Unified Thread Safety Analysis

Dean F. Sutherland & DeLesley Hutchins

Abstract—**Remains to be written**

I. Introduction

<<Intro needs significant re-write to (a) shorten by half, and (b) replace text reused from prior papers. Last two paragraphs of intro propably stay.>>

Many modern software systems employ shared-memory multi-threading and are built using components such as libraries and frameworks. As a result, developers must manage the interactions between multiple threads as they execute within those components. To manage this complexity, developers assume that the libraries and frameworks they use will explicitly specify all of the necessary policies, preconditions, and postconditions ("thread-related policies", hereafter) of the design contract. However, many library and framework interfaces lack explicit specifications regarding threading and concurrency issues. In the absence of explicit specifications, developers cannot comply with the unstated thread-related policies. This can lead to errors such as state corruption and deadlock, and can be a significant cause of security vulnerabilities.

Writing code that obeys thread-related policies is difficult, even when the policies are known. There are many questions that must be answered by the programmer, such as:

> "Does the policy permit me to touch that data from this function or method?[1] Does it permit me to invoke that method from here? Is that data protected by a mutex?[2] Which mutex protects that data? Do I currently hold the mutex that must be held to execute that function? What threads might execute this portion of the code, and should they be permitted to do so?"

The answers to these questions can be both hard to find in documentation and difficult or impossible to glean from inspection of code. To make matters worse, the need to even ask these questions may not be apparent to many programmers. Yet similar questions arise for each and every data reference and method invocation that executes in the context of a multithreaded system.

When failures occur, programmers have trouble debugging multi-threaded code, in part because the symptoms they observe may lack an obvious connection to threading. Symptoms can include intermittent and seemingly-random data corruption, unexpected system or component pauses or freezes, and occasional obscure misbehavior of the application. The manifestation of such errors and failures at runtime is often dependent on conditions such as timing, context, load, and processor architecture. Change the conditions or the timing—by using the debugger, for example—and the symptom may change or disappear. Alternatively, failures or errors may occur on one out of each million executions of the faulty code; this is nearly impossible to isolate in a debugger, but still frequent enough to be an issue for long-running or frequently-run applications.

Because debugging these errors and failures is so difficult, programmers often fall back on code reviews to help locate the underlying faults in the code. However, code reviews have limited effectiveness when addressing non-local properties such as thread-related policy compliance; the consequences of those properties are spread widely and thinly throughout the code. This non-locality also means that every instance of each consequence must be handled correctly; if even one is wrong, the program may fail. These difficulties make static analysis an attractive alternative.

In spite of the rich literature regarding static analysis for lock-based concurrency, C and C++ programmers largely lack access to effective static analysis tools that address concurrency issues. One of the few widely-available C and C++ static analysis tools is the open source Clang static analyzer, which provides a useful thread-safety analysis that covers a limited—but interesting—subset of lock-based thread-safety cases. However, it lacks support for analysis of concurrency issues that are not lock-centric, such as thread confinement of data and expression of thread usage policy.

The goals of this work are to enable C and C++ framework and library authors to provide checkable specifications of their thread-related policies and to create a static analysis tool that can help C and C++ programmers find their concurrency errors for an interesting subset of both lock-based and non-lock-based cases. We accomplish this by (a) extending Clang's existing lock-based thread safety analysis for concurrency to include a policy-based thread role analysis for access control, (b) enhancing Clang's existing lock-based analysis to be more powerful and easier to maintain, (c) providing a cross-translation-unit annotation inference tool that reduces the number of required hand-written annotations, and also (d) improving error reporting by tracking the origin of each finding. We apply previously-known algorithms and techniques in a novel combination to produce actionable analysis results in a timely fashion.

II. Roots

The unified thread safety analysis builds on two prior concurrency analyses. These analyses—described below in brief—addressed related aspects of concurrency. However,

[1] Because our analysis applies to both C and C++ code, this paper uses the terms function and method interchangeably.

[2] Mutex is the typical term used in C and C++ to denote a variable that serves as a lock. We will use the terms mutex and lock interchangeably in this document.

their supported use-cases, and consequently their typical modes of operation, diverged almost to the point of incompatibility.

A. The Clang Thread Safety Analysis

The Clang Thread Safety Analysis that predates this work was based on [2], [12]. It allows users to specify which mutex protects what data. It then checks to ensure that the mutex is held when the data is accessed.

The primary focus of the Clang thread safety analysis tool is on useful analysis of fully-annotated programs. Its primary use-case is as documentation of required locking protocols. Users are interested in *documenting* the locking protocol for code readability and maintenance purposes. The analysis then enforces the documented protocol. It is in daily use on millions of lines of production code at a major technology company, and is publicly available as part of the Clang tool-suite.

This use-case and focus drove many design decisions for the tool, most notably that its analysis operates one method at a time, using annotations on methods to carry requirements across method calls. These decisions serve its primary constituency by providing not only maximum documentation of locking protocols but also the ability to run the analysis on each translation unit (TU, hereafter) in isolation from the rest of the program, thus providing full-fidelity analysis results with maximum parallelism. However, these design decisions fail to address the needs of users seeking extremely low required annotation effort, incremental adoptability, or the ability to analyze programs that are only partially annotated.

Successful design and engineering of a static analysis requires knowledge of the number and scope of the key items being analyzed. Examination of typical uses of the Clang thread safety analysis shows that the universe of mutex names is potentially unbounded, and is large in practice. Additionally, an individual mutex is typically relevant only to a small region of code.

B. Thread Role Analysis for Java

Thread role analysis has been previously implemented for Java, as reported in [22], [21], [23]. Thread role analysis both supports the analysis of concurrent software in lockless situations such as thread confinement and also enforces constraints on architectural coupling and dependency. It allows users to declare various "roles" threads may perform, identify program points where thread roles are added to—or removed from—the set of roles held by "the current thread" (that is, any thread that executes that program point), and identify constraints on methods and/or data that specify which combinations of thread roles are permitted to invoke the annotated method (or access the annotated data). The analysis then checks whether the as-written code and the expressed thread-related policy are consistent.

Thread role analysis differs from lock-based concurrency analyses by expressing role-based *access* control for both code and data without regard to the presence or absence of locks. The user-written thread role constraints for methods, in combination with the analyzed consequences of those constraints, provides two kinds of useful concurrency-related information for data. First, data that is accessed solely from a thread role that is unique is confined to a single thread; therefore concurrency is irrelevant with respect to that data. Second, data that is accessed from one or more non-unique thread roles requires additional concurrency control—such as locks or constraints on methods—because the data is potentially shared among multiple threads.

