# Refactoring for Reentrancy

Jan Wloka[*], Manu Sridharan[†], Frank Tip[†]
[*] Dept. of Computer Science, Rutgers University, Piscataway, NJ 08854, USA
jwloka@cs.rutgers.edu
[†] IBM T. J. Watson Research Center, P.O. Box 704, Yorktown Heights, NY 10598, USA
{msridhar,ftip}@us.ibm.com

## ABSTRACT

A program is *reentrant* if distinct executions of that program on distinct inputs cannot affect each other. Reentrant programs have the desirable property that they can be deployed on parallel machines without additional concurrency control. Many existing Java programs are not reentrant because they rely on mutable global state. We present a mostly-automated refactoring that makes such programs reentrant by replacing global state with thread-local state and performing each execution in a fresh thread. The approach has the key advantage of yielding a program that is *obviously* safe for parallel execution; the program can then be optimized selectively for better performance. We implemented this refactoring in *Reentrancer*, a practical Eclipse-based tool. *Reentrancer* successfully eliminated observed reentrancy problems in five single-threaded Java benchmarks. For three of the benchmarks, *Reentrancer* enabled speedups on a multicore machine without any further code modification.

## Categories and Subject Descriptors

D.2.3 [**Software Engineering**]: Coding Tools and Techniques; D.2.6 [**Software Engineering**]: Programming Environments

## General Terms

Performance, Reliability

## 1. INTRODUCTION

This paper presents a refactoring tool that makes single-threaded programs *reentrant*. A program is reentrant[1] if distinct executions of the program on distinct inputs cannot affect each other, whether run sequentially or concurrently. (Definition 1 in Section 2 provides a more precise definition of the term.) As multicore machines become more pervasive, reentrancy is an increasingly desirable property—reentrant programs can be run in parallel on multicores without additional concurrency control or a separate virtual machine for each program instance. Furthermore, even in single-threaded settings, reentrancy is a desirable property because it ensures that successive executions of an application will behave consistently.

A key obstacle to obtaining increased performance on multicores is the effort required to make legacy single-threaded programs reentrant. Many such programs were written without concurrency in mind and use global state in a non-reentrant manner. Manually introducing reentrancy can be labor-intensive since any use of mutable global state may break reentrancy, and hence such state must either be eliminated or enclosed in carefully-placed synchronization constructs.

We are aware of several approaches for making programs reentrant. First, global state can be encapsulated in one or more "environment objects", and additional method parameters can be introduced to make such objects accessible throughout the application. In our experience, this approach requires a prohibitive amount of intrusive code change. A second approach is to use isolation mechanisms such as separate OS processes or class loaders to ensure non-interference of parallel program executions. However, an important disadvantage of this approach is that it does not allow for selectively removing some isolation for performance (e.g., by sharing some state), and it may introduce high overhead.

This paper employs a third approach based on replacing global mutable state with *thread-local state*. In this approach, each execution thread gets a separate copy of each global variable, and any access to a global variable is implicitly indexed by the current thread. After the introduction of thread-local state, the programmer can ensure reentrancy by (manually) modifying code to use a fresh thread for each execution of the program; the threads can be run sequentially or concurrently.

The use of thread-local state for reentrancy has two key desirable properties. First, the approach is widely applicable, as many platforms provide built-in support for thread-local state, including Java, pthreads, and .NET. Second, if desired (e.g., for performance), a developer can undo the introduction of thread-local state for select global variables (e.g., by introducing carefully synchronized shared state) while leaving the rest as thread-local; this is not possible in isolation-based approaches. This leads to an approach of *correctness before performance* when moving software to

[1] Other definitions of the term "reentrant" can be found in the literature and will be discussed in Section 5.


*ESEC-FSE'09,* August 23–28, 2009, Amsterdam, The Netherlands.

multicores (also espoused elsewhere [6]): the refactoring produces an *obviously* reentrant program (i.e., one with no global mutable state in the application) which can then be optimized as needed. We believe this approach of correctness before performance when parallelizing code will lead to more robust code and greater programmer productivity.

The refactoring tool in which we implemented these ideas, *Reentrancer*, targets the Java programming language and deals with several additional issues that arise when handling real-world programs. One such issue is initialization semantics—Java's static initializers can have fairly intricate behavior that must be preserved when introducing thread-local state.[2] While our transformation does not preserve static initializer semantics in all cases, we have formulated preconditions that ensure that the initializers in an input program are safe for our refactoring; our tool emits warnings when it is able to detect violations of these preconditions. Another challenge is handling library code, which cannot be refactored and allows for accessing possibly-mutable data in the environment (user input, files, etc.); our tool also warns about common library usage that may break reentrancy.

*Reentrancer* was implemented as an extension to the Eclipse Java development tools (JDT)[3] and evaluated by using it to make five Java applications reentrant. Our evaluation showed that *Reentrancer* can successfully refactor realistic programs. For three of the refactored benchmarks, we measured small speedups when the tests associated with the application were executed in parallel on a multicore machine, with no additional performance tuning.

In summary, the contributions of this paper are as follows:

1. a characterization of the causes of non-reentrancy in Java programs,
2. a mostly-automated refactoring for making Java programs reentrant by replacing global state with thread-local state, with associated precondition checking that warns of possible behavioral changes due to fragile static initializer behavior and non-reentrant library usage,
3. an implementation of a refactoring tool, *Reentrancer*, as an extension to the Eclipse JDT, and
4. an evaluation of *Reentrancer* on a set of Java applications, demonstrating its practicability.

# 2. MOTIVATING EXAMPLE

This section presents a motivating example that illustrates our refactoring for making programs reentrant. The refactoring is comprised of the following 5 steps:

1. Removal of non-reentrant accesses to library global state.
2. Encapsulation of static fields in the application in getter/setter methods.
3. Replacing static initializers with explicit lazy initialization.
4. Replacing global state with thread-local state.
5. Transforming the application to use a fresh thread for each execution.

Steps 2-4 are automatic, while steps 1 and 5 must be performed by the developer. In the remainder of this section, we will explain each of these steps in detail using an example program.

*Definitions.* Before discussing our example, we define the key concepts of *reentrancy* and *immutability* as used in this paper.

DEFINITION 1 (REENTRANCY). *Let an execution of a program $\mathcal{P}$ be any external invocation of $\mathcal{P}$, e.g., running $\mathcal{P}$'s* `main` *method or invoking a public API method from $\mathcal{P}$. A program $\mathcal{P}$ is* reentrant *iff for any two executions $e_i$ and $e_j$ of $\mathcal{P}$ such that $e_i$ and $e_j$ have no mutable shared inputs (see Definition 2), the results of $e_i$ and $e_j$ are* unaffected *by how the executions are ordered, including parallel interleavings.*

The goal of our refactoring is to make sequential programs reentrant when each execution is run in a fresh thread (sequentially or concurrently). Note that the refactoring may not preserve program semantics if certain preconditions do not hold (see Section 3.2).

