Detecting Data Race and Atomicity Violation via Typestate-Guided Static Analysis

Yue Yang Anna Gringauze Dinghao Wu Henning Korsholm Rohde

Microsoft Corporation
<jasony,annagrin,dinghao,herohde>@microsoft.com

ABSTRACT

The correctness of typestate properties in a multithreaded program often depends on the assumption of certain concurrency invariants. However, standard typestate analysis and concurrency analysis are disjoint in that the former is unable to understand threading effects and the latter does not take typestate properties into consideration. We combine these two previously separate approaches and develop a novel typestate-driven concurrency analysis for detecting race conditions and atomicity violations.

Our analysis is based on a reformulation of typestate systems in which state transitions of a shared variable are controlled by the locking state of that variable. By combining typestate checking with lockset analysis, we can selectively transfer the typestate to a *transient* state to simulate the thread interference effect, thus uncovering a new class of typestate errors directly related to low-level or high-level data races. Such a concurrency bug is more likely to be harmful, compared with those found by existing concurrency checkers, because there exists a concrete evidence that it may eventually lead to a typestate error as well.

We have implemented a race and atomicity checker for C/C++ programs by extending a NULL pointer dereference analysis. To support large legacy code, our approach does not require *a priori* annotations; instead, it automatically infers the lock/data guardianship relation and variable correlations. We have applied the toolset to check a future version of the Windows operating system, finding many concurrency errors that cannot be discovered by previous tools.

Categories and Subject Descriptors

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

General Terms

Algorithms, Reliability

Keywords

atomicity, data race, concurrency, defect detection

1. INTRODUCTION

Multithreaded programs are hard to get right. A common concurrency error is *data race*, which occurs when two threads concurrently access the same data without synchronization, and at least one of these accesses is a write. However, race-freedom does not guarantee correct synchronization, because program correctness often depends on a stronger non-interference property called *atomicity*, which requires that every concurrent execution of a set of operations is equivalent to some serial execution of the same operations. Atomicity violations, sometimes called *high-level data races*, can cause erroneous behaviors when a consistency requirement exists between multiple pieces of shared data.

Properties such as race-freedom and atomicity can be categorized as *generic* concurrency properties—they are invariants for ensuring proper synchronization on shared-memory data structures. Another important class of program properties, which has been intensively studied in prior research for sequential programs, is related to *typestates* [1]. Typestates extend the ordinary types: For an object created during program execution, its ordinary type does not change through the lifetime of the object but its typestate may be updated during the course of the computation. Typestate errors often indicate violations to safety conditions, which require that operations can only be invoked on objects with appropriate states.

Despite significant advances in both fields, typestate analysis and concurrency analysis are disjoint since the former is unable to understand threading effects and the latter mostly focuses on generic concurrency properties. This introduces two drawbacks. (1) Typestate systems are limited in the presence of concurrency because they would miss many bugs introduced by threading issues. (2) Concurrency tools are often overly conservative, resulting in a large amount of warnings that are benign. This is especially a problem for atomicity analysis as many seeming violations to the atomicity assumption are not harmful because they do not break any critical invariants that programmers care about.

Example. To illustrate an atomicity problem, consider the code snippet of a Win32 program shown in Figure 1. In function `ProcessBuffer`, `p->buffer` is tested at line 3 to make sure it is not `NULL` before being dereferenced at line 10. Although all shared accesses are protected by locking, a problem would arise if function `FreeBuffer`, executed by

```
        typedef struct {
            CRITICAL_SECTION cs;
            int* buffer;
        } DATA;

1       void ProcessBuffer(DATA* p) {
2           EnterCriticalSection(&p->cs);
3           if (p->buffer == NULL) {
4               LeaveCriticalSection(&p->cs);
5               return;
6           }
7           LeaveCriticalSection(&p->cs);
8           // Do something
9           EnterCriticalSection(&p->cs);
10              *p->buffer = 1;
11          LeaveCriticalSection(&p->cs);
12      }

1       void FreeBuffer(DATA* p) {
2           EnterCriticalSection(&p->cs);
3           if (p->buffer) {
4               delete(p->buffer);
5               p->buffer = NULL;
6           }
7           LeaveCriticalSection(&p->cs);
8       }
```

Figure 1: A simplified code snippet of a Win32 program that is race-free but unsafe.

another thread, is interleaved between the two locking segments in `ProcessBuffer` at line 8. In this case, `FreeBuffer` would reset the buffer to `NULL`, causing a crash at line 10 in `ProcessBuffer`.

This code is buggy because it breaks an implicit invariant: The validation against the typestate of an object and the operation on the object that relies on the proper typestate need to be treated as a set of *atomic actions*, *i.e.*, they should appear to be carried out as if the actions were executed sequentially, independent of thread interleaving.

From this example, several interesting observations can be drawn. (1) A traditional typestate checker cannot detect this safety violation. Such a checker would conclude that `p->buffer` cannot be `NULL` at line 10 in `ProcessBuffer` since the function would have returned at line 5 otherwise. (2) A standard race detector cannot catch this error because the code is race-free: Every access to variable `p->buffer` is guarded by lock `p->cs`. (3) Although a strict atomicity checker that assumes every function needs to run atomically would be able to issue an alarm, it is not practical to always enforce atomicity at function boundaries. A common practice is to break down a large piece of code into small fragments of locking blocks for performance reasons. Consequently, a strict atomicity checker would generate too many false alarms, making it hard to find meaningful results.

Concurrency errors, such as the one illustrated here, are often *semantic errors* involving implicit high-level invariants. These subtle bugs, which typically exhibit in corner-case scenarios, are very hard to find, reproduce, and diagnose. Yet, for industrial software that is used daily by millions of people, no code path can be treated as a rare case. With the emerging trend of multicore technology, it is expected that concurrency-related problems will manifest even more frequently.

In this paper, we present a new approach that takes both typestate property and concurrency property into account. We simulate the thread interference effect by reformulating typestate systems such that the state transition of a shared variable is controlled by the locking state of that variable. Our method is based on two key insights.

1. We can simulate the "worst case" scenario due to thread interleaving by killing the state information associated with an object—this is represented by transferring the typestate to a transient state—whenever the object is not guarded by its intended guarding lock. Consider function `ProcessBuffer` in the above example, if the analysis can understand that `p->buffer` should be guarded by `p->cs` and reset the state of `p->buffer` when the guarding lock is released at line 7, the NULL dereference at line 10 would be uncovered.