The primary focus of the prototype thread role analysis tool was on incremental adoption, analysis of partially-annotated programs, parsimony of expression, and fast incremental recomputation of analysis results after a program change. Users were interested in documenting both thread-related policies and constraints on architectural dependency for code readability and maintenance purposes, using as few hand-written annotations as possible. The prototype analysis tool was integrated into the Eclipse IDE and was used for case studies on several million lines of production Java code; to our knowledge, it has never entered production use.

This use-case and focus drove many design decisions for the tool, most notably the decision to use cross-TU inference to reduce the number of annotations to be written by programmers, the design of the analysis lattice to account for missing annotations, and a close focus on informative error reporting even in the face of missing information. These decisions served its primary constituency by reducing required annotation effort and providing actionable results even for partially annotated programs. However, these design decisions fail to address the needs of users who desire maximum documentation or who require full-fidelity results even when running the analysis on individual TUs in isolation from the remainder of the program.

Successful design and engineering of a static analysis requires knowledge of the number and scope of the key items being analyzed. Examination of typical uses of the thread role analysis shows that although the universe of thread role names is potentially unbounded, it tends to be quite small in realistic programs. The maximum number of distinct roles seen in any Java system was less than 32. Additionally an individual thread role is typically relevant to a large region of code, often a substantial fraction of the entire program being analyzed.

III. Single Unified Implementation with Shared Underlying Framework

Our challenge was to design a single analysis framework that can support both maximum analysis parallelism given fully annotated code as well as incremental adoption and analysis of partially-annotated programs given minimally annotated code. The combined analysis unifies the user-visible annotation language, and uses a single underlying representation for its analysis data. It addresses the divergent use-cases by using special “note”-style annotations to reify cross-TU analysis results for TU-at-a-time use.

This unified implementation approach provides benefits both for analysis developers and their users. Analysis developers can provide both analyses—rather than either analysis alone—for very little additional effort, while also supporting an expanded set of use-cases. Analysis users not only gain the benefit of two styles of concurrency analysis with multiple modes of operation, but also gain synergistic benefits beyond the scope of the individual analyses. For example, a combined analysis can automatically introduce a thread role that shadows a locked mutex; this thread role may be exposed to outside code without introducing the encapsulation and security issues that would arise from exposing the mutex itself.

In this section, we introduce the thread safety annotations and provide example uses. We then describe the underlying analysis framework implementation and algorithms; describe cross translation-unit annotation inference implementation; describe support for conflicting user requirements for static analysis at scale; and describe techniques for improved error reporting.

A. Thread Safety Annotation Language

The new combined analysis treats both mutexes and thread roles as specialized flavors of capability, which may be held, not held, or unknown-whether-held individually. The shared underlying framework treats each flavor of capability identically. This design choice allows us to build a combined thread safety analysis that has uniform syntax and semantics. This uniformity not only allows analysis implementation sharing, it also eases user training by supporting uniform expression of thread-related policies.

The combined analysis allows users to establish both data locking policies and thread usage policies using relatively few annotations. Users express their locking discipline by writing annotations that indicate which mutex protects which data as well as annotations that establish partial orders for lock acquisition. User express thread usage policy by writing annotations that declare which thread roles may execute particular code segments or access particular data items. Annotations are also used to establish rules regarding relationships between thread roles. Because the analysis treats both mutexes and thread roles as capabilities, users see a uniform annotation syntax for specifying combinations of capabilities.

A key difference in how the underlying framework treats mutexes and thread roles is driven by the design constraint that the presence or absence of thread safety annotations may never affect the generated code. Consequently, all capabilities must be fully erased before runtime. Thread roles are defined to exist *only* as capabilities; they lack runtime existence. However, for uniformity of implementation certain thread role annotations depend on the presence of notional thread role variables, which exist solely to support the analysis. To meet the requirement of full erasability, our analysis implementation removes these variables before runtime. Conversely, each mutex *capability* has a corresponding mutex *variable* which actually serves as a mutex at runtime. When capabilities are erased, these mutex variables are necessarily untouched; thus, the mutex variables continue to exist at runtime and remain in the program as written.

```
1  class CAPABILITY("mutex") Mutex {
2    void Lock()    ACQUIRE(this) {...}
3    void Unlock()  RELEASE(this) {...}
4  };
5  class CAPABILITY("thread-role") ThreadRole {
6    void acquire() ACQUIRE(this) {}
7    void release() RELEASE(this) {}
8  );
9  class SHARED_CAPABILITY("thread-role") SharedThreadRole
       {...};
10 ThreadRole FlightControl, Logger;
11 SharedThreadRole Worker;
12 Mutex mu;
```

Listing 1. Example Capability Declarations

1) Capability Declarations: Capabilities are declared by introducing variables or objects whose type is a C++ class (or C struct) decorated with the `CAPABILITY` annotation. The types `Mutex`, `ThreadRole`, and `SharedThreadRole`[3] are provided for users in library code; see lines 1, 5, and 9 of listing 1. Declarations of objects of these types introduce capabilities of the stated flavor; see lines 10-12 of listing 1. Listing 1 begins a notional example by declaring thread roles for a system that involves a realtime flight control thread, a data logging thread, and multiple low priority worker threads. It also declares a mutex that will protect data shared between these threads.

To ensure complete erasure of capabilities, all thread safety annotations are wrapped in macros that remove them from the code when thread safety analysis is turned off; these macros are written as `ALL_CAPS(...)` in our examples.[4] Because thread roles are an analysis-time-only concept, all language-level thread role type and object declarations are similarly protected by macros that remove them from the code when thread safety analysis is turned off; these macros are elided from our examples for clarity of presentation. When the analysis is turned on, our implementation ensures that user code refers to the notional thread role objects only as names in annotations or annotation macros; all non-annotation references to these objects are deleted from the compiler's syntax tree before code generation.

2) Acquiring and Releasing Capabilities: Certain methods and functions are annotated to indicate that they acquire (or release) a capability as part of their operation. For mutexes, these are typically the class `Lock()` and `Unlock()` methods (see lines 2 and 3 of listing 1). For thread roles, these are notional methods that are optimized away before runtime because thread roles have no runtime existence (lines 6 and 7). In either case, the capabilities are acquired or released when the method is invoked. The analysis also supports `TRY_ACQUIRE_*` annotations, which decorate functions that are permitted to fail to acquire the capability. One common use for these annotations is to support `TryLock()` style methods, whose specification states that they acquire the lock if it is available and return without waiting if the lock is held by another thread.

