;
                                }
                              { if (phase == MatchingRight) {
                                    assume true; m = 0; if (!E_m) phase = MatchingLeft;
    release(m);     ⟹abs     } else {
                                    assert E_m && true; m = 0;
                                }
                                assert \old(m) == 0 && count == \old(count) + 1;
}
```

the label of an action in the `performs` annotation, or "`begin`" or "`end`". At the beginning of the method, `pc` is initialized to "`begin`". The programmer must insert appropriate updates to `pc` in the method body with the `witness` annotation to identify where actions in the `performs` annotation are said to occur. Figure 3 demonstrates the use of labeled actions and witness annotations.

As before, the translated program begins by storing the value of each variable `x` in `\old(x)` and setting `phase` to `MatchingRight`. The `phase` variable also transitions from `MatchingRight` to `MatchingLeft` in the same way. However, we handle the case in which a non-left-mover is observed when `phase` is `MatchingLeft` differently. This situation occurs when the program reaches the end of a reducible subsequence. At this point, we check the values of the program variables in the current state against their values at the beginning of the reduced sequence (recall that the original values are stored in the `\old` variables). There are three possible outcomes:

1. If all program variables have the same values as at the beginning of the reduced sequence, then we may conclude that the whole sequence is simulated by `skip`.

2. If the variables have been modified according to $\texttt{Spec}_{\texttt{pc}}$, the action from the `performs` annotation labeled with the current value of `pc`, then the whole sequence is simulated by that action. To ensure that we are following a valid path through the specification, we check that `Follows(\old(pc), pc)`. This predicate is true only if the action labeled `pc` is a successor of the action labeled `\old(pc)` in the flow graph of the performs specification.

3. The variables have been modified in a way inconsistent with the `performs` annotation, and an error should be reported.

If no error is found, we prepare for the program to enter the next reducible region by inserting EA^* to model steps from other threads, resetting `phase` to `MatchingRight`, and storing the values of the program variables in the `\old` variables. Thus, we translate `when P do S` according to the following rule.

Figure 8: Check Abstraction

`when P do S;` $\Longrightarrow_{\text{abs}}$

```
if (phase == MatchingRight) {
    assume P; S;
    if (!E_V) phase = MatchingLeft;
} else if (E_V && P) {
    // matching left-movers and found left-mover
    assume P; S;
} else {
    // matching left-movers and found non-left-mover
    assert \old(x⃗) == x⃗ || Follows(\old(pc), pc) && Spec_pc;
    EA*;
    \old(x⃗) = x⃗;
    assume P; S;
    if (E_V) phase = MatchingRight;
}
```

The notation `\old(`$\overrightarrow{x}$`) = `$\overrightarrow{x}$ indicates that x is assigned to `\old(`x`)`, for all variables x that are accessed or modified in the body of the method. We show the translation of `alloc` in Figure 9. At `return` statements, assertions check that the program in the `performs` annotation can end after executing the action labeled `pc`.

4 RELATED WORK

Lipton [28] first proposed reduction as a way to reason about concurrent programs without considering all possible interleavings. He focused primarily on checking deadlock freedom. Doeppner [10], Back [2], and Lamport and Schneider [27] extended this work to allow proofs of general safety properties. Misra [29] has proposed a reduction theorem for programs built with monitors [25] communicating via procedure calls. Cohen and Lamport [7] have extended reduction to allow proofs of liveness properties. These papers focus on the theory of reduction, but 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 language.

Partial-order methods [23, 31] 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 of processes communicating through message passing. Verisoft [22] is one such 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.

