Runtime Model Checking of Multithreaded C/C++ Programs

Yu Yang Xiaofang Chen Ganesh Gopalakrishnan Robert M. Kirby
School of Computing, University of Utah
Salt Lake City, UT 84112, U.S.A.

ABSTRACT

We present `inspect`, a tool for model checking safety properties of multithreaded C/C++ programs where threads interact through shared variables and synchronization primitives. The given program is mechanically transformed into an instrumented version that yields control to a centralized scheduler around each such interaction. The scheduler first enables an arbitrary execution. It then explores alternative interleavings of the program. It avoids redundancy exploration through dynamic partial order reduction(DPOR) [1]. Our initial experience shows that `inspect` is effective in testing and debugging multithreaded C/C++ programs. We are not aware of DPOR having been implemented in such a setting. With `inspect`, we have been able to find many bugs in real applications.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification; D.2.5 [**Software Engineering**]: Testing and Debugging

General Terms

Verification, Threading

Keywords

dynamic partial order reduction, multithreaded, C/C++

1. INTRODUCTION

Writing correct multithreaded programs is difficult. Many "unexpected" thread interactions can only be manifested with intricate low-probability event sequences. As a result, they often escape conventional testing, and manifest years after code deployment. Many tools have been designed to address this problem. They can be generally classified into three categories: dynamic detection, static analysis, and model checking.

Eraser[2] and Helgrind[3] are two examples of data race detectors that dynamically track the set of locks held by shared objects during program execution. They use *locksets* to compute the intersection of all locks held when accessing shared objects. Shared object accesses that have an empty lockset intersection will be reported as being inconsistently protected, and as a result, potentially cause data races. Choi et al.[4] improve the Eraser algorithm by avoiding redundant analysis. Eraser (and similar tools) have been very successful in finding potential data races in multithreaded programs. However, as these tools try to detect potential data races by inferring them based on one feasible execution path, there is no guarantee that the program is free from data races if no error is reported (i.e., full coverage is not guaranteed). Besides, these tools can generate many false warnings. As these tools are designed for identifying data races only, they are not capable of detecting other safety violations such as deadlocks.

Tools such as RacerX[5], ESC/Java[6], and LockSmith[7] detect potential errors in the programs by statically analyzing the source code. Since they do not get the benefit of analyzing concrete executions, the false warning rates of these tools can be high. They also provide no guarantee of full coverage.

Traditional model checking can guarantee complete coverage, but on extracted finite state models (*e.g.*, [8, 9, 10, 11]) or in the context of languages whose interpreters can be easily modified for backtracking (*e.g.*, [12]). However, as far as we know, none of these model checkers can easily check (or be easily adapted to check) general application-level multithreaded C/C++ programs. For instance, if we want to follow Java PathFinder's [12] approach to check multithreaded C/C++ programs, we will have to build a virtual machine that can handle C/C++ programs. This is very involved. Model checkers like Bogor[9], Spin[13], Zing [11], etc. implicitly or explicitly extract a model out of the source code before model checking. However, modeling library functions and the runtime environment of C/C++ programs is very involved as well as error-prone: the gap between modeling languages and programming languages is unbridgeably large in many cases.

Blast[8] and Magic[10] use predicate abstraction and refinement technique to verify concurrent programs. Blast can be used to detect data races in nesC[14] programs. Magic focuses on detecting errors in concurrent programs that communicate via message passing. Again, writing correct library function stubs is a problem for these model checkers. Adapting these ideas to real-world C/C++ programs is also very

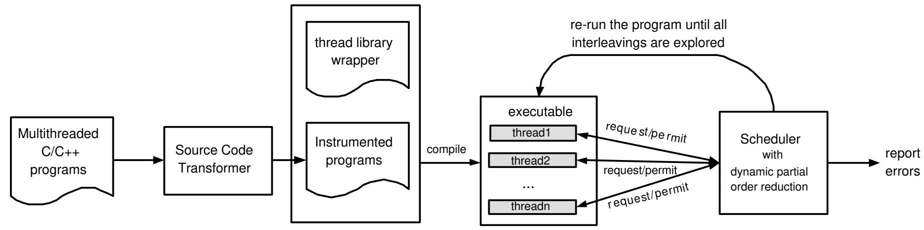

Figure 1: Inspect's workflow

difficult.

To the best of the authors' knowledge, Verisoft [15] is the only model checker that is able to check concurrent C/C++ programs without incurring modeling overheads. Unfortunately, Verisoft focuses on concurrent programs that interact only through inter-process communication mechanisms. In a real-world multithreaded program, the threads can affect each other not only through explicit synchronization/mutual exclusion primitives, but also through read/write operations on shared data objects.

To address these problems, we designed `inspect`, a runtime model checker for systematically exploring all possible interleavings of a multithreaded C/C++ program under a specific testing scenario. In other words, the reactive program under test is closed by providing a test driver, and `inspect` examines *all* thread interleavings under this driver.

An overview of `inspect` is shown in Figure 1. It consists of three parts: a source code transformer to instrument the program at the source code level, a thread library wrapper that helps intercept the thread library calls, and a centralized scheduler that schedules the interleaved executions of the threads. Given a multithreaded program, `inspect` first instruments the program with code that is used to communicate with the scheduler. Then `inspect` compiles the program into an executable and runs the executable repeatedly under the control of the scheduler until all relevant interleavings among the threads are explored. Before performing any operation that might have side effects on other threads, the instrumented program sends a request to the scheduler. The scheduler can block the requester by postponing a reply. We use blocking sockets as communication channels between the threads and the scheduler. As the number of possible interleavings grows exponentially with the size of the program, we implemented an adaption of the dynamic partial order reduction (DPOR [1]) algorithm proposed by Flanagan and Godefroid to reduce the search space. Such an implementation of DPOR in the context of threaded C/C++ programs is our first key contribution. Demonstrating the ability of `inspect` to find bugs in medium-sized public-domain applications is our second key contribution.