DEFINITION 2 (IMMUTABILITY). *We consider an object $o$ to be immutable iff all of $o$'s transitively reachable abstract state (reached by following instance field pointers) cannot be mutated (i.e., it is deeply immutable [20]). A static field is immutable iff (1) it is declared* `final` *and (2) it only points to an immutable object $o$.*

This notion of immutability is often a requirement for safely sharing state between threads (further discussion in [23]).

## 2.1 Example Program

Figure 1 shows a small (contrived) Java program that we shall use as our example. In this program, class `Configuration` represents a configuration as a set of key-value pairs, which are stored in a `HashMap`. A static initializer is used to initialize the configuration with a pair <`"version"`,`"2.0"`> (see lines 5–7). Additional entries can be added using the `setOption()` method and retrieved using the `getOption()` method (see lines 8–10 and 11–13). Class `CheckUserFiles` defines a method `checkDir()` that performs checks on some directory `homedir` by setting a `Configuration` option and calling `runAllChecks()`. The `checkDir()` method throws an exception if the `homedir` option has already been set (see line 18). Class `Tests` defines two JUnit tests, `testFoo()` and `testBar()` that invoke the `checkDir()` method with arguments `"foo"` and `"bar"`, respectively.

The program in Figure 1 is not reentrant: when `testFoo()` and `testBar()` are executed consecutively, `testBar()` fails with an exception (thrown at line 18), but when `testBar()` is executed in isolation, it passes. This lack of reentrancy is due to the use of a global (static) map `conInfo` (declared on line 2) to store configuration items. (Note that though `conInfo` is declared `final`, it points to a mutable `HashMap`, allowing for the problematic usage.) If the tests were run in parallel, data races on `conInfo` could cause other undesirable behaviors.

The use of a *static initializer* in the `Configuration` class is another barrier to reentrancy. A static initializer $s$ is invoked at most once in an execution, typically when the enclosing class $C$ is loaded (a more detailed discussion of static

[2]Similar issues arise in other languages, e.g., C# [3].
[3]http://www.eclipse.org/jdt

```
[1]  class Configuration {
[2]    private static final Map<String, String> conInfo =
[3]      new HashMap<String, String>();
[4]    private static final String versionStr = "2.0";
[5]    static {
[6]      conInfo.put("version", versionStr);
[7]    }
[8]    static void setOption(String name, String value) {
[9]      conInfo.put(name, value);
[10]   }
[11]   static String getOption(String name) {
[12]     return conInfo.get(name);
[13]   }
[14] }
[15] class CheckUserFiles {
[16]   public static boolean checkDir(String homedir) {
[17]     if (Configuration.getOption("homedir") != null) {
[18]       throw new RuntimeException("homedir already set!");
[19]     }
[20]     Configuration.setOption("homedir", homedir);
[21]     return runAllChecks();
[22]   }
[23]   private static boolean runAllChecks() {
[24]     System.out.println("running checks..");
[25]     // details omitted..
[26]     return true;
[27]   }
[28] }
[29] public class Tests {
[30]   @Test
[31]   public void testFoo() {
[32]     Assert.assertTrue(CheckUserFiles.checkDir("foo"));
[33]   }
[34]   @Test
[35]   public void testBar() {
[36]     Assert.assertTrue(CheckUserFiles.checkDir("bar"));
[37]   }
[38] }
```

Figure 1: Example program.

initialization in Java will follow in Section 3.2.1). Static initializers impede reentrancy since after $s$ is executed, later uses of $C$ will not run $s$ again and hence may behave differently. Besides transforming global state into thread-local state, our refactoring must also produce code that initializes each thread-local copy in a manner consistent with existing static initializers.

## 2.2 Removing Accesses to Library Globals

The first step of the refactoring is a manual transformation to remove non-reentrant accesses to global state in libraries. The example program of Figure 1 refers to global library state on line 24, where it calls method `println()` on the static field `System.out`. This gives rise to non-reentrant behavior because if multiple instances of the application are executed concurrently, their output may be intertwined in unpredictable ways. Furthermore, our refactoring cannot remove this issue via thread-local state since (1) library code cannot be transformed and (2) many non-reentrant behaviors occur in native code (console output, file accesses, etc.) that our system cannot analyze.

There are many possible ways to remove accesses to library globals. In this case, one plausible option is to associate a dedicated Java `Logger` object with each execution instance instead of using `System.out`, as shown in Figure 2(a); changed code fragments are underlined. Alternately, the programmer may decide that the original output was unnecessary and delete the call to `println()`, or she may decide that the occurrence of interleaved output from different executions can safely be ignored and leave the code unmodified. Our tool simply reports possibly problematic accesses to global library state and leaves it to the programmer to handle them (see Section 3.2.2).

## 2.3 Encapsulating Static Fields

The next step in our refactoring is the encapsulation of static fields in the application by introducing getter/setter methods and replacing any direct access to such fields with calls to these methods. While this step is not strictly necessary to produce reentrant code, we include it since it enables encapsulation of the lazy initialization code that is introduced in the next step (see Section 2.4). Immutable fields (see Definition 2) that are initialized to constants need not be encapsulated, since they will not be made thread-local. Furthermore, for `final` fields, it suffices to introduce a getter method only. This introduction of getter/setter methods is performed automatically by our *Reentrancer* tool by invoking the standard Encapsulate Field refactoring [9] provided by Eclipse.

Our example program declares two static fields: `conInfo` and `versionStr`. Of these fields, only `conInfo` needs to be encapsulated because `versionStr` is immutable and points to a constant `String`. Furthermore, only a getter method is needed for `conInfo` because it is `final`. The resulting code is shown in Figure 2(b).

## 2.4 Introducing Lazy Initialization

Next, our refactoring replaces static initializers with explicit lazy initialization. As previously mentioned, static initializer methods are an impediment to reentrancy because they are only executed once, upon the first use of the declaring class. Our approach is to move the static initialization code[4] for each class $C$ into a method $C$`.lazyInit()` that is executed when any static field of $C$ is first accessed. This transformation also introduces a boolean field `initRun` that indicates if lazy initialization has already occurred. Once thread-local state is introduced (Section 2.5), the lazy initialization will occur once per thread. The transformation is performed automatically by *Reentrancer*.

