Electronic Communications of the EASST
Volume 72 (2015)

Proceedings of the
15th International Workshop on
Automated Verification of Critical Systems (AVoCS 2015)

ThreadSafe: Static Analysis for Java Concurrency

Robert Atkey and Donald Sannella

15 pages

Guest Editors: Gudmund Grov, Andrew Ireland
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

ThreadSafe: Static Analysis for Java Concurrency

Robert Atkey[1,3] **and Donald Sannella**[2,3]

[1] robert.atkey@strath.ac.uk, http://bentnib.org/
Computer & Information Sciences, University of Strathclyde
Glasgow, United Kingdom

[2] dts@inf.ed.ac.uk, http://homepages.inf.ed.ac.uk/dts/
School of Informatics, University of Edinburgh
Edinburgh, United Kingdom

[3] http://www.contemplateltd.com/
Contemplate Ltd
Edinburgh, United Kingdom

Abstract: ThreadSafe is a commercial static analysis tool that focuses on detection of Java concurrency defects. ThreadSafe's bug-finding capabilities and its look and feel are presented through examples of bugs found in the codebases of two widely-used open source projects.

Keywords: static analysis, concurrency, Java

1 Introduction

It is widely acknowledged that developing reliable concurrent software is very difficult [Goe06, Lee06, HS12]. Concurrency-related defects like data races and deadlocks can be subtle and hard to understand. The fact that concurrent software is inherently non-deterministic means that reliance on testing for software quality assurance is inappropriate and dangerous; intermittent bugs that show up once in a million runs are not uncommon. At the same time, requirements for improved performance force the increased use of application-level concurrency in order to exploit multicore hardware.

Use of static analysis to discover and diagnose concurrency defects during software development is a cost-effective solution to this problem. Static analysis can take all possible execution paths into consideration, not just the ones that are explored by specific test data for a specific scheduling of threads. Static analysis can be applied to a codebase while it is still under construction, long before testing is possible. Bugs that are detected early in the development process are easier and cheaper to fix.

ThreadSafe is a commercial static analysis tool that focuses on detection of Java concurrency defects. By focusing on concurrency bugs, ThreadSafe can find bugs that other static analysis tools, both commercial and freely available, miss or are not designed to look for.

ThreadSafe is fully automatic. It requires no annotations to be added to codebases, but if `@GuardedBy` annotations [Goe06] are present then they are checked for accuracy. Findings are generated from information produced by a class-by-class flow-sensitive, path-sensitive, context-sensitive points-to and lock analysis. Heuristics are used to tune the analysis to avoid false positives. Knowledge of concurrency aspects of the Java library and of some aspects of frameworks including Android has been built in. ThreadSafe has been successfully applied to codebases of more than a million lines. It is being used in a companies across a wide range of industry sectors and in university teaching, for example [Ses14].

In the following sections we give an outline of how ThreadSafe works and present some examples of bugs that it finds in two widely-used open source projects. We then present some of our experiences with developing, using, and observing industrial developers using ThreadSafe, discuss how its performance can be measured, and give a comparison of its performance against FindBugs on a Java concurrency benchmark.

2 How ThreadSafe operates

ThreadSafe analyses `.class` files containing JVM bytecode for evidence of concurrency errors, which are reported back to the user. In theory, because ThreadSafe analyses JVM bytecode, it could be used to analyse the output of any compiler that targets the JVM (for example, the Scala compiler). However, due to the need for heuristics that encode assumptions about the nature of the code being analysed, and built-in knowledge about the compilers, idioms, libraries and frameworks that are common when programming with Java as the source language, ThreadSafe only really operates effectively on the output of a Java compiler.

The operation of ThreadSafe can be thought of as a kind of compiler for JVM bytecode, only instead of producing optimised machine code ThreadSafe produces static analysis warnings. The front-end is the same: JVM `.class` files are read in; translated into a simpler form for analysis; and analysed. Instead of using the results of analysis to generate and optimise bytecode, *checkers* are run over the analysis results to find evidence of problematic code. Below, we explain in more detail the three main stages of ThreadSafe's operation, and discuss the trade-offs that have been made.

Although ThreadSafe is primarily targeted at finding concurrency flaws, there is nothing concurrency focused about the analysis it performs (other than explicit tracking of lock acquisitions) before the checkers are run. It would be relatively straightforward to write new checkers that discover defects such as resource handling errors or information leakage errors on top of the existing analysis.

2.1 Class loading and preprocessing

In order to be scalable to very large code bases, ThreadSafe analyses `.class` files on a by-package basis, only referring to `.class` files in other packages as needed to fill in details in the inheritance hierarchy, or to analyse inherited method bodies. Class files are parsed lazily, as required by the analysis described below, and the bytecode in the body of each method is preprocessed in two phases:

1. The JVM's native stack-based bytecode is translated into a register-based format. This translation enables the interprocedural dataflow analysis to be more efficient, as the local variables in each method can be modelled as a flat array of abstract values, rather than a

stack that changes size dynamically. The translation also makes all control flow in each method body explicit, turning the flat arrays of bytecode instructions into an explicit control flow graph, with one instruction per node. Intra-method subroutines, implemented by the JVM's `jsr` and `ret` instructions, are inlined. The `jsr` and `ret` instructions complicate analysis, since they result in the same segment of bytecode being invoked in several contexts within the same method. Since recursive subroutines via `jsr` and `ret` are not possible, it is safe to inline bytecode subroutines into their callsites.

2. Accessor methods are inlined into their callers. Accessors methods are methods generated by the Java compiler that allow inner (or outer) classes to access private fields and methods in their outer (resp. inner) classes. The access control enforced by the JVM does not allow any class's methods to access another class's private fields or methods (the JVM knows nothing about inner or outer classes), so the Java compiler generates a public "synthetic method" that provides direct access to the underlying private field or method. This results in the actual field or method access appearing in a different bytecode method to the one that appears in the source code.

