Dynamic Test Generation with Static Fields and Initializers

Maria Christakis, Patrick Emmisberger, and Peter Müller

Department of Computer Science,
ETH Zurich, Switzerland
{maria.christakis,peter.mueller}@inf.ethz.ch, empatric@student.ethz.ch

Abstract. Static state is common in object-oriented programs. However, automatic test case generators do not take into account the potential interference of static state with a unit under test and may, thus, miss subtle errors. In particular, existing test case generators do not treat static fields as input to the unit under test and do not control the execution of static initializers. We address these issues by presenting a novel technique in automatic test case generation based on static analysis and dynamic symbolic execution. We have applied this technique on a suite of open-source applications and found errors that go undetected by existing test case generators. Our experiments show that this problem is relevant in real code, indicate which kinds of errors existing techniques miss, and demonstrate the effectiveness of our technique.

1 Introduction

In object-oriented programming, data stored in static fields is common and potentially shared across the entire program. In case developers choose to initialize a static field to a value different from the default value of its declared type, they typically write initialization code. The initialization code is executed by the runtime environment at *some* time prior to the first use of the static field. The time at which the initialization code is executed depends on the programming language and may be chosen non-deterministically, which makes the semantics of the initialization code non-trivial, even to experienced developers.

In C#, initialization code has the form of a static initializer, which may be inline or explicit. The C# code on the right shows the difference: field `f0` is initialized with an inline static initializer, and field `f1` with an explicit static initializer. If any static initializer exists, inline or explicit, the C# compiler always generates an explicit initializer. This compiler-generated explicit initializer first initializes the static fields of the class that are assigned their initial value with inline initializers and then incorporates the code of the original explicit initializer (if any) written by the developer, as shown below for class `C`.

```
class C {
  // inline
  static int f0 = 19;
  static int f1;

  // explicit
  static C() {
    f1 = 23;
  }
}
```

B. Bonakdarpour and S.A. Smolka (Eds.): RV 2014, LNCS 8734, pp. 269–284, 2014.

```
// compiler-generated
static C() {
  f0 = 19;
  f1 = 23;
}
```

However, the semantics of the compiler-generated static initializer depends on whether the developer has indeed written an explicit initializer. If this is the case, the compiler-generated initializer has *precise* semantics: the body of the initializer is executed (*triggered*) exactly on the first access to any (non-inherited) member of the class (that is, static field, static method, or instance constructor). Otherwise, the compiler-generated initializer has *before-field-init* semantics: the body of the initializer is executed no later than the first access to any (non-inherited) static field of the class [3]. This means that the initializer could be triggered by the runtime environment at any point prior to the first static-field access.

In Java, static (initialization) blocks are the equivalent of explicit static initializers with precise semantics in C# [8]. In C++, static initialization occurs before the program entry point in the order in which the static fields are defined in a single translation unit. However, when linking multiple translation units, the order of initialization between the translation units is undefined [2].

Even though static state is common in object-oriented programs and the semantics of static initializers is non-trivial, automatic test case generators do not take into account the potential interference of static state with a unit under test. They may, thus, miss subtle errors. In particular, existing test case generators do not solve the following issues:

1. *Static fields as input*: When a class is initialized before the execution of the unit under test, the values of its static fields are part of the state and should, thus, be treated as inputs to the unit under test. Existing tools fail to do that and may miss bugs when the unit under test depends on the values stored in static fields (for instance, to determine control flow or evaluate assertions).
2. *Initialization and uninitialization*: Existing tools do not control whether static initializers are executed before or during the execution of the unit under test. The point at which the initializer is executed may affect the test outcome since it may affect the values of static fields and any other variables assigned to by the static initializer. Ignoring this issue may cause bugs to be missed. A related issue is that existing tools do not undo the effect of a static initializer between different executions of the unit under test such that the order of executing tests may affect their outcomes.
3. *Eager initialization*: For static initializers with before-field-init semantics, a testing tool should not only control whether the initializer is run before or during test execution; in the latter case, it also needs to explore all possible program points at which initialization of a class may be triggered (non-deterministically).
4. *Initialization dependencies*: The previous issues are further complicated by the fact that the order of executing static initializers may affect the resulting state due to their side effects. Therefore, a testing tool needs to consider all relevant execution orders in order not to miss bugs.

We address these issues by designing and implementing a novel technique in automatic test case generation based on dynamic symbolic execution [7] (*concolic testing* [13]) and static analysis. Our technique treats static fields as input to the unit under test and systematically controls the execution of static initializers. The dynamic symbolic execution collects constraints describing the static-field inputs that will cause the unit under test to take a particular branch in the execution or violate an assertion. It also explores the different program points at which a static initializer might be triggered. The static analysis improves performance by pruning program points at which the execution of a static initializer does not lead to any new behaviors of the unit under test.