3) Lock Ordering and Capability Incompatibility: The `ACQUIRED_BEFORE` and `ACQUIRED_AFTER` annotations (see listing 2) allow programmers to establish partial orders

[3] The distinction between `ThreadRole` and `SharedThreadRole` capabilities is purely declarative; proving by analysis that a thread role is truly exclusive is not computable in full generality.

[4] Some examples include the actual C++11 annotation syntax; this appears as `[[annotation(...)]]`.

```
Mutex mu2 ACQUIRED_AFTER(mu);

[[incompatible(FlightControl, Logger)]]
//equivalent expression is ((FlightControl && !Logger) ||
//(!FlightControl && Logger)||(!FlightControl && !Logger))
INCOMPATIBLE(FlightControl, Logger, Worker)
```

Listing 2. Lock order & Incompatibility

```
void enQueueLogData(...) REQUIRES(FlightControl || Worker);
unsigned long[] deQueueLogData(...) REQUIRES(Logger);
bool moveAileron(...) REQUIRES(FlightControl);
```

Listing 3. Requirements for Code

for lock acquisition; this is a key capability for avoiding the potential for deadlock. The analysis establishes appropriate constraints to enforce the ordering established by these annotations, and checks these constraints on each mutex acquisition to determine whether the required ordering has been obeyed.

Thread usage policies often require knowledge that certain roles must be mutually exclusive; for example, that a given thread may never be both a FlightControl thread and a Logger thread simultaneously. The INCOMPATIBLE annotation (see listing 2) declares this requirement. Incompatibility as a property can apply to any group of capabilities, whether thread roles or mutexes. The analysis implements incompatibility by establishing a Boolean expression that is satisfiable only when either zero or one of the named capabilities is held. This expression is applied as a global constraint to any capability expression or test that involves any of the named capabilities. The [[incompatible(...)]] annotation in listing 2 is accompanied by a comment showing the equivalent Boolean expression; for brevity, this annotation is limited to two thread roles. The subsequent INCOMPATIBLE annotation expresses the actual property required for our ongoing example: no thread may ever hold more than one of the roles FlightControl, Logger, or Worker.

4) Capability Requirements: Methods and data can be annotated to permit invocation or access only from capability environments[5] that satisfy specific combinations of capabilities, using annotations that impose *capability requirements*. Capability requirement annotations express their requirement using Boolean expressions with capability names as variables and any combination of parentheses and the &&, ||, and ! operators (see listing 3). When a capability expression fails to constrain the value of a capability, the unconstrained capability is a "don't-care" value. That is, it could be either held or not-held without effect on the requirement. To correctly invoke an annotated method or access annotated data, callers (or accessors) must hold a combination of capabilities that satisfies the expression; the analysis reports an error for all other callers. Data items that lack a capability requirement are assumed to be unconstrained by design and may be freely accessed from any capability environment.

There is an important distinction, however, between functions whose capability requirement is not yet known—perhaps because a required annotation is missing—and functions that are unconstrained by design. *Unconstrained* methods neither require nor forbid any capability; they are marked with the annotation [[requires(true)]]. The implementation of an unconstrained method must be limited to call only other unconstrained methods. One common case of unconstrained methods is pure functions, whose output depends entirely on their input parameters; for example, many math functions from the standard libraries have this property. The multithreaded Listener pattern provides another example; its methods for adding and removing listeners are invokable from any thread, even though these methods necessarily have side effects.

Any method or function can be annotated with a capability requirement (see listing 3). Methods and functions may require either exclusive or shared capabilities for mutexes, using the REQUIRES and REQUIRES_SHARED annotations, respectively. To indicate that a function requires a mixture of shared and exclusive mutexes, the programmer names the exclusive mutexes in a REQUIRES annotation and the shared mutexes in a REQUIRES_SHARED annotation; the final requirement will be the && of the two expressions. Because the declaration of each thread role states whether it is always-exclusive or always-shared, thread role names can appear in either kind of requirement annotation without regard to their exclusivity. Mixing thread roles and mutexes in a single requirement is common in cases where a method should only be invoked from a limited group of thread roles and also must obey a specified locking discipline (*e.g.*, client-side locking).

Listing 3 continues our notional example, declaring an enQueueLogData method used by FlightControl and Worker threads to safely send logging data to the Logger thread, and a deQueueLogData method used by the Logger thread to receive log data. Finally, it declares a moveAileron method usable only from the FlightControl thread.

Any data item may be protected by a mutex using the GUARDED_BY annotation, which specifies that the named mutex must be held exclusively to *write* the data or held either exclusively or shared to *read* the data. The analysis tracks the exclusivity of mutex capabilities either directly (by observing whether the acquisition of the mutex was exclusive or shared) or transitively (by observing whether the mutex appeared in a REQUIRES or REQUIRES_SHARED annotation on the current function). A data item may similarly require a particular thread role for legitimate access using the REQUIRES annotation; this is a thread usage policy statement that is independent of locking. Unlike mutexes, requiring a thread role lacks any distinction between reads and writes; it simply permits *access* to the data item only from the stated combination of thread roles.

Listing 4 shows the declaration of the data queue used for inter-thread communication; it is protected by mutex mu . The listing also shows a portion of the implementation of enQueueLogData.[6] The thread safety analysis determines that mu is held exclusively at the point of access

[5] A "capability environment" is the conservative static approximation of the various combinations of capabilities that may be held by threads that execute a particular program point. See section III-B1 for a more complete definition.

[6] For clarity, this notional code assumes that the push_back() method never throws an exception.

```
vector<unsigned long> logDataQueue GUARDED_BY(mu)
  REQUIRES(FlightControl || Logger || Worker);

// in enQueueLogData
mu.Lock();
logDataQueue.push_back(input);
mu.Unlock();
```

Listing 4. Requirements for Data

```
class CAPABILITY("mutex") Mutex {
  void assertHeld() ASSERT_CAPABILITY(this) {...}
  bool isCurrentlyHeld() TEST_CAPABILITY(this, !this) {...}
};

bool runningOnLoggerThread() TEST_CAPABILITY(Logger, !
     Logger) {...}
```

Listing 5. Dynamic Tests

to `logDataQueue`, thus satisfying the `GUARDED_BY(mu)` annotation. The analysis also knows that `enQueueLogData` requires `FlightControl || Worker`; this satisfies the `REQUIRES` annotation on `logDataQueue`. Consequently, the access to the queue is permitted. Similarly, we see that `mu` is unlocked before return from the method; consequently, the capability environment is unchanged on method exit. This is consistent with the lack of an `ACQUIRE` or `RELEASE` annotation on the method.

