KISS: Keep It Simple and Sequential

Shaz Qadeer
Microsoft Research
One Microsoft Way
Redmond, WA 98052

Dinghao Wu
Department of Computer Science
Princeton University
Princeton, NJ 08544

ABSTRACT

The design of concurrent programs is error-prone due to the interaction between concurrently executing threads. Traditional automated techniques for finding errors in concurrent programs, such as model checking, explore all possible thread interleavings. Since the number of thread interleavings increases exponentially with the number of threads, such analyses have high computational complexity. In this paper, we present a novel analysis technique for concurrent programs that avoids this exponential complexity. Our analysis transforms a concurrent program into a sequential program that simulates the execution of a large subset of the behaviors of the concurrent program. The sequential program is then analyzed by a tool that only needs to understand the semantics of sequential execution. Our technique *never* reports false errors but may miss errors. We have implemented the technique in KISS, an automated checker for multithreaded C programs, and obtained promising initial results by using KISS to detect race conditions in Windows device drivers.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification—*assertion checkers, model checking, formal methods*; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs—*assertions, mechanical verification*; F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages—*program analysis*

General Terms

Verification, Reliability

Keywords

Model checking, race detection, program analysis, concurrent software, assertion checking

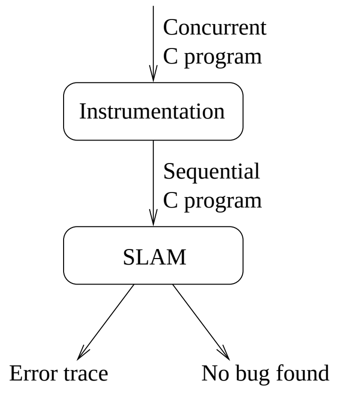

Figure 1: The KISS architecture.

1. INTRODUCTION

The design of concurrent programs is a complex endeavor. The main intellectual difficulty of this task lies in reasoning about the interaction between concurrently executing threads. Concurrency results in insidious programming errors that are difficult to reproduce, locate, and fix. Therefore, analysis techniques that can automatically detect and pinpoint errors in concurrent programs can be invaluable. In this paper, we present and evaluate such an analysis technique for detecting violations of safety properties in concurrent programs.

Traditionally, model checkers [27, 23, 11, 35] have been used to check safety properties of concurrent programs. These tools ensure high coverage of program behavior by exploring all possible thread interleavings. But the large coverage comes at the price of computational complexity. The analysis must explore the set of all reachable control states of the concurrent program. This set grows exponentially with the number of threads, thus severely limiting the scalability of the analysis. The subject of this paper is a new technique for avoiding this exponential complexity in analyzing interleavings.

Our analysis is based on a technique to transform a concurrent program P into a sequential program P' that simulates the execution of a large subset of the behaviors of P. The program P' is then analyzed by a checker that only needs to understand the semantics of sequential execution. In particular, the checker *does not* need to understand thread interleavings. The transformation has the property that if P' goes wrong by failing an assertion, then P must also go wrong by failing an assertion. Moreover, the error trace leading to the assertion failure in P is easily constructed from the error trace in P'.

The problem of checking safety properties on concurrent programs with finite data is undecidable. Ramalingam [33] has shown that the undecidable problem of checking the emptiness of the intersection of context-free languages is reducible to this problem. Our technique provides a complete (no false errors) but unsound (may miss errors) reduction of this undecidable problem to the problem of checking safety properties on sequential programs (with finite data). The latter problem is decidable [37, 34] and there are several efficient tools for solving it [8, 15, 13, 3, 25]. Thus, our technique provides a method for finding errors in concurrent programs by leveraging a variety of analysis techniques developed for sequential programs.

We have implemented our technique in a tool called KISS (Keep It Simple and Sequential!). KISS is an assertion checker for multithreaded C programs built on top of the SLAM [3] model checker for sequential C programs. It would be straightforward to adapt our technique to other similar tools such as PREfix [8], MC [15], ESP [13], and Blast [25]. The architecture of KISS is illustrated in Figure 1. KISS transforms the control flow graph of the input multithreaded C program into the control flow graph of a sequential C program, which is then analyzed by SLAM. An error trace produced by SLAM is transformed into an error trace of the original concurrent program. We have applied KISS to the problem of detecting race conditions in Windows NT device drivers. So far, KISS has analyzed 18 drivers ranging from 1 KLOC to 10 KLOC for a total of 70 KLOC and found 30 race conditions of which several have been determined to be bugs.

2. OVERVIEW

KISS translates a concurrent program P into a sequential program P' that simulates a large subset of the behaviors of P. In a concurrent program, each thread has its own stack. On the other hand, the unique thread in a sequential program has a single stack. Thus, we need to use a single stack to generate behaviors that will simulate interleavings generated by multiple stacks. The sequential program P' is essentially the concurrent program P executing under the control of a nondeterministic scheduler. This scheduler follows a stack discipline for scheduling threads; it supports partial thread resumption. At any point in time, the frames on the unique stack can be partitioned into contiguous blocks. Each contiguous block is the stack of one of the threads executing currently. The scheduler nondeterministically chooses to do one of the following tasks. It may terminate the thread executing at the top of the stack by popping its contiguous block of stack frames and resume the execution of the thread whose block of stack frames is just below that of the terminated thread. Otherwise, it may schedule another thread by calling its starting function. To perform these scheduling decisions, the scheduler requires only a few extra global variables. Keeping the number of extra global variables small is important since the complexity of most sequential-program analyses varies exponentially with the number of global variables.

The algorithm for stack-based nondeterministic scheduling is based on two ideas. The first idea allows us to nondeterministically terminate a thread at any time during its execution. We introduce a fresh global boolean variable *raise* to encode the effect of raising an exception; this variable is initialized to *false*. Preceding every statement in every function, we also introduce new instrumentation code that nondeterministically chooses to either do nothing or set *raise* to *true* and execute a *return* statement. To propagate the return, we add code after each function call to test the value of *raise* and execute another *return* statement if the value of *raise* is *true*.

The second idea allows us to schedule a nondeterministically chosen number of existing threads at any time during the execution of a thread. We introduce a fresh global variable *ts*, whose value is a bounded-size multiset of the starting functions of threads that have been forked but not scheduled. Whenever the concurrent program invokes the function f asynchronously, we add f to *ts* if *ts* is not full. In this case, this function will be executed at some nondeterministically chosen time later on. If the set *ts* is full, then we replace the asynchronous call to f with a synchronous call to f, thus executing it immediately. The instrumentation code preceding every statement, prior to executing the nondeterministic return, performs the following task a nondeterministic number of times: it nondeterministically chooses a function from the set *ts*, invokes it, and sets *raise* to *false* when it returns.