We have implemented our technique as an extension to the testing tool Pex [14] for .NET. We have applied it on a suite of open-source applications and found errors that go undetected by existing test case generators. Our results show that this problem is relevant in real code, indicate which kinds of errors existing techniques miss, and demonstrate the effectiveness of our technique.

Related Work. Most existing automatic test case generation tools ignore the potential interactions of a unit under test with static state. These tools range from random testing (like JCrasher [1] for Java), over feedback-directed random testing (like Randoop [10] for Java), to symbolic execution (like Symbolic Java PathFinder [11]) and dynamic symbolic execution (like Pex for .NET or jCUTE [12] for Java).

To the best of our knowledge, existing testing tools such as the above do not take into account the interference of static state with a unit under test, with the exception of JCrasher. JCrasher ensures that each test runs on a "clean slate"; it resets all static state initialized by any previous test runs either by using a different class loader to load each test, or by rewriting the program under test at load time to allow re-initialization of static state. Nevertheless, JCrasher does not address the four issues described above.

Unit testing frameworks, like NUnit for .NET and JUnit for Java, require the tester to manage static state manually in set-up methods in order to ensure the clean execution of the unit tests. Therefore, the tester must be aware of all interactions of the unit under test with static state. As a result, these frameworks become significantly less automatic for unit tests that interact with static state.

Static analysis tools for object-oriented languages, such as Clousot [5] for .NET and ESC/Java [6] for Java, do not reason about static initialization. An extension of Spec# [9] supports static verification in the presence of static initializers, but requires significant annotation overhead.

We are, therefore, not aware of any tool that automatically takes static state into account and detects the kinds of errors described in this paper.

Outline. Sect. 2 explains how we explore static input state where all relevant classes are initialized. Sects. 3 and 4 show we handle static initializers with precise and before-field-init semantics, respectively. Sect. 5 demonstrates the effectiveness of this technique by applying it on a suite of open-source applications.

2 Static Fields as Input

In this section, we address the issue of treating static fields of *initialized* classes as input to the unit under test. The case that a class is not yet initialized is discussed in the next two sections.

```
1 public class C {
2   public static int F;
3
4   static C() {
5     F = 0;
6   }
7
8   public static void M() {
9     F++;
10    if (F == 2) abort;
11  }
12 }
```

Fig. 1. A C# method accessing static state. To cover all branches, dynamic symbolic execution must treat static field `F` as an input to method `M` and collect constraints on its value.

The example in Fig. 1 illustrates the issue. Existing automatic test case generators do not treat static field `F` of class `C` as input to method `M`. In particular, testing tools based on dynamic symbolic execution generate only one unit test for method `M` since there are no branches on a method parameter of `M`. Since the body of method `M` contains a branch on static field `F` (line 10), they achieve low code coverage of `M` and potentially miss bugs.

Dynamic Symbolic Execution. To address this issue, we treat static fields as inputs to the method under test and assign to them symbolic variables. This causes the dynamic symbolic execution to collect constraints on the static fields and use them to generate inputs that force the execution to explore all branches in the code. As usual with the automatic generation of unit tests, these generated inputs might not occur in any actual execution of the program; to avoid false positives, developers may write specifications (preconditions or invariants) that further constrain the possible values of these inputs.

Treating *all* static fields of a program as inputs is not practical. It is also not modular and defeats the purpose of unit testing. Therefore, we determine at runtime which static fields are read during the execution of a unit test and treat only those as inputs to the unit under test.

We implement this approach in a procedure DSE(UUT, IC), which performs dynamic symbolic execution of the unit under test UUT. IC is the set of classes that have been initialized *before* the execution of the unit under test. For all other classes, initialization may be triggered *during* the execution of the generated unit

tests. The DSE procedure treats the static fields of all classes in the IC set as symbolic inputs. It returns the set TC of classes whose initialization is triggered during the execution of the generated unit tests. The static fields of the classes in $IC \cup TC$ include all static fields that are read by the unit tests. We call the DSE procedure repeatedly to ensure that the static fields of all of these classes are treated as inputs to the unit under test. The precise algorithm for this exploration as well as more details of the DSE procedure are described in the next section.