Our current static analysis tool lacks support for alias analysis. As a consequence, the thread safety analysis cannot distinguish which specific instance of a class or structure is referenced by a particular alias or pointer. However, any access that names the class or structure (or field contained therein) provides sufficient information to support the analysis. Another, more important consequence is that the analysis cannot track references through aliases that point *within* an object or structure directly, without naming the object or structure explicitly; this is a source of unsoundness in the analysis. As a partial remediation for the inability to track aliases, the thread safety analysis also supports the `PT_GUARDED_BY` and `PT_REQUIRES` annotations. When applied to data, these annotations indicate that the variable or object referenced by the pointer is guarded by (for mutexes) or requires (for thread roles) the stated capabilities. When applied to function pointers, the `PT_REQUIRES` and `PT_REQUIRES_SHARED` annotations behave as though there were a `REQUIRES` or `REQUIRES_SHARED` annotation applied to the function referenced by the pointer.

5) Dynamic Capability Tests: Programmers frequently write code that contains predicates that test and report on the internal state of components. Some of these predicates indicate whether particular capabilities are held or not-held. The thread safety analysis supports three forms of dynamic test. The `ASSERT_CAPABILITY` and `ASSERT_SHARED_CAPABILITY` annotations (see listing 5) mark functions that dynamically check whether a mutex is held with appropriate exclusivity and halt the program when it is not-held. The `TEST_CAPABILITY(e1, e2)` annotation marks functions whose return value distinguishes particular capability sets. A return value of `true` indicates that capability expression `e1` holds; a return value of `false` indicates that capability expression `e2` holds. Most common usages of `TEST_CAPABILITY` involve expressions that are duals (e.g., `e1 == !e2`); in some more complicated cases the expressions may appear unrelated. The analysis tracks the results of calls to functions with these annotations and updates its knowledge of capabilities when analyzing basic blocks that are flow-dependent on those results.

B. Thread Safety Analysis Implementation

The thread safety analysis enforces the expressed locking discipline and thread usage policy using a per-method computation of the capability environment, which is then compared with the expressed requirements on methods invoked from and data accessed by that method. Even when operating in this core one-method-at-a-time mode, the analysis tool must nevertheless address issues raised by the interactions between functions. This requires design decisions regarding the origin of capability requirements for functions other than the one currently being analyzed and also how capability annotations interact with inheritance.

1) Core per-method analysis: The capabilities held by a thread at any given moment during program execution are the thread's *capability set*; this is a runtime concept. A thread's capability set could be modeled at runtime as a set of boolean values, using one boolean for each thread role capability, and two for each mutex capability.[7] Because a thread's capability set can change during program execution, the threads that execute a particular program point may have different capability sets on different executions or at different times during a single execution. The set of all capability sets held by all of the threads that execute a program point of interest at any time over all possible executions is the *possible capabilities* for that program point. This is a static concept that may be impossible to compute in full generality. Note that the possible capabilities for a program point is a set of sets of capabilities.

For a successful static analysis we must produce, for each program point of interest, an approximation of the possible capabilities at that point that is not only computable and conservatively correct, but also useful. We call this approximation the *capability environment* for the program point. The capability environment at a program point is also a set of sets of capabilities.

Annotations expressing requirements use Boolean expressions with the names of the capabilities serving as the variables. These capability requirements can be equivalently considered as defining sets of thread capability sets that are acceptable to the requirement. As with the capability environment, sets of thread capability sets are effectively sets of sets of capabilities. Consequently, to check whether the capability environment satisfies an expressed requirement, the thread safety analysis must determine whether the capability environment is a subset of the set expressed by the requirement. This check is the static equivalent of the dynamic question "does the current

[7] Mutexes require two booleans because each mutex capability can have three potential states: not-held, exclusive, or shared.

thread's capability set satisfy the requirement?". Note that the set inclusion property described above is exactly equivalent to checking whether the Boolean expression that represents the capability environment implies the Boolean expression that states the requirement. This equivalence allows the analysis to operate entirely on Boolean expressions, without ever needing to build the potentially large set form of the capability environments and of capability requirements. For efficiency, we implement these expressions using Binary Decision Diagrams (BDDs, hereafter) [10].

To check a single method, the thread safety analysis must compute a capability environment for each program point in the method. In a fully-annotated program, each method or function has an annotation stating the capabilities it requires.[8] The thread safety analysis uses this requirement as the initial capability environment on entry to the method.

The algorithm used for computing the capability environment for each program point within a method is a standard iterative forwards flow analysis. The lattice for the iteration is a disjunction lattice with Boolean expressions as members, `false` as Bottom, and a special `UNKNOWN` value as Top. The Boolean expressions reify sets of thread capability sets. The iteration converges towards a maximum; that is, the largest (equivalently, least precise) set of thread capability sets. We cannot use the expression `true` for Top, because `true` is the constraint expression (and thus the initial capability environment) for unconstrained methods; nevertheless, our iterative flow analysis requires that Top must be less precise than any legal lattice value.

The algorithm traverses each basic block in the control flow graph (*CFG*, hereafter), looking for annotations that modify the current capability environment. These are the acquire-, release-, assert- and test-style annotations described above. The updated lattice value at the end of each basic block flows to its successors by way of the `join()` operator, which is the logical-or of the Boolean expressions (extended so that `join(anything, UNKNOWN)` always yields `UNKNOWN`, as expected for Top).

Once the algorithm has converged, we make a final pass through the method during which we (a) check that the capability environment satisfies each capability requirement we encounter, (b) check each change to the capability environment to ensure that all global constraints imposed by incompatibility and lock ordering annotations are obeyed, (c) ensure that all exits from strongly-connected regions in the CFG have identical capability environments, (d) ensure that all exits from the method have identical capability environments, and (e) ensure that the capability environment on function return matches that specified by its annotations. That is, for each function whose annotations neither acquire nor release capabilities, the capability environment on return must be identical to the entry environment; similarly, for each function whose annotations specify capability acquisition or release, the capability environment on return must reflect exactly and only the changes specified by the annotations.

Representing both mutexes and thread roles as Boolean capabilities allows a single analysis implementation to perform both a traditional lock-based thread safety analysis and a non-lock-based thread role analysis. Using Boolean expressions to represent capability requirements and the capability environment (as in the prior thread role analysis work) provides improved expressive power for the lock-based analysis, particularly through improved support for negative capabilities and by supporting expressions involving the `||` operator. The flow-sensitivity drawn from the lock-based analysis provides greater expressive power for the thread role analysis.