`Inspect` can check application-level C/C++ programs that use POSIX threads[16]. `Inspect` supports not only mutual exclusive lock/unlock, but operations on condition variables, including wait, signal and broadcast. The errors that `inspect` can detect include data races, deadlocks, and incorrect usages of thread library routines. When an error is located, `inspect` reports the error along with the trace that leads to the error, which facilities debugging. The key features of our work, and some of the challenges we overcame are as follows:

- We design `inspect`, an *in situ* runtime model checker, that can efficiently check multithreaded C/C++ programs. `Inspect` not only supports mutual exclusive locks, but also wait/signal, and read/write locks. The ability to model check programs containing these constructs makes `inspect` a unique tool.
- The *in situ* model checking capability runs the actual code, and not a formal model thereof. This eliminates the tedium of model extraction, and lends itself to an adaptation of the DPOR algorithm. Such a DPOR algorithm has, previously, not been implemented in the context of debugging large C/C++ programs that communicate using shared memory in a general way.
- We have evaluated `inspect` on a set of benchmarks and confirmed its efficacy in detecting bugs.
- We have designed and implemented an algorithm to automate the source code instrumentation for runtime model checking.
- Since `inspect` employs stateless search, it relies on re-execution to pursue alternate interleavings. However, during re-execution, the runtime environment of the threaded code can change. This can result in the threads not being allotted the same id by the operating system. Also, dynamically-created shared objects may not reside in the same physical memory address in different runs. We have suitably address these challenges in our work.
- We employ lock-sets, and additionally sleep-sets (the latter is recommended in [1]), to eliminate redundant backtrack points during DPOR.

2. BACKGROUND

2.1 Multithreaded Programs in C/C++

Threading is not part of the C/C++ language specification. Instead, it is supported by add-on libraries. Among many implementations of threading, POSIX threads [16] are perhaps the most widely used.

Mutex and *condition variable* are two common data structures for communication between threads. Mutexes are used to give threads exclusive access to critical sections. Condition variables are used for synchronization between threads. Each condition variable is always used together with an associated mutex. When a thread requires a particular condition to be true before it can proceed, it waits on the associated condition variable. By waiting, it gives up the lock and blocks itself. The operations of releasing the lock and blocking the thread should be atomic. Any thread that subsequently causes the condition to be true may then use the condition variable to notify a thread waiting for the condition. A thread that has been notified regains the lock and can then proceed. [1]

The POSIX thread library also provides *read-write locks* and *barriers*. A read-write lock allows concurrent read access to an object but requires exclusive access for write operations. Barrier provides explicit synchronization for a set of threads. As barriers seem not to be frequently used in multithreaded programs (we did not encounter any uses, except in some tutorials), we do not consider them here.

2.2 Formal Model of Multithreaded Programs

A multithreaded program can be modeled as a concurrent system, which consists of a finite set of *threads*, and a set of *shared objects*. Shared objects include *mutexes*, *condition variables*, *read/write locks*, and *data objects*. The state of a mutex, $Mutexes$, can be captured as a function from the mutex id($MutexId$) to the thread(Tid) which holds the mutex, and the set of threads(2^{Tid}) that are waiting for acquiring the mutex. A condition variable($Conds$) can be modeled in terms of the associated mutexes, along with the set of threads that are waiting on the condition variable. A read-write lock can be modeled in terms of a writer thread or a set of read threads, along with a set of threads waiting for read, and the other set of threads waiting for write. We denote this as $Rwlocks$.

$$
\begin{array}{rcl}
ObjId,\ Tid & \subset & \mathbb{N} \\
MutexId & \subset & ObjId \\
Mutexes & = & MutexId \rightarrow Tid \times 2^{Tid} \\
Conds & = & ObjId \rightarrow MutexId \times 2^{Tid} \\
Rwlocks & = & ObjId \rightarrow Tid \times 2^{Tid} \times 2^{Tid} \times 2^{Tid}
\end{array}
$$

Threads communicate with each other only through shared objects. Operations on shared objects are called *visible operations*, while the rest are *invisible operations*. The execution of an operation is said to *block* if it results in putting the calling threads into the waiting queue of a mutex, a condition variable, or a read/write lock. We assume that only the following can block a thread: visible operations on acquiring a mutex; acquiring a read/write lock; or waiting for a condition variable signal.

A state of a multithreaded program consists of the global state $Global$ of all shared objects, and the local state $Local$ of each thread. $Global$ includes $Mutexes$, $Conds$, and $Rwlocks$.

$$
\begin{array}{rcl}
Locals & = & Tid \rightarrow Local \\
State & = & Global \times Locals \\
Program & = & (State, s_0, \Delta) \\
\Delta & = & State \rightarrow State
\end{array}
$$

A *transition* moves the program from one state to the next state, by performing one visible operation of a certain thread, followed by a finite sequence of invisible operations, ending just before the next visible operation of that thread. A multithreaded program as a whole is denoted as $Program$, which is a triplet: the state space $State$, the initial state s_0, and the transition relation Δ.

[1] POSIX threads have condition wait and signal routines named `pthread_cond_wait`, `pthread_cond_signal`, `pthread_cond_broadcast`. On Microsoft Windows platforms, the correspondent APIs of the same semantics are `SleepConditionVariableCS`, `WakeConditionVariable` and `WakeAllConditionVariable`.

2.3 Runtime Model Checking

Model checking is a technique for verifying a transition system by exploring its state space. Cycles in the state space are detected by checking whether a state has been visited before or not. Usually the visited states information is stored in a hash table. Runtime model checkers explore the state space by executing the program concretely and observing its visible operations. Runtime model checkers do not keep the search history because it is not easy to capture and restore the state of a program which runs concretely. As a result, runtime model checkers are not capable of checking programs that have cyclic state spaces.

`Inspect` follows the common design principles of a runtime model checker, and uses a depth-first strategy to explore the state space. As a result, `inspect` can only handle programs that can terminate in a finite number of steps. Fortunately the execution of many multithreaded programs terminates eventually. [2]

2.4 Dynamic Partial Order Reduction

Partial order reduction (POR) techniques[17] are those that avoid interleaving independent transtions during search.

Given a state s and a transition t, let $t.tid$ denote the identity of the thread that executes t, and $next(s, t)$ refer to the state which is reached from s by executing t. Let $s.enabled$ denote the set of transitions that are enabled from s, and $s.sleep$ (*sleep sets* [18]) the set of transitions that are enabled in s but will not be executed from s (because doing so would only interleave independent transitions). A thread p is enabled in a state s if there exists transition t such that $t \in s.enabled$ and $t.tid = p$. Let $s.backtrack$ be the backtrack set at state s (Figure 2). $\{t \mid t.tid \in s.backtrack\}$ is the set of transitions which are enabled but have not been executed from s. Let $s.done$ be the set of threads examined at s, and let $\{t \mid t.tid \in s.done\}$ be the set of transitions that have been executed from s. Given the set of enabled transitions from a state s, partial order reduction algorithms try to explore only a (proper) subset of $s.enabled$, and at the same time guarantee that the properties of interest will be preserved. Such a subset is called *persistent set*, i.e. $s.persistent$.

In a finite transition sequence $T = t_1 t_2 ... t_n$, we say t_i happens before t_j if $i < j$ in every member of the equivalence class (Mazurkeiwitz trace set) of T obtained by permuting independent transitions [18].

