Keshmesh: A Tool for Detecting and Fixing Java Concurrency Bug Patterns

Mohsen Vakilian Stas Negara Samira Tasharofi Ralph E. Johnson
University of Illinois, Urbana, IL, USA
{mvakili2, snegara2, tasharo1, rjohnson}@illinois.edu

Abstract

Developing concurrent software is error prone. Others have cataloged common bug patterns in concurrent Java programs. But, there are no tools for detecting complex concurrency bug patterns accurately, and concurrent programs are full of similar bugs. We have been developing a tool called *Keshmesh* for detecting complex concurrency bug patterns in Java programs statically. Keshmesh is the first tool that accurately detects a few of the top concurrency bug patterns of the SEI CERT catalog [3] and suggests automated fixers for some of them. Keshmesh is fast enough to be used interactively, produces few false alarms and helps Java programmers to quickly find and fix common concurrency bug patterns in their programs.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification; D.1.3 [*Programming Techniques*]: Concurrent Programming

General Terms Algorithm, Design, Verification

Keywords bug, pattern, detector, fixer, concurrency, parallelism, static analysis, program analysis

1. Introduction

Multi-cores have encouraged more programmers to write concurrent software. But, writing concurrent software is difficult and similar bugs keep showing up in concurrent programs. Cataloging common bug patterns in concurrent programs is an active research domain, and the SEI CERT catalog [3] is a recent and comprehensive catalog of concurrency bug patterns in Java. The existence of a bug pattern does not necessarily imply a bug. In other words, some bug patterns are considered as bad practices that might lead to future bugs as the software evolves. Nonetheless, it is valuable to detect such problems as early as possible.

Keshmesh is the first tool that automatically detects complex concurrency bug patterns and suggests ways of fixing them. Complex concurrency bug patterns usually cross the boundaries of methods and involve indirect accesses to shared data via references. The SEI CERT catalog has rated the *severity*, *likelihood* and *remediation cost* of each bug pattern. We have prioritized the bug patterns based on these three attributes and selected five of the top ten bug patterns. Keshmesh provides automated detectors for the generalized forms of five bug patterns and fixers for two of them. Keshmesh improves the state of the art in static analysis for detecting concurrency bug patterns by looking for instances of the bug patterns interprocedurally and employing context sensitive points-to analysis. Keshmesh is an Eclipse plug-in that uses WALA [2] as its underlying static analysis engine and FindBugs [1] as its user interface. Keshmesh is open source and available at `http://keshmesh.cs.illinois.edu`.

SPLASH'11 Companion, October 22–27, 2011, Portland, Oregon, USA.
ACM 978-1-4503-0940-0/11/10.

2. Detecting and Fixing Bug Patterns

Keshmesh extends the Eclipse plug-in of FindBugs [1] by a few concurrency bug pattern detectors and fixers. By extending FindBugs, Keshmesh inherits all the nice features of FindBugs such as filtering and integration into build systems and becomes easily usable by the users of FindBugs. Even though Keshmesh extends FindBugs to report bug patterns and suggest fixes to the user, it does not use the analysis engine of FindBugs. Instead, Keshmesh uses WALA [2], which is a much more powerful engine for analyzing Java byte code.

WALA enables Keshmesh to detect complex bug patterns with high accuracy. WALA requires the user to designate some methods as entry points. Typically, the `main` methods and test methods serve as entry points. Keshmesh expects the user to specify the entry points by using the custom method annotation `@EntryPoint`. Knowing the entry points, WALA builds the call graph and computes the points-to sets starting from the entry points. Then, every Keshmesh detector inspects the results of WALA, i.e. the IR, call graph and points-to sets to find the bug patterns. Keshmesh reports the bug patterns to the user as FindBugs bug reports, and if an automated fix is available, the user can apply the automated fix using the quick fix mechanism of Eclipse. In the following, we briefly describe the bug patterns and their detectors and fixers in Keshmesh.

2.1 LCK01-J. Do not synchronize on objects that may be reused

Objects such as interned strings and primitive literals may be reused. Therefore, such reusable objects cannot be safely used as locks. Keshmesh examines the allocation sites to detect reusable objects, and reports any `synchronized` block whose lock expression may point to a reusable object as an instance of LCK01-J. For example, Keshmesh reports a `synchronized` block whose lock object is the variable `intLock`, where `Integer intLock = 0`, because this object is autoboxed and allocated inside `Integer.valueOf()`.

2.2 LCK02-J. Do not synchronize on the class object returned by `getClass()`

The bug pattern LCK02-J recommends not to use the return value of `Object.getClass()` as a lock.

Keshmesh detects an object returned by `Object.getClass()` by inspecting the allocation site of the object. If the type of the allocation is `Class` and the object is allocated inside `Object.getClass()`, Keshmesh recognizes the object as one returned by `Object.getClass()`. Keshmesh reports any `synchornized` block whose lock expression may point to an object returned by `Object.getClass()` as a instance of LCK02-J. Moreover, Keshmesh suggests automated fixes to the user to replace the lock expression by the class literal of the objects that it may point to, if there is only one possible class literal.