Consider the dynamic symbolic execution DSE(`M`, {}) of method `M` from Fig. 1. This dynamic symbolic execution generates one unit test that calls method `M`. The execution of this unit test triggers the initialization of class `C` due to the access to static field `F` (line 9). Therefore, procedure DSE returns the singleton set {`C`}. As a result, our exploration algorithm will call DSE(`M`, {`C`}). This second dynamic symbolic execution treats static field `F` as a symbolic input to method `M` and collects constraints on its value. For instance, assuming that the first unit test of the second dynamic symbolic execution executes `M` in a state where `F` is zero, the conditional statement introduces the symbolic constraint $\neg(\texttt{F} + 1 = 2)$. The dynamic symbolic execution subsequently negates and solves the symbolic constraints on `M`'s inputs. Consequently, a second unit test is generated that first assigns the value 1 to field `F` and then calls `M`. The second unit test now reaches the `abort` statement and reveals the bug. We will see in the next section that, even though the second call to DSE is the one that explores the unit under test for different values of static field `F`, the first call to DSE is also important; besides determining which static fields should be treated symbolically, it is also crucial to handle uninitialized classes.

3 Initialization with Precise Semantics

In the previous section, we addressed the issue of treating static fields of initialized classes as input to the unit under test. In this section, we explain how our technique (1) controls the execution of static initializers and (2) explores executions that trigger static initializers. Here, we consider only static initializers with precise semantics; initializers with before-field-init semantics are discussed in the next section.

3.1 Controlling Initialization

In order to explore the interaction between a unit under test and static initializers, we must be able to control for each execution of a unit test which classes are initialized before the execution of the unit test and which ones are not. This could be achieved by restarting the runtime environment (virtual machine) before each execution of a unit test and then triggering the initialization of certain classes. To avoid the high performance overhead of this naïve approach, we instrument the unit under test such that the execution *simulates* the effects of triggering an initializer and restarting the runtime environment.

Initialization. We insert calls to the dynamic symbolic execution engine at all points in the entire program where a static initializer could be triggered according to its semantics. For static initializers with precise semantics, we insert instrumentation calls to the dynamic symbolic execution engine on the first access to any (non-inherited) member of their class. Where to insert these instrumentation calls is determined using the inter-procedural control-flow graph of the unit under test. This means that we might insert an instrumentation call at a point in the code where, along certain execution paths, the corresponding class has already been initialized. Note that each .NET bytecode instruction triggers at most one static initializer; therefore, there is at most one instrumentation call at each program point.

For an exploration DSE(UUT, IC), the instrumentation calls in UUT have the following effect. If the instrumentation call is made for a class C that is in the IC set, then C has already been initialized before executing UUT and, thus, the instrumentation call has no effect. Otherwise, if this is the first instrumentation call for C in the execution of this unit test, then we use reflection to explicitly invoke C's static initializer. That is, we execute the static initializer no matter if the runtime environment has initialized C during the execution of a previous unit test or not. Moreover, we add class C to the TC set of classes returned by procedure DSE. If the same unit test has already initialized C during its execution, the instrumentation call has no effect.

In method `M` from Fig. 1, we add instrumentation calls for class `C` before the two accesses to static field `F`, that is, between lines 8 and 9 and between lines 9 and 10. (Our implementation omits the second instrumentation call in this example, but this is not always possible for methods with more interesting control flow.) Consider again the exploration DSE(`M`, {}). During the execution of the generated unit test, the instrumentation call at the first access to static field `F` calls `C`'s static initializer such that the unit test continues with `F` = 0. The instrumentation call for the second access to `F` has no effect since this unit test already initialized class `C`. DSE returns the set {`C`} as described above.

Note that an explicit call to a static initializer is itself an access to a class member and, thus, causes the runtime environment to trigger another call to the same initializer. To prevent the initializer from executing twice (and thereby duplicating its side effects), we instrument each static initializer such that its body is skipped on the first call, as shown on the right.

```
static C() {
  if (/* this is the first call */)
    return;
  // body of original
  // static initializer
}
```

This instrumentation decouples the execution of a unit test from the initialization behavior of the runtime environment. Static initializers triggered by the runtime environment have no effect and, thus, do not actually initialize the classes, whereas our explicit calls to static initializers initialize the classes even in cases where the runtime environment considers them to be initialized already.

Uninitialization. To avoid the overhead of restarting the runtime environment after each unit test, we simulate the effect of a restart through code instrumentation. Since our technique does not depend on the behavior of the runtime environment to control class initialization, we do not have to actually uninitialize classes. It is sufficient to reset the static fields of all classes initialized by the unit under test to the default values of their declared types after each execution of a unit test. Therefore, the next execution of the static initializer during the execution of the next unit test behaves as if it ran on an uninitialized class.

Existing automatic test case generators (with the exception of JCrasher) do not reset static fields to their initial values between test runs. For code like in Fig. 1, Pex emits a warning that the unit under test might not leave the dynamic symbolic execution engine in a clean state. Therefore, the deterministic re-execution of the generated unit tests is not guaranteed. In fact, the Pex documentation suggests that the tester should mock all interactions of the unit under test with static state. However, this requires the tester to be aware of these interactions and renders Pex significantly less automatic.