Static POR algorithms compute the persistent set of a state s immediately after reaching it. In our context, persistent sets computed statically will be excessively large because of the limitations of static analysis. For instance, if two transitions leading out of s access an array `a[]` by indexing it at locations captured by expressions `e1` and `e2` (i.e., `a[e1]` and `a[e2]`), a static analyzer may not be able to decide whether `e1=e2`. Flanagan and Godefroid introduced dynamic partial-order reduction (DPOR) [1] to dynamically compute smaller persistent sets (smaller persistent sets are almost always better).

In DPOR, given a state s, $s.persistent$ is not computed immediately after reaching s. Instead, DPOR explores the states that can be reached from s with depth-first search, and dynamically computes $s.persistent$. Assume $t \in s.enabled$

[2] If termination is not guaranteed, `inspect` can still work by depth-bounding the search.

```
StateStack S;
TransitionSequence T;
Transition t;

DPOR( ) {
    State s = 𝕊.top;
    update_backtrack_info(s);
    if (∃ thread p, ∃t ∈ s.enabled, t.tid = p) {
        s.backtrack = {p};
        s.done = ∅;
        while (∃q ∈ s.backtrack) {
            s.done = s.done ∪ {q};
            s.backtrack = s.backtrack \ {q};
            let t_n ∈ s.enabled, t_n.tid = q;
            T.append(t_n);
            S.push(next(s, t_n));
            DPOR();
            T.pop_back();
            S.pop();
        }
    }
}

update_backtrack_info(State s) {
    for each thread p {
        let t_n ∈ s.enabled, t_n.tid = p;
        t_d = the latest transition in T that dependent
            and may be co-enabled with t_n;
        if (t_d ≠ null) {
            s_d = the state in 𝕊 from which t_d is executed;
            E = {q ∈ s_d.enabled | q = p or ∃t_j ∈ T, t_j
                happened after t_d, and is dependent with
                some transition of process p in 𝕋 that is
                happened after t_j }
            if (E ≠ ∅)
                add any q ∈ E to s_d.backtrack
            else
                add all enabled threads to s_d.backtrack;
        }
    }
}
```

Figure 2: Dynamic partial-order reduction

is the transition which the model checker chose to execute, and t' is a transition that can be enabled with DFS from s by executing t. For each to-be-executed transition t', DPOR will check whether t' and t are dependent and can be enabled concurrently (i.e. *co-enabled*). If t' and t are dependent and can be co-enabled, $t'.tid$ will be added to the $s.backtrack$. Later, when backtracking during DFS, if a state s is found with non-empty $s.backtrack$, DPOR will pick one transition t such that $t \in s.enabled$ and $t.tid \in s.backtrack$, and explore a new branch of the state space by executing t. Figure 2 recapitulates the DPOR algorithm (this is the same as the one given, as well as proved correct in [1]; we merely simplified some notations). In Section 4 we show how to adapt the DPOR algorithm for checking multithreaded C/C++ programs.

3. AN EXAMPLE

In this section we consider the following example, which captures a common concurrent scenario in database systems: Suppose that a shared database supports two distinct classes of operations, A and B. The semantics of the two operations allow multiple operations of the same class to run concurrently, but operations belongs to different classes cannot be run concurrently. Figure 3 is an implementation that attempts to solve this problem. `a_count` and `b_count` are the number of threads that are performing operations A and B respectively. Here, `lock` is used for the mutual exclusion between threads, and `mutex` is used for the mutual exclusion between threads of the same class. Could this code deadlock?

shared variables among threads:

```
pthread_mutex_t mutex, lock;
int a_count = 0, b_count = 0;
```

class A operation:

```
pthread_mutex_lock(&mutex);
a_count++;
if (a_count == 1) {
   pthred_mutex_lock(&lock);
}
pthread_mutex_unlock(&mutex);
performing class A operation;
pthread_mutex_lock(&mutex);
a_count--;
if (a_count == 0){
   pthread_mutex_unlock(&lock);
}
pthread_mutex_unlock(&mutex);
```

class B operation:

```
pthread_mutex_lock(&mutex);
b_count++;
if (b_count == 1){
  pthred_mutex_lock(&lock);
}
pthread_mutex_unlock(&mutex);
performing class B operation;
pthread_mutex_lock(&mutex);
b_count--;
if (b_count == 0){
   pthread_mutex_unlock(&lock);
}
pthread_mutex_unlock(&mutex);
```

Figure 3: An example on concurrent operations in a shared database

Conventional testing might miss the error as it runs with random scheduling. In general it is difficult to get a specific scheduling that will lead to the error. To systematically explore all possible interleavings, `inspect` needs to take the control of scheduling away from the operating system. We do this by instrumenting the program with code that is used to communicate with a central scheduler. As only visible operations in one thread can have side effects on other threads, we only need to instrument before each visible operation is performed. The instrumented code sends a request to the scheduler. The scheduler can then decide whether the request should be granted permission immediately, or be delayed. The scheduler works as an external observer. In addition, it needs to be notified about the occurrences of the thread start, join, and exit events. Figure 4 shows the code after instrumentation for threads that performs class A operations. As shown in the figure, each call to the pthread library routines is replaced with a wrapper routine. As `a_count` is a shared variable that multiple threads can access, we insert a read/write request to the scheduler before each access of `a_count`. A call to `inspect_thread_start` is inserted at the entry of the thread to notify that a new thread is started. Similarly, a call to `inspect_thread_end` is inserted at the end of the thread routine.

```
inspect_thread_start();
   ...
inspect_mutex_lock(&mutex);
inspect_obj_write( (void*)&a_count );
a_count++;
inspect_obj_read( (void*)&a_count );
if (a_count == 1)
   inspect_mutex_lock(&lock);
inspect_mutex_unlock(&mutex);
  ...
inspect_mutex_lock(&mutex);
inspect_obj_write( (void*)&a_count );
a_count--;
inspect_obj_read( (void*)&a_count );
if (a_count == 0)
   inspect_mutex_unlock(&lock);
inspect_mutex_unlock(&mutex);
  ...
inspect_thread_end();
```

Figure 4: Instrumented code for class A threads shown in Figure 3

After compiling the instrumented program, `inspect` obtains an executable that can be run under the central scheduler's control and monitoring. Firstly `inspect` lets the program run randomly and collects a sequence of visible operations, which reflects a random interleaving of threads. If it happens that an interleaving that can lead to errors, these errors will be reported immediately. (When `inspect` encounters an error, it does not stop immediately. Section 4.6 presents the details.) Otherwise, `inspect` will try to find a backtrack point out of the trace (as described in Figure 2), and begins monitoring the executable runs, now obtained through another interleaving.

Assume the program shown in Figure 3 has only one class A thread, and one class B thread. In the first run of the instrumented program, `inspect` may observe the visible operation sequence shown in Figure 5, beginning at "(*thread a*)" (which does not contain any errors).