Using *raise*, *ts*, and the instrumentation described above, we can simulate a large number of the executions of the concurrent program. The set *ts* provides a tuning knob to trade off coverage for computational cost of analysis. Increasing the size of *ts* increases the number of simulated behaviors at the cost of increasing the global state space of the translated sequential program. In practice, we expect to start KISS with a small size for *ts* and then increase it as permitted by the computational resources available for debugging.

The generated sequential program only simulates a subset of all executions of the concurrent program. The static nature of our analysis allows us to formalize this unsoundness: given a 2-threaded concurrent program, the sequential program simulates *all* executions with at most two context switches. Thus, the unsoundness of our analysis is qualitatively different from that of dynamic techniques which may explore executions with multiple context switches but cannot provide any guarantees. In Section 4, we present a precise characterization of the executions of a general concurrent program (with many threads) that are simulated by the transformed sequential program.

We illustrate our technique by showing how it can be used to find concurrency errors in Windows device drivers. A device typically processes a number of concurrent requests from the operating system. These requests share a data structure called *device extension*, which is allocated once when the device starts up. Concurrent conflicting accesses to the fields of the device extension is a significant source of errors. A second source of errors is in the synchronization required for reference counting of allocated resources. The operating system may issue a request to stop the device at any time. The thread performing this request must wait until all other threads executing in the driver have finished their work and then free all allocated resources. We illustrate both these types of errors on a simplified model of one of the Bluetooth drivers in Windows NT. To keep the example simple yet illustrative, we have abstracted away a lot of the complexity of the data structures but modeled the synchronization used in the driver accurately.

```
struct DEVICE_EXTENSION {
    int  pendingIo;
    bool stoppingFlag;
    bool stoppingEvent;
};

bool stopped;

void main() {
    DEVICE_EXTENSION *e =
        malloc(sizeof(DEVICE_EXTENSION));
    e->pendingIo = 1;
    e->stoppingFlag = false;
    e->stoppingEvent = false;
    stopped = false;
    async BCSP_PnpStop(e);
    BCSP_PnpAdd(e);
}

void BCSP_PnpAdd(DEVICE_EXTENSION *e) {
    int status;
    status = BCSP_IoIncrement (e);
    if (status == 0) {
        // do work here
        assert !stopped;
    }
    BCSP_IoDecrement(e);
}

void BCSP_PnpStop(DEVICE_EXTENSION *e) {
    e->stoppingFlag = true;
    BCSP_IoDecrement(e);
    assume e->stoppingEvent;
    // release allocated resources
    stopped = true;
}

int BCSP_IoIncrement(DEVICE_EXTENSION *e) {
    if (e->stoppingFlag)
        return -1;
    atomic {
        e->pendingIo = e->pendingIo + 1;
    }
    return 0;
}

void BCSP_IoDecrement(DEVICE_EXTENSION *e) {
    int pendingIo;
    atomic {
        e->pendingIo = e->pendingIo - 1;
        pendingIo = e->pendingIo;
    }
    if (pendingIo == 0)
        e->stoppingEvent = true;
}
```

Figure 2: The simplified model of Bluetooth driver.

2.1 Bluetooth driver

The simplified model of the Bluetooth driver is shown in Figure 2. The device extension of the driver has four fields—`pendingIo`, `stoppingFlag`, `stoppingEvent`. The integer field `pendingIo` keeps count of the number of threads that are currently executing in the driver. It is initialized to 1, incremented atomically whenever a thread enters the driver, and decremented atomically whenever a thread exits the driver. The boolean field `stoppingFlag` is initialized to *false* and set to *true* by a thread that is trying to stop the driver. New threads are not supposed to enter the driver once this field is *true*. The boolean field `stoppingEvent` models an event. This field is initialized to *false*, and set to *true* when the event happens. The event fires when a decrement of `pendingIo` results in a value of 0. Finally, the global variable `stopped` is introduced to conveniently specify an important safety property of the driver. This field is initialized to *false*. The thread stopping the driver sets this field to *true* after it has established that there are no other threads executing in the driver. Other threads assert that this field is *false* just before starting their work in the driver.

There are two dispatch functions in the simplified driver—`BCSP_PnpAdd` and `BCSP_PnpStop`. The first dispatch function `BCSP_PnpAdd` is a prototypical routine called by the operating system to perform I/O in the driver. The second dispatch function `BCSP_PnpStop` is called to stop the driver. The `assume` statement in its body blocks until the condition `e->stoppingEvent` becomes *true*; this statement is explained in more detail in Section 3. We model concurrent execution of the driver by a program that begins by calling the function `main`. This function allocates a device extension, initializes its fields, forks off a thread to asynchronously execute `BCSP_PnpStop` and then calls `BCSP_PnpAdd`.

2.2 Race detection

We now show how our analysis detects a race condition on the field `stoppingFlag` of the device extension. For this example, a size 0 for the multiset *ts* is enough to expose the race. In the translation, bounding *ts* by 0 effectively replaces the asynchronous function calls in the function `main` by the corresponding synchronous function calls. In addition, the instrumentation before every statement in every function allows a function to return from any control location after setting *raise* to *true*.

When we run the model checker on the transformed sequential program, it explores the following erroneous path. The path starts execution at the beginning of `main` and makes a synchronous function call to `BCSP_PnpStop`. In the function `BCSP_PnpStop`, just after the write to `stoppingFlag`, *raise* is set to true and a return statement is executed. When control returns to `main`, *raise* is reset to *false* and `BCSP_PnpAdd` is called. The function `BCSP_PnpAdd` calls the function `BCSP_IoIncrement`, where there is a read of the field `stoppingFlag`. Thus, this execution exposes a race condition on the field `stoppingFlag`.

2.3 Assertion checking

We now show how our analysis detects the violation of the assertion in the dispatch function `BCSP_PnpAdd`. The error trace leading to the assertion violation cannot be simulated by the transformed sequential program if the size of *ts* is 0. However, the error trace can be simulated if the size of *ts* is increased to 1.