Figure 2(c) shows the example program after the introduction of lazy initialization. Class `Configuration` now declares a field `initRun` (line 92) and a method `lazyInit()` (lines 93–99). Method `lazyInit()` checks if `initRun` is `false`, and if so, sets it to `true` and executes all static initialization code, i.e., the field initializer for `conInfo` and the static block (lines 3 and 5–7 in Figure 1). Furthermore, method `getConInfo()` is transformed to invoke `lazyInit()`, ensuring that the initialization is performed before `conInfo` is first accessed. Class `CheckUserFiles` is transformed similarly for the logger field that was manually introduced by the programmer in Figure 2(a).

Since lazy initialization may alter the point at which initialization code executes, this refactoring step may alter program behavior (e.g., if the initialization code reads a mutable static field in some other class). *Reentrancer* performs a static analysis to reveal such cases and issues warnings to the programmer if needed. This issue will be discussed further in Section 3.2.1.

[4] The static initialization code for class $C$ includes all static initializers declared in class $C$, as well as the initializers associated with `static` fields in class $C$, in textual order.

```
[39] class Configuration { ... } // same as before
[40] class CheckUserFiles {
[41]   private static Logger logger = null;
[42]   public static Logger getLogger(){ return logger; }
[43]   public static void setLogger(Logger l){ logger = l; }
[44]   public static boolean checkDir(String homedir) {
[45]     if (Configuration.getOption("homedir") != null) {
[46]       throw new RuntimeException("homedir already set!");
[47]     }
[48]     Configuration.setOption("homedir", homedir);
[49]     return runAllChecks();
[50]   }
[51]   private static boolean runAllChecks() {
[52]     getLogger().info("running checks..");
[53]     // details omitted..
[54]     return true;
[55]   }
[56] }
[57] public class Tests {
[58]   @Test public void testFoo() {
[59]     CheckUserFiles.setLogger(Logger.getLogger("foo"));
[60]     Assert.assertTrue(CheckUserFiles.checkDir("foo"));
[61]   }
[62]   @Test public void testBar() {
[63]     CheckUserFiles.setLogger(Logger.getLogger("bar"));
[64]     Assert.assertTrue(CheckUserFiles.checkDir("bar"));
[65]   }
[66] }
```

**(a)**

```
[67] class Configuration {
[68]   private static final Map<String,String> conInfo =
[69]     new HashMap<String,String>();
[70]   private static Map<String,String> getConInfo() {
[71]     return conInfo;
[72]   }
[73]   private static final String versionStr = "2.0";
[74]   static {
[75]     getConInfo().put("version", versionStr);
[76]   }
[77]   static void setOption(String name, String value) {
[78]     getConInfo().put(name, value);
[79]   }
[80]   static String getOption(String name) {
[81]     return getConInfo().get(name);
[82]   }
[83] }
[84] class CheckUserFiles { ... } // same as before
[85] public class Tests { ... } // same as before
```

**(b)**

```
[86]  class Configuration {
[87]    private static Map<String,String> conInfo;
[88]    private static Map<String,String> getConInfo() {
[89]      lazyInit(); return conInfo;
[90]    }
[91]    private static final String versionStr = "2.0";
[92]    private static boolean initRun = false;
[93]    private static void lazyInit() {
[94]      if (!initRun) {
[95]        initRun = true;
[96]        conInfo = new HashMap<String,String>();
[97]        getConInfo().put("version", versionStr);
[98]      }
[99]    }
[100]   // methods setOption(), getOption() same as before
[101] }
[102] public class CheckUserFiles {
[103]   private static Logger logger;
[104]   private static boolean initRun = false;
[105]   private static void lazyInit() {
[106]     if (!initRun) {
[107]       initRun = true; logger = null;
[108]     }
[109]   }
[110] ...
```

```
[111] ...
[112]   public static Logger getLogger(){
[113]     lazyInit(); return logger;
[114]   }
[115]   public static void setLogger(Logger l){
[116]     lazyInit(); logger = l;
[117]   }
[118]   // methods checkDir(), runAllChecks() same as before
[119] }
[120] public class Tests { ... } // same as before
```

**(c)**

```
[121] class Configuration {
[122]   private static ThreadLocal<Map<String,String>> conInfo =
[123]     new ThreadLocal<Map<String,String>>();
[124]   private static Map<String,String> getConInfo() {
[125]     lazyInit(); return conInfo.get();
[126]   }
[127]   private static final String versionStr = "2.0";
[128]   private static ThreadLocal<Boolean> initRun =
[129]     new ThreadLocal<Boolean>() {
[130]       protected Boolean initialValue() { return false; }
[131]     };
[132]   private static void lazyInit() {
[133]     if (!(initRun.get())) {
[134]       initRun.set(true);
[135]       conInfo.set(new HashMap<String,String>());
[136]       getConInfo().put("version", versionStr);
[137]     }
[138]   }
[139]   // methods setOption(), getOption() same as before
[140] }
[141] public class CheckUserFiles {
[142]   private static ThreadLocal<Logger> logger =
[143]     new ThreadLocal<Logger>();
[144]   private static ThreadLocal<Boolean> initRun =
[145]     new ThreadLocal<Boolean>() {
[146]       protected Boolean initialValue() { return false; }
[147]     };
[148]   private static void lazyInit() {
[149]     if (!initRun.get()) {
[150]       initRun.set(true); logger.set(null);
[151]     }
[152]   }
[153]   public static Logger getLogger(){
[154]     lazyInit(); return logger.get();
[155]   }
[156]   public static void setLogger(Logger l){
[157]     lazyInit(); logger.set(l);
[158]   }
[159]   // methods checkDir(), runAllChecks() same as before
[160] }
[161] public class Tests { ... } // same as before
```

**(d)**

```
[162] public class Configuration { ... } // same as before
[163] public class CheckUserFiles { ... } // same as before
[164] public class Tests {
[165]   @Test public void testFoo() {
[166]     TestRunner.runInChildThread(new TestRunner() {
[167]       protected void runTest() {
[168]         CheckUserFiles.setLogger(Logger.getLogger("foo"));
[169]         Assert.assertTrue(CheckUserFiles.checkDir("foo"));
[170]       }
[171]     });
[172]   }
[173]   @Test public void testBar() {
[174]     TestRunner.runInChildThread(new TestRunner() {
[175]       protected void runTest() {
[176]         CheckUserFiles.setLogger(Logger.getLogger("bar"));
[177]         Assert.assertTrue(CheckUserFiles.checkDir("bar"));
[178]       }
[179]     });
[180]   }
[181] }
```

**(e)**

Figure 2: Successive refactoring steps applied to make the program of Figure 1 reentrant: (a) the program after replacing an access to `System.out` with the use of decidated `Logger`s; (b) the program after encapsulating field accesses; (c) the program after eliminating of static initializers; (d) the program after replacing global state with thread-local state; (e) the example program after introducing fresh threads. In each step, changes since the previous version of the program are shown underlined.