3.2 Dynamic Symbolic Execution

The core idea of our exploration is as follows. Assume that we knew the set *classes* of all classes whose initialization may be triggered by executing the unit under test UUT. For each subset $IC \subseteq classes$, we perform dynamic symbolic execution of UUT such that the classes in IC are initialized before executing UUT and their static fields are symbolic inputs. The classes in $classes \setminus IC$ are not initialized (that is, their initializers may be triggered when executing a unit test). We can then explore all possible initialization behaviors of UUT by testing it for each possible partition of *classes* into initialized and uninitialized classes.

Algorithm. Alg. 1 is a dynamic-symbolic-execution algorithm that implements this core idea, but also needs to handle the fact that the set of relevant classes is not known upfront, but determined during the execution. Procedure EXPLORE takes as argument a unit under test UUT, which has been instrumented as described above. Local variable *classes* is the set of relevant classes determined

Alg. 1 Dynamic symbolic execution for exploring the interactions of a unit under test with static state.

```
1  procedure EXPLORE(UUT)
2    classes ← {}
3    explored ← {}
4    while ∃IC ⊆ classes · IC ∉ explored do
5      IC ← choose({IC | IC ⊆ classes ∧ IC ∉ explored})
6      TC ← DSE(UUT, IC)
7      classes ← classes ∪ TC
8      explored ← explored ∪ {IC}
9    end while
10 end procedure
```

so far, while local variable *explored* is the set of sets of classes that have been treated as initialized in the exploration so far; that is, *explored* keeps track of the partitions that have been explored. As long as there is a partition that has not been explored (that is, a subset IC of *classes* that is not in *explored*), the algorithm picks any such subset and calls the dynamic symbolic execution procedure DSE, where classes in IC are initialized and their static fields are treated symbolically. If this procedure detects any classes that are initialized during the dynamic symbolic execution, they are added to *classes*. The EXPLORE procedure terminates when all possible subsets of the relevant classes have been explored.

Initialization Dependencies. Alg. 1 enumerates all combinations of initialized and uninitialized classes in the input state of the method under test, that is, all possible partitions of *classes* into IC and $classes \setminus IC$. This includes combinations that cannot occur in any actual execution. If the static initializer of a class E triggers the static initializer of a class D, then there is no input state in which E is initialized, but D is not. To avoid such situations and, thus, false positives during testing, we trigger the static initializers of all classes in IC before invoking the method under test. In the above example, this ensures that both E and D will be initialized in the input state of the method under test, and D's initializer will not be triggered during the execution of the method. Since the outcome of running several static initializers may depend on the order in which they are triggered, we explore all orders among dependent static initializers.

The triggering of the static initializers of the classes in IC happens at the beginning of the set-up code that precedes the invocation of the method under test in every generated unit test. This set-up code is also responsible for creating the inputs for the method under test, for instance, for allocating objects that will be passed as method arguments. Therefore, the set-up code may itself trigger static initializers, for instance, when a constructor reads a static field. To handle the dependencies between set-up code and initialization, we treat set-up code as a regular part of the unit test (like the method under test itself), that is, apply the same instrumentation and explore all possible execution paths during dynamic symbolic execution.

Handling dependencies between static initializers is particularly useful in C++, where static initialization happens before the program entry point. When linking multiple translation units, the order of initialization between the translation units is undefined. By exploring all orders of execution of dependent initializers, developers can determine dependencies that may crash a program before its entry point is reached.

Example. The example in Fig. 2 illustrates our approach. The assertion on line 13 fails only if `N` is executed in a state in which class `D` is initialized (such that the if-statement may be executed), the static field `G` is negative (such that the if-statement will be executed and `E`'s initialization will be triggered), and class `E` is not initialized (such that its static initializer will affect the value of `G`).

We will now explain how Alg. 1 reveals such subtle bugs. In the first iteration, IC is the empty set, that is, no class is considered to be initialized. Therefore,

```
1  public class D {
2    public static int G;
3
4    static D() {
5      G = 0;
6    }
7
8    public static void N() {
9      if (G < 0) {
10       E.H++;
11       G = -G;
12     }
13     assert 0 <= G;
14   }
15 }
```

```
16 public class E {
17   public static int H;
18
19   static E() {
20     H = 0;
21     D.G = 1;
22   }
23 }
```

Fig. 2. An example illustrating the treatment of static initializers with precise semantics. We use the special `assert` keyword to denote Code Contracts [4] assertions. The assertion on line 13 fails only if `N` is called in a state where class `D` is initialized, but `E` is not.

when the DSE procedure executes method `N`, class `D` is initialized right before line 9. Consequently, static field `G` is zero, the if-statement is skipped, and the assertion holds. DSE returns the set $\{\texttt{D}\}$.