The erroneous path explored by the model checker in the transformed sequential program is as follows. After the `main` function finishes initializing the allocated device extension,

function names	f	$::=$	$f_0 \mid f_1 \mid \ldots$
integers	i	$::=$	$\ldots \mid -1 \mid 0 \mid 1 \mid \ldots$
boolean constants	b	$::=$	$\mathit{true} \mid \mathit{false}$
constants	c	$::=$	$i \mid b \mid f$
primitives	op	$::=$	$+ \mid - \mid \times \mid ==$
variables	v	$::=$	$v_0 \mid v_1 \mid \ldots$
values	u	$::=$	$v \mid c$
statements	s	$::=$	$v_0 = c$
		$\mid$	$v_0 = \& v_1$
		$\mid$	$v_0 = * v_1$
		$\mid$	$* v_0 = v_1$
		$\mid$	$v_0 = v_1 \ op \ v_2$
		$\mid$	$\mathit{assert}(v_0)$
		$\mid$	$\mathit{assume}(v_0)$
		$\mid$	$\mathit{atomic}\{s\}$
		$\mid$	$v = v_0()$
		$\mid$	$\mathit{async}\ v_0()$
		$\mid$	return
		$\mid$	$s_1; s_2$
		$\mid$	$\mathit{choice}\{s_1 \ [] \ \ldots \ [] \ s_n\}$
		$\mid$	$\mathit{iter}\{s\}$

Figure 3: A parallel language.

it adds `BCSP_PnpStop` to *ts* and then calls `BCSP_PnpAdd`. The function `BCSP_PnpAdd` calls `BCSP_IoIncrement` which reads the field `stoppingFlag`. Since the value of `stoppingFlag` is *false*, control moves to the beginning of the atomic increment. At this point, `BCSP_PnpStop` is removed from *ts* and its execution begins. It sets `stoppingFlag` to *true* and calls `BCSP_IoDecrement`. The function `BCSP_IoDecrement` decrements `pendingIo` to 0 and sets `stoppingEvent` to *true*. Consequently, when `BCSP_IoDecrement` returns, the assume statement in `BCSP_PnpStop` does not block and `stopped` is set to *true*. Now the execution of `BCSP_PnpStop` terminates. At this point, the execution stack has two entries—`BCSP_IoIncrement` at the top and `BCSP_PnpAdd` at the bottom. The function `BCSP_IoIncrement` resumes execution, increments `pendingIo` to 1, and returns 0. Since the returned value is 0, the conditional of the if statement in `BCSP_PnpAdd` is true and control moves to the assertion, which is violated since the value of `stopped` is *true*.

The examples discussed in this section suggest that the executions explored by the transformed sequential program, although a subset of the set of all behaviors of the concurrent program, are still varied enough to catch a variety of common concurrency errors.

3. A PARALLEL LANGUAGE

KISS is capable of handling general multithreaded C programs. To succinctly present the main idea behind KISS, we formalize its analysis for a simple parallel language that is capable of modeling conventional concurrent programs. The syntax for this language is shown in Figure 3. We essentially have a procedural language extended with asynchronous procedure calls (*async*), atomic statements (*atomic*), and blocking statements (*assume*). The language is equipped with pointer operations for taking the address of a variable and for obtaining the contents of an address. Fields have been omitted for simplicity of exposition; however, KISS can handle them just as well.

The asynchronous function call $\mathit{async}\ v_0()$ creates a new thread whose starting function is the value of the variable v_0. After this thread is created, its actions get interleaved with the actions of the existing threads.

The statement $\mathit{assume}(v_0)$ blocks until the variable v_0 becomes *true*. In a sequential program, if control arrives at $\mathit{assume}(v_0)$ when v_0 is false, then the program is blocked forever. However, in a concurrent program a thread blocked at the assume statement might get unblocked if another thread changes the value of v_0.

The statement $\mathit{atomic}\{s\}$ executes just like s except that the execution may not be interrupted in the middle by other threads. The *atomic* statement, together with the *assume* statement, is used to implement synchronization primitives, such as *lock_acquire* and *lock_release*.

$$\begin{aligned}\mathit{lock\_acquire}(l) &\overset{\text{def}}{=} \mathit{atomic}\{\mathit{assume}(*l == 0); *l = 1;\}\\ \mathit{lock\_release}(l) &\overset{\text{def}}{=} \mathit{atomic}\{*l = 0;\}\end{aligned}$$

Here, the type of the variable l is a pointer to a integer.

The nondeterministic *choice* statement makes choices between different branches; exactly one branch will be executed nondeterministically. The iteration statement $\mathit{iter}\{s\}$ does not have a termination condition. It executes s a nondeterministic number of times. The conventional conditional and loop statements can be modeled as follows:

$$\begin{aligned}\mathit{if}\ (v)\ s_1\ \mathit{else}\ s_2 &\overset{\text{def}}{=} \mathit{choice}\{\mathit{assume}(v); s_1 \ [] \ \mathit{assume}(\neg v); s_2\}\\ \mathit{while}\ (v)\ s &\overset{\text{def}}{=} \mathit{iter}\{\mathit{assume}(v); s\}; \mathit{assume}(\neg v)\end{aligned}$$

Here, the symbol $\neg$ is logical negation as usual. There are no explicit boolean expressions in the language; decisions are made on variables. Decisions on an expression can be modeled by first assigning the expression to a fresh variable.

We assume that other analyses are used to check that a program written in this parallel language is well-typed. In addition to the usual requirements, we also require that the statement s in $\mathit{atomic}\{s\}$ is free of function calls (both synchronous and asynchronous), return statements, and nested atomic statements. This requirement does not pose any restriction on the expressiveness of the language since the most common use of the *atomic* statement is to implement synchronization primitives.

4. PROGRAM TRANSFORMATION

In this section, we present our method for translating a concurrent program into a sequential program. For the purpose of this section, a concurrent program is one expressible in the parallel language of Section 3 and a sequential program is one expressible in the parallel language without using asynchronous function calls and atomic statements.

The translation function $[\![\cdot]\!]$ for translating a concurrent program to a sequential program is defined recursively in Figure 4. For any statement s, the transformed code $[\![s]\!]$ is a statement without any asynchronous function calls and synchronization statements. For any function f in the program with body s, we introduce a new function $[\![f]\!]$ with body $[\![s]\!]$.

To achieve nondeterministic termination of a thread at any time during its execution, we introduce a fresh global boolean variable *raise*. To terminate a thread, this variable is set to *true* and a return statement is executed. The