Figure 9: Check abstraction for alloc

```
static int alloc() {
                                                  phase = MatchingRight;
                                                  pc = "begin";
                                                  \old(pc) = pc; \old(l) = l; \old(bits) = bits;
  for (int i = 0; i < NBLOCKS; i++) {    ⟹abs   for (int i = 0; i < NBLOCKS; i++) {
                                                    if (phase == MatchingRight) {
                                                        assume l[i]==0; acquire(l[i]);
                                                        if (!E_l[i]) phase = MatchingLeft;
                                                    } else if (E_l[i] && l[i]==0) {
  acquire(l[i]);                         ⟹abs       assume l[i]==0; acquire(l[i]);
                                                    } else {
                                                        YIELD;
                                                        assume l[i]==0; acquire(l[i]);
                                                        if (E_l[i]) phase = MatchingRight;
                                                    }
                                                    if (phase == MatchingRight) {
                                                        assume true; test = bits[i];
                                                        if (!E_bits[i]) phase = MatchingLeft;
                                                    } else if (E_bits[i] && true) {
  test = bits[i];                        ⟹abs       assume true; test = bits[i];
                                                    } else {
                                                        YIELD;
                                                        assume true; test = bits[i];
                                                        if (E_bits[i]) phase = MatchingRight;
                                                    }
  if (!test) {                           ⟹abs     if (!test) {
                                                      if (phase == MatchingRight) {
                                                          assume true; bits[i] = true;
                                                          if (!E_bits[i]) phase = MatchingLeft;
                                                      } else if (E_bits[i] && true) {
      bits[i] = true;                    ⟹abs           assume true; bits[i] = true;
                                                      } else {
                                                          YIELD;
                                                          assume true; bits[i] = true;
                                                          if (E_bits[i]) phase = MatchingRight;
                                                      }
      //@ witness "Allocated";           ⟹abs       pc = "Allocated";
                                                      \result = i;
      return i;                          ⟹abs       YIELD;
                                                      assert Follows(pc, "end");
                                                      return i;
    }                                               }
                                                    if (phase == MatchingRight) {
                                                        assume true; release(l[i]);
                                                        if (!E_l[i]) phase = MatchingLeft;
                                                    } else if (E_l[i] && true) {
    release(l[i]);                       ⟹abs       assume true; release(l[i]);
                                                    } else {
                                                        YIELD;
                                                        assume true; release(l[i]);
                                                        if (E_l[i]) phase = MatchingRight;
                                                    }
  }                                               }
  //@ witness "NoBlocks";                ⟹abs   pc = "NoBlocks";
                                                  \result = -1;
  return -1;                           ⟹abs     YIELD;
                                                  assert Follows(pc, "end");
                                                  return -1;
}

YIELD  def=  assert \old(pc) == pc && \old(l) == l && \old(bits) == bits
             || Follows(\old(pc), pc) && Spec_pc;
         EA*;
         \old(pc) = pc; \old(l) = l; \old(bits) = bits;
```

Dwyer et al. extended partial-order reduction to take advantage of object escape and locking information [11].

Using ideas from reduction and partial-order methods, Bruening [4] 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 [37] 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 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 [38]. A similar approach has recently been pursued by Flanagan and Qadeer [16].

Insights gained while developing Calvin-R have recently led us to develop several light-weight verification tools that focus on identifying procedures in multithreaded programs that may be considered to execute atomically. Flanagan and Qadeer use reduction in a syntactic type-based analysis to identify atomic procedures [18, 17]. The type system scales more easily because it requires fewer and less complex program annotations, but it is limited to checking this single atomicity property. Flanagan and Freund have implemented the Atomizer tool [13], which uses reduction to identify atomic blocks of code dynamically. Although the Atomizer is unsound because it does not check all execution paths, it requires little or no programmer-supplied information and has identified atomicity errors in a number of large systems. In other work, the notions of reduction and atomicity are used by Qadeer et al. [32] to infer concise procedure summaries in an analysis for multithreaded programs. Calvin-R's combination of simulation and reduction enable it to check many properties beyond the capabilities of these other checkers.

Hatcliff et al. [24] explore an alternative approach for verifying atomicity using model-checking. In addition to using Lipton's theory of reduction, they also investigate an approach based on partial-order reductions. Their experimental results suggest that the model-checking approach for verifying atomicity is feasible for unit-testing, where the reachable state space is smaller than in integration-testing. In related work, Robby et al. [34] demonstrate how to check some properties of multithreaded code with standard pre- and post-conditions by refactoring code into methods that do not update global state from critical sections nested inside methods.

A number of static tools have been designed to detect data races. These include several type systems [12, 3], Warlock [36], and a race detector for SPMD programs [1]. Dynamic race detection tools [35, 5] 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 [39, 6]. Access predicates provide a simple, general method for specifying and verifying a wider variety of synchronization mechanisms than allowed by these tools.

Several tools [8, 40] 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.

5 CONCLUSIONS

Calvin-R provides a way for programmers to specify and check complex properties of multithreaded programs in a simple, intuitive way. The access predicates used by Calvin-R enable synchronization mechanisms to be expressed in a simple and uniform way. In addition, the abstraction relation based on simulation and reduction permits both data and control abstraction in method specifications. Many properties of multithreaded code can be expressed concisely and checked with this technique.

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. Stephen Freund's work was supported in part by faculty research funds granted by Williams College and by the NSF under Grant CCR-0341387.

REFERENCES

[1] A. Aiken and D. Gay. Barrier inference. In *Proceedings of the ACM Symposium on the Principles of Programming Languages*, pages 243–354, 1998.

[2] R.-J. Back. A method for refining atomicity in parallel algorithms. In *PARLE 89: Parallel Architectures and Languages Europe*, volume 366 of *Lecture Notes in Computer Science*, pages 199–216. Springer-Verlag, 1989.

[3] 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, 2001.

[4] D. Bruening. Systematic testing of multithreaded Java programs. Master's thesis, Massachusetts Institute of Technology, 1999.

[5] 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.