While observing the random visible operation sequence, `inspect` will, for each visible operation, update the backtrack set for each state in the search stack. In the above trace, as event b_1 may happen before a_6, and b_1 and a_6 are both lock acquire operation on shared object `mutex`, `inspect` will put the backtracking information after event a_5 (i.e., just before a_6). Before `inspect` backtracks from a state, if its backtrack set is not empty, `inspect` will try to enable

(*thread a*) $\Rightarrow$	a_1 :	*acquire* `mutex`	
	a_2 :	`count_a` $++$	
	a_3 :	`count_a` $==$ 1	
	a_4 :	*acquire* `lock`	
	a_5 :	*release* `mutex`	
			$\leftarrow (\{b\}, \{a\})$
	a_6 :	*acquire* `mutex`	
	a_7 :	`count_a` $--$	
	a_8 :	`count_a` $==$ 0	
	a_9 :	*release* `lock`	
	a_{10} :	*release* `mutex`	
(*thread b*) $\Rightarrow$	b_1 :	*acquire* `mutex`	
	b_2 :	`count_b` $++$	
	b_3 :	`count_b` $==$ 1	
	b_4 :	*acquire* `lock`	
	b_5 :	*release* `mutex`	
	b_6 :	*acquire* `mutex`	
	b_7 :	`count_b` $--$	
	b_8 :	`count_b` $==$ 0	
	b_9 :	*release* `lock`	
	b_{10} :	*release* `mutex`	

Figure 5: A possible interleaving of one class A thread and one class B thread

the transition in the backtrack set to start exploring another branch of the state space. It, however, accomplishes this by *marking* that such a branch must be tried (the actual exploration is done following a re-execution from the initial state). In this example, (i) `inspect` will first re-execute the instrumented program and allow thread a run through $a_1 - a_5$. (ii) now, since the backtrack set contains b, the scheduler will block thread a until thread b has performed the visible operation b_1. Then it will allow thread a and thread b run randomly until all threads ends. In our example, we will observe the following alternate sequence of visible operations generated as a result of re-execution:

			$\leftarrow (\{b\}, \{a\})$
(*thread a*) $\Rightarrow$	a_1 :	*acquire* `mutex`	
	a_2 :	`count_a` $++$	
	a_3 :	`count_a` $==$ 1	
	a_4 :	*acquire* `lock`	
	a_5 :	*release* `mutex`	
(*thread b*) $\Rightarrow$	b_1 :	*acquire* `mutex`	
	b_2 :	`count_b` $++$	
	b_3 :	`count_b` $==$ 1	

After thread b performs the visible operation b_3, thread a is trying to acquire `mutex` which is held by thread b; at the same time, thread b is waiting to acquire `lock`, which is held by thread a. *`Inspect` will report this deadlock scenario at this point, and start backtracking.* As for the new interleaving, b_1 may happen before a_1, `inspect` will start another backtracking by having b_1 execute first. In general, `inspect` will continue the process of finding the backtrack point and re-running the program under test until there are no backtrack points in the search stack.

4. ALGORITHMS

4.1 Identifying threads and shared objects out of multiple runs

When `inspect` runs the program under test repeatedly, as the runtime environment may change across re-executions, each thread may not be allocated to the same id by the operating system. Also, dynamically-created shared objects may not reside in the same physical memory address in different runs.

One observation about thread execution is this: given the same external inputs, if the two runs of a thread program generate the same sequence of visible operations, then the constituent threads in this program should be created in the same order. Banking on this fact, we can identify threads (which may be allocated different thread IDs in various re-executions) across two different runs by examining the sequence of thread creations. In our implementation, we make each thread register itself in a mapping table, from system-allocated thread ids to integers. If the threads are created in the same sequential order in different runs, each thread will be assigned to the same id by this table. In the same manner, if two runs of the program have the same visible operation sequence, the shared objects will also be created with malloc, etc., in the same sequence. As a result, the same shared objects between multiple runs can be recognized in a similar way as threads.

4.2 Communicating with the scheduler

As explained before, `inspect` works by having the instrumented threads calling the scheduler before executing visible operations, and moving forward only based on the permissions being granted by the scheduler. The requests that a thread can send to the scheduler can be classified into four classes: 1) thread-management related events, including thread creation, thread destruction, thread join, etc.; 2) mutex-events, include mutex init, destroy, acquire, release; 3) read-write lock init, destroy, reader lock, writer lock,and unlock operations; 4) cond-related events, include condition variable creation, destroy, wait, signal, broadcast; and 5) data object related events, include the creation of the data object, read and write operations.

4.3 Handling wait and signal

Condition variable and the related wait/signal/broadcast routines are a necessity in many threading libraries. The wait/signal routines usually obey the following rules: (i) the call to a condition wait routine shall block on a condition variable; (ii) They shall be called with a mutex locked by the calling thread; (iii) The condition wait function atomically release mutex and causes the calling thread to block on the condition variable; and (iv) Upon successful return, the mutex shall have been locked and shall be owned by he calling thread.

A common problem (user bug) related to *wait* and *signal* operations is the "lost wake-up" caused by a thread executing a signal operation before the waiter goes into the waiting status. This can cause the waiter to block forever, since condition signals do not pend. We now explain how `inspect` takes care of this *so as to not mask such bugs.*

`Inspect` handles the condition variable related events by splitting the wait routine into three sub-operations: `pre_wait`, `wait`, and `post_wait`. Here, `pre_wait` releases the mutex that is held by the caller thread; `wait` changes the thread into blocking status and put the thread into the waiting queue of the correspondent condition variable; and `post_wait` tries to re-acquire the mutex again. The wrapper function for the `pthread_cond_wait` routine is shown in Figure 6.

In an instrumented program, when the wait routine is invoked, the calling thread t first sends a `pre_wait` request to the scheduler; the scheduler records that t releases the mutex, and set t's status as blocking. The scheduler will not send an "*unblocking*" permit to t until some other threads send out a related signal and t is picked out by the scheduler from the waiting queue. In t, after receiving the *unblocking* permit from the scheduler, t will send a `post_wait` request to acquire the mutex.

inspect_cond_wait(cond, mutex) {
  send *pre_wait* request;
  receive *pre_wait* permit;
  receive unblocking permit;
  send *post_wait* request;
  receive *post_wait* permit;
}

Figure 6: Wapper function for `pthread_cond_wait`

4.4 Avoiding Redundant Backtracking

Following [1], we assume that two transitions t_1 and t_2 are dependent if and only if they access the same communication object. We treat *wait* and *signal* operations on the same condition variable as dependent transitions. Also, a thread join operation is dependent with the correspondent thread exit.