## 2.5 Introducing Thread-Local State

In the next step, global state is replaced with thread-local state. This is accomplished by wrapping each appropriate static field in a `java.lang.ThreadLocal` object and using the methods `ThreadLocal.get()` and `ThreadLocal.set()` to read/write the value of the wrapped object. As a result of this transformation, each thread will have its own "copy" of each wrapped static field.

For our example, the resulting code is shown in Figure 2(d). In class `Configuration`, the type of field `conInfo` has been changed from `Map<String,String>` to `ThreadLocal<Map<String,String>>` (see line 122), and the read and write of the field have been replaced with calls to `ThreadLocal.get()` and `ThreadLocal.set()` on lines 125 and 135, respectively. Similarly, the type of field `initRun` is wrapped in a `ThreadLocal` object.[5] Moreover, the initial value of the field `initRun` must now be defined by overriding the method `ThreadLocal.initialValue()`. These changes are shown on lines 128–131. Furthermore, the accesses to `initRun` on lines 133 and 134 are transformed appropriately. Class `CheckUserFiles` is transformed similarly, and class `Tests` is not affected by this transformation.

## 2.6 Introducing Threads

The final step of the transformation involves the use of a fresh thread for each execution of the application. This will ensure that each execution observes a different "copy" of the static fields that have been made thread-local and that the code in the `lazyInit()` methods is re-executed for each copy. There are many possible ways to create these fresh threads, and different situations may require different solutions. For example, a stand-alone application may create the threads in its `main()` method, but for a set of tests, it may be desirable to create a special test-runner object that automatically creates a new thread for each test. Therefore, this is a step that must be performed manually.

Figure 2(e) shows how our example program has been modified to create a new thread for each test. This is accomplished by creating a special `TestRunner` that creates a new thread for each test and editing each test to utilize the `TestRunner`. The code for class `TestRunner` is straightforward and the class is provided with our tool; we omit the code here due to space limitations.

The reader may verify that, after this final step of the transformation, the program is reentrant. In particular, the behavior of `testFoo()` and `testBar()` no longer depends on whether they are executed in isolation or not, i.e., `testBar()` passes even if executed after `testFoo()` or if the tests are executed in parallel.

# 3. REFACTORING FOR REENTRANCY

We will now present our refactoring algorithm, including the preconditions checked before the code is transformed. Section 3.1 presents the algorithm for the automatic steps of our refactoring, previously illustrated in Sections 2.3–2.5. Then, Section 3.2 describes static analyses to check preconditions that ensure the refactoring is safe; the user is warned about detected violations before the refactoring is run.

[5]Note that the type `boolean` must be boxed into a type `Boolean`; this is due to the limitations of Java generics, which do not permit parameterized types to be instantiated with primitive types.

## 3.1 The Refactoring

Figure 3 gives pseudocode for the refactoring algorithm. The AddAccessorMethods, IntroduceLazyInit, and IntroduceThreadLocals routines correspond to the steps of the refactoring illustrated in Sections 2.3, 2.4, and 2.5 respectively. The routines are designed to be run in sequence, as shown by the MakeProjectReentrant routine that calls them. Note that the algorithm may not be semantics-preserving if the preconditions to be described in Section 3.2 do not hold for the input program $\mathcal{P}$.

The AddAccessorMethods routine transforms accesses to eligible static fields into calls to generated getter/setter methods (see Figure 2(b) for example output). The procedure returns a mapping $\mathcal{M}$ from types to the generated methods, used when introducing lazy initialization. The MustTransform procedure indicates that a field $f$ must be transformed unless it is both immutable (see Definition 2) *and* initialized to a constant value, e.g., the `versionStr` field at line 4 of Figure 1. The check for a constant value is required to handle cases where $f$ is immutable but its value is obtained via other mutable state, e.g., a hash-consing lookup table $T$.[6]

The IntroduceLazyInit method causes the initialization code for each type $t$ to be run lazily, i.e., the first time any transformed static field in $t$ is accessed (once per thread after thread-local state is introduced). It creates a new method `lazyInit()` for $t$[7] that includes all of $t$'s static initialization code and a check to ensure it is only run once (e.g., see lines 93–99 in Figure 2(c)). Note that to employ lazy initialization, we must remove the **final** modifier from the corresponding fields (line 7); if we desired, we could easily generate code to enforce a write-once property at runtime. Since getters and setters have already been introduced, it suffices to only add calls to `lazyInit()` in those methods (lines 16–17).

Finally, IntroduceThreadLocals transforms eligible static fields to become thread-local using Java's `java.lang.ThreadLocal` type. The routine changes the type `T` of any such field $f$ to `ThreadLocal<T>` and initializes it appropriately.[8] Line 3 checks if $f$ has a field initializer and if so preserves it by overriding `ThreadLocal.initValue()`. Assuming IntroduceLazyInit has run, the only fields with initializers should be the `initRun` flags for guarding lazy initialization (line 2 of IntroduceLazyInit). Finally, all reads and writes of $f$ are transformed to calls to `ThreadLocal.get()` and `ThreadLocal.set()`.

## 3.2 Preconditions

As mentioned, a number of preconditions must hold to ensure the safety our refactoring. Section 3.2.1 discusses restrictions on static initializers, and Section 3.2.2 presents conditions on the usage of libraries. Finally, Section 3.2.3 discusses other potential semantic changes (mostly common to all refactorings) that we currently do not check for.

[6]In this case, $f$ must be transformed since the hash-consing table will be made thread-local, and hence $f$ must be initialized separately for each copy of the table, thereby preserving the behavior of reference-equality checks involving $f$.

[7]We ignore name conflict issues here; they are handled by the implementation.