2. To precisely simulate the thread interference effect, the central question is how to determine when the state of a variable needs to be brought to the transient state—too much (*e.g.*, when triggered by unrelated lock release) would lead to over-approximation and too little (*e.g.*, when failing to update correlated variables) would result in under-approximation. To address this issue, we apply inference techniques to automatically identify lock/data and data/data correlations.

The advantages of the combined approach are two-fold. First, the "thread-sensitive" aspect of the analysis extends traditional typestate checking from the sequential context to the multithreaded context. Second, the "typestate-driven" aspect of the analysis enables a more focused checking by pinpointing concurrency problems that could eventually lead to typestate bugs.

Ideally, an analysis tool must meet several challenging and sometimes conflicting goals at once.

- The analysis should offer enough precision. Unlike Java or C#, C/C++ programs do not require locking to be syntactically scoped. As a result, *flow-sensitivity* is important. Also, since lock acquire and lock release operations can be conditional, *path-sensitivity* is critical as well. These analysis features are all supported by our tools. Additionally, our typestate-driven approach further improves precision by distinguishing harmful bugs from benign warnings.

- The analysis must scale to large programs with millions lines of code. We support this by applying modular (intra-procedural) checking that analyzes one function at a time.

- Modular checking requires annotations and we need to support large legacy code base that is infeasible to annotate by hand. The conventional wisdom is that any annotation-based approach is cumbersome and hard to deploy. We strive to change this perception by showing that manual annotation effort can be largely replaced by a series of automatic inference techniques.

The technical merit of this paper lies in addressing these challenges and integrating practical techniques to produce

useful results. In summary, we make the following contributions:

- We present a reformulation of typestate systems in which state transitions of a variable are subject to the locking state of that variable. In particular, we show how to combine a NULL pointer dereference typestate checker with a lockstate checker to detect data races and atomicity violations. To the best of our knowledge, this is the first effort that combines typestate analysis with lockset analysis.
- We present an inference algorithm for determining correlated variables that should be bundled. Although the notion of annotating lock/data guardianship relation with the `__guarded_by` annotations is not new, applying a judicious choice of heuristics to infer such annotations and further use them to reveal atomicity violations is a unique contribution.
- We have implemented a set of concurrency tools, including a concurrency annotation inference and checking tool EspC and an atomicity checker EspA, and applied them to large real-world software systems.

The remainder of the paper is organized as follows. In Section 2, we review the techniques applied in our approach. In Section 3, we present the concurrency annotation inference methods. In Section 4, we describe the implementation of our tools. In Section 5, we discuss experimental results. In Section 6, we review related work. We conclude in Section 7.

2. APPROACH

In this section, we use an important safety property—freedom from NULL pointer dereferences—as a concrete example to illustrate our approach.

2.1 Conventional Typestate Analysis

A typestate property can be captured by a finite state machine where the nodes represent typestates and the arcs correspond to operations that lead to state transitions. Formally, a typestate automaton for a variable, say p, is a tuple

$$A = \langle Q, \Sigma, init, \delta, Err \rangle,$$

where Q is the set of all states, Σ is an alphabet denoting operations on edges, $init$ is the initial state, $\delta : Q \times \Sigma \rightarrow Q$ is the state transition function for p, Err is a set of terminating states representing error states, $\forall e \in Err, o \in \Sigma.\ \delta(e, o) = e$.

2.1.1 Notations

A variable, represented by a C/C++ path expression, can be classified as either a *data variable* or a *lock variable*. In Win32 programs, a lock variable can be a critical section declared as `CRITICAL_SECTION`, a mutex, a spin lock, or a user-defined lock. Variables are mapped to the corresponding memory locations. In the definitions below, for example, data variable p is mapped to memory location m and lock variable *lock* is mapped to memory location l.

As a convention, we use superscripts to distinguish various automata. Specifically, we use *null* to represent the NULL pointer dereference automaton, *ls* to represent the lockset automaton, and c to represent the combined concurrency automaton. In addition, we use subscripts to represent different instantiations. For example, A_m represents an instantiation of automaton A for memory location m.

2.1.2 Analysis Definitions

Let $Var = \{p_1 \dots p_i\}$ be the set of all variables in the analyzed function F. Let $Mloc = \{m_1 \dots m_j\}$ be the set of all memory locations in F, and function $\rho : Var \rightarrow Mloc$ be the mapping function giving memory locations for each variable. Then, for every memory location m, the finite state machine for variable p can be instantiated with m to yield a specialized execution automaton

$$A_m = \langle Q_m, \Sigma_m, init_m, \delta_m, Err_m \rangle,$$

where $A_m = A[m/p]$, $Q_m = Q[m/p]$, $\Sigma_m = \Sigma[m/p] = \{o[m/p], o \in \Sigma\}$, and the state of the program at program point x is given by the tuple $State(x) \in Q_{m_1} \times \dots \times Q_{m_j}$. We denote by $State(x)_m$ the element of the tuple $State(x)$ for location m.

2.1.3 NULL Pointer Dereference Automaton

The state machine for a standard NULL pointer dereference analysis is illustrated in Figure 2(a). Every variable, say p, starts with a `MaybeNULL` state, and can be transferred to other states depending on various operations occurred to that variable. We use $toNULL(p)$ and $toNonNULL(p)$ to represent operations that lead p to `NULL` and `NonNULL`, respectively. An error occurs when a dereference is taken from the `NULL` state or the `MaybeNULL` state.

The automaton for NULL pointer dereference is defined as $A^{null} = \langle Q^{null}, \Sigma^{null}, init^{null}, \delta^{null}, Err^{null} \rangle$, where

$Q^{null} = \{$`MaybeNULL`, `NULL`, `NonNULL`, `MaybeNULLDeref`, `NULLDeref`$\}$,
$\Sigma^{null} = \{toNULL(p), toNonNULL(p), *p\}$,
$init^{null} =$ `MaybeNULL`,
$Err^{null} = \{$`MaybeNULLDeref`, `NULLDeref`$\}$,
and δ^{null} is defined as follows (for state-changing edges):
$\forall s \notin Err.\delta^{null}(s, toNULL(p)) =$ `NULL`
$\forall s \notin Err.\delta^{null}(s, toNonNULL(p)) =$ `NonNULL`
$\delta^{null}($`NULL`$, *$`p`$) =$ `NULLDeref`
$\delta^{null}($`MaybeNULL`$, *$`p`$) =$ `MaybeNULLDeref`