   If a static analysis author is not careful, the presence of synthetic compiler-generated methods can lead to confusing reports from a static analyser that naively reports which method a problematic field access appears in. Moreover, while it would be possible to just let the normal interprocedural analysis handle accessor methods, it is more efficient to preemptively inline the methods into their callers.

   Therefore, THREADSAFE inlines all accessor methods into their callers. This process is not entirely straightforward, because the compilation scheme for accessor methods is not specified by the Java Language Specification, and different compilers have slightly different strategies. THREADSAFE handles the schemes used by the standard OpenJDK `javac` compiler, and Eclipse's `ecj` compiler.

2.2 Interprocedural per-class analysis

After translation, a context-sensitive interprocedural points-to and lock analysis is run on each public entry point of each instantiable class. The idea is that each class is a self-contained entity whose instances can be invoked arbitrarily via its public entry points. A method is deemed to be a "public entry point" if it is either declared as `public`, or has been specially marked for the analysis because it will be called by some framework or library method. An example of a non-public "public entry point" is the protected `doInBackground` method from the Android `AsyncTask` class, which is invoked by the framework code as an asynchronous background task (see Section 3.3 for an example of misuse of this API that THREADSAFE catches).

This analysis is interprocedural, but to keep the overall analysis scalable, only calls to private and protected methods on the same class are followed. All other calls are treated symbolically. This again follows the idea that each public method of a class is special in that it represents how classes are used by other classes, while private and protected methods are part of a class's implementation. A disadvantage of this approach is that it relies heavily on the programmer having good taste when it comes to assigning access qualifiers to methods. A class whose methods are all public when they could be declared as `private` or `protected` will act the same during execution, but will produce very different analysis results. A possible mitigation, which we have not yet attempted, is to run a whole-program analysis to infer a tightest possible access qualifier for each method, ignoring the programmer's annotations.

Given a set of entry points, the interprocedural analysis computes, for each object-manipulating instruction, an approximation of the objects that will be accessed and the set of locks that are held when that instruction is executed. Objects (whether locks or directly manipulated) are represented as paths relative to one of: the symbolic `this` reference pointing to the instance of the class currently under analysis; the parameters of the public entry point method being analysed; or to references acquired from globally accessible static fields and methods.

The abstract lock sets computed for each instruction consist of representations of monitors, i.e., the JVM's intrinsic lock associated with every object, and first-class locks as represented by implementations of the `java.util.concurrent.locks.Lock` interface. Monitors are slightly simpler to model because they must strictly nest with respect to method calls, so it is easier to accurately track whether or not they are held through recursive method calls.

The points-to and lock analysis are intentionally unsound in their handling of aliasing and mutation. For example, the sequence of code `this.f.lock(); ... this.f.unlock();` is assumed to lock and unlock the same object, even if the field `f` could be updated in between. A later checker checks to see whether fields containing locks are ever mutated while the corresponding lock is held, because this is often a mistake by the programmer.

Once the analysis of each public entry point of each instantiable class is complete, some general summary information is collected: fields that contain collection objects are identified, and classified according to whether the collection object is known to be suitable for concurrent use or not. For example, the concurrent collections from the `java.util.concurrent` package are known to be safe for concurrent use (though see the description of the Get-Check-Put checker below), while instances of a collection class like `java.util.ArrayList` are definitely known to be unsafe if accessed concurrently without synchronisation.

2.3 Checkers

Finally, to produce the analysis findings that are presented to the user, *checkers* are run over the information gathered by the interprocedural analysis. THREADSAFE contains over a dozen checkers that look for specific instances of wrong or problematic code that uses concurrency. Two of the checkers are the *Inconsistent Synchronisation* checker and the *Get-Check-Put* checker.

Inconsistent synchronisation The Inconsistent Synchronisation checker examines, for each field, all the accesses to that field, and determines if they all share a common lock. A warning is produced if either: all accesses are locked, but do not share a common lock; or some accesses are locked and some are not, with the ratio of locked/total being over some threshold.

The inconsistent synchronisation analysis was originally based on the inconsistent synchronisation analysis in the FindBugs tool, but has been extended to have a more precise determination of which locks are held, and to handle accesses to thread unsafe collections stored in fields, as well as direct accesses to fields. In particular, THREADSAFE's analysis can handle cases where all accesses are locked, but with different locks.

Get-Check-Put The Get-Check-Put checker looks for sequences of method calls on an instance of a concurrent collection, e.g., a `ConcurrentHashMap`, that check the status of the collection, and then mutate the collection based on the result of the check. If other threads are not prevented from mutating the collection between the check and the mutation, then there is a potential race condition, because the information gathered from the check will have become outdated. The name "Get-Check-Put" comes from the common pattern where a `.get(key)` method is invoked to find out whether `key` is in the map, the result is compared to `null` (the "check"), and if so, the `.put` method is used to associate `key` with a new value in the map. Such a pattern is often used to maintain caches of objects that are expensive to create.

Instances of Get-Check-Put are detected by performing another interprocedural analysis over the results of the first interprocedural analysis, tracking the evolution of method invocations on each collection object using a typestate-style analysis.