$$
\begin{array}{lcl}
[\![v_0 = c]\!] & = & \mathit{schedule}();\mathit{choice}\{\mathit{skip} \; [\!] \; \mathit{RAISE}\}; v_0 = c \\
[\![v_0 = \&v_1]\!] & = & \mathit{schedule}();\mathit{choice}\{\mathit{skip} \; [\!] \; \mathit{RAISE}\}; v_0 = \&v_1 \\
[\![v_0 = *v_1]\!] & = & \mathit{schedule}();\mathit{choice}\{\mathit{skip} \; [\!] \; \mathit{RAISE}\}; v_0 = *v_1 \\
[\![*v_0 = v_1]\!] & = & \mathit{schedule}();\mathit{choice}\{\mathit{skip} \; [\!] \; \mathit{RAISE}\}; *v_0 = v_1 \\
[\![v_0 = v_1 \; op \; v_2]\!] & = & \mathit{schedule}();\mathit{choice}\{\mathit{skip} \; [\!] \; \mathit{RAISE}\}; v_0 = v_1 \; op \; v_2 \\
[\![\mathit{assert}(v_0)]\!] & = & \mathit{schedule}();\mathit{choice}\{\mathit{skip} \; [\!] \; \mathit{RAISE}\}; \mathit{assert}(v_0) \\
[\![\mathit{assume}(v_0)]\!] & = & \mathit{schedule}();\mathit{choice}\{\mathit{skip} \; [\!] \; \mathit{RAISE}\}; \mathit{assume}(v_0) \\
[\![\mathit{atomic}\{s\}]\!] & = & \mathit{schedule}();\mathit{choice}\{\mathit{skip} \; [\!] \; \mathit{RAISE}\}; s \\
[\![v = v_0()]\!] & = & \mathit{schedule}();\mathit{choice}\{\mathit{skip} \; [\!] \; \mathit{RAISE}\}; v = [\![v_0]\!](); \mathit{if} \; (\mathit{raise}) \; \mathit{return} \\
[\![\mathit{async} \; v_0()]\!] & = & \mathit{schedule}();\mathit{choice}\{\mathit{skip} \; [\!] \; \mathit{RAISE}\}; \\
 & & \mathit{if} \; (\mathit{size}() < \mathit{MAX}) \; \mathit{put}(v_0) \\
 & & \mathit{else} \; \{[\![v_0]\!](); \mathit{raise} = \mathit{false}\} \\
[\![\mathit{return}]\!] & = & \mathit{schedule}(); \mathit{return} \\
[\![s_1; s_2]\!] & = & [\![s_1]\!]; [\![s_2]\!] \\
[\![\mathit{choice}\{s_1 [\!] \ldots [\!] s_n\}]\!] & = & \mathit{choice}\{[\![s_1]\!] \; [\!] \; \ldots \; [\!] \; [\![s_n]\!]\} \\
[\![\mathit{iter}\{s\}]\!] & = & \mathit{iter}\{[\![s]\!]\}
\end{array}
$$

Figure 4: Instrumentation for assertion checking.

RAISE statement performs this task.

$$\mathit{RAISE} \overset{\text{def}}{=} \mathit{raise} = \mathit{true}; \mathit{return}$$

Let *skip* represent the statement *assume*(*true*). At every control location, we introduce the following code to execute the *RAISE* statement nondeterministically.

$$\mathit{choice}\{\mathit{skip} \; [\!] \; \mathit{RAISE}\}$$

A thread may return after setting *raise* to *true* with a number of functions on its stack. To terminate the thread, all of these functions need to be popped. Therefore, we introduce after each function call, a statement that checks the value of *raise* and returns if the value is *true*. As described later, the variable *raise* is reset to *false* when all stack frames of the terminated thread have been popped.

In addition to terminating a thread nondeterministically, we also need to schedule other threads at any point during a thread's execution. We introduce a fresh global variable *ts* to keep track of the set of threads that have been forked but not scheduled yet. The variable *ts* is a multiset of function names; each name indicates the start function of an unscheduled thread. Our translation is parameterized by MAX, the maximum size of this set. There are three special functions to access and modify the variable *ts*. The function *get* requires that *ts* is not empty; it removes and returns a nondeterministically chosen element of *ts*. The function *put* requires that the number of elements in *ts* is less than MAX; it takes as argument a function name and adds it to *ts*. The function *size* returns the number of elements in *ts*.

The function *schedule* allows us to schedule a nondeterministically chosen set of threads from *ts*.

```
schedule() {
    var f;
    iter {
        if (size() > 0) {
            f = get();
            [[f]]();
            raise = false;
        }
    }
}
```

This function gets an arbitrary number of existing threads in the set *ts* and executes them one after another. After a thread returns, potentially because it set *raise* to *true*, the variable *raise* is reset to *false*.

The function *schedule* encapsulates the scheduling policy for the concurrent program. The implementation of this function presented above assumes a completely nondeterministic scheduler. A more sophisticated scheduler can be provided by writing a different implementation of *schedule*.

Before most statements, the translation function inserts a call to *schedule* followed by the nondeterministic execution of *RAISE*. The instrumentation for asynchronous and synchronous function calls is noteworthy. When a function f is called asynchronously, the instrumentation checks the size of *ts*. If *ts* is full, the function $[\![f]\!]$ is called synchronously. Otherwise, the function f is added to *ts*. Thus, the maximum size of *ts* determines the number of possible executions of the original program that can be simulated by the translated program. When a function f is called synchronously, the instrumentation calls $[\![f]\!]$ instead. When $[\![f]\!]$ returns, the value of *raise* is checked and if it is *true*, a return statement is executed to propagate the "exception."

Given a concurrent program s, the sequential program to be analyzed is defined as follows:

$$\mathit{Check}(s) \overset{\text{def}}{=} \mathit{raise} = \mathit{false}; \; \mathit{ts} = \emptyset; \; [\![s]\!]; \; \mathit{schedule}();$$

The program *Check*(s) initializes *raise* and *ts* appropriately, executes $[\![s]\!]$, and finally schedules the remaining unscheduled threads.

For a sequential program with boolean variables, the complexity of model checking (or interprocedural dataflow analysis) is $O(|C| \cdot 2^{g+l})$, where $|C|$ is the size of the control-flow graph, g is the number of global variables, and l is the maximum number of local variables in scope at any location. Our instrumentation introduces a small constant blowup in the control-flow graph of the concurrent program and adds a small constant number of global variables. Thus, the complexity of using KISS on a concurrent program of a certain size is about the same as using a standard interprocedural dataflow analysis or model checking on a sequential program of the same size.