[8]Note that we use the static initializer for this initialization, but its run-once behavior causes no problems since only `ThreadLocal` fields are initialized.

```
AddAccessorMethods(P)
 1  M ← []  // map from types to field access methods
 2  for t ∈ Types(P)
 3    do for each f ∈ StaticFields(t) ∧ MustTransform(f)
 4         do g ← CreateGetterMethod(f, t)
 5            M[t] ← M[t] ∪ {g}
 6            cr ← CreateMethodCall(g)
 7            for r ∈ FindReads(f, P)
 8              do Replace(r, cr)
 9            if final ∉ Modifiers(f)
10              then s ← CreateSetterMethod(f, t)
11                   M[t] ← M[t] ∪ {s}
12                   cw ← CreateMethodCall(s)
13                   for w ← FindWrites(f, P)
14                     do Replace(w, cw)
15  return M

IntroduceLazyInit(P, M)
 1  for t ∈ Types(P) ∧ M[t] ≠ ∅
 2    do i ← CreateStaticField("boolean", "initRun", "false", t)
 3       l ← CreateStaticMethod("void", "lazyInit", t)
         // Return if 'lazyInit' has run already.
 4       AddIfCondition(l, i)
 5       for m ∈ Members(t)
 6         do if m ∈ StaticFields(t) ∧ MustTransform(m)
 7              then RemoveModifier(final, m)
 8                   FI ← GetFieldInitialization(m)
 9                   AddStatements(FI, l)
10                   RemoveInitialization(m)
11            elseif m ∈ StaticBlocks(t)
12              then SB ← GetStatements(m)
13                   AddStatements(SB, l)
14                   Remove(m)
15       c ← CreateMethodCall(l)
16       for m ∈ M[t]
17         do InsertStatementToFront(c, m)

IntroduceThreadLocals(P)
 1  for each f ∈ StaticFields(P) ∧ MustTransform(f)
 2    do wrap type of f with type ThreadLocal
 3       if f has a field initializer
 4         then init f to new ThreadLocal
                with overridden initValue()
 5         else  init f to new ThreadLocal
 6       g ← CreateMethodCall(f, "get()")
 7       s ← CreateMethodCall(f, "set()")
 8       for a ∈ Reads(f, P)
 9         do Replace(a, g)
10       for a ∈ Writes(f, P)
11         do Replace(a, s)

MustTransform(f)
1  return IsMutable(f) ∨ ¬ InitToConstant(f)

MakeProjectReentrant(P)
1  M ← AddAccessorMethods(P)
2  IntroduceLazyInit(P, M)
3  IntroduceThreadLocals(P)
```

Figure 3: Refactoring pseudocode.

### 3.2.1 Static Initializers

Static initialization semantics can make our transformation to lazy initialization unsafe, since the refactoring may change when static initialization code executes. In Java, several events can cause the static initializer for a type $T$ to be run, including allocation of a $T$ object, invocation of a static method in $T$, access of a non-constant static field of $T$ (a field is constant if it is final and initialized to a compile-time constant [15, 4.12.4]), or an initialization event for a subclass of $T$ [15, 12.4.1]. After applying our refactoring, type $T$ is initialized lazily, i.e., when some static field of $T$ is accessed. Therefore, the refactoring may change the initialization point of $T$ (e.g., if some allocation of a $T$ object previously caused its initialization). Since static initializers may execute arbitrary code, this altered timing may change program behavior in unexpected ways.

```
[182] public class Main {
[183]   public static void main(String[] args){
[184]     B b = new B();
[185]     C.y = 5;
[186]     System.out.println(B.x);
[187]   }
[188] }
[189] class B {
[190]   static int x = C.y;
[191] }
[192] class C {
[193]   static int y = 0;
[194] }
```

Figure 4: Illustration of fragile static initializers.

For example, consider the program of Figure 4. As written, the allocation of a `B` object at line 184 causes `B`'s static initializer (line 190) to run. The initializer accesses `C.y`, which in turn causes `C`'s initializer (line 193) to run. Hence, the `B.x` field gets the value 0, which is printed on line 186. With the lazy initialization introduced by our refactoring, `B`'s initializer would not run until line 186 (due to the read of `B.x`), and hence because of the write to `C.y` at line 185, `B.x` will get the value 5, a change in behavior.

Cyclic dependences between static initializers may also make our refactoring unsafe. Consider this example:

```
class D { static final int x = E.y + 1; }
class E { static final int y = D.x + 1; }
```

The static initializers of `D` and `E` are mutually dependent. If `E` is initialized first, it will cause `D`'s initializer to run. Since `E`'s initialization is not complete, `D`'s initializer reads 0 from `E.y`, yielding `D.x = 1` and `E.y = 2` when initialization completes. In contrast, if `D` were initialized first, at completion `D.x` would be 2 and `E.y` would be 1. Lazy initialization does not preserve this fragile behavior.

We use a static analysis to detect if static initializers in a program may cause our refactoring to change its behavior, and our tool issues a warning if this is the case. Assume for the moment that the input program $\mathcal{P}$ for the refactoring uses no library code, i.e., there is no code that cannot be transformed (we will discuss libraries in Section 3.2.2). Then, the change to lazy initialization is safe if the following two conditions hold:

1. There are no cyclic dependences between initializers in $\mathcal{P}$, where initializer $s_1$ is dependent on initializer $s_2$ if execution of $s_1$ may cause $s_2$ to run. Such dependences can lead to fragile behavior not preserved by the transformation, as discussed above.

2. For any static initializer $s$ for some class $C$, the only mutable static fields (see Definition 2) accessed by $s$

and its explicit transitive callees must be in $C$. Explicit callees do *not* include other static initializers invoked due to classloading during execution of $s$. This condition flags programs such as that of Figure 4 where `B`'s static initializer reads the mutable static field `C.y`.

The exclusion of static initializers invoked due to classloading in condition (2) is important for avoiding false positive warnings. Without the exclusion, the analysis would emit a warning every time a static initializer for some class $C$ referred to another class $D$ with an initializer, even if $D$'s initializer only initialized immutable fields in $D$. Other static initializers are safe to exclude since if conditions (1) and (2) hold, all static initializers can only read immutable static fields from other classes, and hence their relative execution ordering can no longer affect behavior. (Note that condition (1) is important, since with cyclic initializer dependences a `final` static field can be read before its initialization completes, as illustrated earlier.)

A possible alternative to our lazy initialization transformation would be to preserve the timing of initialization in the refactored code via insertion of explicit calls at each point the initializer could have run. We rejected this approach for two reasons. First, preserving initializer timings at the source level can require ugly code transformations in some cases.[9] Second, we believe that true violations of our precondition checks (e.g., the code of Figure 4) reflect highly fragile code that should be avoided independent of our refactoring.

### 3.2.2 Library Usage

Certain usage of library code can make our refactoring unsafe because (i) static fields and initializers in the libraries cannot be transformed, and (ii) libraries enable access to the environment (user input, the file system, etc.), which can be viewed as another form of mutable global state. We will now discuss the checks we perform to warn about potentially unsafe uses of library code.

Ideally, our analysis would check for the following conditions regarding library usage, which would ensure that libraries do not affect the safety of our refactoring:

- Static initializers for all library classes used by the application must pass the check of Section 3.2.1. This would ensure that the lazy initialization transformation does not change the behavior of initializers in the library.
- The program, including used library code, must not access any mutable static fields in the library or mutable environment state through library calls (e.g., files). We must prohibit accesses to such state since it cannot be made thread-local.

Unfortunately, it is not practical to check the above conditions precisely in a realistic tool. First, the notion of mutability becomes murky when dealing with environmental inputs like files. Consider a file containing configuration settings with unchanged contents during execution. The file is "immutable" as far as the program is concerned, but it would be very difficult for a tool to discover this fact without annotations. Second, even reasoning about what library code is executed by an application is difficult in a refactoring tool—precise call graph construction algorithms (which require precise pointer analysis) are not sufficiently scalable.