3 Examples

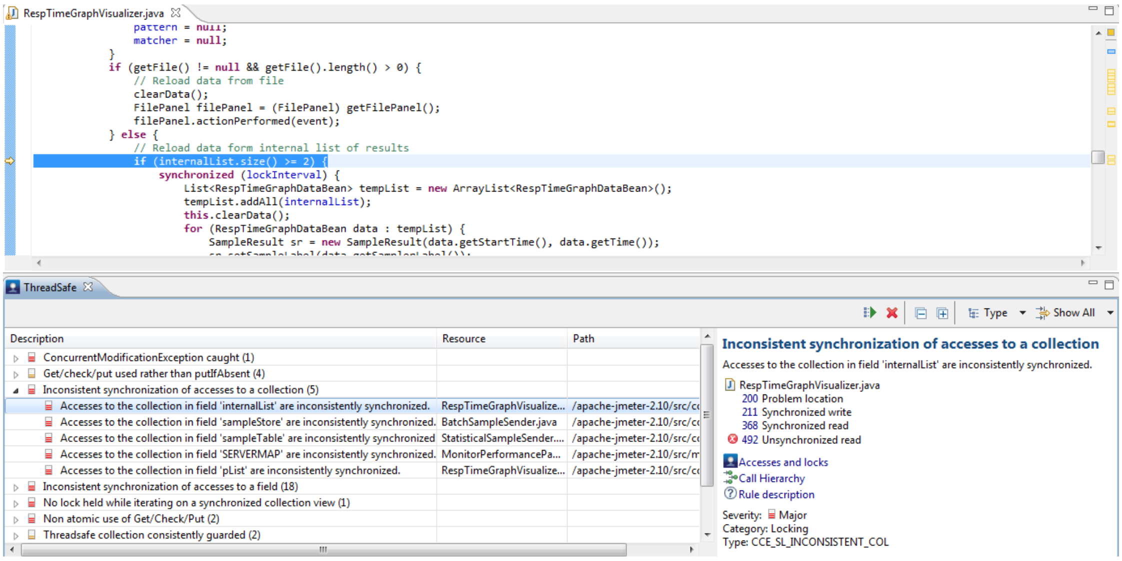
The following sections give a few examples of bugs and potential bugs that THREADSAFE finds in open source codebases. As far as we are aware, none of these bugs is found by any other static analysis tool. Our aim in presenting these examples is to give an impression of the kinds of bugs that THREADSAFE is able to find as well as a feeling for the overall look and feel of the tool.

3.1 Example: incorrect synchronization of collection accesses

Java provides a rich assortment of collection classes, each with its own requirements on whether or not synchronization is required for concurrent access to the collection.

Inconsistent synchronization on collections can be particularly harmful to program behaviour. While incorrectly synchronizing accesses to a field may "only" result in missed updates or stale information, incorrectly synchronizing accesses to collections that have not been designed for concurrent use can lead to violations of the collections' internal invariants. This may not immediately cause visible effects, but may cause odd behaviour, including infinite loops or corrupted data, at a later point in the program's execution. See [Tym09, Ora12] for an example.

An example of inconsistent use of synchronization when accessing a shared collection is present in version 2.10 of Apache JMeter, an open source tool for testing application performance under load. THREADSAFE produces 44 findings for JMeter, including the one shown below in THREADSAFE's Eclipse plug-in.



RespTimeGraphVisualizer.java

```
                pattern = null;
                matcher = null;
            }
            if (getFile() != null && getFile().length() > 0) {
                // Reload data from file
                clearData();
                FilePanel filePanel = (FilePanel) getFilePanel();
                filePanel.actionPerformed(event);
            } else {
                // Reload data form internal list of results
                if (internalList.size() >= 2) {
                    synchronized (lockInterval) {
                        List<RespTimeGraphDataBean> tempList = new ArrayList<RespTimeGraphDataBean>();
                        tempList.addAll(internalList);
                        this.clearData();
                        for (RespTimeGraphDataBean data : tempList) {
                            SampleResult sr = new SampleResult(data.getStartTime(), data.getTime());
```

ThreadSafe

Description	Resource	Path
▷ ConcurrentModificationException caught (1)		
▷ Get/check/put used rather than putIfAbsent (4)		
◢ Inconsistent synchronization of accesses to a collection (5)		
Accesses to the collection in field 'internalList' are inconsistently synchronized.	RespTimeGraphVisualize...	/apache-jmeter-2.10/src/co
Accesses to the collection in field 'sampleStore' are inconsistently synchronized.	BatchSampleSender.java	/apache-jmeter-2.10/src/co
Accesses to the collection in field 'sampleTable' are inconsistently synchronized	StatisticalSampleSender....	/apache-jmeter-2.10/src/co
Accesses to the collection in field 'SERVERMAP' are inconsistently synchronized.	MonitorPerformancePa...	/apache-jmeter-2.10/src/m
Accesses to the collection in field 'pList' are inconsistently synchronized.	RespTimeGraphVisualize...	/apache-jmeter-2.10/src/co
▷ Inconsistent synchronization of accesses to a field (18)		
▷ No lock held while iterating on a synchronized collection view (1)		
▷ Non atomic use of Get/Check/Put (2)		
▷ Threadsafe collection consistently guarded (2)		

Inconsistent synchronization of accesses to a collection

Accesses to the collection in field 'internalList' are inconsistently synchronized.

RespTimeGraphVisualizer.java
200 Problem location
211 Synchronized write
368 Synchronized read
492 Unsynchronized read

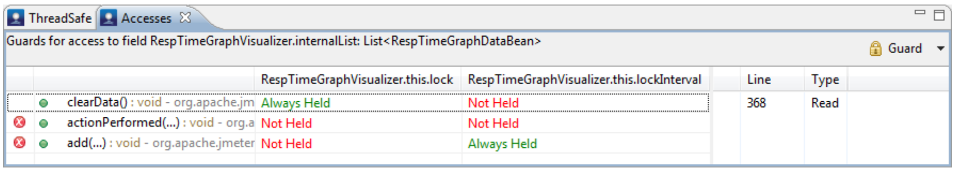
Accesses and locks
Call Hierarchy
Rule description

Severity: Major
Category: Locking
Type: CCE_SL_INCONSISTENT_COL

This finding suggests a possible mistake in synchronizing accesses to the collection stored in the field `RespTimeGraphVisualizer.internalList`.