$$
\begin{array}{lcl}
\llbracket v = c \rrbracket & = & \mathit{schedule}();\ \mathit{choice}\{\mathit{skip}\ []\ \mathit{check}_w(\&v); \mathit{RAISE}\};\ v = c \\
\llbracket v = \&v_1 \rrbracket & = & \mathit{schedule}();\ \mathit{choice}\{\mathit{skip}\ []\ \mathit{check}_w(\&v); \mathit{RAISE}\};\ v = \&v_1 \\
\llbracket v = *v_1 \rrbracket & = & \mathit{schedule}(); \\
 & & \mathit{choice}\{\mathit{skip}\ []\ \mathit{check}_r(\&v_1); \mathit{RAISE}\ []\ \mathit{check}_r(v_1); \mathit{RAISE}\ []\ \mathit{check}_w(\&v); \mathit{RAISE}\}; \\
 & & v = *v_1 \\
\llbracket *v = v_1 \rrbracket & = & \mathit{schedule}(); \\
 & & \mathit{choice}\{\mathit{skip}\ []\ \mathit{check}_r(\&v_1); \mathit{RAISE}\ []\ \mathit{check}_r(\&v); \mathit{RAISE}\ []\ \mathit{check}_w(v); \mathit{RAISE}\}; \\
 & & *v = v_1 \\
\llbracket v = v_1\ op\ v_2 \rrbracket & = & \mathit{schedule}(); \\
 & & \mathit{choice}\{\mathit{skip}\ []\ \mathit{check}_r(\&v_1); \mathit{RAISE}\ []\ \mathit{check}_r(\&v_2); \mathit{RAISE}\ []\ \mathit{check}_w(\&v); \mathit{RAISE}\}; \\
 & & v = v_1\ op\ v_2 \\
\llbracket \mathit{assert}(v) \rrbracket & = & \mathit{schedule}(); \mathit{choice}\{\mathit{skip}\ []\ \mathit{check}_r(\&v); \mathit{RAISE}\};\ \mathit{assert}(v) \\
\llbracket \mathit{assume}(v) \rrbracket & = & \mathit{schedule}(); \mathit{choice}\{\mathit{skip}\ []\ \mathit{check}_r(\&v); \mathit{RAISE}\};\ \mathit{assume}(v) \\
\llbracket \mathit{atomic}\{s\} \rrbracket & = & \mathit{schedule}(); \mathit{choice}\{\mathit{skip}\ []\ \mathit{RAISE}\};\ s \\
\llbracket v = v_0() \rrbracket & = & \mathit{schedule}(); \\
 & & \mathit{choice}\{\mathit{skip}\ []\ \mathit{check}_r(\&v_0); \mathit{RAISE}\ []\ \mathit{check}_w(\&v); \mathit{RAISE}\}; \\
 & & v = \llbracket v_0 \rrbracket(); \\
 & & \mathit{if}\ (\mathit{raise})\ \mathit{return} \\
\llbracket \mathit{async}\ v_0() \rrbracket & = & \mathit{schedule}(); \\
 & & \mathit{choice}\{\mathit{skip}\ []\ \mathit{check}_r(\&v_0); \mathit{RAISE}\}; \\
 & & \mathit{if}\ (\mathit{size}() < \mathit{MAX})\ \mathit{put}(v_0) \\
 & & \mathit{else}\ \{\llbracket v_0 \rrbracket(); \mathit{raise} = \mathit{false}\} \\
\llbracket \mathit{return} \rrbracket & = & \mathit{schedule}(); \mathit{return} \\
\llbracket s_1; s_2 \rrbracket & = & \llbracket s_1 \rrbracket; \llbracket s_2 \rrbracket \\
\llbracket \mathit{choice}\{s_1 [] \ldots [] s_n\} \rrbracket & = & \mathit{choice}\{\llbracket s_1 \rrbracket\ []\ \ldots\ []\ \llbracket s_n \rrbracket\} \\
\llbracket \mathit{iter}\{s\} \rrbracket & = & \mathit{iter}\{\llbracket s \rrbracket\}
\end{array}
$$

Figure 5: Instrumentation for race checking.

4.1 Coverage analysis

Every path in the sequential program *Check*(s) simulates a potential execution of the original program s. However, the sequential program *Check*(s) does not simulate all executions of s. Therefore, if an assertion is violated in the translated sequential program, it is violated in some execution of the multithreaded program as well, but not vice versa. In this section, we characterize more formally the set of executions of s that are covered by *Check*(s).

We first introduce some notation to enable the formal characterization. Suppose that each thread created by a concurrent program has a unique identifier in the set $N = \{1, 2, \ldots\}$. For any finite set $X \subseteq N$, we define a language $L_X \subseteq N^*$ recursively as follows:

$$
\begin{array}{rcl}
L_X & = & \{i^* \cdot L_{X_1} \cdot \ldots \cdot i^* \cdot L_{X_k} \cdot i^* \ \mid \\
 & & \quad \{i\},\ X_1,\ \ldots\ ,\ X_k\ \mathit{form\ a\ partition\ of}\ X\}
\end{array}
$$

where "$\cdot$" denotes concatenation over strings and string languages as usual, and "$*$" stands for Kleene closure. A string in N^* is *balanced* if it belongs to L_X for some finite $X \subseteq N$.

An execution of a concurrent program is a finite sequence of states and transitions, where each transition is labeled with the identifier of the thread that performed the transition. From each execution, we generate a string in N^* by concatenating the identifiers labeling the transitions in the execution. An execution is *balanced* if the corresponding string is balanced. We can now state the following theorem characterizing the executions simulated by *Check*(s).

THEOREM 1 (COMPLETENESS). *Suppose the multiset ts is unbounded. If a* balanced *execution of a concurrent program s goes wrong by failing an assertion, then the sequential program Check(s) also goes wrong, and vice versa.*

5. RACE DETECTION

In this section, we augment the translation of Section 4 with auxiliary information to enable the detection of race conditions. We fix a distinguished variable r and describe instrumentation that checks for race conditions on r.

In addition to the variables added by the instrumentation described in Section 4, the new instrumentation also requires a fresh global variable *access* that takes values from the set $\{0, 1, 2\}$. The value of *access* is 0 if no access to r has occurred, 1 if a read access has occurred, and 2 if a write access has occurred. We also need two functions, check_r and check_w, which are defined below.

```
$check_r(x)$ {
    $if\ (x == \&r)$
        $\{assert(\neg(access == 2)); access = 1;\}$
}
```

A call $\mathit{check}_r(x)$ checks that there is no race on r due to a read of the address contained in x. If the address x is an alias of $\&r$, the address of the variable being checked for race conditions, the function check_r asserts that there has not been a write access to r. Since we check for read/write and write/write races, two simultaneous read accesses are allowed. If the assertion passes, the variable *access* is set to 1 to indicate that a read access to r has happened.

```
$check_w(x)$ {
    $if\ (x == \&r)$
        $\{assert(access == 0); access = 2;\}$
}
```

Similarly, a call $\mathit{check}_w(x)$ checks that there is no race on r due to a write to the address contained in x. If the address

Driver	KLOC	Fields	Races	No Races
tracedrv	0.5	3	0	3
moufiltr	1.0	14	7	7
kbfiltr	1.1	15	8	7
imca	1.1	5	1	4
startio	1.1	9	0	9
toaster/toastmon	1.4	8	1	7
diskperf	2.4	16	2	14
1394diag	2.7	18	1	17
1394vdev	2.8	18	1	17
fakemodem	2.9	39	6	31
gameenum	3.9	45	11	24
toaster/bus	5.0	30	0	22
serenum	5.9	41	5	21
toaster/func	6.6	24	7	17
mouclass	7.0	34	1	32
kbdclass	7.4	36	1	33
mouser	7.6	34	1	27
fdc	9.2	92	18	54
Total	69.6	481	71	346

Table 1: Experimental results (I).