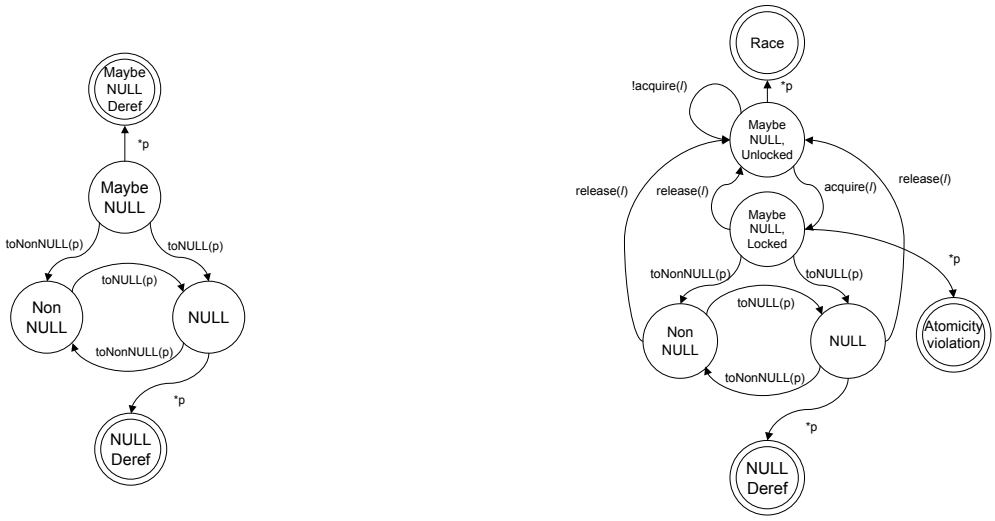
2.2 Lockset Analysis

Our analysis needs to understand the lock/data guardianship relation as well as correlations between data variables to guide state transitions. We obtain this information via concurrency annotations, which can be either manually added or automatically inferred. We explain the need of concurrency annotations in Section 2.2.1 and describe our lock analysis in Section 2.2.2.

2.2.1 Concurrency Annotations

A fundamental limitation of the mainstream programming languages in use today is that they do not directly support the specification of concurrency requirements. Programmers have to rely on informal documentation to express their intention regarding the usage of locks. To improve the quality of multithreaded software, we have designed *Concurrency SAL* as an extension to Microsoft's Standard Annotation Language (SAL) [2]. Concurrency SAL defines a set of annotations that make the implicit locking rules explicit in C/C++ programs. In this section, we describe a small subset of Concurrency SAL that is related to this paper.

Data Protection. An effective technique for avoiding race conditions is to always acquire the guarding lock before ac-



(a) (b)

Figure 2: Finite state machines for checking NULL pointer dereference. Figure (a) shows the traditional state machine for thread-insensitive analysis. Figure (b) shows the extended state machine for thread-sensitive analysis.

```
typedef struct {
    CRITICAL_SECTION cs;
    __guarded_by(cs) int* buffer;
} DATA;
```

Figure 3: The `__guarded_by` annotation clarifies the lock/data guardianship relation.

cessing a shared variable. The `__guarded_by` annotation is used to specify which lock is intended for guarding the shared data. For example, the guardianship relation between `cs` and `buffer` in Figure 1 can be formalized with the `__guarded_by` annotation, as illustrated in Figure 3.

Caller/Callee Locking Contract. Whenever a lock is needed to protect some shared access, it is important to clarify whether the function itself or its caller is responsible for acquiring the lock. Annotation `__requires_lock_held` on function `Foo` states that the caller of `Foo` must acquire the lock prior to calling `Foo`.

Locking Side Effect. C/C++ programs do not enforce lock acquires and lock releases to be syntactically scoped. This free-style locking mechanism is a common source for bugs. Annotations `__acquires_lock` and `__releases_lock` formalize two important locking side effects of a function: acquiring a lock and releasing a lock, respectively.

2.2.2 Lockset Automaton

We now define a lockset-based analysis for data race detection. Let $acquire(lock)$ and $release(lock)$ represent the acquire operations and release operations for lock variable $lock$, and $access(p)$ represent the access operations for data variable p. The `__guarded_by` annotations, either manually added by programmers or inferred by our tool, yield the following relation:

$GuardedBy \in Mloc \times Mloc$, where $(m, l) \in GuardedBy$ iff $\rho(p) = m$ and $\rho(lock) = l$ and the `__guarded_by(lock)` annotation is applied to p.

Based on the `__guarded_by`($lock$) annotation on variable p, we can define the lockset algorithm as a typestate system with automaton A^{ls}.

$A^{ls} = \langle Q^{ls}, \Sigma^{ls}, init^{ls}, \delta^{ls}, Err^{ls} \rangle$,
$Q^{ls} = \{\texttt{Unlocked}, \texttt{Locked}, \texttt{Race}\}$,
$\Sigma^{ls} = \{acquire(lock), release(lock), access(p)\}$,
$init^{ls} = \texttt{Unlocked}$,
$Err^{ls} = \{\texttt{Race}\}$,
and δ^{ls} is defined as follows (for state-changing edges):
$\delta^{ls}(\texttt{Unlocked}, acquire(lock)) = \texttt{Locked}$
$\delta^{ls}(\texttt{Locked}, release(lock)) = \texttt{Unlocked}$
$\delta^{ls}(\texttt{Unlocked}, access(p)) = \texttt{Race}$

For each data location m protected by lock location l, we create a specialized automaton $A^{ls}_{m,l} = A^{ls}[m/p, l/lock]$ ($s.t.$ $GuardedBy(m, l)$ is true). Then the lockset at program point x can be defined as

$$LS_x = \{l : \exists m. GuardedBy(m, l) \ \wedge \ State(x)^{ls}_m = \texttt{Locked}\}$$

2.3 Combined Concurrency Analysis

We now define an extended typestate system for concurrency analysis. Given a traditional typestate automaton A, we construct a combined automaton that is a modification of a product automaton from A and A^{ls}, with the addition of a transient state for each protected variable. When a variable

is in the transient state, it indicates that the variable is not currently guarded and thus it can be modified by another thread. We assume that the initial state *init* in automaton A always represents the unknown state for the variable, *i.e.*, it can be any possible state of the variable. Hence, we can reuse the initial state as the transient state for the newly created concurrency automaton.

For example, from the NULL pointer dereference automaton A^{null} and the lockset automaton defined in Section 2.2.2, the resultant combined automaton is similar to the one shown in Figure 2(b). For conciseness, Figure 2(b) is simplified to reduce the number of states. This automaton is created only for shared data that is explicitly annotated with the `__guarded_by` annotations. For un-annotated data, the original typestate automaton is reused.