We can click on the "Problem location" to display the declaration of this field or on any of the locations in the list of accesses. We have clicked above on the location of the unsynchronized read access and the relevant line of code is highlighted. Detailed documentation is available under "Rule description" on the finding, the risks that it raises and their potential severity, and remediation options.

We can ask THREADSAFE to show us the accesses to this field along with the locks that are held, by clicking on "Accesses and locks":

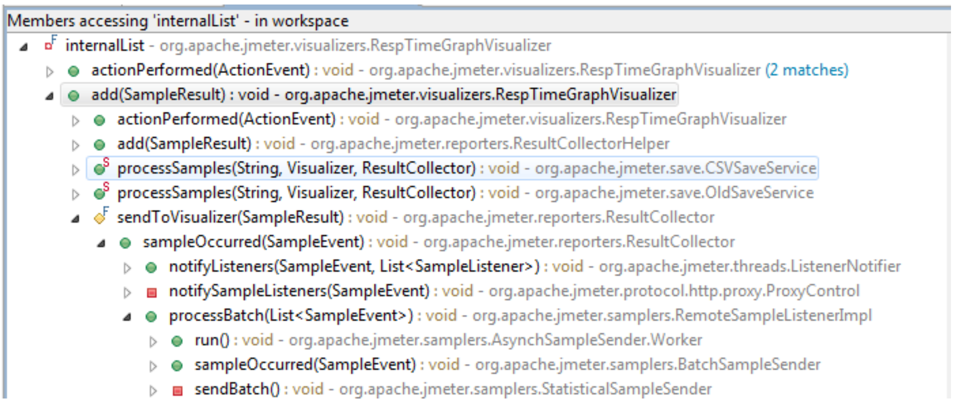


Guards for access to field RespTimeGraphVisualizer.internalList: List<RespTimeGraphDataBean>

	RespTimeGraphVisualizer.this.lock	RespTimeGraphVisualizer.this.lockInterval	Line	Type
clearData() : void - org.apache.jm	Always Held	Not Held	368	Read
actionPerformed(...) : void - org.a	Not Held	Not Held		
add(...) : void - org.apache.jmeter	Not Held	Always Held		

There are three methods that access the collection stored in the field `internalList`. One of these methods is `actionPerformed`, which will be invoked by the Swing GUI framework on the UI thread.

Another method that accesses the collection stored in `internalList` is `add()`. By investigating the possible callers of this method in the call hierarchy, we can see that it is indeed called from the `run()` method of a thread that is not the main UI Thread of the application, indicating that synchronization ought to have been used.

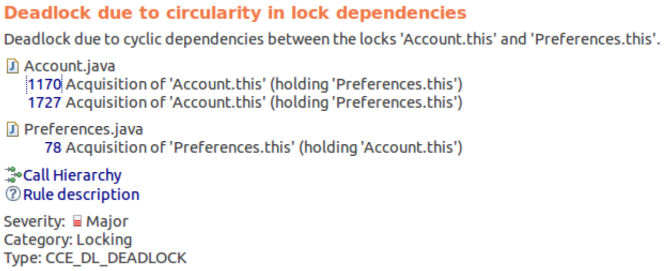


```
Members accessing 'internalList' - in workspace
◢ internalList - org.apache.jmeter.visualizers.RespTimeGraphVisualizer
  ▷ actionPerformed(ActionEvent) : void - org.apache.jmeter.visualizers.RespTimeGraphVisualizer (2 matches)
  ◢ add(SampleResult) : void - org.apache.jmeter.visualizers.RespTimeGraphVisualizer
    ▷ actionPerformed(ActionEvent) : void - org.apache.jmeter.visualizers.RespTimeGraphVisualizer
    ▷ add(SampleResult) : void - org.apache.jmeter.reporters.ResultCollectorHelper
    ▷ processSamples(String, Visualizer, ResultCollector) : void - org.apache.jmeter.save.CSVSaveService
    ▷ processSamples(String, Visualizer, ResultCollector) : void - org.apache.jmeter.save.OldSaveService
    ◢ sendToVisualizer(SampleResult) : void - org.apache.jmeter.reporters.ResultCollector
      ◢ sampleOccurred(SampleEvent) : void - org.apache.jmeter.reporters.ResultCollector
        ▷ notifyListeners(SampleEvent, List<SampleListener>) : void - org.apache.jmeter.threads.ListenerNotifier
        ▷ notifySampleListeners(SampleEvent) : void - org.apache.jmeter.protocol.http.proxy.ProxyControl
        ◢ processBatch(List<SampleEvent>) : void - org.apache.jmeter.samplers.RemoteSampleListenerImpl
          ▷ run() : void - org.apache.jmeter.samplers.AsynchSampleSender.Worker
          ▷ sampleOccurred(SampleEvent) : void - org.apache.jmeter.samplers.BatchSampleSender
          ▷ sendBatch() : void - org.apache.jmeter.samplers.StatisticalSampleSender
```

3.2 Example: potential deadlock

K9Mail is an Android email client, written in Java, that uses concurrent threads to prevent the user interface from becoming unresponsive during communication with the network.