Driver	Races
moufiltr	0
kbfiltr	0
imca	1
toaster/toastmon	1
diskperf	0
1394diag	1
1394vdev	1
fakemodem	6
gameenum	1
serenum	2
toaster/func	5
mouclass	1
kbdclass	1
mouser	1
fdc	9
Total	30

Table 2: Experimental results (II).

x is an alias of $\&r$, it asserts that there has not been either a read or a write access to r. Moreover, if the assertion passes, it sets *access* to 2 to indicate that a write access to r has happened.

The translation function is shown in Figure 5. The translation looks very similar to that for assertion checking, except that we introduce a call $\mathit{check}_r(\&v)$ for each read access to variable v, and a call $\mathit{check}_w(\&v)$ for each write access to v. Every such call by a thread is followed immediately by *RAISE* which causes the thread to terminate. Thus, an assertion in one of these calls is violated only if there are conflicting (read/write or write/write) accesses by two *different* threads.

We use a static alias analysis [12] to optimize away most of the calls to check_r and check_w. If the alias analysis determines that the variable v being accessed cannot be aliased to the distinguished variable r, then the call to check_r (or check_w) has no effect and is therefore omitted in the instrumentation. Thus, our analysis generates a separate sequential program for detecting race conditions on each shared variable.

Given a concurrent program s, the sequential program to be analyzed is as follows:

$$\mathit{Check}(s) \stackrel{\text{def}}{=} \mathit{raise} = \mathit{false}; \mathit{ts} = \emptyset; \mathit{access} = 0; [\![s]\!]; \mathit{schedule}();$$

As for the translation in Section 4, every path in $\mathit{Check}(s)$ simulates a potential execution of the original program s, but $\mathit{Check}(s)$ does not capture all possible executions of s. If an assertion is violated in $\mathit{Check}(s)$, there is an execution of s in which either an assertion is violated or there is a race condition on r.

6. EVALUATION

We have used KISS to detect race conditions in a number of device drivers in the Windows Driver Development Kit. In this section, we present the results of our experiments. As mentioned earlier in Section 2, each device driver has a data structure called the device extension that is shared among the various threads executing in it. For each device driver, we checked for race conditions on each field of the device extension separately. A device driver is written as a library of dispatch routines that may be called by the operating system. For each device driver, we created a concurrent program with two threads, each of which nondeterministically calls a dispatch routine. To have a complete concurrent program, we also need models for the routines of the operating system called by the driver. SLAM already provided stubs for these calls; we augmented them to model the synchronization operations accurately. Some of the synchronization routines we modeled were `KeAcquireSpinLock`, `KeWaitForSingleObject`, `InterlockedCompareExchange`, `InterlockedIncrement`, etc. Guided by the intuition of the Bluetooth driver example in Section 2.2, we set the size of *ts* to 0.

We performed the experiments on a 2.2 GHz PC running Windows XP. For each run of KISS on a device driver and a field of the device extension, we set a resource bound of 20 minutes of CPU time and 800MB of memory. Table 1 gives a summary of our results. For each driver, we give the code size in KLOC (thousand lines of code), the number of fields in the device extension, the number of fields on which a race condition was detected, and the number of fields on which the analysis terminated within the resource bound without reporting any errors. We ran KISS on 18 device drivers ranging from 0.5 KLOC to 9.2 KLOC for a total of 69.6 KLOC. The tool reported at least one race condition in 15 drivers for a total of 71 race conditions.

Due to the large number of reported race conditions, it was infeasible to carefully review them all. Therefore, we showed a small subset of these race conditions to the Windows driver quality team. We found that KISS was reporting spurious race conditions primarily because of the imprecision of the concurrent harness executing dispatch functions in the device driver. Drivers are written under the (typically undocumented) assumption that certain pairs of dispatch routines cannot be called concurrently by operating system, but out harness allowed all such pairs to be executed concurrently. The most important such assumptions mentioned by the driver quality team are the following:

A1. Two Pnp IRPs (interrupt request packets) will not be sent by the operating system concurrently.

A2. The operating system will not send any IRP concurrently with a Pnp IRP for starting or removing a device.

A3. There are two categories of Power IRPs—system and device. Two Power IRPs sent concurrently by the operating system must belong to different categories.

The above rules are general and applicable to all drivers. But some rules are specific to particular drivers, as in the case of kbfiltr and moufiltr. The error traces for all race conditions reported by KISS on these two drivers involved two concurrent Ioctl IRPs. However, the position of these two drivers in the driver stack ensures that they will never receive two concurrent Ioctl IRPs; consequently, the race conditions reported by KISS were spurious.

We used the feedback from the driver quality team to refine the harness and ran KISS again on the fields on which race conditions were reported in the first set of experiments. We present the results of this second set of experiments in Table 2. The total number of reported race conditions went down from 71 to 30.

After examining a subset of the remaining race conditions, the driver quality team confirmed that the race conditions in toaster/toastmon, mouclass, and kbdclass are bugs. In addition, the race conditions on three fields of fdc generated a lot of debate and were considered serious enough to be tabled for further discussion. The feedback from the driver quality team suggested that the warnings produced by KISS were useful for more than just finding concurrency bugs. These warnings also served to focus costly manual code inspection resources on tricky areas of the driver code.

We illustrate the errors found by KISS with the race condition on the field `DevicePnPState` of the device extension in toaster/toastmon. This field is accessed in most places while holding a lock but there is an unprotected read to it as well. The read/write race condition is exposed by the concurrent execution of the two dispatch functions shown in Figure 6.

Another interesting source of spurious warnings are benign race conditions. Consider the race condition found by KISS on the field `OpenCount` of the device extension of the fakemodem driver. This field keeps track of the number of threads executing in the driver. In all places but one, `OpenCount` is incremented while holding a lock. But there is a single unprotected access in which a decision is based on whether the value read for `OpenCount` is 0. The read operation is atomic already; performing it while holding the protecting lock will not reduce the set of values that may be read. So the programmer chose to not pay for the overhead of locking.

We are continuing our dialogue with the driver quality team to establish which races are benign. In future work, we intend to deal with the problem of benign races by allowing the programmer to annotate an access as benign. KISS can then use this annotation as a directive to not instrument that access. We are also planning to use the ideas behind the type system for atomicity [20] to automatically prune such benign race conditions.

We have also used KISS to find concurrent reference counting errors in device drivers, as exemplified by the assertion violation in our simple model of the bluetooth driver (Section 2.3). Just as in the simple model, we manually introduced into the code of each driver an auxiliary boolean global variable `stopped` to model the stopping of the driver. We introduced assignments initializing `stopped` to *false* and updating it to *true* when the driver is stopped, and assertions at appropriate places in the code of the various dispatch routines stating that this variable is *false*. Guided by the intuition of Section 2.3, we set the size of *ts* to 1.