We assume that the set of errors in the original typestate automaton is divided into "may errors" and "must errors": $Err = MayErr \cup MustErr$. The distinction between "may errors" and "must errors" is based on whether there exists a direct evidence of a bug. For instance, suppose a pointer p starts with `NULL` in function `Foo`. If `Foo` makes the dereference of p right away, it is a "must error". On the other hand, if `Foo` calls an external function that may or may not properly initialize p and then makes the dereference, it is a "may error".

For the combined concurrency automata, we define an automaton per location and then make connections between state machines for locations of the same *bundle*, which represents a set of correlated locations. We will formalize the notion of bundle and describe how to compute it in Section 3.2.

We use $A_l^{ls} = A_{m_1}^{ls} \times \cdots \times A_{m_k}^{ls} = \langle Q_l^{ls}, \Sigma_l^{ls}, init_l^{ls}, \delta_l^{ls}, Err_l^{ls} \rangle$ to represent a standard product automaton related to the same data bundle B_l, associated with lock l, where $m_1 \ldots m_k$ belong to B_l. Given a data location m guarded by a lock location l, the combined automaton is defined as follows:

$A_m^c = \langle Q_m^c, \Sigma_m^c, init_m^c, \delta_m^c, Err_m^c \rangle$,
$Q_m^c = Q_m \times Q_l^{ls}$,
$\Sigma_m^c = \Sigma_m \cup \Sigma_l^{ls}$,
$init_m^c = \langle \mathtt{init_m}, \mathtt{Unlocked_l} \rangle$,
$Err_m^c = (\mathtt{MustErr_m} \times \mathtt{Q_l^{ls}}) \cup (\mathtt{MayErr_m} \times \{\mathtt{Race_l}, \mathtt{Locked_l}\})$,

and δ_m^c is defined as follows:

(1) Simulate the thread interference effect by entering the transient, unprotected state when unlocked:

$\forall s \in Q_m.\delta_m^c(\langle s, \mathtt{Locked_l} \rangle, release(lock)) = \langle \mathtt{init_m}, \mathtt{Unlocked_l} \rangle$

(2) Stay in the transient, unprotected state until locked:

$\forall o \notin acquire(lock)$.
$\delta_m^c(\langle \mathtt{init_m}, \mathtt{Unlocked_l} \rangle, \mathtt{o}) = \langle \mathtt{init_m}, \mathtt{Unlocked_l} \rangle$

(3) Go to the protected, unknown state when locked:

$\forall s \in Q_m.\delta_m^c(\langle s, \mathtt{Unlocked_l} \rangle, acquire(lock)) = \langle \mathtt{init_m}, \mathtt{Locked_l} \rangle$

(4) Perform as the original automata when protected:

$\forall s \in Q_m, o \in \Sigma_m.\delta_m^c(\langle s, \mathtt{Locked_l} \rangle, \mathtt{o}) = \langle \delta_\mathtt{m}(\mathtt{s}, \mathtt{st}), \mathtt{Locked_l} \rangle$

(5) Go to the error state from the transient state for any operations leading to errors before:

$\forall s \in Q_m, o \in \Sigma_m.\delta_m(s, o) = e, e \in \mathtt{MayErr_m}$
$\delta_m^c(\langle \mathtt{init_m}, \mathtt{Locked_l} \rangle, \mathtt{o}) = \langle \mathtt{e}, \mathtt{Locked_l} \rangle$ (atomicity violation)
$\delta_m^c(\langle \mathtt{init_m}, \mathtt{Unlocked_l} \rangle, \mathtt{o}) = \langle \mathtt{e}, \mathtt{Unlocked_l} \rangle$ (race condition)

Those additional errors, revealed by the combined concurrency automaton but not by the original typestate automaton, are generated from the transient state induced by threading effects, *i.e.*, they are potential typestate violations directly related to concurrency. With the capability of uncovering a whole new class of concurrency-related typestate errors, our extended typestate system improves the soundness of a traditional typestate system. Driven by typestate errors, our approach also improves the precision of a conventional concurrency checker. This unique balance between the soundness/precision tradeoff offers a sweet spot for checking high-level and low-level race conditions.

Although this paper has focused on extending the NULL pointer dereference state machine, our approach is generic and can be extended to other typestate properties.

3. LOCK INFERENCE

Since state transitions in the combined concurrency analysis are driven by the activities associated with guarding locks, finding lock/data correlations is key to the precision of our analysis. If the analysis over-kills the state information following unrelated lock releases, false positives would be generated. On the other hand, if the analysis misses any lock/data guardianship relation, it would result in false negatives. In the extreme case, if there does not exist any `__guarded_by` annotation, the concurrency automaton would degenerates to the original property automaton.

Because manually annotating legacy code can be very costly, we apply inference techniques to automatically extract the implicit locking rules.

3.1 Data Protection Inference

For each analyzed function, we infer the `__guarded_by` annotations for formal parameters or global variables accessed at the function. Our heuristic-based inference algorithm is motivated by the observation that programmers are mostly correct, thus we can infer their assumptions based on some strong evidence exhibited by certain code paths.

For instance, our "high-confidence" seeding `__guarded_by` annotations are generated as follows: For every locking block, we conclude that the first access after the lock acquire and the last access before the lock release need to be protected by the guarding lock. The intuition is that developers do not use locking liberally; they try to minimize the locking scope to improve performance and avoid deadlocks. In Figure 1, for example, the shared variable `p->buffer` is accessed right after `p->cs` is acquired. If `p->buffer` does not need to be guarded by `p->cs`, the access could have been moved before the lock acquire operation.

Our algorithm also allows the inference to be more aggressive (which can potentially be more noisy) by selecting the best fitting lock from a set of candidate locks.

At each protected access to a structure field with location m, we sort the currently held locks into the following buckets:

- Priority 1: lock locations which are fields of the parent of m.
- Priority 2: lock locations which are reachable from the parent of m, but are not at the same level with m.
- Priority 3: lock locations reachable from formal parameter locations.
- Priority 4: lock locations reachable from global locations.

We choose the guarding lock from a non-empty bucket with the highest priority (the priority with the smallest number). We ignore the locks reachable from m as they are probably intended to protect fields of m, but not m itself.

3.2 Bundle Computation

In addition to lock/data correlations, data/data correlations are also important for atomicity analysis. Suppose the x and y coordinates of a `Point` object need to be updated atomically, fragmenting the locking blocks in updating x and y would lead to data inconsistency.

To be able to discover this kind of atomicity violation, we introduce the notion of *bundle*. A bundle collects a set of data locations whose updates need to be grouped atomically. Whenever a variable is brought into the transient state, the atomicity invariant could be broken, hence all variables in the same bundle are also transferred to the unknown state. For the `Point` data structure, our analysis needs to recognize x and y as a bundle.