Our tool warns the user about potentially unsafe uses of library methods via a blacklisting approach. Our blacklist includes potentially problematic library methods that access files, read system properties, paint GUI objects, manipulate threads, etc. We flag any call to a blacklisted method from the application as dangerous and warn the user. We also flag all accesses of library mutable static fields from the application as potentially dangerous. While these checks are not complete in that they do not flag all unsafe uses of the library, in our experience they have been sufficient and they can be implemented scalably.

### 3.2.3 Other Issues

Our transformation may not preserve the exceptional behavior of a program. For example, if a static initializer $s$ throws an exception, making $s$ run lazily may change the stack trace of the exception, or the exception may not be thrown at all (e.g., if the corresponding static fields are never accessed.) Many other refactorings (e.g., extracting an expression to a local variable) do not preserve exceptional behavior, and typically this does not pose problems.

Also note that our transformation may change the semantics of combining results from different executions of $\mathcal{P}$. Say, e.g., that $\mathcal{P}$ is already reentrant and that it hash-conses objects of type `T` via a global lookup table. After our refactoring, $\mathcal{P}'$ will perform thread-local hash-consing of `T` objects, and hence equality checks on `T` objects obtained from distinct threads may not operate as before. If results of different reentrant executions must be combined programmatically, the developer must ensure that our refactoring does not transform state such as hash-consing tables.

There are other corner cases in which our refactoring may not preserve program semantics, e.g., if the program uses reflection or native code. Such issues are common to all refactorings, and it is therefore standard practice to rely on a test suite to ensure that important behavior has not changed [9]. All benchmarks used in our evaluation (see Section 4.2) had a test suite which we used to check the safety of our refactoring.

## 4. IMPLEMENTATION AND EVALUATION

We implemented our refactoring in a tool called *Reentrancer* as an extension to the Eclipse Java development tools (JDT). Section 4.1 discusses some details of the implementation and Section 4.2 presents our evaluation of *Reentrancer* on several single-threaded, non-reentrant benchmarks.

## 4.1 Implementation

*Reentrancer*'s functionality shows up as a Make Project Reentrant refactoring in the `Refactor` menu. Similar to Infer Generic Type Arguments [11], *Reentrancer* targets a whole Java project rather than one program element.

*Reentrancer*'s analyses for checking the refactoring preconditions are implemented using the WALA framework [21]. We use WALA primarily to compute call graphs from static initializers to discover their transitive callees (see Section 3.2). The implementation reflects the discussion in

[9]For example, consider a constructor that explicitly invokes its superclass constructor via `super`. To preserve initializer timings, the transformed code must run the subclass static initializer before the superclass constructor. However, an explicit `super` call must appear first in a constructor.

Section 3.2, and *Reentrancer* issues a warning for each violation. The user is allowed to proceed with the refactoring in spite of the warnings, since some may be false positives.

The MAKE PROJECT REENTRANT refactoring is composed of several smaller refactoring steps, reflecting the transformation of the motivating example in Section 2.

**Encapsulating fields.** We use the ENCAPSULATE FIELD refactoring (see [9, p.206]) from Eclipse to encapsulate accesses to mutable[10] or non-constant static fields (see Section 3.1). Such fields may be declared in Java interface types, which cannot have methods with bodies; in this case, we first move the fields to a fresh abstract class. Several pragmatic issues may arise in the process of encapsulating fields, including name clashes for getter/setter methods (handled by renaming) and the transformation of field accesses in postfix expressions, e.g., `var++` (handled by introducing temporary variables).

**Introducing lazy initialization.** This step is a straightforward implementation of the INTRODUCELAZYINIT algorithm of Section 3.1. One notable issue is that certain forms of array initializers can only be used in variable declarations, e.g., `int[][] foo = {{1,2},{3,4}}`; we introduce new array creation expressions before moving them into `lazyInit()` (e.g., `int[][] foo = new int[]{new int[]{1,2},new int[]{3,4}}`).

**Introducing thread-locals.** This step closely reflects algorithm INTRODUCETHREADLOCALS of Section 3.1. As discussed in Section 2.5, we must "box" any field of primitive type into the corresponding reference type before wrapping the type with `java.lang.ThreadLocal`. Similarly, primitive literals have to be replaced with their boxed equivalents.

## 4.2 Evaluation

We evaluated our refactoring by observing its behavior on several single-threaded, non-reentrant benchmarks. For each benchmark, we first confirmed that reentrancy problems existed, either by observing failures when running existing tests in parallel or by writing new tests that exposed problems. We then ran our refactoring, ensured that the preconditions were met for each of the benchmarks, and confirmed that the original tests still passed. Finally, we confirmed that the observed reentrancy problems were fixed in the refactored version of the benchmark and checked if running tests in parallel on a dual-core processor yielded a performance benefit.

Table 1 describes our benchmarks,[11] including a short description, the version used, and the size in thousands of lines of code. The table also shows a number of size measures for each benchmark, including the number of types, methods, and fields, the number of static initializer blocks, the number of fields with static initializer expressions, and the total number of tests (and where relevant, the number of tests we added to expose reentrancy problems is shown in parentheses). For `bcel`, we chose a recent unreleased version since it included a test suite. We manually unwrapped two existing thread-local fields in `bcel`; they were presumably made thread-local in an aborted attempt to add thread safety, as the `bcel` documentation explicitly states that it is not thread safe [1]. For `coco/r`, we chose an old version explicitly labeled as non-reentrant [2]. We used a version of `xml-security` from the Software-Artifact Infrastructure Repository [7]. We made small modifications to the benchmarks to work around bugs in the Eclipse refactorings we rely on (see Section 4.1).

Table 2 presents our results. We first discuss the **Warnings** column, which states the number of warnings issued by *Reentrancer* as $a(b)/c$: $a$ denotes the number of warnings for accessing external mutable state from a static initializer (Section 3.2.1), $b$ is the number of warnings from $a$ where the accessed state is truly mutable, and $c$ is the number of dangerous library usage warnings (Section 3.2.2). We did not find any instances of circular initializer dependences in the benchmarks. For `bcel` and `wala`, we currently report many false warnings (the difference between $a$ and $b$) due to an insufficiently strong check for immutable static fields (described in Section 4.1). Arrays are particularly problematic, as the immutability of array contents cannot be expressed with the Java type system. More sophisticated mutability inference techniques [18, 23] may reduce the false positive rate.