2) Analysis for Multiple Methods in a TU: In basic operation, the core per-method analysis depends on explicit annotations to carry capability environment information across methods; that is, it expects fully annotated programs. This mode of operation provides two key benefits: first, it provides in-code documentation of all thread safety policies and requirements; second, it supports running the thread safety analysis as part of every system build. The associated cost is that users must annotate many (for mutexes) or substantially all (for thread roles) functions or methods in their programs. This cost can be prohibitive for many development teams.

One partial mitigation of the core analysis' expectation of fully annotated programs is provided by inheritance. In keeping with the "is-a" relationship typical of inheritance, thread safety requirements are inherited from parent to child. As a consequence, overriders inherit and must conform to the thread safety requirements found on their ancestors. Previous work with thread role analysis [23], for example, showed that inheritance from library and framework methods allows users to avoid writing significant numbers of thread safety annotations.

C. Capability Requirement Inference

Many development teams are unwilling or unable to incur the substantial cost of fully annotating their programs. Capability requirement inference can reduce this cost by up to an order of magnitude [21], [23]. This section first discusses our basic capability inference algorithm, then discusses the choice of a suitable inference scope. Finally, we describe our support for users with fundamentally conflicting requirements for the operation of static analyses at scale.

1) Capability Inference Basics: Performing sound inference on a collection of source code requires at least the following properties:

1) a known starting point for the properties being inferred;
2) complete access to all code that is part of the collection;
3) proof that the inference algorithm has adequate knowledge of all paths whereby control-flow can enter the code from outside.

These properties are fundamental requirements, regardless of the scale of the collection of code being analyzed.

Satisfying the second and third requirements requires some form of closed scope (an *inference scope*, hereafter) as well as knowledge of the methods and functions that are its *entire interface* to the outside world (its *API*, hereafter).[9] To discuss

[8] We discuss analysis for partially-annotated programs in section III-C.

[9] We defer discussion of how to satisfy these requirements and how to choose a suitable inference scope size to section III-C2.

the operation of the inference algorithm, we will assume that these requirements are met. Further, we assume that the inference analysis can see capability requirement annotations for each method that is part of the API of the inference scope[10] and also that these annotations serve as analysis cut-points. That is, we can assume their correctness on one side of the interface and check it on the other. Under these circumstances, we can infer the capability requirements of all methods in the inference scope that neither are part of its API nor have a specific capability requirement (the *inferable internal methods*, hereafter).

The inference algorithm is essentially identical to the iterative flow analysis used inside individual methods. Specifically, it uses the same lattice and `join()` operator with the same iterative forwards flow analysis. The key difference is that the inference algorithm iterates over the call graph of the inference scope using the annotations on API methods as its initial values. Where the checking algorithm traverses basic blocks, the inference algorithm traverses functions and methods. Where the checking algorithm propagates lattice values across control-flow edges in the CFG, the inference algorithm propagates lattice values across edges in the call graph. Because of this similarity, we take the obvious implementation approach of recursively invoking the per-method analysis to accomplish traversal of methods and functions.

The behavior of the inference algorithm when an API function's known capability requirement is missing provides the second motivation for using `UNKNOWN` as Top. When an API method lacks a known requirement, the analysis uses `UNKNOWN` in place of the missing capability requirement. Consequently, the initial capability environment inside the method is also `UNKNOWN`. When such a method calls an inferable internal method, the calling environment of the internal method must similarly be `UNKNOWN`; this is exactly the behavior provided by Top. Thus, using `UNKNOWN` as Top propagates the lack of knowledge throughout the inference scope exactly as expected. This differs from the constraint `true`, which is used for unconstrained methods. We use Top as the initial calling environment for all methods that both lack a predecessor in the call graph and also lack a known capability requirement.

When the algorithm finishes, it has computed an observed calling environment for all methods in the call graph; this result is necessarily complete only for an inference scope's internal methods. We use the calling environment as the capability requirement (and consequently the initial capability environment) for each inferable internal method; internal methods that already have known capability requirements keep those requirements. When the inference algorithm produces a calling environment of Top for an inferable internal method, we say that the method has an *unknown capability environment*.

The inference algorithm provides additional motivation for inheriting thread safety requirements from parent to child. When the definition, implementation and call site are all in the same inference scope, the inference algorithm can directly see everything it requires to determine both the methods that are potentially invoked from the call site and their capability requirements (if any). Inference of missing annotations can succeed in this case. Alternatively, when the implementation and/or the call site are in a different inference scope from the definition, the definition must be part of the API of some visible inference scope. Consequently, there must be a known capability requirement that applies to the method definition (whether due to a predefined default assumption, an explicitly written annotation, inheritance from a parent, or some other mechanism); this knowledge provides the necessary starting point for the inference algorithm even when the implementation is in a different inference scope from the definition and/or the call site.

2) Choosing a suitable inference scope: Our discussion of capability requirement inference thus far has carefully avoided the issue of choosing a suitable inference scope. Because the inference algorithm requires a closed scope with a known API, there are two obvious choices for the inference scope in C and C++ programs. The simplest choice is to define the inference scope as the TU. The language definition then tells us exactly which methods are inferable internal methods; everything else is the API. Unfortunately, this choice provides very few inferable internal methods; the potential reduction in the number of annotations the user must write is too small to be worth the implementation effort.

The second obvious choice for the inference scope is the entire program. With this choice of inference scope, programmers write only those capability requirements that represent true design commitments; everything else in the program is inferable—after all, the scope *must* be closed because it contains all the code in the entire program. Although this approach initially appears attractive, analysis developers and users quickly encounter a number of pragmatic impediments. These include:

Analysis Tool Capacity: A static analysis tool on any given host platform has a capacity limit; for current tools on current hardware this may fall somewhere between a few hundred thousand and perhaps one or two million lines of code. For any given capacity limit, there always seems to be somebody who would like to analyze an important program that is larger than the limit.

Source Code Availability: Few projects have access to all of the source code for all of the libraries they use. The source code of proprietary operating systems and proprietary libraries, for example, is rarely available to outside developers. Consequently, realistic analysis systems must support some form of analysis cut-point.

Analyzing Libraries: Library *authors* who wish to analyze their own code *fundamentally cannot* have access to "the entire program." After all, much of the client code for their library hasn't been written yet.

Analysis Time: Committing to whole-program analysis essen-

[10] This assumption simplifies the description of the inference algorithm. That said, the *specific origin* of these known requirements is an independent decision. Whether they arise from a well-chosen default assumption, from explicit user-written annotations, or from some other origin entirely is immaterial to the inference algorithm.