In the second iteration, IC will be $\{\texttt{D}\}$, that is, the static initializer of class `D` is triggered by the set-up code, and static field `G` is treated symbolically. Since there are no constraints on the value of `G` yet, the dynamic symbolic execution executes method `N` with an arbitrary value for `G`, say, zero. This unit test passes and produces the constraint $\texttt{G} < 0$ for the next unit test. For any such value of `G`, the unit test will now enter the if-statement and initialize class `E` before the access to `E`'s static field `H`. This initialization assigns 1 to `G`, such that the subsequent negation makes the assertion fail, and we have detected the bug. The call to the DSE procedure in the second iteration returns $\{\texttt{D}, \texttt{E}\}$.

The two remaining iterations of Alg. 1 cover the cases that IC is $\{\texttt{E}\}$ or $\{\texttt{D}, \texttt{E}\}$. The former case illustrates how we handle initialization dependencies. The static initializer of class `E` accesses static field `G` of class `D`. Therefore, when `E`'s initializer is called by the set-up code of the generated unit test, `D`'s initializer is also triggered (recall that the set-up code and all static initializers are instrumented like the method under test). This avoids executing `N` in the impossible situation where `E` is initialized, but `D` is not. The rest of this iteration is analogous to the first iteration, that is, class `D` gets initialized (this time while executing the set-up code), the if-statement is skipped, and the assertion holds.

Finally, for $IC = \{\texttt{D}, \texttt{E}\}$, all relevant classes are initialized. The dynamic symbolic execution will choose negative and non-negative values for `G`. The assertion holds in either case.

Discussion. Alg. 1 can be implemented in any testing tool based on dynamic symbolic execution. We have implemented it in Pex, whose existing dynamic symbolic execution engine is invoked by our DSE procedure. Alg. 1 could also be implemented in jCUTE for testing how static fields and static blocks in Java interact with a unit under test. Moreover, this algorithm can be adjusted to perform all dynamic tasks statically for testing tools based on static symbolic execution. For instance, Symbolic Java PathFinder could then be extended to take static state into account.

By treating static fields symbolically, our technique gives meaning to specifications that refer to static fields, like assertions or preconditions. For example, an assertion about the value of a static field is now treated as a branch by the symbolic execution. One could also support preconditions that express which classes are required to be initialized before the execution of a method.

As part of the integration with unit testing frameworks, many automatic test case generators support defining set-up methods for a unit under test. Such methods allow testers to initialize and reset static fields manually. Since set-up methods might express preconditions on static fields (in the form of code), we extended our technique not to override the functionality of these methods. That is, when a set-up method assigns to a static field of a class C, we do not trigger the initialization of class C and do not treat its static fields symbolically. We do, however, reset the values of all static fields in class C after each execution of a unit test such that the next execution of the set-up method starts in a fresh state.

This technique could also be used in existing frameworks for detecting whether a set-up method allows for any static fields to retain their values between runs of the unit under test. This is achieved by detecting which static fields are modified in the unit under test, but have not been manually set up. If such fields exist, an appropriate warning could be emitted by the unit testing framework.

4 Initialization with Before-Field-Init Semantics

The technique presented in the previous section handles static initializers with precise semantics. Static initializers with before-field-init semantics, which may be triggered at any point before the first access to a static field of the class, impose two additional challenges. First, they introduce non-determinism because the static initializer of any given class may be triggered at various points in the unit under test. Second, in addition to the classes that have to be initialized in order to execute the unit under test, the runtime environment could in principle choose to trigger any other static initializer with before-field-init semantics, even initializers of classes that are completely unrelated. In this section, we describe how we solve these challenges. Our solution uses a static program analysis to determine the program points at which the execution of a static initializer with before-field-init semantics may affect the behavior of the unit under test. Then, we use a modified dynamic symbolic execution procedure to explore each of these possibilities.

As the running example of this section, consider method `P` in Fig. 3. The static initializer of class `D` has before-field-init semantics and must be executed before the access to field `D.Fd` on line 10. If the initializer runs on line 5 or 9, then the assertion on line 8 succeeds. If, however, the initializer runs on line 7, the assertion fails because the value of field `C.Fc` has been incremented (line 15) and is no longer equal to 2. This bug indicates that the unit under test is affected by the non-deterministic behavior of a static initializer with before-field-init semantics. Such errors are particularly difficult to detect with standard unit testing since they might not manifest themselves reproducibly.

```
1 public static class C {
2   static int Fc = 0;
3
4   public static void P() {
5     // static initializer of 'D'
6     Fc = 2;
7     // static initializer of 'D'
8     assert Fc == 2;
9     // static initializer of 'D'
10    if (D.Fd == 3)
11      Fc = E.Fe;
12  }
13
14  static class D {
15    public static int Fd = C.Fc++;
16  }
17
18  static class E {
19    public static int Fe = 11;
20  }
21 }
```

Fig. 3. A C# example illustrating the non-determinism introduced by static initializers with before-field-init semantics. The assertion on line 8 fails if `D`'s static initializer is triggered on line 7.