We ran KISS on two drivers—bluetooth and fakemodem. Not surprisingly, since our example in Section 2.3 is based on the bluetooth driver, KISS found the assertion violation in the actual driver as well. The bug is in the implementation of the `BCSP_IoIncrement` function. After fixing the bug as

```
NTSTATUS
ToastMon_DispatchPnp (
    IN PDEVICE_OBJECT DeviceObject,
    IN PIRP           Irp  )
{
      ⋮
    status = IoAcquireRemoveLock
        (&deviceExtension->RemoveLock, Irp);
      ⋮
    switch (irpStack->MinorFunction) {
        ⋮
      case IRP_MN_QUERY_STOP_DEVICE:
        // Race: write access
        deviceExtension->DevicePnPState =
                        StopPending;
        status = STATUS_SUCCESS;
        break;
        ⋮
    }
    ⋮
    IoReleaseRemoveLock
        (&deviceExtension->RemoveLock, Irp);
    return status;
}

NTSTATUS
ToastMon_DispatchPower(
    IN PDEVICE_OBJECT    DeviceObject,
    IN PIRP              Irp  )
{
   ⋮
  // Race: read access
  if (Deleted == deviceExtension->DevicePnPState) {
    PoStartNextPowerIrp(Irp);
    Irp->IoStatus.Status =  STATUS_DELETE_PENDING;
    IoCompleteRequest(Irp, IO_NO_INCREMENT );
    return STATUS_DELETE_PENDING;
  }
  ⋮
}
```

Figure 6: Race condition in toaster/toastmon.

suggested by the driver quality team, we ran KISS again and this time KISS did not report any errors. KISS did not report any errors in the fakemodem driver. We examined the code dealing with reference counting in the fakemodem driver and observed that it behaved exactly according to the fixed implementation of `BCSP_IoIncrement`. Hence, we believe that the fakemodem driver does not have this error.

6.1 Discussion

Our experience with KISS shows that it is a useful and viable approach for finding errors in concurrent programs. Although unsound, the KISS approach has managed to uncover a number of subtle concurrency errors in device drivers. We believe that the flexibility of our approach sets it apart from other existing work on race-detection.

Flexibility in implementation: Our checker can conveniently support a variety of synchronization mechanisms and is easily extensible to new ones. For analyzing systems code, this flexibility is essential as illustrate by our experience with NT drivers. To analyze these drivers, we modeled several synchronization mechanisms such locks, events, interlocked compare and exchange, etc. Most existing race-detection tools, both static and dynamic, are based on the lockset algorithm which can handle only the simplest synchronization mechanism of locks.

Flexibility in environment modeling: Our experience with checking low-level systems code indicates that to avoid being inundated with false alarms, care must be taken in modeling the environment of the module being analyzed. Our tool provides a flexible mechanism for writing an experimenting with such environments.

Flexibility in specification: Our tool is a general assertion checker for concurrent programs and can check safety properties other than just race-freedom. For example, the reference counting error in the bluetooth driver manifested itself through an assertion violation. This error is not a race condition according to the traditional definition of race-freedom.

7. RELATED WORK

There has been a substantial amount of research on the problem of debugging and verifying concurrent programs. Here, we discuss the more relevant research along several axes.

Model checking: Model checkers systematically explore the state space of a model of the concurrent program. The model is constructed either manually or extracted automatically by a tool. The model checker SPIN [27] checks models written in the Promela [26] modeling language. Other model checkers such as the JPF [23, 39], Bandera [11], and Bogor [35] directly analyze multithreaded Java programs. These model checkers exploit partial-order reduction techniques [31, 21] to reduce the number of explored interleavings. But in the worst case, they must still explore an exponential number of control states. The model checkers SLAM [3] and Blast [25] analyze sequential C programs. Recently, Blast has been extended to check properties of multi-threaded C programs using the approach of thread-modular model checking [19, 24]. This approach is sound but may report false errors. Our technique is complementary because although it is unsound, it will never report false errors.

Static analysis: Static analysis tools for concurrent programs are typically based on type systems and dataflow analyses. In general, they do not directly analyze thread interleavings. Consequently, they are less precise but more scalable than model checkers. Warlock [38] is a static race detection tool for ANSI C programs. Aiken and Gay [1] investigate static race detection in the context of SPMD programs. The Race Condition Checker (RccJava) [16, 17] uses a type system to catch race conditions in Java programs. This approach has been extended [6, 5] and adapted to other languages [22]. Engler and Ashcraft [14] have developed a static tool RacerX based on interprocedural dataflow analysis to detect both race conditions and deadlocks. The Extended Static Checker for Java (ESC/Java) [18], which uses a verification-condition generator and an automatic theorem prover to find errors, catches a variety of software defects in addition to race conditions. ESC/Java has been extended to catch "higher-level" race conditions, where a stale value from one synchronized block is used in a subsequent synchronized block [7]. Flanagan and Qadeer [20] have developed a type and effect system, which is a synthesis of Lipton's theory of reduction [29] and type systems for race detection, for checking atomicity of methods in multithreaded Java programs. The advantage of our approach over these static analyses is that it is more precise and it can check more general specifications such as program assertions.

Dynamic analysis: Dynamic tools work by instrumenting and executing the program. They are easy to use but their coverage is typically small since only a few executions are explored. Several methods [2, 30, 32] have been developed to detect race conditions by computing Lamport's *happens-before* relation [28] dynamically. Eraser [36] is a dynamic race-detection tool aimed at the lock-based synchronization. Eraser finds races by keeping track of locks held during program execution. This algorithm has been extended to object-oriented languages [40] and improved for precision and performance [10]. A race detection tool has also been developed for Cilk programs [9]. Although both the dynamic approach and our static approach are unsound, the coverage provided by them seem to be complementary in nature. Our approach can schedule threads only according to the stack discipline but for each such schedule all possible paths in each thread are explored. A dynamic approach may allow schedules not allowed by our approach but for each schedule only a small number of paths in each thread are explored.

Others: Bouajjani *et al.* [4] present a generic approach to the static analysis of concurrent programs, focusing on the synchronous message-passing mechanism. Their verification method is not automated whereas the approach described in this paper is fully automated.

8. CONCLUSION

We have introduced a novel technique for checking assertions in multithreaded programs. The technique *never* reports false errors but may miss errors. The key idea of our analysis is the transformation of a concurrent program into a sequential program which simulates a large subset of the behaviors of the concurrent program. The transformed sequential program may then be checked by any sequential analysis tool.