> tially guarantees that the analysis will be slow, because whole programs—including all required libraries—tend to be quite large. Developers generally prefer to get actionable results in a timely manner; they are unlikely to adopt tools that appear to "run forever."

As a consequence of these impediments, practical cross-TU analysis must fall back on a divide-and-conquer approach, using some scope larger than a single TU, but smaller than an entire program. Current C and C++ language semantics provide little help here. Fortunately, subgroup 2 of the ISO C++ committee is developing a specification for modules as an opt-in feature for C++ [1]. In its current form, the module specification focuses on identifying the header files that are the API for various modules in a system. This information is used to support pre-compilation of the API headers, consequently avoiding the need to repeatedly compile imported header files in every TU. This capability has been implemented in Clang for C, C++ and Objective-C as of the 3.3 release.

We have chosen modules as our inference scope. The API of a module provides the identified interface required for cross-TU capability requirement inference. However, modules in their current form fail to satisfy the other two requirements for sound inference: they lack identification of the TUs that implement the module, and they lack a completeness guarantee for the API.

Although there is ongoing standards activity that may add these missing pieces, at the moment we rely upon the user and the build system to provide three pieces of information. First, we need the list of all TUs that implement the module in question; this is presented on the command line when users invoke the analysis tool in inference mode. Second, the user must provide on the command line the name of the module those TUs implement. Finally, we rely upon users to place an annotation in the source code of each of these TUs that also names the module that the TU partially implements. Our inference results are thus conditionally-sound; they depend upon the correctness and completeness of the list of TUs presented for analysis, as well as depending on the user's promise to invoke functions inside a module only through its declared interface.

The user-written annotation in each TU provides two benefits. First, the inference tool can use it as input to a sanity check that ensures that all of the TUs presented for analysis are annotated as being "part of” the module to be analyzed. More importantly, it represents an experimental step towards representing this information in a language-defined fashion, rather than (or perhaps as well as) keeping it externally in the build system.

3) Conflicting User Requirements for Static Analysis at Scale: Some development teams ("TU-at-a-time" teams) require that the static analyses they adopt have the ability to produce full-fidelity results even when processing individual TUs. This allows them to integrate the analysis into their build system "as though" it were part of the compiler; thus the analysis is performed on every compilation of every TU. This model of analysis permits maximum parallelization of builds across dozens (or thousands) of distinct processors and machines. This comes at a cost, however. Each method that can be invoked from outside its home TU requires sufficient annotation to permit independent analysis (thread safety annotations, in our case). Further, for many such development teams, these annotations are hand-written and hand-maintained. Experience shows that writing all required annotations for a large source base is a daunting task; maintaining an existing large fully-annotated source base by hand is difficult but feasible.

Other development teams ("annotation minimizer" teams) want to gain the benefit of static analysis while minimizing their adoption cost in terms of programmer effort. Achieving this goal in practice requires both support for analysis of partially-annotated programs as well as analysis techniques that minimize the number of programmer-written annotations required for analysis. Past research [5] has shown that cross-TU inference of missing annotations over large inference scopes can reduce the number of required programmer-written annotations by roughly a factor of 10; this reduction represents a crucial adoptability issue for development teams that wish to minimize the number of annotations they must write. However, using cross-TU analysis also entails costs not present in the per-TU model. Inference scopes must be identified and described to the tooling. Integrating a cross-TU analysis tool may require modification of build systems. Some analysis tools using this approach attempt whole-program analysis; others choose inference scopes that are smaller than whole programs. Whatever the approach, the cross-TU analysis is necessarily less parallel than is per-TU analysis (by a factor of #TUs/inference-scope, on average).

We believe that to be successful, static analysis tools must support both usage styles described above. Accordingly, we have developed an approach that can support both TU-at-a-time and cross-TU analysis in a single tool. Given the capability to perform cross-TU inference of capability requirement annotations at large scale, the key innovation is to allow users to choose between either (a) performing the thread safety analysis checks directly over the large cross-TU inference scope using the inferred annotations (which is a trivial addition to the cross-TU analysis), which supports annotation-minimizer teams; or (b) recording the inferred thread safety requirement annotations for otherwise-unannotated methods in the source code. This makes the results of the inference manifest in the source code, thus supporting TU-at-a-time development teams.

We support recording inferred requirements by introducing an additional `*_NOTE` version of each of our capability requirement annotations—a `REQUIRES_NOTE` to match `REQUIRES`, and so on. The regular requirement annotations are considered to be design commitments; they are an important part of the interface of the functions and methods they apply to, and should be considered to be "essential" rather than "incidental." Changes to such design commitments are typically infrequent. Note-style annotations, by comparison, represent incidental knowledge that is purely a consequence of an essential design commitment somewhere else; they are inferred based upon the design requirements of the methods that transitively invoke them. Consequently, the content of these note-style annotations may change for non-local reasons—because a code modification elsewhere in the inference scope adds a new path that reaches some internal method, for example.

The cross-TU inference ignores note-style annotations on input. It always recomputes the inferred capability requirements from the non-note annotations. To support TU-at-a-time analysis, the cross-TU inference uses Clang's "rewriter" tool[11] to remove old note-style annotations, if necessary, and to write the newly inferred note-style capability requirements into the source code.

When performing thread safety checking of a method, the thread safety analysis treats note-style annotations as though they were ordinary requirement annotations. The only difference arises during error reporting: when a note-style annotation is implicated in a failed consistency check, the tool reports the inconsistency along with a comment that this may be a spurious failure due to an out-of-date note-style annotation. The annotation can be brought up-to-date either by running the cross-TU inference analysis, or by hand-editing the annotation. If the analysis continues to report an inconsistency after the inference results are updated, it is a true failure.

Note-style annotations provide two new capabilities for developers using the TU-at-a-time analysis approach. First, the distinction between incidental note-style annotations and true design commitments allows developers to focus on the the true design commitments that are important to them, while maintaining the benefits of TU-at-a-time analysis. Second, the inference tool's ability to re-write note-style annotations when necessary frees programmers from the chore of propagating changes to incidental annotations through their source code by hand.

D. Improving error reporting by tracking deponents

The thread safety analysis produces user-visible error messages when it detects an inconsistency between the thread safety annotations and the as-written code. A typical message may indicate, for example, a method call site and report that the caller fails to satisfy the capability requirement of the invoked method. The message typically provides the capability environment at the point of inconsistency and the unsatisfied requirement as additional information for the programmer. A notional example might look like `(compute && mu1) !==> (mu2)`.