For these benchmarks, we determined that all the $b$ warnings were safe due to the structure of the code. These accesses were often similar to the following:

```
class A {
  static Map cache = new HashMap();
  public static A findOrCreateA(String s) {
    A result = cache.get(s);
    if (result == null) { result = new A(s);
                          cache.put(s, result); }
    return result; } }
class B {
  static final A a = A.findOrCreateA("test"); }
```

Here, the static initializer of `B` accesses the mutable static field `A.cache` by calling `findOrCreateA()`. While making `B`'s initializer lazy could affect whether `B.a` gets a cached or fresh `A` object, it should not otherwise affect behavior.

We also inspected all the dangerous library usage warnings $c$ and found that none required a code change in order to make the tests for the benchmarks pass. The output of some print statements could be interleaved in unexpected ways in parallel executions; such behavior could be fixed by using loggers, as in Section 2.2. Some library usage required careful invocation of the benchmarks in parallel executions; e.g., `javacc` writes several output files, and different executions must specify different output directories for the files.

The next three columns of Table 2 measure the amount of code changed by the refactoring. The `LOC changed` column gives the number of lines of code changed. The `#methods added/changed` column states the number of methods introduced by *Reentrancer* (e.g., `lazyInit()` methods) and the number of existing methods changed due to insertion of getter/setter calls. Finally, the `fields added/wrapped` column counts the number of `initRun` fields added and the number of fields changed to be `ThreadLocal`. From these results, it is clear that the transformation involves significant

[10] We determine if a static field $f$ is immutable (see Definition 2) in a type-based manner: $f$ must be **final**, and either the declared type of $f$ must be an immutable class [23] (e.g., `java.lang.String`) or $f$ must be initialized to an object of immutable class type.

[11] Note that we had fairly stringent criteria for benchmarks; they had to be single-threaded, had to have reentrancy issues, and had to have a test suite to enable checking for regressions.

| Benchmark | Description | Version | kLOC | #types/ #methods/ #fields | #static init blocks | #static field inits | #tests |
|---|---|---|---|---|---|---|---|
| `coco/r` | Parser generator [2] | Non-reentrant | 3.3 | 23/210/191 | 0 | 71 | 15 |
| `xml-security` | XML security checks [22] | 1.0.71 | 31 | 199/1555/574 | 3 | 454 | 86 (2) |
| `bcel` | Bytecode transformation [1] | SVN 694866 | 33 | 454/3562/804 | 3 | 612 | 81 (6) |
| `javacc` | Parser generator [13] | 4.2 | 39 | 156/2106/697 | 20 | 268 | 29 |
| `wala` | Program analysis [21] | SVN 3243 | 83 | 1118/8413/2538 | 16 | 1356 | 166 |

Table 1: Benchmark characteristics.

| Benchmark | Warnings | LOC changed | #methods added/changed | #fields added/wrapped | processing time | running time (i)/(ii)/(iii) |
|---|---|---|---|---|---|---|
| `coco/r` | 1(1)/10 | 1680 | 168/168 | 16/77 | 111 | 0.61/0.34/0.24 |
| `xml-security` | 26(26)/28 | 2321 | 380/420 | 90/145 | 328 | 14.5/14.6/14.1 |
| `bcel` | 14(1)/29 | 2723 | 472/435 | 34/219 | 447 | 0.50/0.73/0.66 |
| `javacc` | 0(0)/20 | 7583 | 654/537 | 26/316 | 492 | 1.40/2.41/2.13 |
| `wala` | 82(10)/41 | 5795 | 988/838 | 165/423 | 888 | 380/416/338 |

Table 2: Results for the benchmarks of Table 1. All times are in seconds.

amounts of code change—making all these changes manually would be a tedious and error-prone undertaking. Note that as previously mentioned, our tool's immutability checking is sometimes weak, and hence it may transform immutable fields unnecessarily; with better immutability inference, the amount of change may decrease for some benchmarks.

The **processing time** column in Table 2 states how long it took *Reentrancer* to refactor each benchmark. *Reentrancer* currently requires nearly 15 minutes to process the largest of the benchmarks; however, the implementation is currently completely untuned, and we expect to be able to improve the running time of our tool significantly.

In all cases, our refactoring fixed the observed reentrancy problems with the benchmarks. For `bcel` and `xml-security`, the additional tests we added to expose reentrancy issues passed after the refactoring when run in fresh threads. For `wala` and `javacc`, some tests originally failed in a parallel run of the test suite, but all passed after the refactoring.

`coco/r` differed from our other benchmarks in that the program could not be invoked multiple times in the same JVM at all, since many static fields were never reset. Before applying our refactoring, we ran `coco/r`'s test suite in a single JVM instance by using a fresh class loader per execution. After the refactoring, running each execution in a fresh thread was sufficient. Note that removing the fresh class loaders led to a performance *improvement* after our refactoring, discussed further below.

The final **running time** column in Table 2 states the time required for running the test suite associated with each benchmark for the following 3 scenarios:[12] (i) execution time for the test suite for the unrefactored benchmark on a single core, (ii) execution time for the test suite for the refactored benchmark on a single core, and (iii) execution time for the test suite for the refactoring benchmark on two cores. Hence, comparing (ii) and (i) indicates the slowdown caused by the refactoring (introduction of thread-locals, getters and setters, etc.), and comparing (iii) to $min$((i),(ii)) indicates the speedup that is *enabled* by our refactoring by switching from one core to two cores.[13]

As can be seen from the results for (i) and (ii), the performance overhead of introducing thread-locals varies widely and is often not large, though it can be significant (e.g., `javacc` tests run 1.72X slower). For frequently accessed static fields, the transformation to thread-locals could be undone manually, yielding improved performance while retaining the advantages of thread-locals elsewhere. The refactoring yielded a 44% speed *improvement* for `coco/r`, as we were able to replace the fresh classloader per execution required for that benchmark (discussed above) with a fresh thread, which had lower overhead.

For three of the benchmarks, the (iii) numbers show we were able to speed up the test suite through a parallel run without any modifications to the refactored code. The speedups were 29% for `coco/r`, 11% for `wala`, and 3% for `xml-security`[14]. For `bcel` and `javacc`, the overheads from the refactoring outweighed the benefits of a parallel run on two processors. (With more processors, the parallelism benefits may be greater.) We strongly suspect that further improvements in parallel running times are possible by replacing some thread-local state with properly synchronized shared state.

## 5. RELATED WORK

The term 'reentrant' is used with various slightly different meanings in the literature, each alluding to the fact that a process attempting to "re-enter" a function concurrently can do so safely. Our use of the term is most consistent with the notion of reentrant procedures in systems programming [19, p.49]. We are only concerned with reentrancy of external calls to API methods of a program, not recursive

[12] In each case, we ran the test suite 5 times, discarded the fastest and the slowest run, and computed the average of the remaining 3 runs

[13] All tests are run on a MacBook Pro with a 2.6GHz Intel Core 2 Duo processor and 4GB RAM.