In runtime model checking, backtracking is an expensive operation as we need to restart the program, and replay the program from the initial state until the backtrack point. To improve efficiency, we want to avoid backtracking as much as possible. Line 21 in Figure 2 is the place in DPOR where a backtrack point is identified. It treats t_d, which is dependent and may-be co-enabled with t_n as a backtrack point. However, *if two transitions that may be co-enabled are never co-enabled, we may end up exploring redundant backtrackings, and reduce the efficiency of DPOR.* Our two solutions – the use of locksets, and the use of sleep sets – are now discussed.

We use lockset to eliminate exploring transitions pairs that may not be co-enabled. Take the following trace as an example. It shows a program of two threads, both of which are trying to acquire locks `p` and `q`, and then release them. In thread b, before it acquires the lock `p`, it updates some shared variable `a`.

$$
\begin{array}{lll}
(\textit{thread } a) \Rightarrow & a_1: & \textit{acquire } \texttt{p} \\
 & a_2: & \textit{acquire } \texttt{q} \\
 & a_3: & \textit{release } \texttt{p} \\
 & a_4: & \textit{release } \texttt{q} \\
(\textit{thread } b) \Rightarrow & b_0: & \texttt{a++;} \\
 & b_1: & \textit{acquire } \texttt{p} \\
 & b_2: & \textit{acquire } \texttt{q} \\
 & b_3: & \textit{release } \texttt{p} \\
 & b_4: & \textit{release } \texttt{q}
\end{array}
$$

The algorithm in [1] relies on clock vectors, and the status of the process after taking a transition to infer whether two transitions may be co-enabled or not. Although that is a safe approximation, and will not affect correctness, this may lead to the result that the state before taking a_2 is a backtrack point because of the dependency between a_2 and b_2. However, this will lead to a redundant backtracking as

a_2 and b_2 cannot be co-enabled. (More specifically, as in [1], an attempt to run thread b starting with `a++` just before $a2$ is superfluous.)

To solve this problem, we associate with each transition t the set of locks that are held by the thread which executes t. Testing whether the intersection of the locksets that are held by the threads right after a_1 and b_1 can help us safely judge that the two transitions are mutually exclusive, and avoid redundant backtracking after a_1.

4.5 Using Sleep Sets for Further Reduction

While using locksets can avoid some redundant backtracking, conditional dependency and cond/signal make filtering out false "may-be co-enabled" transition pairs more difficult. Instead, we use sleep sets to further reduce the search space to detect the false "may-be co-enabled" transitions at runtime.

Figure 7 shows an example that has redundant backtrack points that is hard to detect by checking the transition sequence. It is a simplified dining philosopher problem: two philosophers competing for forks $f1$ and $f2$, both follow the order of get $f1$ first, and then $f2$. Two condition variables, `a_free` and `b_free` are used for synchronization between the philosophers. The lower part of Figure 7 shows a trace that `inspect` may explore with DPOR. In this trace, thread a first acquires the forks, and then releases $f1$, thread b takes $f1$ right after thread a releases it, and waits on thread a to release $f2$. After that, thread a releases $f2$ and notifies thread b that the fork is available. We only show the first two context switches between threads in Figure 7. In this trace, two lower backtrack points have been explored. The next backtrack points to be explored is right before a_5, as a_5 and b_5 are dependent transitions that appear to be co-enabled (actually they are not). However, enabling b_1, b_2 after a_4 will have thread b attaining blocking status waiting for signal $f1\_free$, a_5 will be enabled again. In this situation, stopping further depth-first search will not affect the correctness of model checking as the assumption that a_5 and b_5 may be co-enabled is wrong. We use sleep sets to achieve this.

The sleep set is a mechanism used to avoid the interleaving of independent transitions. It works by maintaining a set of transitions *sleep* such that whenever a new transition *new* is considered, if *new* is in sleep, then moving *new* can be considered to be un-necessary, and hence avoided [16]. Line 38-45 in Figure 8 shows how the sleep set are computed. In this example, while backtracking right after a_4, and executes b_1 and b_2, we will reach a state in which thread b is blocked, and the transition a_5 in the sleep set, which is not going to be executed. At this point, since there is no transition available for further exploration, we can backtrack immediately.

4.6 Runtime Model Checking with DPOR

Figure 8 shows how DPOR is adapted in the context of model checking multithreaded C/C++ programs. The algorithm has two phases: In the first phase, `inspect` executes the program for the first time under the monitoring of the scheduler, and collects a random visible operation sequence (lines 7-17). Any errors encountered will be reported. In the second phase, `inspect` does backtrack checking until all backtrack points are explored (lines 18-26).

In the replay mode in the second phase, we first rerun the program until the latest backtrack point (lines 31-36). The multithreaded program must be able to precisely follow the transition sequence. After that, we choose a new transition t from the backtrack set of the backtracking state s (line 37). In lines 38-40, we update *s.backtrack*, *s.done* and initialize *s.sleep*. After that, we will continue the depth-first search

shared variables :

```
pthread_mutex_t  f1, f2;
pthread_cond_t  f1_free, f2_free;
int f1_owner, f2_owner;
```

thread routine :

```
pthread_mutex_lock(&f1);
while ( f1_owner != 0 )
  pthread_cond_wait(&f1_free, &f1);
f1_owner = thread_id;
pthread_mutex_unlock(&f1);

pthread_mutex_lock(&f2);
while ( f2_owner != 0 )
  pthread_cond_wait(&f2_free, &f2);
f2_owner = thread_id;
pthread_mutex_unlock(&f2);

pthread_mutex_lock(&f1);
left_owner =0;
pthread_mutex_unlock(&f1);
pthread_cond_signal(&f1_free);

pthread_mutex_lock(&f2);
right_owner = 0;
pthread_mutex_unlock(&f2);
pthread_cond_signal(&f2_free);
```

			$\leftarrow (\{b\}, \{a\})$
$(thread\ a) \Rightarrow$	a_1 :	*acquire* `f1`	
	a_2 :	(`f1_owner` $!=0$)?	
	a_3 :	`f1_owner` $=$ `thread_id`	
	a_4 :	*release* `f1`	
			$\leftarrow (\{b\}, \{a\})$
	a_5 :	*acquire* `f2`	
	a_6 :	(`f2_owner`$!=0$)?	
	a_7 :	`f2_owner` $=$ `thread_id`	
	a_8 :	*release* `f2`	
			$\leftarrow (\{\}, \{a, b\})$
	a_9 :	*acquire* `f1`	
	a_{10} :	`f1_owner` $= 0$	
	a_{11} :	*release* `f1`	
	a_{12} :	*signal* `f2_free`	
$(thread\ b) \Rightarrow$	b_1 :	*acquire* `f1`	
	b_2 :	(`f1_owner` $!=0$)?	
	b_3 :	`f1_owner` $=$ `thread_id`	
	b_4 :	*release* `f1`	
			$\leftarrow (\{\}, \{a, b\})$
	b_5 :	*acquire* `f2`	
	b_6 :	(`f2_owner` $!=0$)?	
	b_7 :	*pre_wait* `f2_free`	
	b_8 :	*wait*	
$(thread\ a) \Rightarrow$	a_{13} :	*acquire* `f2`	
	a_{14} :	`f2_owner` $= 0$	
		...	