We have implemented our debugging technique in KISS (Keep It Simple and Sequential!), an automated checker for multithreaded C programs built on top of the SLAM [3] model checker for sequential C programs. It is straightforward to adapt our technique to other similar tools such as PREfix [8], MC [15], ESP [13], and Blast [25]. Thus, our technique is a general framework for checking safety properties of concurrent programs, that can leverage a variety of analysis techniques developed for sequential programs. We have applied KISS to the problem of detecting race conditions in Windows NT device drivers and obtained promising initial results. KISS has analyzed 18 drivers for a total of 70 KLOC and found 30 race conditions of which several have been determined to be bugs.

Acknowledgments

We gratefully acknowledge the help of our colleagues, Tom Ball, Byron Cook, Jakob Lichtenberg, and Sriram Rajamani, for answering numerous questions about SLAM and for helping with the implementation of KISS. We also thank Trishul Chilimbi for suggesting the name KISS, and Andrew Appel, Jim Larus, Xinming Ou, Sriram Rajamani, and anonymous reviewers for their feedback on earlier drafts of this paper.

9. REFERENCES

[1] A. Aiken and D. Gay. Barrier inference. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'98)*, pages 243–354. ACM Press, 1998.

[2] Anne Dinning and Edith Schonberg. An empirical comparison of monitoring algorithms for access anomaly detection. In *Proceedings of the Second ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 1–10, 1990.

[3] T. Ball and S. Rajamani. The SLAM project: debugging system software via static analysis. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'02)*, pages 1–3. ACM Press, 2002.

[4] A. Bouajjani, J. Esparza, and T. Touili. A generic approach to the static analysis of concurrent programs with procedures. In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'03)*, pages 62–73. ACM Press, 2003.

[5] C. Boyapati, R. Lee, and M. Rinard. Ownership types for safe programming: preventing data races and deadlocks. In *Proceedings of the 17th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'02)*, pages 211–230. ACM Press, 2002.

[6] C. Boyapati and M. Rinard. A parameterized type system for race-free Java programs. In *Proceedings of the 16th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'01)*, pages 56–69, Tampa Bay, FL, 2001.

[7] M. Burrows and K. R. M. Leino. Finding stale-value errors in concurrent programs. Technical Note 2002-4, Compaq Systems Research Center, May 2002.

[8] W. R. Bush, J. D. Pincus, and D. J. Sielaff. A static analyzer for finding dynamic programming errors. *Software-Practice and Experience*, 30(7):775–802, June 2000.

[9] G.-I. Cheng, M. Feng, C. E. Leiserson, K. H. Randall, and A. F. 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. ACM Press, 1998.

[10] J.-D. Choi, K. Lee, A. Loginov, R. O'Callahan, V. Sarkar, and M. Sridharan. Efficient and precise datarace detection for multithreaded object-oriented programs. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI'02)*, pages 258–269, June 2002.

[11] J. Corbett, M. Dwyer, J. Hatcliff, S. Laubach, C. Păsăreanu, Robby, and H. Zheng. Bandera: extracting finite-state models from Java source code. In *International Conference on Software Engineering*, pages 439–448, 2000.

[12] M. Das. Unification-based pointer analysis with directional assignments. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation (PLDI'00)*, pages 35–46. ACM Press, 2000.

[13] M. Das, S. Lerner, and M. Seigle. ESP: Path-sensitive program verification in polynomial time. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI'02)*, pages 57–69, 2002.

[14] D. Engler and K. Ashcraft. RacerX: effective, static detection of race conditions and deadlocks. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, pages 237–252. ACM Press, 2003.

[15] D. Engler, B. Chelf, A. Chou, and S. Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *Proceedings of the 4th Symposium on Operating Systems Design and Implementation (OSDI'00)*, pages 1–16. Usenix Association, 2000.

[16] C. Flanagan and M. Abadi. Types for safe locking. In *Proceedings of European Symposium on Programming*, pages 91–108, March 1999.

[17] C. Flanagan and S. Freund. Type-based race detection for Java. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation (PLDI'00)*, pages 219–232, 2000.

[18] 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 SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI'02)*, pages 234–245. ACM Press, 2002.

[19] C. Flanagan and S. Qadeer. Thread-modular model checking. In *Proceedings of the SPIN Workshop on Software Verification*, pages 213–224, 2003.

[20] C. Flanagan and S. Qadeer. A type and effect system for atomicity. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation (PLDI'03)*, pages 338–349, 2003.

[21] P. Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem.* Lecture Notes in Computer Science 1032. Springer-Verlag, 1996.

[22] D. Grossman. Type-safe multithreading in Cyclone. In *Proceedings of the 2003 ACM SIGPLAN International Workshop on Types in Language Design and Implementation*, pages 13–25. ACM Press, 2003.

[23] K. Havelund and T. Pressburger. Model checking Java programs using Java PathFinder. *Software Tools for Technology Transfer (STTT)*, 2(4):72–84, 2000.

[24] T. A. Henzinger, R. Jhala, R. Majumdar, and S. Qadeer. Thread-modular abstraction refinement. In *CAV 2003: Computer Aided Verification*, pages 262–274, 2003.

[25] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'02)*, pages 58–70. ACM Press, 2002.

[26] G. Holzmann. *Design and Validation of Computer Protocols*. Prentice-Hall, 1991.

[27] G. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997.

[28] L. Lamport. Time, clocks, and the ordering of events in a distributed program. *Communications of the ACM*, 21(7):558–565, 1978.

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

[30] J. Mellor-Crummey. On-the-fly detection of data races for programs with nested fork-join parallelism. In *Proceedings of the 1991 ACM/IEEE conference on Supercomputing*, pages 24–33. ACM Press, 1991.

[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] D. Perkovic and P. Keleher. Online data-race detection via coherency guarantees. In *Proceedings of the 2nd Symposium on Operating Systems Design and Implementation (OSDI'96)*, pages 47–57, 1996.

[33] G. Ramalingam. Context-sensitive synchronization-sensitive analysis is undecidable. *ACM Transactions on Programming Languages and Systems*, 22(2):416–430, 2000.

[34] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 49–61. ACM, 1995.

[35] Robby, M. B. Dwyer, and J. Hatcliff. Bogor: an extensible and highly-modular software model checking framework. In *FSE 2003: Foundations of Software Engineering*, pages 267–276, 2003.

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

[37] M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis. In *Program Flow Analysis: Theory and Applications*, pages 189–233. Prentice-Hall, 1981.

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

[39] S. Stoller. Model-checking multi-threaded distributed Java programs. In *Proceedings of the 7th International SPIN Workshop on Model Checking and Software Verification*, Lecture Notes in Computer Science 1885, pages 224–244. Springer-Verlag, 2000.

[40] C. von Praun and T. Gross. Object-race detection. In *Proceedings of the 16th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'01)*, pages 78–82, Tampa Bay, FL, October 2001.