Bundle identification can be viewed as a technique to generalize our atomicity analysis. It uncovers *high-level data dependence*, which is different from traditional data dependence induced by reads and writes.

Our bundle inference technique is based on the following observation: Mutual exclusion locks remain the *de facto* mechanism for ensuring data consistency, partially due to the lack of direct support for atomicity from programming languages. As a result, the same lock is likely to be used to guard a group of variables that need to be processed atomically. Therefore, we use the guarding lock as a hint to detect a data bundle: $B_l = \{m : \ (m, l) \in \mathit{GuardedBy}\}$. Alias analysis allows us to identify path expressions with same locations.

3.3 Locking Side Effects

The free-style locking mechanism in C/C++ poses a challenge for annotation inference because all the locking rules and locking side effects are intertwined—one misunderstanding would lead to other wrong conclusions. As a result, we also need to infer other concurrency annotations, *e.g.*, those related to locking side effects.

For `__acquires_lock`, we infer that a function acquires *lock* if $\rho(\mathit{lock})$ is added to the lockset along every path reaching the function exit. Similarly, for `__releases_lock`, we infer that a function releases *lock* if $\rho(\mathit{lock})$ is removed from the lockset along every path reaching the function exit.

4. IMPLEMENTATION

To enable rapid development of custom checkers, we have built a program analysis platform called *the Esp analysis framework*. Using this framework, we have implemented our techniques in several tools, including EspC (a concurrency checker and an annotation inference tool), NullPtr (a NULL pointer dereference checker), and EspA (a race and atomicity checker).

4.1 Esp Analysis Framework

The architecture of the Esp platform is illustrated in Figure 4. The framework consists of a common intermediate representation (IR) layer based on Control Flow Graphs (CFGs), as well as a rich layer of analysis components. With a simple interface, the platform provides an easy-to-use intra-procedural, path-sensitive dataflow engine with integrated alias analysis (a flow-insensitive, field-sensitive Andersen-style alias analysis [3, 4]), build integration, extensive annotation handling, automatic defect rendering, and refutation of infeasible paths [5]. The platform appeals to domain experts by allowing them to focus almost exclusively on the domain-specific aspects when developing a custom checker. The Esp framework has been applied to power many widely-used analysis tools at Microsoft.

4.2 EspC

EspC is an intra-procedural, path-sensitive lock analyzer, analyzing one function at a time, and between different functions via function preconditions and postconditions expressed as annotations. It computes locksets at every program points and checks against concurrency violations. A function assumes that all of its preconditions hold at entry, and must ensure that all of its postconditions hold at exit. At a call site, the caller must ensure that all the callee's preconditions hold before the callee is called, and assumes that the callee's postconditions hold when the callee returns.

We define all Win32 locking APIs using our pattern matching specification language called OPAL. This makes EspC understand the intrinsic locking behavior of a program even without explicit source annotations. We then instrument the CFGs by adding the matched locking event nodes. This allows the analysis engine to track those specified APIs. When a particular code pattern, say `EnterCriticalSection(&cs)`, is encountered, a special event node is injected into the CFG. With such information, EspC implements the lockset algorithm in Section 2.2.2 by computing locks acquired at every program point.

To improve scalability, EspC employs a *selective merge* algorithm [6]: At a merge point, we evaluate the locksets from incoming paths—if the locksets are the same, we merge the incoming states; otherwise, we keep them separate and propagate them precisely. This allows our tool to gain path-sensitivity while avoiding the exponential blow up.

We also apply many algorithms to optimize the performance. For example, if a function does not contain Concurrency SAL annotations and locking operations, the analysis phase is skipped. In addition, the symbolic path simulation is only turned on on-demand when a potential error is detected to ensure the path is feasible.

EspC uses an extensive set of algorithms to bucket locking anomalies into different warning categories with a corresponding confidence level. For instance, a race warning from a function that does not use locking at all is less likely to be a real bug compared with a race warning from a function that uses extensive locking. This bucketing mechanism allows a user to apply EspC either as a precise bug-finding tool or as a comprehensive validation tool by selectively monitoring at different warning levels.

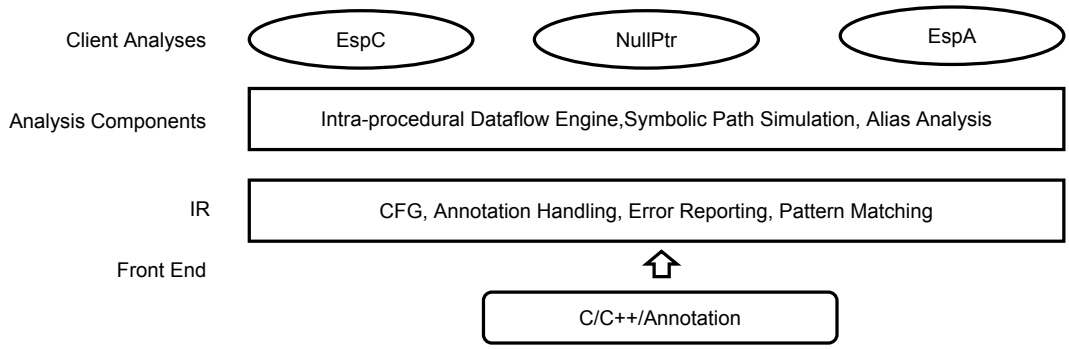


Figure 4: The Esp analysis framework.

In addition to detecting concurrency errors such as race conditions and locking mismatch errors, EspC can also find missing concurrency annotations using the inference algorithms outlined in Section 3. The inferred annotations can be automatically patched to the source code.

4.3 NullPtr

NullPtr is an intra-procedural and path-sensitive NULL pointer dereference checker. This tool tracks pointer values according to the state machine in Figure 2(a). It understands ordinary (non-concurrency) annotations as well as the behavior of certain un-annotated system and library APIs. For the purpose of this paper, the brevity and domain-specificity of NullPtr makes it an ideal basis of a typestate-guided concurrency analysis.

4.4 EspA

EspA is a NullPtr-based race and atomicity checker. It tracks the state machine in Figure 2(b). The typestate-guided concurrency analysis using EspA goes through a staged process.

1. Run EspC with the inference mode turned on to infer concurrency annotations.
2. Run NullPtr to generate `MustNULL` warnings.
3. Run EspA to generate `MaybeNULL` warnings.
4. Make a "diff" between EspA warnings and NullPtr warnings, and output those extra warnings produced by the EspA run but not by the NullPtr run.