To resolve the inconsistency, the programmer must determine whether the inconsistency is due to incorrect code, to incorrect annotations, or both. This determination requires understanding the origin of the capability environment (the left-hand side of the relation) and of the violated requirement (the right-hand side of the relation). Given this understanding, the programmer can quickly determine, for example, that the problem is that the code has failed to acquire mutex `mu2`.

In fully annotated programs, the requirement is written directly and is easy to find. Similarly, when the current method both has an explicit capability requirement of its own and also has relatively simple control-flow, it is straight-forward to understand why the capability environment has the particular capabilities shown. However, in large functions with complicated control flow, the origin of the capability environment or requirement may be far distant from the violation, making it difficult for programmers to understand the error message presented by the analysis.

Using the inference analysis adds another source of possible confusion. For example, when the analysis is being used in annotations-minimization mode, the explicit annotations giving rise to the inferred requirements that produce the current method's capability environment may be distant from the method in terms both of the call graph and also of the file in which they appear. This confusion due to annotation inference both resembles—on a much larger scale—and compounds with, that posed by large functions with complicated control flow. In medium-sized or larger modules—over lOOKLOC with potentially complex call graphs—understanding the error message can require not only substantial effort, but also broad understanding of large portions of the system under analysis.

This error reporting issue is not unique to our thread safety analysis; it also appears in other systems that present flow-analysis-based results to developers. For example, the error reports provided by compilers for languages using strong type inference are notoriously confusing when the inference fails.

Developers need assistance determining, for each variable in the inconsistency report, why the thread safety analysis has the particular values in the capability environment (and/or requirement) at the point of inconsistency. We address this issue by tracking the deponents of each increment of knowledge gained during the flow analysis.[12] We provide these deponents as additional information attached to each error report. This additional information provides developers with a direct answer to the question above.

To compute the deponents during our flow analysis (whether within a method or across TUs), the analysis maintains for each relevant variable in the lattice, the set of most-recent annotations that justify the variable's presence in the lattice. The rules for tracking deponent information are:

- When an annotation explicitly sets a value for a capability (mutex or thread role acquired/released, etc.), the analysis drops the old set for the variable representing the capability and starts anew by placing the annotation in the (newly emptied) set.
- When control-flow forks, the analysis sends the deponent sets down all outgoing paths (behaving identically to the lattice).
- When control-flow joins, the analysis calculates the union of the deponent sets for each variable (this also matches the handling of the lattice).

This computation is another forwards iterative flow analysis that can be performed simultaneously with the main flow analysis. Astute observers will already have noted that it computes the reaching definitions for each variable that is relevant to the current lattice expression, where annotations

[11] This is a standard part of the Clang tool suite that accepts xml-style input indicating changes to be made to existing source files. It was originally written to support the Clang source code formatting tool, but has since found many other uses.

[12] A deponent is one who attests to the truth of an assertion, typically in a deposition given under oath.

serve as definitions.

At any given point, the set of all such annotations for all variables in the support of the BDD are the answer to the question "why do we think this is the current lattice expression?" During the core per-method thread safety analysis checking pass, the lattice expression at any particular program point is the current capability environment at that program point. When the cross-TU inference analysis reaches its fixed point, the lattice expression at any particular function or method entry is the observed calling environment for that method or function. In either case, the annotations in the deponent sets are the most recent deponents that attest to the truth of the current lattice expression; they are the reason for the analysis' knowledge.

These deponent sets are insufficient to solve an additional problem. Recall that the cross-TU inference analysis requires known capability requirements for the APIs functions and methods exported from each module. It uses the `UNKNOWN` value when one or more of these requirements are missing. The `UNKNOWN` requirements from these locations 'poison' the lattice down the call-graph and flow-graph because `join( anything, UNKNOWN)` always yields `UNKNOWN`. In past case studies, this led to a frequently-experienced and vexing problem when analyzing partially-annotated programs: the analysis reported an inconsistency when a method of interest makes a function call, providing a message such as `(UNKNOWN ) !==> (GUI)`. This message indicates that the analysis was unable to determine the capability environment for the current method. To resolve the issue, we needed to add one or more currently-missing annotations. The missing annotations are probably capability requirements for API methods of the current module. In the absence of deponent information, programmers had to trace backwards not only through the per-method control-flow, but also through the module's call graph until they reached one or more API methods that lacked capability requirements.

To address this challenge, we extend our algorithm for tracking deponents by adding one additional deponent set to represent the deponents of `UNKNOWN`. The locations tracked in this set are those where the analysis expected to find required capability information, but the information was missing. These are exactly the locations that introduce the `UNKNOWN` value into the analysis lattice.

IV. Related Work

Our work uses thread roles (based on [22], [21], [23]) to manage threading-related access control, and locking discipline (based on [12], [2]) to avoid race conditions. Among the rich body of prior work on locking and data races, [14], [15] assure the absence of data races through the use of models of locking behavior; [3], [9], [8] assure the absence of data races using type systems; [11], [16], [18], [19] assure the absence of data races through the use of data flow-based and context-sensitive methods; and [20] reports on an early static lockset-based analysis to assure the absence of data races. Other work on thread roles includes [24], [13]. None of the prior work combines lock-based and non-lock-based concurrency control in a single analysis.

Our approach to modeling the external constraints imposed on an interface by a hidden mutex through use of a publicly-visible thread role as a safe proxy for the mutex is similar to permission and type-state based work such as [6], [7], [4], [5]. Analyses based on these concepts may offer a more direct approach to expressing these permissions. However, such analyses have not yet been demonstrated at scale.

V. Limitations and Future Work

This paper is the result of the authors' efforts to understand and design a system that combines their prior work. The unified analysis is funded and currently being built. There are several important limitations in the current design other than its partially-constructed state. First, it lacks support for C++ lambda functions. Second, its current design provides only limited support for function pointers and templates. Users can apply a capability requirement to a function pointer field or variable; this causes the analysis to check that requirement on all assignments to the function pointer. This approach suffices for many basic uses of function pointers, but cannot support analysis of such interesting cases as an RAII object that releases a mutex when it goes out of scope.

Third, many interesting uses of templates require tracking the specific functions available to each instantiation of the template. This often requires analysis based on the actual arguments passed to the constructor call used to create a particular object of the template type. Also, effective analysis of templates often requires the ability to distinguish between cases such as `vector<int GUARDED_BY(mu)>`, `vector <int>`, and `vector<int GUARDED_BY(mu)REQUIRES (role)>`, even though these types all share a single instantiation of the vector class (because our annotations are fully erased). Our tool is unable to analyze templates in these cases.