Critical Points. A static initializer with before-field-init semantics may be triggered at any point before the first access to a static field of its class. To reduce the non-determinism that needs to be explored during testing, we use a static analysis to determine the *critical points* in a unit under test, that is, those program points where triggering a static initializer might actually affect the execution of the unit under test. All other program points can be ignored during testing because no new behavior of the unit under test will be exercised.

A critical point is a pair consisting of a program point i and a class C. It indicates that there is an instance or static field f that is accessed both by the instruction at program point i and the static initializer of class C such that the instruction or the static initializer or both modify the field. In other words, a critical point indicates that the overall effect of executing the static initializer

of C and the instruction at i depends on the order in which the execution takes place. Moreover, a pair (i, C) is a critical point only if program point i is not dominated in the control-flow graph by an access to a static field of C, that is, if it is possible to reach program point i without initializing C first.

In the example of Fig. 3, there are five critical points: (6, `C`), (6, `D`), (8, `D`), (10, `D`), and (11, `E`), where we denote program points by line numbers. Note that even though the static initializer of class `E` could be triggered anywhere before line 11, there is only one critical point for `E` because the behavior of method `P` is the same for all these possibilities.

We determine the critical points in a method under test in two steps. First, we use a simple static analysis to compute, for each program point i, the set of classes with before-field-init initializers that might get triggered at program point i. This set is denoted by $prospectiveClasses(i)$. In principle, it includes all classes with before-field-init initializers in the entire program, except those that are definitely triggered earlier. Since it is not feasible to consider all of them during testing, we focus on those classes whose static fields are accessed by the method under test. This is not a restriction in practice: even though the Common Language Infrastructure *standard* [3] allows more initializers to be triggered, the Common Language Runtime *implementation*, version 4.0, triggers the initialization of exactly the classes whose static fields are accessed by the method. Therefore, in Fig. 3, $prospectiveClasses(8)$ is the set $\{\texttt{D}, \texttt{E}\}$.

Second, we use a static analysis to determine for each program point i and class C in $prospectiveClasses(i)$ whether (i, C) is a critical point. For this purpose, the static analysis approximates the read and write effects of the instruction at program point i and of the static initializers of all classes in $prospectiveClasses(i)$. The *read effect* of a statement is the set of fields read by the statement or by any method the statement calls directly or transitively. Analogously, the *write effect* of a statement is the set of fields written by the statement or by any method the statement calls directly or transitively. The pair (i, C) is a critical point if (1) i's read effect contains a (static or instance) field f that is included in the write effect of C's static initializer, or (2) i's write effect contains a (static or instance) field f that is included in the read or write effect of C's static initializer. For instance, for line 8 of our example, (8, `D`) is a critical point because the statement on line 8 reads field `Fc`, which is written by the static initializer of class `D`, and `D` is in $prospectiveClasses(8)$. However, even though class `E` is in $prospectiveClasses(8)$, (8, `E`) is not a critical point because the effects of the statement on line 8 and of `E`'s static initializer are disjoint.

Read and write effects are sets of fully-qualified field names, which allows us to approximate them without requiring alias information. Our static effect analysis is inter-procedural. It explores the portion of the whole program it can access (in particular, the entire assembly of the method under test) to compute a call graph that includes information about dynamically-bound calls. Therefore, our analysis may miss critical points (for instance, when it fails to consider a method override in an assembly that is not accessible to the analysis) and, thus, testing might not explore all possible behaviors. It may also yield irrelevant critical

points (for instance, when the instruction and the static initializer both have an instance field f in their effects, but at runtime, access f of different objects) and, thus, produce redundant unit tests.

A critical point (i, C) indicates that the dynamic symbolic execution should trigger the initialization of class C right before program point i. However, C's static initializer might lead to more critical points, because its effects may overlap with the effects of other static initializers and because it may trigger the initialization of additional classes, which, thus, must be added to *prospectiveClasses*. To handle this interaction, we iterate over all options for critical points and, for each choice, inline the static initializer and recursively invoke our static analysis.

Dynamic Symbolic Execution. We instrument the unit under test to include a marker for each critical point (i, C). We enhance the DSE procedure called from Alg. 1 to trigger the initialization of class C when the execution hits such a marker. If there are several markers for one class, the DSE procedure explores all paths of the unit under test for each possible point. Conceptually, one can think of adding an integer argument n_C to the unit under test and interpreting the n-th marker for class C as a conditional statement `if` $(n_C == n)$ { $\texttt{init}_C$ }, where $\texttt{init}_C$ calls the static initializer of class C if it has not been called earlier during the execution of the unit test. Dynamic symbolic execution will then explore all options for the initialization of a class C by choosing different values for the input n_C.