To illustrate, we now walk through these steps for the example in Figure 1. We first run EspC on the un-annotated code with the inference mode turned on. Using the algorithm described in Section 3, EspC will infer the missing `__guarded_by` annotation and patch it to the source code, resulting in the modified code shown in Figure 3. (At this stage, EspC can be optionally run again to uncover all potential race conditions and locking mismatch errors, but it cannot reveal atomicity issues.) We then run NullPtr, followed by EspA. Warnings generated by EspA but not by NullPtr, if any, consists of the atomicity bugs and a refined set of race conditions. In the case of Figure 1, a warning at line 10 in function `ProcessBuffer` will be reported. All these steps are automated by a scripting command utility.

5. RESULTS

Concurrency SAL and EspC have been deployed for the Windows division at Microsoft. We evaluated EspC across the whole Windows code base. We ran the typestate-guided concurrency analysis with EspA on several networking subcomponents, and drilled down into the results with a process of manually examining every warning to collect detailed data.

The Windows code base consists of around 60 million lines of code and spans across 6000 binaries (each binary is a DLL, EXE, or SYS component). Our experiment was carried out on an 8-core machine with a 2.66GHz Xeon CPU and 8GB of RAM, running Windows Server 2003 Enterprise x64 Edition. The analysis was spawned on 8 processes. All confirmed bugs have been or will be fixed for the production code.

5.1 Lock Analysis with EspC

We ran EspC across Windows to test the performance, precision, and scalability of our lockset analysis. The original code base is un-annotated. For the experimental run reported here, we have turned on the detection mode for "failing to release critical section", "releasing un-held critical section", as well as the inference mode for finding lock wrapper functions.

Problem Description. In Win32 programs, critical sections are light-weight synchronization objects for achieving mutual exclusion among threads from the same process. If a thread fails to release a critical section, the lock becomes "orphaned". Unfortunately, the operating system does not provide any mechanism to recover an orphaned critical section. Therefore, this is a bug pattern that may lead to deadlocks.

Performance. The checking phase of EspC completed in 12 hours and 30 minutes for the whole operating system, comparable to the time it took to build the system. With the build integration capability, EspC can be run on the background while the programs are being compiled. This allows developers to find concurrency errors right away while developing their code.

Precision. A total number of 6813 warnings were issued, including 942 warnings for "failing to release critical section", 472 warnings for "releasing un-held critical section", and 5399 instances for lock wrapper functions that need to be annotated with `__acquires_lock` and `__releases_lock`.

Component	Number of Files	Time (min)	Total Warnings	Bugs	False Positives	False Positive Rate
A	7415	23:25	92	65	27	29%
B	5564	32:54	35	18	17	49%
C	2196	14:21	5	4	1	20%
D	3920	24:24	9	7	2	22%
E	3488	14:56	16	11	5	31%
F	2968	12:33	9	5	4	44%
G	10361	35:12	34	30	4	12%
H	1496	6:45	63	46	17	27%

Figure 5: Experimental results of EspC. Bugs include "failing to release critical section" and "releasing un-held critical section".

```
1  ACQUIRE_SPIN_LOCK(&ClientListLock, &OldIrql);
2  ClientBlockLink = ClientList.Flink;
3  while (ClientBlockLink != &ClientList)
4  {
5    // Do something
6    RELEASE_SPIN_LOCK(&ClientListLock, OldIrql);
7    // Do something
8    ACQUIRE_SPIN_LOCK(&ClientListLock, &OldIrql);
9    ClientBlockLink = ClientBlockLink->Flink;
10 }
11 RELEASE_SPIN_LOCK(&ClientListLock, OldIrql);
```

Figure 6: An example of atomicity vulnerability. The assumed loop invariant at line 3 could be violated at line 7 due to thread interference.

The wrapper annotation inference is highly accurate. For the bug warnings, we reviewed many components with the corresponding product teams. Table 5 summarizes the results for 8 Windows components. Among these components, there are 186 confirmed bugs and 77 false positives. The confirmed bugs usually reside in error paths. There is a common scenario that introduces false positives: A flag in a structure is sometimes used to record whether a lock in the same structure has been acquired and the lock acquisition status is updated by some high-level design logic beyond the comprehension of EspC. We are considering to support conditional field annotations to express such structure-level invariants.

5.2 Atomicity Analysis with EspA

We have run the typestate-guided concurrency analysis with EspA on several networking subcomponents, following the process described in Section 4.4.

Performance. The performance statistics is summarized in Table 7, with a break down among EspC inference time, NullPtr checking time, and EspA checking time. For the largest subcomponent in this experiment (subcomponent c with 839K LOC), the whole analysis took less than 25 minutes. If annotation inference can be omitted, *i.e.*, if the annotations have been previously added, the process would have taken less than 6 minutes.

Precision. The number of inferred `__guarded_by` annotations, race warnings, and atomicity warnings are summarized in Table 8. The "Total Difference" column lists the additional warnings from EspA. Out of the 99 warnings, 45 are harmful bugs, 39 are benign warnings, and 15 are false positives.

Bug Example. Figure 6 illustrates a simplified bug example for an atomicity violation. In this code, the `while` loop tests the loop invariant at line 3. However, when the guarding lock is released at line 6, another thread could update the shared data, thus causing an access violation at line 9.

To illustrate how the typestate-driven method can help distinguish harmful races, consider a common coding pattern in Win32 programs called double-checked-locking[1], as illustrated below:

```
1   void AccessBuffer(DATA* p) {
2       if (p->buffer == NULL) {
3           return;
4       }
5       EnterCriticalSection(&p->cs);
6       if (p->buffer != NULL) {
7           cout << *p->buffer << endl;
8       }
9       LeaveCriticalSection(&p->cs);
10  }
```

The program tries to minimize blocking by only acquiring a lock when necessary. It is critical to double-check the null-ness of the buffer at line 6 because even though it is not `NULL` at line 2, it could have been reset to `NULL` by another thread by the time the lock is acquired. In this scenario, the race at line 2 is intentional and benign. However, if the double-check is missing at line 6, a real race bug would arise. A standard race detector, *e.g.*, EspC, would report the warning at line 2; a standard NULL dereference checker, *e.g.*, NullPtr, would miss the real race when the double-check is omitted. The combined approach, on the other hand, covers the sweet spot by only identifying the harmful data race. In one case, we found such a code fragment where the second NULL check is wrapped with a `#ifdef DEBUG` macro, suggesting that the programmer has been trying to debug a related crash. Ironically, the bug cannot be reproduced in the debug build but would slip into the release build.

In summary, our experiments have validated the following hypotheses: (1) Our approach is scalable and precise enough to produce useful results. (2) Annotation inference can reduce, or even eliminate, the human annotation effort. (3)

[1]The double-checked-locking programming idiom is unsafe for certain relaxed architectures such as Alpha, but is fine for X86.