Fourth, the tool lacks support for general alias analysis; it is limited to type-based field accesses and cases noted by users via the `PT_GUARDED_BY`, `PT_REQUIRES`, and `PT_REQUIRES_SHARED` annotations. Although this unsoundness in our analysis leads to false negatives in some cases, our existing support for guarding data and function pointers nevertheless discovers many actual violations of locking and thread usage policies.

Finally, use of note-style annotations in TU-at-a-time analysis mode entails an unfortunate limitation on our reporting of deponents. Although the deponents for each particular note-style annotation are known during annotation inference, our current implementation is unable to record those deponents along with the annotation itself. Consequently, when running in TU-at-a-time mode, the analysis can only track deponents back to the beginning of each method. The usability impact of this limitation remains to be evaluated.

Potential future work is primarily based around addressing the above limitations. A pluggable type system for C++ along the lines of Java's JSR308 [17] would provide the support needed to address our function-pointer and template related limitations, using a combination of dependent types and generic capabilities. This would clearly entail a major design and implementation effort that has been beyond the scope of our project to date. Integrating our unified thread safety analysis with a high-quality alias analysis for C and C++ would enable us to improve our discovery of thread safety violations involving data. We believe that our error-reporting limitations for note-style annotations in TU-at-a-time analysis mode can be addressed by using a side-car data file for each TU to transmit the deponent information from the per-module inference tool to the per-TU analysis. Additional areas for future work include support for C++ lambdas and extension to support Objective-C.

VI. Conclusion

This work describes five primary research contributions. First, it demonstrates how to combine support for two related static analyses—lock-based concurrency analysis and thread role analysis—in a single implementation using a single analysis framework. Second, it demonstrates the viability of static analysis of C and C++ at scales larger than a translation unit but smaller than the entire program, by using cross-TU annotation inference to reduce the number of required handwritten annotations. Third, it demonstrates an *ad hoc* approach to identifying and grouping the TUs that implement modules for C++. Knowledge of the implementation of each module provides key semantic guarantees to support our cross-TU analysis. Fourth, it demonstrates a technique for supporting both TU-at-a-time and cross-TU analysis in a single tool, thus serving the needs of a variety of user communities. Finally, it demonstrates how to improve error reporting in flow-based analyses by tracking deponents.

References

[1] ISO C++ SG2 - Modules.

[2] Martin Abadi, Cormac Flanagan, and Stephen N. Freund. Types for safe locking: Static race detection for java. *ACM Trans. Program. Lang. Syst.*, 28(2):207–255, March 2006.

[3] R. Agarwal and S. D. Stoller. Type inference for parameterized race-free Java. In *Proc. Conference on Verification, Model Checking and Abstract Interpretation*, pages 149–160, 2004.

[4] Nels E. Beckman, Kevin Bierhoff, and Jonathan Aldrich. Verifying correct usage of atomic blocks and typestate. In *Proceedings of the 23rd ACM SIGPLAN Conference on Object-oriented Programming Systems Languages and Applications*, OOPSLA '08, pages 227–244, New York, NY, USA, 2008. ACM.

[5] Nels E. Beckman and Aditya V. Nori. Probabilistic, modular and scalable inference of typestate specifications. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 211–221, New York, NY, USA, 2011. ACM.

[6] Kevin Bierhoff and Jonathan Aldrich. Lightweight Object Specification with Typestates. In *FSE*, pages 217–226, September 2005.

[7] Kevin Bierhoff and Jonathan Aldrich. Modular typestate checking of aliased objects. In *Proceedings of the 22Nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications*, OOPSLA '07, pages 301–320, New York, NY, USA, 2007. ACM.

[8] Chandrasekhar Boyapati, Robert Lee, and Martin Rinard. Ownership types for safe programming: preventing data races and deadlocks. In *OOPSLA*, pages 211–230, 2002.

[9] Chandrasekhar Boyapati and Martin Rinard. A parameterized type system for race-free Java programs. In *OOPSLA*, pages 56–69, 2001.

[10] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Comput.*, 35(8):677–691, August 1986.

[11] Dawson Engler and Ken Ashcraft. RacerX: Effective, static detection of race conditions and deadlocks. In *SOSP*, pages 237–252, 2003.

[12] Cormac Flanagan and Stephen N. Freund. Type-based race detection for Java. In *PLDI*, 2000.

[13] Colin S. Gordon, Werner Dietl, Michael D. Ernst, and Dan Grossman. Javaui: Effects for controlling ui object access. In *Proceedings of the 27th European Conference on Object-Oriented Programming*, ECOOP'13, pages 179–204, Berlin, Heidelberg, 2013. Springer-Verlag.

[14] Aaron Greenhouse. *A Programmer-oriented Approach to Safe Concurrency*. PhD thesis, Carnegie Mellon, May 2003.

[15] Aaron Greenhouse, T. J. Halloran, and William L. Scherlis. Observations on the assured evolution of concurrent Java programs. *Sci. Comput. Program.*, 58(3):384–411, 2005.

[16] Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. Race checking by context inference. In *PLDI*, pages 1–13, 2004.

[17] JSR308 Expert Group. Jsr308:.

[18] Mayur Naik, Alex Aiken, and John Whaley. Effective static race detection for Java. In *PLDI '06*, pages 308–319, 2006.

[19] Polyvios Pratikakis, Jeffrey S. Foster, and Michael Hicks. Locksmith: Context-sensitive correlation analysis for race detection. In *PLDI '06*, pages 320–331, 2006.

[20] N. Sterling. Warlock: A static data race analysis tool. In *USENIX Winter Technical Conference*, pages 97–106, 1993.

[21] Dean F. Sutherland. *The Code of Many Colors: Semi-automated Reasoning about Multi-Thread Policy for Java*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA 15213, May 2008.

[22] Dean F. Sutherland, Aaron Greenhouse, and William L. Scherlis. The code of many colors: relating threads to code and shared state. In *PASTE '02: Proceedings of the 2002 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 77–83, New York, NY, USA, 2002. ACM.

[23] Dean F. Sutherland and William L. Scherlis. Composable thread coloring. In *PPoPP '10: Proceedings of the 15th ACM SIGPLAN symposium on Principles and practice of parallel programming*, New York, NY, USA, 2010. ACM.

[24] Sai Zhang, Hao Lü, and Michael D. Ernst. Finding errors in multithreaded gui applications. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, ISSTA 2012, pages 243–253, New York, NY, USA, 2012. ACM.