2.3 LCK03-J. Do not synchronize on the intrinsic locks of high-level concurrency objects

Instances of classes that implement `Condition` or `Lock` are high-level concurrency objects. And, using such concurrency objects as the lock objects of `synchronized` blocks is a bad practice.

Keshmesh reports `synchronized` blocks whose lock expressions may point to a high-level concurrency object as instances of LCK03-J. In addition, if the high-level concurrency object is a `Lock`, Keshmesh provides the user with an automated fixer that replaces the `synchronized` block on the high-level object by a block that invokes `Lock.lock()` and `Lock.unlock()` at the beginning and end, respectively.

2.4 LCK06-J. Do not use an instance lock to protect shared `static` data

Instance locks cannot protect shared `static` data because multiple instances of the class may make the locks different. Therefore, LCK06-J warns about the use of instance locks to protect shared `static` data.

Let S be the set of objects that the `static` fields may point to. A `synchronized` block is *safe* if the points-to set of its lock object is a nonempty subset of S. Similarly, a `synchronized` method is *safe* if it is either `static` or the points-to set of its lock object, `this`, is a nonempty subset of S. A byte code instruction is *unsafe* if it is not inside any safe `synchronized` blocks and modifies a non-`final` `static` field or an object that some `static` field may point to.

Keshmesh computes the unsafe instructions of each method and performs an interprocedural data flow analysis to propagate these instructions up the call graph. Finally, it reports the `static` fields affected by the unsafe instructions to the user.

2.5 VNA00-J. Ensure visibility when accessing shared primitive variables

An unprotected access to a non-`volatile` shared primitive variable is an instance of VNA00-J. We have generalized VNA00-J by not restricting the shared data to primitive variables. Keshmesh considers any class that extends `Thread`, implements `Runnable`, or contains `synchronized` methods or blocks as a thread-safe class. And, it treats all objects reachable from the fields of nonlocal instances of thread-safe classes as shared data. Keshmesh looks for unsafe accesses to the shared data and marks every line of code that directly or indirectly, i.e. through method invocations, makes an unsafe access as an instance of VNA00-J. Keshmesh propagates unsafe accesses across the boundaries of methods by solving a data flow problem. This formulation of the data flow problem propagates unsafe accesses from the callee to the caller if the call site is not protected by a `synchronized` block and at least one argument of the method invocation may be shared data.

3. Related Work

FindBugs [1] detects a variety of bug patterns, including several concurrency bug patterns. However, the capability of FindBugs for detecting bug patterns is limited by its intraprocedural analysis engine.

Luo et al. [4] developed a tool to statically find concurrency bug patterns in Java. Their tool finds most of the bug patterns intraprocedurally by AST pattern matching, and it uses WALA's points-to analysis only for one bug pattern.

Naik et al. [5] proposed a static analysis to find a specific kind of bug pattern, i.e. data races, in Java programs. Their tool looks for actual bugs rather than bug patterns that might lead to bugs in future.

4. Future Work

A challenge in developing an effective static analysis tool for finding bug patterns is to make the right balance between accuracy on one hand and performance and scalability on the other hand. We plan to evaluate Keshmesh on real-world software, adjust the identification criteria of its bug patterns and tune the context sensitivity of WALA [2] to improve the accuracy, performance and scalability of Keshmesh.

5. Description of the Demonstration

The SPLASH community has been actively participating in shaping the future of multi-core software engineering. Keshmesh is relevant to the SPLASH community since it is a step towards correct concurrent software.

In our demonstration session, we plan to present the bug patterns supported by Keshmesh. We will do a live demo of Keshmesh on some examples. Specifically, we will run Keshmesh on several example programs containing instances of some of the bug patterns supported by Keshmesh. Then, we will show how the results are presented to the user, including the fix information, where it is available. We will show the audience how to use Keshmesh to find instances of the bug patterns and fix them. Then, we will explain the underlying algorithms of Keshmesh for detecting the bug patterns. We will compare Keshmesh with an existing tool for detection of concurrency bug patterns such as FindBugs [1]. Also, we will describe the techniques that we have employed to achieve the right number of false alarms and tune WALA [2] for getting good precision and performance.

6. Presenters

Mohsen Vakilian and Samira Tasharofi are PhD students at the University of Illinois. Mohsen is interested in improving the programming environments for parallel programming. In addition to developing better tools for finding and fixing concurrency bug patterns, he has been working on tools for migrating sequential programs to parallel languages. Samira is interested in concurrency bug patterns and testing concurrent programs. She has been working on partial order reduction and tools for testing message-passing programs.

References

[1] FindBugs. URL http://findbugs.sf.net/.

[2] T.J. Watson Libraries for Analysis (WALA). URL http://wala.sf.net/.

[3] F. Long, D. Mohindra, R. C. Seacord, and D. Svoboda. Java Concurrency Guidelines. Technical report, 2010. URL http://www.sei.cmu.edu/reports/10tr015.pdf.

[4] Z. D. Luo, L. Hillis, R. Das, and Y. Qi. Effective Static Analysis to Find Concurrency Bugs in Java. In *10th IEEE Working Conference on Source Code Analysis and Manipulation*, 2010.

[5] M. Naik, A. Aiken, and J. Whaley. Effective Static Race Detection for Java. In *Proceedings of the conference on programming language design and implementation*, 2006.