Sub Component	Files	Functions	LOC	Inference Time (min)	NullPtr Time (min)	EspA Time (min)
a	60	3066	179303	6.65	5.18	5.50
b	163	17744	163088	10.92	3.65	3.78
c	1323	16860	839465	18.70	2.93	2.92
d	78	773	82779	1.38	0.60	0.83
e	446	8448	411908	15.25	3.63	4.00
f	215	5070	223885	7.50	2.17	2.75

Figure 7: Performance statistics for the typestate-guided concurrency analysis.

Sub Component	Inferred `__guarded_by`	NullPtr Warnings	EspA Warnings	Total Difference	Data Race	Atomicity Violation	Harmful Bugs	Benign Warnings	False Positives
a	139	89	108	19	1	18	5	9	5
b	8	11	22	11	5	6	10	1	0
c	4	8	20	12	11	1	7	2	3
d	16	6	11	5	5	0	5	0	0
e	84	21	37	16	10	6	8	5	3
f	30	16	52	36	23	13	10	22	4

Figure 8: Warning statistics for the typestate-guided concurrency analysis.

The typestate-guided method is effective in detecting hard-to-find semantic errors.

6. RELATED WORK

In this section, we discuss related work spanning across many fields.

Typestate Checking. Typestate systems [1, 7, 6, 8] track the states each object goes through during its lifetime. Typestate has also been extended to *roles* [9]: The role of an object captures its typestate as well as its involvement in aliasing relationships. Our approach can be viewed as an extended typestate analysis that is thread-sensitive.

Dynamic Race Detection. Data races can be found using dynamic tools, *e.g.*, Eraser [10], Racetrack [11], Locksmith [12], and iDNA [13]. Static and dynamic approaches are complementary, with some well-known tradeoffs, *e.g.*, static analysis generally provides better coverage and dynamic analysis tends to offer more precision. The requirement of supporting DLLs and device drivers has motivated us to apply static dataflow analysis, which is more favorable for analyzing open programs.

Static Race Detection. A variety of static techniques have been driven by the need for analyzing Java, which offers built-in support for threads with syntactically scoped locks. Rccjava [14, 15] detects data races based on a race-free type system. *Ownership* [16] is supported to make the type systems more expressive. Chord [17] detects races by pruning the set of memory access pairs via a staged analysis. Based on that, the notion of *conditional must not alias* [18] is proposed: A race occurs if two memory locations are aliased when the two guarding locks are not aliased. There is also a large body of work on race detection for C/C++ programs. Early works include Warlock [19] and RacerX [20]. These analyses are not path-sensitive. As a result, much effort has to be spent in filtering the results, and only a small number of bugs have been found. In comparison, the unstructured usage of locking in C/C++ programs prompts us to employ a path-sensitive and flow-sensitive analysis. This enables our tools to discover a large number of concurrency bugs with a favorable false positive rate. A race-free type system is developed for Cyclone [21]. Model checking techniques are used in Blast [22] and Kiss [23]. In a recent line of work, Locksmith [12] conducts a correlation analysis using a constraint-based technique. RELAY [24] uses a bottom up approach in finding races. RADAR [25], implemented based on RELAY, is a framework that automatically converts a dataflow analysis for sequential programs into one for concurrent programs. While sharing a similar spirit, our approach applies lock inference techniques for refining race detection and finding atomicity violations. As far as we know, our toolset is the first that combines annotation inference, patching, and checking in an industrial setting, with the capability of finding atomicity errors.

Atomicity Analysis. Flanagan and Qadeer [26] develop a type system to enforce atomicity based on Lipton's theory of reduction [27]. Atomizer [28] is a dynamic atomicity checker that uses simple heuristic to determine atomic blocks. Another type system for atomicity is developed by combining a more expressive type system for analyzing data races [29]. A dynamic atomicity checker is described in [30] for finding atomicity violations. VYRD [31] applies a runtime technique for checking conformance to atomicity between specifications and implementations. Compared with type-based atomicity tools, our typestate-based approach is less restrictive, thus more precise. One way to view our strategy is that we give up the enforcement on the strict atomicity requirement for all program side effects and instead focus on a semantically defined atomicity requirement treating typestates as the observable behavior. Our approach has the practical benefit of being able to pinpoint harmful atomicity bugs.

Correlation Inference. The SVD tool [32] develop a dynamic technique that uses heuristics to infer computation unit (atomic regions) based on data and control flow. It reports bugs when interleavings with unserializable writes are detected. AVIO [33] extracts *Access-Interleaving* invariant to infer atomicity intension and detects violations at runtime. MUVI [34] detects inconsistency via pattern analysis on multi-variable access correlations. JNuke [35] detects stale-value concurrency errors. Vaziri *et al.* [36] present a new definition of data races in terms of 11 problematic interleaving scenarios, and introduces the notion of *atomic sets of locations* to let programmers specify the existence of consistency properties between fields in objects. Our bundle-based variable correlation inference draws insights from these efforts by sharing the *data-centric* approach, *i.e.*, we concentrate on finding the bundling relation between variables. However, our technique is directly based on the locking evidence exhibited in code paths.

New Concurrency Model. Researchers have explored using optimistic concurrency [37, 38], or software transactions, as a means to implement concurrency control. However, development and adoption of a new programming paradigm takes time and such research efforts do not directly solve legacy threading issues, which is the main problem addressed by this paper. Furthermore, atomicity analysis is still critical with the new programming model since atomicity errors may still occur if the programmer fails to properly define a transaction. A recent line of work [39, 40, 41] explores lock allocation techniques to support pessimistic concurrency. It will be interesting to explore if our method for finding vulnerable atomic regions can be used as an inference technique to serve lock allocation purposes.

7. CONCLUSIONS

We have presented a new typestate-guided approach to concurrency analysis. In particular, we have shown how to combine a NULL dereference checker with lockset analysis and demonstrated its effectiveness in detecting race conditions and atomicity vulnerabilities. In the future, we plan to integrate our techniques with a generic typestate checker such as ESP [6] so that any user-specified typestate property can be used to drive the combined analysis.

8. ACKNOWLEDGMENTS

We wish to thank Brian Hackett, whose insightful ideas have been inspirational for this work. We are deeply grateful to Stephen Adams, Zhe Yang, Daniel Wang, Vikram Dhaneshwar, and Manuvir Das, who have made major contributions to our analysis infrastructures. We also thank Frances Perry and Aquinas Hobor for their work in developing EspC prototypes, and Francesco Faggioli for his support in several pilot projects.