K9Mail consists of over 1000 classes. It would be impractical to go through all of them to look for potential deadlock cycles. Running the ThreadSafe Eclipse plug-in on K9Mail[1] produces the following deadlock warning:



Deadlock due to circularity in lock dependencies

Deadlock due to cyclic dependencies between the locks 'Account.this' and 'Preferences.this'.

```
Account.java
  1170 Acquisition of 'Account.this' (holding 'Preferences.this')
  1727 Acquisition of 'Account.this' (holding 'Preferences.this')
Preferences.java
    78 Acquisition of 'Preferences.this' (holding 'Account.this')
```

Call Hierarchy
Rule description

Severity: Major
Category: Locking
Type: CCE_DL_DEADLOCK

We can see from this report that the intrinsic locks associated with objects of the `Preferences` and `Account` classes form a circular relationship. Investigating this finding in the call hierarchy, we learn that the circularity can arise from the following pieces of code:

1. In the synchronized method `Preferences.getAvailableAccounts()`, there is a call on line 95 to the synchronized method `Account.isEnabled()`.

2. In the synchronized method `Account.save(Preferences)`, there is a call on line 679 to the synchronized method `Preferences.getAccounts()`.

If two threads invoke the `Preferences.getAvailableAccounts()` method and the `Account.save()` method concurrently, then there is a real chance that a deadlock will result.

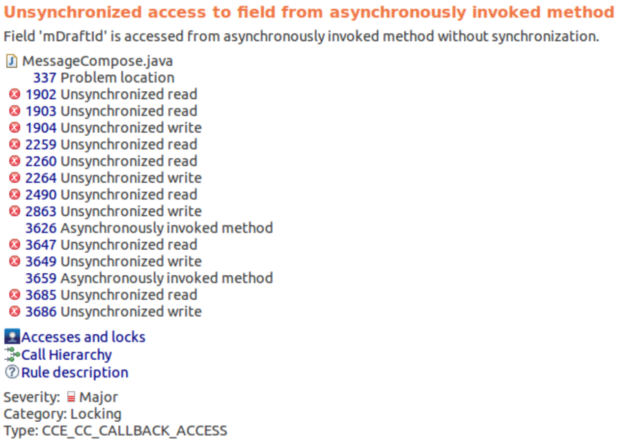
It will take further investigation to determine whether or not this scenario is actually possible, but ThreadSafe has successfully narrowed down the number of lines of code that we have to examine.

[1] The version used here and in Subsection 3.3 was taken from `github.com/k9mail/k-9` and has commit SHA1 id `cc8353d25572b5f1c19047c0c093371f5ac721b4`.

3.3 Example: missing synchronization in Android

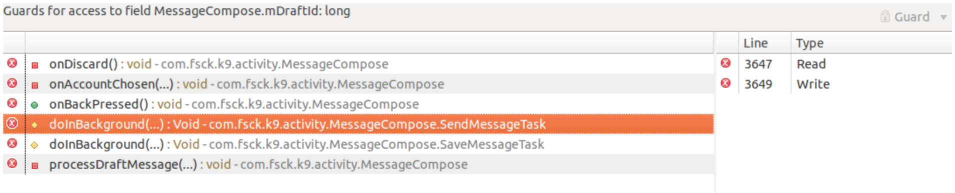
The concurrent environment in which an application may run is almost never under the complete control of the application developer. Frameworks invoke various parts of applications in response to user, network, or other external events, and often have implicit requirements about which methods may be invoked on which threads.

An example of the incorrect use of frameworks is visible in K9Mail. Running THREADSAFE on K9Mail yields the following finding, indicating that the field `mDraftId` appears to be accessed from an Android background thread, and another thread, with no synchronization.



```
Unsynchronized access to field from asynchronously invoked method
Field 'mDraftId' is accessed from asynchronously invoked method without synchronization.
MessageCompose.java
    337 Problem location
  1902 Unsynchronized read
  1903 Unsynchronized read
  1904 Unsynchronized write
  2259 Unsynchronized read
  2260 Unsynchronized read
  2264 Unsynchronized write
  2490 Unsynchronized read
  2863 Unsynchronized write
  3626 Asynchronously invoked method
  3647 Unsynchronized read
  3649 Unsynchronized write
  3659 Asynchronously invoked method
  3685 Unsynchronized read
  3686 Unsynchronized write
Accesses and locks
Call Hierarchy
Rule description
Severity: Major
Category: Locking
Type: CCE_CC_CALLBACK_ACCESS
```

Clicking on “Accesses and locks”, we can see that the field `mDraftId` has been accessed from a `doInBackground` method.



```
Guards for access to field MessageCompose.mDraftId: long                                   Guard

                                                                                      Line   Type
onDiscard() : void - com.fsck.k9.activity.MessageCompose                              3647   Read
onAccountChosen(...) : void - com.fsck.k9.activity.MessageCompose                     3649   Write
onBackPressed() : void - com.fsck.k9.activity.MessageCompose
doInBackground(...) : Void - com.fsck.k9.activity.MessageCompose.SendMessageTask
doInBackground(...) : Void - com.fsck.k9.activity.MessageCompose.SaveMessageTask
processDraftMessage(...) : void - com.fsck.k9.activity.MessageCompose
```

The `doInBackground` method is part of the Android framework’s `AsyncTask` facilities for running time-consuming tasks in the background separately from the main UI thread. Correct use of `AsyncTask.doInBackground(..)` can ensure that Android applications remain responsive to user input, but care must be taken to ensure that interaction between the background thread and the main UI thread is correctly synchronized.

Investigating further in Eclipse’s call hierarchy, we discover that the `onDiscard()` method, which also accesses the `mDraftId` field, is called by the `onBackPressed()` method. This method is in turn always invoked by the Android framework on the main UI thread, not the background thread for running `AsyncTasks`, indicating the presence of a potential concurrency defect.