Figure 7: Simplified dining philosophers

while updating the sleep sets associated with each state. Line 45 shows how sleep sets are updated.

In the second phase, if a data race is detected, it will be reported on the fly. If a deadlock detected, *FoundDeadlockException*will be thrown out. When `inspect` catches such an exception, it will abort the current execution and start explore another backtrack point.

The backtrack point information is updated each time a new transition is appended to the transition sequence. We do not show the pseudo-code here. In `inspect`, we use the clock vector, lockset, along with the thread creation/join information to decide whether a pair of transitions may be co-enabled or not. As we divide *cond_wait* into three subtransitions, for a lock acquire, the appropriate backtracking points include not only the preceding lock acquire, but *post_wait* which is also a lock acquiring operation. For a *cond_signal*, the appropriate backtrack point is the latest *wait* on the signal.

4.7 Automated Instrumentation

`Inspect` needs to capture every visible operation to guarantee that it is not missing any bugs in the program. Incorrect instrumentation can make the scheduler fail to observe visible operations (viz., before execution of some visible operations, the program under test does not notify the scheduler). To automate the instrumentation process, we designed an algorithm as shown in Figure 9.

The automated instrumentation is primarily composed of three steps: (i) replace the call to the thread library routines with the call to the wrapper functions; (ii) before each visible operation on a data object, insert code to send a request to the scheduler; (iii) add *thread start* at the entry of every thread, and *thread end* at each exit point.

To achieve this, we need to know whether an update to a data object is a visible operation or not. The may-escape analysis [19] is used to discover the shared variables among threads. Because the result of may-escape analysis is an over-approximation of all-possible shared variables among threads, our instrumentation is safe for intercepting all visible operations in the concrete execution.

4.8 Detecting Bugs

`Inspect` detects data races and deadlocks while updating the backtracking information. If two transitions on a shared data object are enabled in the same state, `inspect` will report a data race. Deadlocks are detected by checking whether there is a cycle in resource dependency. `Inspect` keeps a resource dependent graph among threads, checks and updates the graph before every blocking transition.

Besides races and deadlocks, `inspect` can report incorrect usages of synchronization primitives. The incorrect usages include: (1) using an uninitialized mutex/condition variable; (2) not destroying mutex/condition variables after all threads exit; (3) releasing a lock that is held by another thread; (4) waiting on the same condition variable with different mutexes; (5) missing a pthread-exit call at the end of function `main`.

5. IMPLEMENTATION

`Inspect` is designed in a client/server style. The server side is the scheduler which controls the program's execution. The client side is linked with the program under test to communicate with the scheduler. The client side includes a wrapper for the pthread library, and facilities for commu-

```
1:  TransitionSequence T, T';
2:  StateStack S;
3:  State s, s';

4:  runtime_mc_with_DPOR( ) {
5:      run P, which is the program under test;
6:      s = the initial state of the program;
7:      try {
8:          while (s.enabled ≠ ∅) {
9:              S.push(s);
10:             choose t ∈ s.enabled;
11:             s = next(s, t);
12:             T.append(t);
13:             update_backtrack_info(); //defined in Figure 2
14:         }
15:     }
16:     catch(FoundDeadlockException){ ... }
17:     catch(AssertViolationException) { ... }

18:     while (¬S.empty()) {
19:         s = S.pop();
20:         T.pop_back(); // remove the last element of T
21:         if (s.backtrack ≠ ∅) {
22:             restart the program P;
23:             backtrack_checking(s);
24:             T = T';
25:         }
26:     }
27: }

28: backtrack_checking(State s_bt) {
29:     initialize T' to empty;
30:     try {
31:         s = the initial state of the program;
32:         while (s ≠ s_bt) {
33:             t = T.pop_front(); // remove the head of T
34:             T'.append(t);
35:             s = next(s, t);
36:         }
37:         choose t, t ∈ s.enabled ∧ t.tid ∈ s.backtrack;
38:         s.backtrack = s.backtrack \ {t.tid};
39:         s.sleep = {t ∈ s.enabled | t.tid ∈ s.done};
40:         s.done = s.done ∪ {t.tid};
41:         repeat
42:            S.push(s);
43:            T'.append(t);
44:            s' = next(s, t);
45:            s'.sleep = { t' ∈ s.sleep | (t, t') are indepen-
                 dent};
46:            s'.enabled = s'.enabled \ s'.sleep;
47:            s = s';
48:            update_backtrack_info();
49:            choose t ∈ s.enabled;
50:         until (s.enabled = ∅)
51:     }
52:     catch (FoundDeadlockException) { ... }
53:     catch (AssertViolationException) { ... }
54: }
```

Figure 8: Runtime model checking with DPOR

```
auto_instrument(program P) {
    have an inter-procedural escape analysis on P to find
      out all possible shared variables among threads;

    for each call of the thread library routines
        replace the call with the call to the correspondent
         wrapper function;
    for each  access of a shared variable v {
        if (read access)
            insert a reading request for v before reading v;
        else
            insert a write request for v before updating v;
    }
    for each entry of threads
        insert a thread start notification to the scheduler,
         before the first statement in the thread;
    for each exit point of threads
        insert a thread end notification after the last state-
         ment of the thread;
}
```

Figure 9: Automated instrumentation

nication with the scheduler.

We have the scheduler and the program under test communicate using Unix domain sockets. Comparing with Internet domain sockets, Unix domain sockets are more efficient as they do not have the protocol processing overhead, such as the network headers to add or remove, the check sums to calculate, the acknowledgments to send, etc. Besides, the Unix domain datagram service is reliable. Messages will neither be lost nor be delivered out of order.

In the automated instrumentation part, we first use CIL [20] as a pre-processor to simplify the code. Then we use our own program analysis and transformation framework based on `gcc`'s C front end to do the instrumentation. We first have an inter-procedural flow-sensitive alias analysis to compute the alias information. With the alias information, we use an inter-procedural escape analysis to discover the shared variables among threads. Finally we follow the algorithm in Figure 9 to do the source code transformation. Right now the automatic instrumentation can only work for C programs because of the lack of a front end for C++.

6. EXPERIMENTS AND EVALUATION

We evaluate `inspect` on two sets of benchmarks. The first set includes two benchmarks in [1]. The second set contains several small applications that use pthread on `sourceforge.net` and `freshmeat.net`[21, 22, 23, 24].