9. REFERENCES

[1] Robert E. Strom and Shaula Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Transactions on Software Engineering*, 12(1):157–171, 1986.

[2] Sal Annotations. http://msdn2.microsoft.com/en-us/library/ms235402.aspx.

[3] David J. Pearce, Paul H. J. Kelly, and Chris Hankin. Efficient field-sensitive pointer analysis for C. In *PASTE '04: Proceedings of the 5th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, pages 37–42, 2004.

[4] Suan Hsi Yong, Susan Horwitz, and Thomas Reps. Pointer analysis for programs with structures and casting. In *Proceedings of the 1999 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 91–103, 1999.

[5] Hari Hampapuram, Yue Yang, and Manuvir Das. Symbolic path simulation in path-sensitive dataflow analysis. In *PASTE '05: Proceedings of the 6th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, pages 52–58, 2005.

[6] Manuvir Das, Sorin Lerner, and Mark Seigle. ESP: Path-sensitive program verification in polynomial time. In *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 57–68, 2002.

[7] Robert DeLine and Manuel Fahndrich. Enforcing high-level protocols in low-level software. In *Proceedings of the 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2001.

[8] J. Field, D. Goyal, G. Ramalingam, and E. Yahav. Typestate verification: Abstraction techniques and complexity results. In *Proceedings of the 10th International Static Analysis Symposium*, 2003.

[9] Viktor Kuncak, Patrick Lam, and Martin Rinard. Role analysis. In *the ACM Symposium on Principles of Programming Languages*, 2002.

[10] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: a dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems (TOCS)*, 15(4):391–411, 1997.

[11] Yuan Yu, Tom Rodeheffer, and Wei Chen. Racetrack: efficient detection of data race conditions via adaptive tracking. In *SOSP*, pages 221–234, 2005.

[12] Polyvios Pratikakis, Jeffrey S. Foster, and Michael W. Hicks. Locksmith: context-sensitive correlation analysis for race detection. In *Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 320–331, 2006.

[13] Satish Narayanasamy, Zhenghao Wang, Jordan Tigani, Andrew Edwards, and Brad Calder. Automatically classifying benign and harmful data races using replay analysis. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2007.

[14] Cormac Flanagan and Martin Abadi. Types for safe locking. In *ESOP*, 1999.

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

[16] Chandrasekhar Boyapati, Robert Lee, and Martin Rinard. Ownership types for safe programming: preventing data races and deadlocks. In *Proceedings of*

the 17th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, 2002.

[17] Mayur Naik, Alex Aiken, and John Whaley. Effective static race detection for Java. In *Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2006.

[18] Mayur Naik and Alex Aiken. Conditional must not aliasing for static race detection. In *the ACM Symposium on Principles of Programming Languages*, pages 327–338, 2007.

[19] Nicolas Sterling. Warlock: A static data race analysis tool. *USENIX Winter Technical Conference*, 1993.

[20] Dawson Engler and Ken Ashcraft. RacerX: effective, static detection of race conditions and deadlocks. In *Proceedings of the 9th ACM Symposium on Operating Systems Principles*, 2003.

[21] Dan Grossman. Type-safe multithreading in cyclone. *TLDI*, pages 13–25, 2003.

[22] Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. Race checking by context inference. In *Proceedings of the 2004 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 1–13, 2004.

[23] Shaz Qadeer and Dinghao Wu. KISS: Keep it simple and sequential. In *Proceedings of the 2004 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 14–24, 2004.

[24] Jan Wen Voung, Ranjit Jhala, and Sorin Lerner. RELAY: static race detection on millions of lines of code. In *Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 205–214, 2007.

[25] Ravi Chugh, Jan W. Voung, Ranjit Jhala, and Sorin Lerner. Dataflow analysis for concurrent programs using datarace detection. In *Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2008.

[26] Cormac Flanagan and Shaz Qadeer. A type and effect system for atomicity. In *Proceedings of the 2003 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 256–267, 2003.

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

[28] Cormac Flanagan and Stephen Freund. Atomizer: a dynamic atomicity checker for multithreaded programs. In *the ACM Symposium on Principles of Programming Languages*, 2003.

[29] Amit Sasturkar, Rahul Agarwal, Liqiang Wang, and Scott D. Stoller. Automated type-based analysis of data races and atomicity. In *Proceedings of the 10th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2005.

[30] Liqiang Wang and Scott D. Stoller. Runtime analysis of atomicity for multithreaded programs. *IEEE Transactions on Software Engineering*, 32(2):93–110, 2006.

[31] Tayfun Elmas, Serdar Tasiran, and Shaz Qadeer. VYRD: verifying concurrent programs by runtime refinement-violation detection. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 27–37, 2005.

[32] Min Xu, Rastislav Bodík, and Mark D. Hill. A serializability violation detector for shared-memory server programs. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2005.

[33] Shan Lu, Joseph Tucek, Feng Qin, and Yuanyuan Zhou. AVIO: detecting atomicity violations via access interleaving invariants. *ASPLOS*, pages 37–48, 2006.

[34] Shan Lu, Soyeon Park, Chongfeng Hu, Xiao Ma, Weihang Jiang, Zhenmin Li, Raluca A. Popa, and Yuanyuan Zhou. MUVI: automatically inferring multi-variable access correlations and detecting related semantic and concurrency bugs. In *SOSP*, pages 103–116, 2007.

[35] Cyrille Artho, Klaus Havelund, and Armin Biere. Using block-local atomicity to detect stale-value concurrency errors. *ATVA*, pages 150–164, 2004.

[36] Mandana Vaziri, Frank Tip, and Julian Dolby. Associating synchronization constraints with data in an object-oriented language. In *the ACM Symposium on Principles of Programming Languages*, pages 334–345, 2006.

[37] Tim Harris and Keir Fraser. Language support for lightweight transactions. In *Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, and Applications*, 2003.

[38] Michael F. Ringenburg and Dan Grossman. Atomcaml: First-class atomicity via rollback. *ICFP*, 2005.

[39] Michael Hicks, Jeffrey S. Foster, and Polyvios Pratikakis. Inferring locking for atomic sections. *The ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing (TRANSACT)*, 2006.

[40] Bill McCloskey, Feng Zhou, David Gay, and Eric Brewer. Autolocker: Synchronization inference for atomic sections. In *the ACM Symposium on Principles of Programming Languages*, pages 346–358, 2006.

[41] Michael Emmi, Jeffrey S. Fischer, Ranjit Jhala, and Rupak Majumdar. Lock allocation. In *the ACM Symposium on Principles of Programming Languages*, pages 291–296, 2007.