4 Experiences with developing and using THREADSAFE

4.1 What is a concurrency bug, and how do developers react to static analysis?

In our experience, the primary difficulty in developing an unsound and incomplete static analysis like THREADSAFE is in determining exactly what ought to be reported, given the information that we are able to compute statically from the program.

For example, the Inconsistent Synchronisation checker described above only discovers places where the programmer has been inconsistent in their use of locks when accessing a given field. It does not attempt to determine whether or not the two accesses can actually happen in parallel. Nor does it attempt to determine any particular severity of the potential race condition. There may be several reasons why accesses to a field use locks inconsistently:

1. The programmer has genuinely made a mistake, and the unlocked, or incorrectly locked accesses, represent real data races that will manifest at runtime;

2. The programmer has genuinely made a mistake, but the unlocked, or incorrectly locked accesses are extremely rare at runtime, and are very unlikely to produce a real reliability issue when the code is in production;

3. The field is only ever accessed by one thread, but other fields are accessed by multiple threads, and the locks used to protect them are irrelevantly being held while accessing the single-thread field, giving the impression that it has been locked unnecessarily;

4. The methods that access the fields without locking are never executed in parallel with those that do. For example, initialisation methods or shutdown methods may be guaranteed to have exclusive access to their object;

5. The programmer has been able to determine via other means that this particular field access is safe, perhaps by reasoning based on the Java Memory Model.

In case 2, some developers that we observed using THREADSAFE were loath to change their code to assuage the analysis. Changing code has costs of its own, and if the bug is extremely unlikely (at least in the developer's estimation), then why bother? Other users of THREADSAFE take a longer view, reasoning that a bug that is unlikely now may become more likely in the future after the structure and intended purpose of the code evolves. Leaving concurrency bugs hidden inside code that is believed to be reliable creates technical debt that obstructs future development.

In cases 3 and 5, it could be argued that the design of the code is at fault. If the locking strategy used by the code is sufficiently complex that subtle reasoning is required to determine when access to a field requires a lock or not, then the code ought to be changed.

In our experience, different developers differ widely in their responses to static analysis results. Some actively dislike false positives, treating them as noise; some appreciate the warnings, but require a reliable way of turning them off for code they have hand-checked to be safe; while others will rewrite code to remove static analysis warnings, perhaps in the belief that code that confuses a static analyser will confuse a human too. In general, however, we observed a lean towards conservatism in most developers: unless a particular static analysis warning obviously indicates a real, costly, bug, then it was usually felt better to leave apparently working and tested code alone until time could be set aside to work on it properly. Performing quick fixes piecemeal on complex code, following the warnings produced by a static analyser, may actually decrease reliability. Accidental reduction in reliability is especially likely if it is done without an understanding of what the code is meant to do, a common situation when the original developers of some code have left the organisation.

At a level above data races on individual fields, it can be very difficult to determine the difference between intended behaviour and unintended behaviour. In some cases, non-deterministic behaviour may be exactly what the programmer intends to happen in a concurrency environment, but in other cases it may be a mistake. THREADSAFE is designed to operate on code that has no formal specification describing what the intended behaviour is (and, often, no written informal specification either). Arguably, the most damaging concurrency errors are not the low-level data races and race conditions on concurrent collections, but high-level design errors that lead to user-visible data inconsistencies. However, in the absence of information about what ought to happen, these bugs are the most difficult for a static analysis tool to find.

4.2 Testing a static analyser

Beyond the basic question of what kinds of errors THREADSAFE ought to catch, or not catch, a practical aspect of the development of any heuristic static analysis is how to maintain and track the behaviour of the analysis as it is added to and bugs in the analyser are fixed. It is all too easy to fix a bug that removes some false positive, while at the same time also removing many more valuable true positives. Whether or not such trade-offs are a net positive is one of the hardest parts of developing a static analyser intended for a mass audience. In order to be able to make judgements about such tradeoffs, we need to have visibility into the effect of changes we make during development of THREADSAFE.

To ensure that development of THREADSAFE does not accidentally introduce new false positives, or remove valuable true positives, we maintain a large test suite of over 500 tests identifying specific true/false positives and negatives, and for each one whether it is expected to be reported or not. Since most of the checkers in THREADSAFE rely to a greater or lesser extent on heuristics, the only practical way to document and maintain the desirable behaviour of checkers is via testing — there is no formal specification of how THREADSAFE ought to behave.

Since the individual test cases are not representative of real Java projects, THREADSAFE is also systematically tested against a portfolio of open source and proprietary Java code bases. THREADSAFE is tested against the benchmark suite before and after each change to determine the impact of altered heuristics, or new checkers. We do not attempt to classify all the findings of THREADSAFE on the benchmark suite — this turns out to be too difficult and time consuming for a small company, see *What is a Concurrency Bug?* above — but we do inspect the difference in the analysis results before and after the change, to determine whether the proposed change has an overall positive impact. The benchmark suite is also used to track the runtime of THREADSAFE on realistic code bases for regressions.

5 Performance

5.1 Measuring static analysis tool performance

The performance of a static analysis tool like THREADSAFE can be measured in a number of ways. For example:

- What percentage of the real bugs in a given codebase does the tool find (true positives) and miss (false negatives)?
- What is the potential impact of the bugs found versus the ones missed?
- How difficult would it be to find the same bugs by other means?
- What percentage of findings produced are false alarms (false positives)?
- How easy is it to investigate a finding to discover if it is a genuine bug and how to fix it?
- How much time and space does the tool consume?
- How well does it scale for use with very large codebases?
- How smoothly is the tool integrated with a software developer's existing workflow?