The performance of `inspect` for the benchmarks in [1] is shown in Table 1. The first program, *indexer*, captures the scenarios in which multiple threads insert messages into a hash table concurrently. The second benchmark, *fsbench*, is an abstraction of the synchronization idiom in Frangipani file system. We re-wrote the code using C and the POSIX thread library. The source code is available at [25]. In the original *indexer* benchmark, a compare-and-swap is used. As C does not have such an atomic routine, we replaced it with a function and used a mutex to guarantee the mutual exclusion. The execution time was measured on a PC with two Intel Pentium CPUs of 3.0GHz, and 2GB of memory. `inspect` was compiled with gcc-3.3.5 at optimization level `-O2`.

Table 1: Checking indexer and fsbench

	threads	runs	transitions	time(s)	runs/sec
indexer	1-11	1	$\leqslant$ 272	0.01	-
	13	64	6,033	1.44	44.44
	14	512	42,635	12.58	40.69
	15	4,096	351,520	108.74	37.68
	16	32,768	2,925,657	988.49	33.15
fsbench	1-13	1	$\leqslant$ 209	0.01	-
	16	8	1,242	0.14	-
	18	32	4,893	0.64	50
	20	128	20,599	2.76	46.38
	22	512	84,829	11.94	42.88
	24	2,048	367,786	54.82	37.36
	26	8,192	1,579,803	261.40	31.33

In Table 1, it shows that when the number of threads increases, more conflicts among threads slow down the program. However, `inspect` can still explore more than 30 different interleavings per second.

We also tried `inspect` on several small applications that use pthread on `sourceforge.net` and `freshmeat.net`. Table 2 shows the result. Application *aget*[21] is an ftp client in which multiple threads are used to download different segments of a large file concurrently. Application *pfscan*[22] is a multithreaded file scanner that combines the functionality of `find`, `xargs`, and `fgrep`. It uses multiple threads to search in parallel through directories. Application *tplay*[23] is a multimedia player that uses one thread to prefetch the audio data, and the other thread to play the audio. Finally, *libcprops*[24] is a C prototyping tools library which provides thread-safe data structures such as linked list, AVL tree, hash list, as well as a thread pool and thread management framework.

In Table 2, the second column LOC (lines of code) for each application is counted with `wc`. The right most column shows the number of errors we found. As `inspect` may report the same error multiple times while backtracking and re-executing the program repeatedly, we only count the unique number of errors.

For *aget*, we found one data race on writing the statistic data `bwritten` to a file. This data race is also reported in [7]. When testing *aget* with `inspect`, we need to construct a closed environment for it. As the network may introduce non-determinism to the environment, we reduced the size of the data package, which *aget* gets from the ftp server, to 512 bytes.

In *pfscan*, we found four errors. One error is that a condition variable is used without initialization. This is a dangerous behavior, and may completely mess up synchronization among threads and end up with incorrect results. In addition, two mutexes that are initialized at the beginning of the program never get released, which results in resource leakage. Also, we found that a `pthread_exit` was missing at the end of main. As a result, when the main thread exits, some worker threads may be killed before they completely finish their work. .

As for *libcprops*, it is a thread-safe library and test drivers are required for testing it. We adapted the test cases in *libcprops* release into multithreaded versions, and used them

Table 2: Checking real applications

benchmark		LOC	threads	Errors		
				races	deadlock	other errors
aget-0.4		1,098	3	1	0	0
pfscan-1.0		1,073	4	1	0	4
tplay-0.6.1		3,074	2	0	0	0
libcprops-0.1.6	avl	1,432	1-3	2	0	0
	heap	716	1-3	0	0	0
	hashlist	1,953	1-3	1	0	1
	linked_list	1,476	1-3	1	0	0
	splay_tree	1,211	1-3	1	0	0

as test drivers. `Inspect` revealed several data races in the code. After manually examining the source code, we found that most of the races are benign races. Besides, we also found that in *hashlist*, a condition variable is destroyed without initialization. This may lead to undefined behaviors.

6.1 Discussion

Our experiments show that `inspect` can be very helpful in testing and debugging multithreaded C/C++ applications. However, it also has limitations. First, `inspect` needs a set of test cases incorporated in its test driver to get good coverage of the code being verified. Secondly, runtime monitoring puts an overhead on the program, especially in programs that have a lot of visible operations on shared data objects. Also, the intrusive instrumentation limits `inspect` from checking programs that have strict timing requirements. As `inspect` checks the program's behavior by monitoring the concrete executions of the program, is not able to check system-level code like RacerX, LockSmith, etc., can do.

It is obvious that to check a program, we must be able to concretely execute the program. When doing our experiments, however, we also tried running several other open-source applications. Unfortunately, some problems were encountered: (i) some programs kept crashing because of other existing bugs; (ii) it is inconvenient to construct a closed world for server programs such as http servers. Other than these limitations, we think `inspect` is a powerful assistant tool in the process of unit testing and debugging for multithreaded software.

7. OTHER RELATED WORK

Lei et al.[26] designed RichTest, which used reachability testing to detect data races in concurrent programs. Reachability testing views an execution of a concurrent program as a partially-ordered synchronization sequence. Instead, dynamic partial order reduction views it as an interleaving of visible operations from multiple threads. Compared with RichTest, `inspect` focuses on checking multithreaded C/C++ programs, and it can detect not only data races, but also deadlocks and other errors. However, `inspect` cannot yet handle send/receive events between multiple processes.

CMC[27] verifies C/C++ programs by using a user-model Linux as a virtual machine. CMC captures the virtual machine's state as the state of a program. Unfortunately, CMC is not fully-automated. As CMC takes the whole kernel plus the user space as the state, it is not convenient for CMC to adapt the dynamic partial order reduction method.

ConTest[28] debugs multithreaded programs by injecting context switching code to randomly choose the threads to be executed. As randomness does not guarantee all interleavings will be explored for a certain input, it is possible that ConTest can miss bugs.

jCute[29] uses a combination of symbolic and concrete execution to check a multithreaded Java program by feeding it with different inputs and replaying the program with different schedules. jCute is more powerful in discovering inputs that can have the program execution take different paths. We think the difference between our work and jCute is in the implementation part. jCute uses the Java virtual machine to intercept visible operations of a multithreaded Java program. Here we use socket communication and an external scheduler for C/C++ programs.

Helmstetter et al.[30] show how to generate scheduling based on dynamic partial order reduction. We think that the differences between our work and theirs lie in: (i) We are focusing on application-level multithreaded C programs, while they focused on the schedulings of SystemC simulations; and (ii) Instead of generating the scheduling only, our work reruns the program and tries to verify safety properties.