Since (8, `D`) is a critical point in our example, DSE will trigger the initialization of class `D` right before line 8 during the symbolic execution of method `P`. As a result, the assertion violation is detected.

5 Experimental Evaluation

We have evaluated the effectiveness of our technique on 30 open-source applications written in C#. These applications were arbitrarily selected from applications on Bitbucket, CodePlex, and GitHub. Our suite of applications contains a total of 423,166 methods, 47,515 (11%) of which directly access static fields. All classes of these applications define a total of 155,632 fields (instance and static), 28,470 (18%) of which are static fields; 14,705 of the static fields (that is, 9% of all fields) are static read-only fields. There is a total of 1,992 static initializers, 1,725 (87%) of which have precise semantics, and 267 (13%) of which have before-field-init semantics.

To determine which of the 47,515 methods that directly access static fields are most likely to have bugs, we implemented a lightweight scoring mechanism. This mechanism statically computes a score for each method and ranks all methods by their score. The score for each method is based on vulnerability and accessibility scores. The *vulnerability score* of a method indicates whether the method directly accesses static fields and how likely it is to fail at runtime because of a static field, for instance, due to failing assertions, or division-by-zero and arithmetic-overflow exceptions involving static fields. This score is computed based on nesting levels

Tab. 1. Summary of our experiments. The first column shows the name of each application. The second column shows the total number of tested methods from each application. The two rightmost columns show the number of errors detected without and with treating static fields as inputs to the unit under test, respectively.

Application	Methods	Number of errors	
		init	init&input
Boggle	60	-	24
Boogie	21	-	6
Ncqrs	38	1	1
NRefactory	37	-	9
Scrabble	64	-	2
Total	220	1	42

of expressions and how close a static field is to an operation that might throw an exception. The *accessibility score* of a method indicates how accessible the method and the accessed static fields are from potential clients of the application. In particular, this score indicates the level of accessibility from the public interface of the application, and suggests whether a potential bug in the method is likely to be reproducible by clients of the application. The final score for each method is the product of its vulnerability and accessibility scores.

To compare the number of errors detected with and without our technique, we ran Pex with and without our implementation on all methods with a non-zero score. There were 454 methods with a non-zero score in the 30 applications. Tab. 1 summarizes the results of our experiments on the applications in which bugs were detected. The first column of the table shows the name of each application[1]. The second column shows the total number of methods with a non-zero score for each application. The two rightmost columns of the table show the number of errors that our technique detected in these methods. These errors do not include errors already detected by Pex without our technique; they are all caused by interactions of the methods under test with static state.

More specifically, column "init" shows the number of errors detected by simply triggering static initializers at different points in the code. These errors are, thus, caused by calling static initializers (with both semantics) during the execution of the unit tests without treating static fields as inputs. Column "init&input" shows the number of errors detected by our technique, that is, by treating static fields symbolically and systematically controlling the execution of static initializers.

As shown in the last column of the table, our technique detected 42 bugs that are not found by Pex. Related work suggests that existing test case generators would not find these bugs either. A failed unit test does not necessarily mean that the application actually contains code that exhibits the detected bug; this

[1] The applications can be found at:
http://boggle.codeplex.com, rev: 20226
http://boogie.codeplex.com, rev: e80b2b9ac4aa
http://github.com/ncqrs/ncqrs, rev: 0102a001c2112a74cab906a4bc924838d7a2a965
http://github.com/icsharpcode/NRefactory, rev: ae42ed27e0343391f7f30c1ab250d729fda9f431
http://wpfscrabble.codeplex.com, rev: 20226

uncertainty is inherent to unit testing since methods are tested in isolation rather than in the context of the entire application. However, all of the detected bugs may surface during maintenance or code reuse. In particular, for 25 of the 42 detected bugs, both the buggy method and the accessed static fields are public. Therefore, when the applications are used as libraries, client code can easily exhibit these bugs.

We have also manually inspected static initializers from all 30 applications and distilled their three most frequent usage patterns. Static initializers are typically used for:

1. Initializing static fields of the same class to constants or simple computations; these initializers are often inline initializers, that is, have before-field-init semantics. However, since they neither read static fields of other classes nor have side effects besides assigning to the static fields of their class, the non-determinism of the before-field-init semantics does not affect program execution.
2. Implementing the singleton pattern in a lazy way; these initializers typically have precise semantics.
3. Initializing public static fields that are mutable; these fields are often meant to satisfy invariants such as non-nullness. However, since they are public, these invariants can easily be violated by client code or during maintenance. This pattern is especially susceptible to static-field updates after the initialization, a scenario that we cover by treating static fields as inputs of the unit under test.