[14] All percentages compare the fastest single-threaded run (either (i) or (ii)) with a parallel run.

method calls internal to the program (work by Fähndrich et al. focuses on the latter issue for object invariants [8]). Also, since we focus on user-level programs running on a JVM, we are not concerned with low-level issues such as self-modifying code and interrupts that systems programmers must address. Our work does *not* concern reentrant locks, which are locks that can be acquired multiple times by the same thread without blocking (like Java's implicit monitor locks [15]).

A related concept to reentrancy is that of a *thread-safe* function, commonly used to mean a function that protects shared resources from concurrent access via locks to avoid data races and other concurrency errors. Reentrancy implies thread-safety, but thread-safety may not imply reentrancy because even the behavior of sequential executions may interfere with each other through global state.

Separate class loaders are often used to introduce isolation and reentrancy in Java programs, e.g., for J2EE servlets. Much work has been done on reducing the overhead of running multiple isolated programs on a single JVM [14]. As discussed in Section 1, a major disadvantage of this approach is the difficulty of selectively introducing shared state to improve performance. Also, on standard JVMs, a fresh class loader per execution may introduce more overhead than a fresh thread, as indicated by the results for `coco/r` benchmark in our evaluation (Section 4.2).

In recent years, significant advances have been made in the area of automated tool support for refactoring [4, 12, 16, 17]. However, to our knowledge, we are the first to present semi-automatic refactoring support for making existing Java programs reentrant.

Dig et al. [5] present work on refactoring sequential Java programs for concurrency using the `java.util.concurrent` utilities. Two of their refactorings (converting `int` to `AtomicInteger` and converting `HashMap` to `ConcurrentHashMap`) make shared data accesses thread-safe, but they do not necessarily make the program reentrant. Their work complements ours: our technique for introducing thread-local state could be used for state that need not be shared among executions, while their technique could be used to share other state safely.

Recently, Frigo et al. have introduced *hyperobjects* as a language construct to aid in parallelization of code that uses global variables [10]. Their holder hyperobjects are a generalization of thread-local variables. Our refactoring could be applied to a language with hyperobjects by replacing global variables with holders and then allowing the programmer to employ other hyperobjects like reducers as needed.

# 6. CONCLUSIONS AND FUTURE WORK

We presented a refactoring that makes single-threaded programs reentrant by replacing global state with thread-local state and static initialization with explicit lazy initialization. The refactoring enables a "correctness before performance" approach to deploying such programs on multicore machines: the refactored program is obviously safe for multicores, and it can be optimized by selectively introducing shared state. The refactoring is implemented in a tool called *Reentrancer* in the context of Eclipse JDT. The refactoring is mostly automatic, and warnings are given for issues that may require manual intervention. We used *Reentrancer* to refactor several single-threaded Java applications with observed reentrancy problems and demonstrated that these problems are eliminated by the refactoring and that the refactoring alone can enable parallel speedups.

Future work includes the use of more sophisticated immutability checking [18] in order to reduce the number of fields that need to be wrapped into thread-local objects. Also, we aim to develop an incremental workflow for our refactoring in which only small amounts of code are transformed at a time, making the transformations easier for developers to understand.

*Acknowledgments.* We thank the anonymous reviewers for their detailed comments and Doug Lea, Stephen Fink, and David Grove for early discussions about this work.

# References

[1] BCEL: The Byte Code Engineering Library. http://jakarta.apache.org/bcel/.

[2] Static version of Coco/R. http://www.ssw.uni-linz.ac.at/coco/static.html.

[3] C# language specification, 2006. http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-334.pdf.

[4] D. Dig. *Automated upgrading of component-based applications*. PhD thesis, University of Illinois at Urbana-Champaign, 2007.

[5] D. Dig, J. Marrero, and M. Ernst. Refactoring sequential Java code for concurrency. In *Proc. ICSE'09*, 2009.

[6] D. Dig, J. Marrero, and M. D. Ernst. How do programs become more concurrent? a story of program transformations. Technical Report MIT-CSAIL-TR-2008-053, MIT CSAIL, Cambridge, MA, Sept. 5, 2008.

[7] H. Do, S. Elbaum, and G. Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Soft. Eng.*, 10(4), 2005.

[8] M. Fähndrich, D. Garbervetsky, and W. Schulte. A static analysis to detect re-entrancy in object oriented programs. *J. Object Technology*, 7(5), 2008.

[9] M. Fowler. *Refactoring. Improving the Design of Existing Code*. Addison-Wesley, 1999.

[10] M. Frigo, P. Halpern, C. E. Leiserson, and S. Lewin-Berlin. Reducers and other Cilk++ hyperobjects. In *Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2009.

[11] R. Fuhrer, F. Tip, A. Kieżun, J. Dolby, and M. Keller. Efficiently refactoring Java applications to use generic libraries. In *Proc. ECOOP'05*, pages 71–96, Glasgow, Scotland, 2005.

[12] W. G. Griswold. *Program Restructuring as an Aid to Software Maintenance*. PhD thesis, University of Washington, 1991. Tech. Rep. 91-08-04.

[13] JavaCC. https://javacc.dev.java.net/.

[14] M. Jordan, G. Czajkowski, K. Kouklinski, and G. Skinner. Extending a J2EE server with dynamic and flexible resource management. In *Proc. 5th ACM/IFIP/USENIX Int. Conf. on Middleware (Middleware'04)*, pages 439–458, 2004.

[15] D. H. Ken Arnold, James Gosling. *The Java Language Specification*. Prentice Hall, 3rd edition, 2005.

[16] T. Mens and T. Tourwé. A survey of software refactoring. *IEEE Trans. on Softw. Eng.*, 30(2):126–139, Feb. 2004.

[17] W. F. Opdyke. *Refactoring Object-Oriented Frameworks*. PhD thesis, University Of Illinois at Urbana-Champaign, 1992.

[18] J. Quinonez, M. S. Tschantz, and M. D. Ernst. Inference of reference immutability. In *Proc. ECOOP'08*, pages 616–641, Paphos, Cyprus, 2008.

[19] W. Stallings. *Operating Systems: Internals and Design Principles*. Prentice Hall, 2008.

[20] M. S. Tschantz and M. D. Ernst. Javari: Adding reference immutability to Java. In *Proc. OOPSLA'05*, San Diego, CA, 2005.

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

[22] Apache XML Security. http://santuario.apache.org/.

[23] Y. Zibin, A. Potanin, M. Ali, S. Artzi, A. Kieżun, and M. D. Ernst. Object and reference immutability using Java generics. In *Proc. ESEC/FSE'07*, 2007.