CHESS[31] is the work which is probably most similar to ours. The difference between CHESS and our work lies in the instrumentation part and how to take control of the scheduling away from the operation system. In CHESS, the instrumentation allocates a semaphore for each thread that is created. It also requires an invariant to be preserved: that at any time every thread but one is blocked on its semaphore. In contrast, we do the instrumentation at the source code level, and use blocking sockets to communicate between scheduler and the threads.

8. CONCLUSION

In this paper, we propose a new approach to model check safety properties including deadlocks and stuttering invariants in multithreaded C/C++ programs. Our method works by automatically enumerating all possible interleavings of the threads in a multithreaded program, and forcing these interleavings to execute one by one. We use dynamic partial-order reduction to eliminate unnecessary explorations. Our preliminary results show that this method is promising for revealing bugs in real multithreaded C programs. Finally, `inspect` is available from [25].

In the future, `inspect` can be improved in many ways. We can combine the static analysis techniques with the dynamic partial order reduction to further reduce the number of interleavings we need to explore to reveal errors. `Inspect` can also adapt more efficient algorithms such as Goldilocks[32] for computing happen-before relations to improve efficiency. The automated instrumentation part can be improved by employing more efficient and precise pointer-alias analysis.

Acknowledgments: We thank Subdoh Sharma for helping implement the escape analysis part, and Sarvani Vakkalanka for comments. This work is funded by NSF CNS-0509379, SRC 2005-TJ-1318, and a grant from Microsoft.

9. REFERENCES

[1] Cormac Flanagan and Patrice Godefroid. Dynamic partial-order reduction for model checking software. In Jens Palsberg and Martín Abadi, editors, *POPL*, pages 110–121. ACM, 2005.

[2] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: a dynamic data race detector for multithreaded programs. *ACM Trans. Comput. Syst.*, 15(4):391–411, 1997.

[3] Nicholas Nethercote and Julian Seward. Valgrind: A program supervision framework. *Electr. Notes Theor. Comput. Sci.*, 89(2), 2003.

[4] Jong-Deok Choi, Keunwoo Lee, Alexey Loginov, Robert O'Callahan, Vivek Sarkar, and Manu Sridharan. Efficient and precise datarace detection for multithreaded object-oriented programs. In *Proceedings of the ACM SIGPLAN Conference on Programming language design and implementation*, pages 258–269, New York, NY, USA, 2002. ACM Press.

[5] Dawson Engler and Ken Ashcraft. Racerx: effective, static detection of race conditions and deadlocks. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 237–252, New York, NY, USA, 2003. ACM Press.

[6] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for java. In *Proceedings of the ACM SIGPLAN Conference on Programming language design and implementation*, pages 234–245, New York, NY, USA, 2002. ACM Press.

[7] Polyvios Pratikakis, Jeffrey S. Foster, and Michael Hicks. Locksmith: context-sensitive correlation analysis for race detection. In *Proceedings of the ACM SIGPLAN conference on Programming language design and implementation*, pages 320–331, New York, NY, USA, 2006. ACM Press.

[8] Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. Race checking by context inference. In *PLDI '04: Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, pages 1–13, New York, NY, USA, 2004. ACM Press.

[9] Robby, Matthew B. Dwyer, and John Hatcliff. Bogor: an extensible and highly-modular software model checking framework. In *ESEC / SIGSOFT FSE*, pages 267–276, 2003.

[10] Sagar Chaki, Edmund M. Clarke, Alex Groce, Somesh Jha, and Helmut Veith. Modular verification of software components in c. In *ICSE*, pages 385–395. IEEE Computer Society, 2003.

[11] Tony Andrews, Shaz Qadeer, Sriram K. Rajamani, Jakob Rehof, and Yichen Xie. Zing: A model checker for concurrent software. In *Computer Aided Verification, 16th International Conference, CAV 2004, Boston, MA, USA, July 13-17, 2004, Proceedings*, volume 3114 of *Lecture Notes in Computer Science*, pages 484–487. Springer, 2004.

[12] Willem Visser, Klaus Havelund, Guillaume P. Brat, and Seungjoon Park. Model checking programs. In *ASE*, pages 3–12, 2000.

[13] Gerard J. Holzmann. *The Spin Model Checker: Primer and Reference Manual*. Addison-Wesley, 2004.

[14] http://nescc.sourceforge.net/.

[15] Patrice Godefroid. Model checking for programming languages using verisoft. In *POPL*, pages 174–186, 1997.

[16] David R. Butenhof. *Programming with POSIX Threads*. Addison-Wesley, 1998.

[17] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. MIT Press, 2000.

[18] Patrice Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1996. Foreword By-Pierre Wolper.

[19] Alexandru Salcianu and Martin Rinard. Pointer and escape analysis for multithreaded programs. In *PPoPP '01: Proceedings of the eighth ACM SIGPLAN symposium on Principles and practices of parallel programming*, pages 12–23, New York, NY, USA, 2001. ACM Press.

[20] http://manju.cs.berkeley.edu/cil/.

[21] http://freshmeat.net/projects/aget/.

[22] http://freshmeat.net/projects/pfscan.

[23] http://tplay.sourceforge.net/.

[24] http://cprops.sourceforge.net/.

[25] http://www.cs.utah.edu/∼yuyang/inspect.

[26] Yu Lei and Richard H. Carver. Reachability testing of concurrent programs. *IEEE Trans. Software Eng.*, 32(6):382–403, 2006.

[27] Madanlal Musuvathi, David Y. W. Park, Andy Chou, Dawson R. Engler, and David L. Dill. Cmc: A pragmatic approach to model checking real code. In *OSDI*, 2002.

[28] Orit Edelstein, Eitan Farchi, Evgeny Goldin, Yarden Nir, Gil Ratsaby, and Shmuel Ur. Framework for testing multi-threaded java programs. *Concurrency and Computation: Practice and Experience*, 15(3-5):485–499, 2003.

[29] Koushik Sen and Gul Agha. Concolic testing of multithreaded programs and its application to testing security protocols. Technical Report UIUCDCS-R-2006-2676, University of Illinois at Urbana Champaign, 2006.

[30] Claude Helmstetter, Florence Maraninchi, Laurent Maillet-Contoz, and Matthieu Moy. Automatic generation of schedulings for improving the test coverage of systems-on-a-chip. *fmcad*, 0:171–178, 2006.

[31] http://research.microsoft.com/projects/CHESS/.

[32] Tayfun Elmas, Shaz Qadeer, and Serdar Tasiran. Goldilocks: Efficiently computing the happens-before relation using locksets. In *Formal Approaches to Software Testing and Runtime Verification, LNCS*, pages 193–208, Berlin, Germany, 2006. Springer.