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

[7] E. Cohen and L. Lamport. Reduction in TLA. In *International Conference on Concurrency Theory*, pages 317–331, 1998.

[8] 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.

[9] E. Dijkstra. *A Discipline of Programming.* Prentice-Hall, 1976.

[10] T. W. Doeppner, Jr. Parallel program correctness through refinement. In *Proceedings of the ACM Symposium on the Principles of Programming Languages*, pages 155–169, 1977.

[11] M. B. Dwyer, J. Hatcliff, V. R. Prasad, and Robby. Exploiting object escape and locking information in partial order reduction for concurrent object-oriented programs. *Formal Methods in System Design Journal*, 2004. (to appear).

[12] C. Flanagan and S. N. Freund. Type-based race detection for Java. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, pages 219–232, 2000.

[13] C. Flanagan and S. N. Freund. Atomizer: A dynamic atomicity checker for multithreaded programs. In *Proceedings of the ACM Symposium on the Principles of Programming Languages*, pages 256–267, 2004.

[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, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, pages 234–245, 2002.

[16] C. Flanagan and S. Qadeer. Transactions for software model checking. In *Proceedings of the Workshop on Software Model Checking*, pages 338–349, 2003.

[17] C. Flanagan and S. Qadeer. A type and effect system for atomicity. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, pages 338–349, 2003.

[18] C. Flanagan and S. Qadeer. Types for atomicity. In *Proceedings of the ACM Workshop on Types in Language Design and Implementation*, pages 1–12, 2003.

[19] C. Flanagan, S. Qadeer, and S. Seshia. A modular checker for multithreaded programs. In *CAV 02: Computer Aided Verification*, pages 180–194, 2002.

[20] C. Flanagan and J. B. Saxe. Avoiding exponential explosion: Generating compact verification conditions. In *Proceedings of the ACM Symposium on the Principles of Programming Languages*, pages 193–205, 2001.

[21] S. N. Freund and S. Qadeer. Checking concise specifications for multithreaded software. Technical Note 01-2002, Williams College, 2002.

[22] P. Godefroid. Model checking for programming languages using VeriSoft. In *Proceedings of the ACM Symposium on the Principles of Programming Languages*, pages 174–186, 1997.

[23] 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.

[24] J. Hatcliff, Robby, and M. B. Dwyer. Verifying atomicity specifications for concurrent object-oriented software using model-checking. In *Proceedings of the International Conference on Verification, Model Checking and Abstract Interpretation*, pages 175–190, 2004.

[25] C. Hoare. Monitors: an operating systems structuring concept. *Communications of the ACM*, 17(10):549–557, 1974.

[26] 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.

[27] L. Lamport and F. B. Schneider. Pretending atomicity. Research Report 44, DEC Systems Research Center, 1989.

[28] R. J. Lipton. Reduction: A method of proving properties of parallel programs. *Communications of the ACM*, 18(12):717–721, 1975.

[29] J. Misra. *A Discipline of Multiprogramming: Programming Theory for Distributed Applications*. Springer-Verlag, 2001.

[30] C. Nelson. Techniques for program verification. Technical Report CSL-81-10, Xerox Palo Alto Research Center, 1981.

[31] 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.

[32] S. Qadeer, S. K. Rajamani, and J. Rehof. Summarizing procedures in concurrent programs. In *Proceedings of the ACM Symposium on the Principles of Programming Languages*, pages 245–255, 2004.

[33] M. C. Rinard. Analysis of multithreaded programs. In *SAS 01: Static Analysis Symposium*, Lecture Notes in Computer Science 2126, pages 1–19. Springer-Verlag, 2001.

[34] Robby, E. Rodriguez, M. B. Dwyer, and J. Hatcliff. Checking strong specifications using an extensible software model checking framework. Technical Report SAnToS-TR-2003-10, Kansas State University, 2004.

[35] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. E. Anderson. Eraser: A dynamic data race detector for multi-threaded programs. *ACM Transactions on Computer Systems*, 15(4):391–411, 1997.

[36] N. Sterling. WARLOCK — a static data race analysis tool. In *USENIX Technical Conference Proceedings*, pages 97–106, Winter 1993.

[37] S. D. Stoller. Model-checking multi-threaded distributed Java programs. In *Workshop on Model Checking and Software Verification*, pages 224–244, 2000.

[38] S. D. Stoller and E. Cohen. Optimistic synchronization-based state-space reduction. In *Proceedings of the 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 489–504, 2003.

[39] C. von Praun and T. Gross. Object race detection. In *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages and Applications*, pages 70–82, 2001.

[40] E. Yahav. Verifying safety properties of concurrent Java programs using 3-valued logic. In *Proceedings of the ACM Symposium on the Principles of Programming Languages*, pages 27–40, 2001.