In none of these common usage patterns do initializers typically have side effects besides assigning to static fields of their class. This might explain why we did not find more bugs that are caused by static initialization alone (column "init" in Tab. 1); it is largely irrelevant when such initializers are triggered.

An interesting example of the third pattern was found in application Boggle, which uses the Caliburn.Micro library. This library includes a public static field `LogManager.GetLog`, which is initialized by `LogManager`'s static initializer to a non-null value. `GetLog` is read by several other static initializers, for instance, the static initializer of class `Coroutine`, which assigns the value of `GetLog` to a static field `Log`. If client code of the Caliburn.Micro library assigned null to the public `GetLog` field before the initialization of class `Coroutine` is triggered, the application might crash; `Coroutine` will then initialize `Log` with the null value, which causes a null-pointer exception when `Coroutine`'s `BeginExecute` method dereferences `Log`. Our technique reveals this issue when testing `BeginExecute`; it explores the possibility that `LogManager` is initialized before `BeginExecute` is called whereas `Coroutine` is not, and it treats `GetLog` as an input to `BeginExecute` such that the dynamic symbolic execution will choose null as a possible value. Note that this issue is indeed an initialization problem. Since `Coroutine.Log` is not public, a client could not cause this behavior by assigning null directly to `Log`.

6 Conclusion

To automatically check the potential interactions of static state with a unit under test, we have proposed a novel technique in automatic test case generation based on static analysis and dynamic symbolic execution. Our technique treats static fields as input to the unit under test and systematically controls the execution of static initializers. We have implemented this technique as an extension to Pex and used it to detect errors in open-source applications. As future work, one could prune redundant explorations more aggressively; this is promising since our evaluation suggests that many static initializers have very small read and write effects and, thus, very limited interactions with the unit under test.

Acknowledgments. We are grateful to Nikolai Tillman and Jonathan "Peli" de Halleux for sharing the Pex source code with us. We also thank Dimitar Asenov, Valentin Wüstholz, and the reviewers for their constructive feedback.

References

1. Csallner, C., Smaragdakis, Y.: JCrasher: An automatic robustness tester for Java. SPE 34, 1025–1050 (2004)
2. Du Toit, S.: Working Draft, Standard for Programming Language C++ (2013), http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3691.pdf
3. ECMA. ECMA-335: Common Language Infrastructure (CLI). ECMA (2012)
4. Fähndrich, M., Barnett, M., Logozzo, F.: Embedded contract languages. In: SAC, pp. 2103–2110. ACM (2010)
5. Fähndrich, M., Logozzo, F.: Static contract checking with abstract interpretation. In: Beckert, B., Marché, C. (eds.) FoVeOOS 2010. LNCS, vol. 6528, pp. 10–30. Springer, Heidelberg (2011)
6. Flanagan, C., Leino, K.R.M., Lillibridge, M., Nelson, G., Saxe, J.B., Stata, R.: Extended static checking for Java. In: PLDI, pp. 234–245. ACM (2002)
7. Godefroid, P., Klarlund, N., Sen, K.: DART: Directed automated random testing. In: PLDI, pp. 213–223. ACM (2005)
8. Gosling, J., Joy, B., Steele, G., Bracha, G., Buckley, A.: The Java Language Specification. Oracle, java SE, 7th edn. (2012)
9. Leino, K.R.M., Müller, P.: Modular verification of static class invariants. In: Fitzgerald, J.S., Hayes, I.J., Tarlecki, A. (eds.) FM 2005. LNCS, vol. 3582, pp. 26–42. Springer, Heidelberg (2005)
10. Pacheco, C., Lahiri, S.K., Ernst, M.D., Ball, T.: Feedback-directed random test generation. In: ICSE, pp. 75–84. IEEE Computer Society (2007)
11. Păsăreanu, C.S., Mehlitz, P.C., Bushnell, D.H., Gundy-Burlet, K., Lowry, M., Person, S., Pape, M.: Combining unit-level symbolic execution and system-level concrete execution for testing NASA software. In: ISSTA, pp. 15–26. ACM (2008)
12. Sen, K., Agha, G.: CUTE and jCUTE: Concolic unit testing and explicit path model-checking tools. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 419–423. Springer, Heidelberg (2006)
13. Sen, K., Marinov, D., Agha, G.: CUTE: A concolic unit testing engine for C. In: ESEC, pp. 263–272. ACM (2005)
14. Tillmann, N., de Halleux, J.: Pex—White box test generation for .NET. In: Beckert, B., Hähnle, R. (eds.) TAP 2008. LNCS, vol. 4966, pp. 134–153. Springer, Heidelberg (2008)