Many of these are difficult to measure. For instance: in any very large real-world codebase, computing the percentage of real bugs found assumes knowledge of all of the real bugs that it contains. Determining whether or not a finding reports a real bug or is a false alarm, and the potential impact of a real bug, often relies on expert knowledge of the architecture of the system and/or the way that it is intended to be used. Some aspects are somewhat subjective and a matter of taste or attitude. Finally, measurements of performance on small or textbook examples may bear little relation to performance on large real-world codebases.

For a commercial static analysis tool like THREADSAFE, intended for use in industrial software development, the tradeoffs between these factors are different from what an academic researcher might expect. When working with a million-line codebase, developers are primarily interested in finding and fixing very high-impact bugs quickly. Their experience tells them that some bugs will remain, no matter what they do. And since fixing a bug might introduce a new bug, as explained earlier, it might well be prudent to leave a low-impact bug unfixed. A tool that reports large numbers of low-impact bugs alongside a few high-impact bugs may therefore be less useful than one that reports only the high-impact bugs. But filtering lists of findings according to type will sometimes allow likely high-impact bugs to be separated from low-impact bugs.

Any static analysis tool needs to balance false positives against false negatives: higher sensitivity will tend to reveal more potential bugs, but some of those found will turn out to be false alarms. If the aim is to eliminate high-impact bugs quickly, then a high level of false positives is more damaging than a moderate level of false negatives. Developers will quickly lose patience with a tool if they need to investigate a long list of false alarms before encountering a genuine high-impact bug.

More interesting than the absolute performance of a tool is its performance in comparison to competing tools. Unfortunately, a head-to-head comparison between THREADSAFE and competing commercial tools is not possible because their licenses forbid their use for this purpose. Our experiments suggest that THREADSAFE outperforms other commercial static analysis tools with respect to detection of Java concurrency defects, and users of other tools are invited to undertake a comparison themselves. We provide a comparison below between THREADSAFE and the non-commercial FindBugs tool [Fin15], the most widely-used static analysis tool for Java.

5.2 Performance on IBM benchmark

The IBM concurrency bugs benchmark [EHSU07, ETU08, IBM15] is a collection of Java programs with concurrency bugs, meant for use in comparing bug-finding tools; it is the most widely-used benchmark for this purpose. Most are small programs produced by students at the University of Haifa as coursework for a software testing course and the bugs vary in complexity. See Table 1 for a list. An example of a simple bug is a use of the double-checked locking anti-pattern [Pug00] in the program `DCL`. An example of a bug that seems well beyond the reach of an automatic static-analysis tool is an atomicity bug in the program `Lottery` that leads to violation to an implicit logical invariant. An earlier version of the benchmark was annotated with information about the location(s) and type(s) of bugs in each program, but this information and some of the programs in the benchmark have been lost [TB15].

THREADSAFE and FindBugs (with only “multithreaded correctness” checkers enabled) were both applied to each program, first with default settings (first column) and then with sensitivity increased (second column). For THREADSAFE, sensitivity was increased by reducing the threshold for inconsistent synchronisation from 70% to 50%. For FindBugs, sensitivity was increased by setting analysis effort to “maximal” and setting the minimum rank to report so that all findings would be reported. Findings produced by both tools were manually classified as true positives or false positives and the number of concurrency bugs in each program was counted. The TP/FP classification and count of bugs are admittedly somewhat subjective.

The results produced by THREADSAFE and Findbugs on this benchmark are given in Table 1. There are many bugs in the benchmark, including the atomicity bug in `Lottery` mentioned above, that neither tool is able to detect. THREADSAFE finds about twice as many bugs as FindBugs does, with about the same false positive rate.

There are a few bugs that FindBugs detects which THREADSAFE, despite its more sophisticated analysis, misses. An example is the bug in `Piper`, where there is an invocation of the method `java.lang.Object.wait()` that is not in a loop. FindBugs contains a checker for exactly this bug pattern. In developing THREADSAFE, we decided not to duplicate checkers in FindBugs unless we were able to significantly improve on its results, since it is easy to use FindBugs alongside THREADSAFE. This is just such a case.

Spathoulas [Spa14] did a similar comparison of THREADSAFE with FindBugs and Chord [Cho12] on the IBM benchmark and on concurrency examples from *The CERT Oracle Secure Coding Standard for Java* [LMS$^+$11], comparing THREADSAFE with default sensitivity against FindBugs with increased sensitivity and the datarace and deadlock analyses in Chord. His comparison did not classify findings into true and false positives. He found that Chord was the most accurate of the three tools but that it was not scalable for use with codebases of even modest

	ThreadSafe	ThreadSafe	FindBugs	FindBugs	
	default	sensitive	default	sensitive	# bugs
	TP / FP	TP / FP	TP / FP	TP / FP	
`A_B_push_pop`	0 / 0	0 / 0	0 / 0	0 / 0	1
`Account`	0 / 0	1 / 0	0 / 0	1 / 0	1
`Airlines Tickets`	0 / 0	0 / 0	0 / 1	0 / 1	1
`AllocationVector`	0 / 0	0 / 0	0 / 0	0 / 0	1
`BoundedBuffer`	0 / 3	0 / 4	0 / 0	0 / 3	1
`BubbleSort`	1 / 0	1 / 0	1 / 0	1 / 0	1
`BubbleSort2`	1 / 0	1 / 0	0 / 0	0 / 0	1
`BufWriter`	1 / 3	2 / 5	0 / 0	0 / 0	2
`Critical`	0 / 0	0 / 0	1 / 0	1 / 0	1
`DCL`	1 / 0	1 / 0	1 / 0	1 / 0	1
`Deadlock`	0 / 0	0 / 0	0 / 1	0 / 1	1
`DeadlockException`	0 / 0	0 / 0	0 / 0	0 / 0	1
`Fiforeadwritelock`	0 / 0	0 / 0	0 / 0	0 / 0	1
`FileWriter`	0 / 0	2 / 0	1 / 0	1 / 0	2
`GarageManager`	1 / 0	1 / 2	0 / 0	0 / 0	1
`Hierarchy Example`	0 / 1	0 / 1	0 / 0	0 / 1	1
`LinkedList`	0 / 0	0 / 0	0 / 0	0 / 0	1
`Liveness`	0 / 0	0 / 0	0 / 0	0 / 0	1
`Lottery`	1 / 0	2 / 0	0 / 0	0 / 0	2
`Manager`	2 / 3	2 / 3	0 / 0	0 / 0	2
`MergeSort`	0 / 1	0 / 1	0 / 0	0 / 0	1
`MergeSortBug`	0 / 0	0 / 0	0 / 0	0 / 0	1
`PingPong`	0 / 0	0 / 0	0 / 0	0 / 0	1
`Piper`	0 / 0	0 / 0	1 / 0	1 / 0	1
`ProducerConsumer`	0 / 0	0 / 0	0 / 0	0 / 0	1
`Shop`	1 / 0	1 / 0	0 / 0	1 / 0	2
`Suns Account`	0 / 0	0 / 0	0 / 0	0 / 0	1
`XtangoAnimation`	5 / 0	5 / 0	2 / 3	2 / 5	5
Total	14 / 11	19 / 16	7 / 5	9 / 11	37
False positive rate: FP / (FP+TP)	44%	46%	42%	55%	
Percentage of bugs found	38%	51%	19%	24%	

Table 1: ThreadSafe and FindBugs applied to the IBM benchmark

size. He also found that THREADSAFE was much better than FindBugs at resisting attempts to hide bugs using very simple obfuscation methods. This is to be expected because of the comparatively simple bug-pattern detection methods used by FindBugs [HP04], but it suggests that the performance gap between THREADSAFE and FindBugs will widen for codebases containing more complex and hidden bugs.

Although comparison of performance on small examples like the ones in the IBM benchmark provides a useful data point, is not a reliable guide to performance on large real-world examples. At least, that is our experience with deadlock detection. The heuristics that THREADSAFE uses to achieve a good rate of deadlock detection in reasonable time sometimes yield poor performance on small "textbook-style" examples. For real industrial-scale examples, its results tend to be much better. THREADSAFE's heuristics could be adjusted to perform better on the small examples, but this would increase the false positive rate on larger examples. We have not yet found a way to achieve good results in both cases.

6 Packaging

THREADSAFE is tightly integrated with Eclipse, as shown above, and with the SonarQube quality platform. It can also be used from the command line to generate an HTML report containing information similar to what is available in Eclipse.

Contemplate's website http://www.contemplateltd.com/threadsafe provides more information about THREADSAFE and access to free two-week trials.

Bibliography

[Cho12] Chord 2.1. 2012.
http://www.cc.gatech.edu/~naik/chord.html

[EHSU07] Y. Eytani, K. Havelund, S. D. Stoller, S. Ur. Towards a framework and a benchmark for testing tools for multi-threaded programs. *Concurrency and Computation: Practice and Experience* 19(3):267–279, 2007.
http://dx.doi.org/10.1002/cpe.1068

[ETU08] Y. Eytani, R. Tzoref, S. Ur. Experience with a Concurrency Bugs Benchmark. In *First International Conference on Software Testing Verification and Validation, ICST 2008, Lillehammer, Norway, April 9-11, 2008, Workshops Proceedings*. Pp. 379–384. 2008.
http://dx.doi.org/10.1109/ICSTW.2008.17

[Fin15] FindBugs 3.0.1. 2015.
http://findbugs.sourceforge.net

[Goe06] B. Goetz. *Java Concurrency in Practice*. Addison-Wesley, 2006.

[HP04] D. Hovemeyer, W. Pugh. Finding Bugs is Easy. *SIGPLAN Notices* 39(12):92–106, Dec. 2004.
http://doi.acm.org/10.1145/1052883.1052895

[HS12] M. Herlihy, N. Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann, 2012.

[IBM15] IBM. Concurrency benchmark. Accessed Aug 2015.
http://researcher.watson.ibm.com/researcher/view_person_subpage.php?id=5722

[Lee06] E. A. Lee. The problem with threads. *IEEE Computer* 39(5):33–42, 2006.
http://dx.doi.org/10.1109/MC.2006.180

[LMS$^+$11] F. Long, D. Mohindra, R. C. Seacord, D. F. Sutherland, D. Svoboda. *The CERT Oracle Secure Coding Standard for Java*. Addison Wesley, 2011.

[Ora12] JDK-7027300 : Unsynchronized HashMap access causes endless loop. 2012. Oracle bug report.
http://bugs.java.com/view_bug.do?bug_id=7027300

[Pug00] W. Pugh. The "double-checked locking is broken" declaration. 2000.
http://www.cs.umd.edu/users/pugh/java/memoryModel/DoubleCheckedLocking.html

[Ses14] P. Sestoft. Practical Concurrent and Parallel Programming. 2014. Course at IT University of Copenhagen.
http://www.itu.dk/people/sestoft/itu/PCPP/E2014/

[Spa14] A. Spathoulas. Assessing Tools for Finding Bugs in Concurrent Java. Master's thesis, 2014. School of Informatics, University of Edinburgh.
http://homepages.inf.ed.ac.uk/dts/students/spathoulas/spathoulas.pdf

[Tym09] P. Tyma. A beautiful race condition. 2009. Mailinator blog post.
http://mailinator.blogspot.co.uk/2009/06/beautiful-race-condition.html

[TB15] R. Tzoref-Brill. Private communication. August 2015.