# Dynamic test generation with static fields and initializers

Patrick Emmisberger

Bachelor's Thesis

Chair of Programming Methodology
Department of Computer Science
ETH Zurich

http://www.pm.inf.ethz.ch/

07/15/2013

**Supervised by:**
Maria Christakis
Prof. Dr. Peter Müller

**Chair of Programming Methodology**
inf | Informatik
Computer Science

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

# Abstract

*Static fields*, also known as *class variables*, have long been part of object oriented programming languages and are widely used in libraries and applications. Especially in the field of unit testing their reputation of being inherently hard to test precedes them. A recent approach to software testing is *Dynamic Symbolic Execution*, a form of white-box testing, to generate test cases by exploring the method under test automatically. This thesis looks at the results of combining *static fields* and *Dynamic Symbolic Execution* for the concrete case of the Microsoft .NET-Framework and Pex, an automated testing tool developed by Microsoft Research.

In the core part, the focus is on extending the support for static fields and initializers by Pex. Additionally analysis approaches for eager initialization and detecting dependencies between types are introduced. Afterwards the utility of the new analysis features is evaluated on a set of open source projects and the evaluation concludes by presenting general usage patterns of static fields and initializers that emerged in the process.

Chapter 1 introduces the basic concepts and gives an overview of *Dynamic Symbolic Execution* allowing people without prior experience with Pex to follow the rest of this report. Chapter 2 describes a set of extensions for the testing tool Pex aiming towards the full support for static fields and initialization. Chapter 3 continues by presenting a newly developed tool for determining the effectiveness of the extensions while the results of this evaluation are found in Chapter 4. Chapter 5 mentions related work and this report concludes by summarizing the findings in Chapter 6

# Contents

# Chapter 1

# Introduction

## 1.1 Static Fields and Initializers

Static fields are variables that keep their value for the complete lifetime of a process[1]. Since static fields are normal members of a class, they are subject to the same visibility checking as instance fields. In contrast to an instance field, where every instance can store a different value, the value of static fields is shared between all instances of the declaring type and all its subtypes. This property of static fields allows the implementation of classes as seen in Listing 1.1.

```
class UniqueId {
  static int maximumId;
  int instanceId;

  public UniqueId() {
    instanceId = ++maximumId;
  }

  public InstanceId { get { return instanceId; } }
  public static MaximumId { get { return maximumId; } }
}

var a = new UniqueId();
var b = new UniqueId();
Console.WriteLine(a.InstanceId); // Output: 1
Console.WriteLine(b.InstanceId); // Output: 2
Console.WriteLine(UniqueId.MaximumId); // Output: 2
var c = new UniqueId();
Console.WriteLine(UniqueId.MaximumId); // Output: 3
```

Listing 1.1: Class with a static and an instance field

The class `UniqueId` assigns a unique numeric identifier to all its instances by keeping a common counter variable in form of a static field. Each time an new instance is created, the counter is incremented by one and the new value is copied into the instance field `instanceId`. Because static

[1] This is not entirely true for programs that use the CLI, there the lifetime of a static field is limited by the lifetime of the AppDomain[15]. However, most applications only use a single AppDomain where the lifetime of the AppDomain is the same as the lifetime of the process.

members are shared between all instances of a class, no instance must be provided when referring to a static member. To provide the lexical scope when a static member is referenced outside of a method of its declaring type, the class name is used.

Most languages initialize the value of a static field to the default value of the respective type (in case of the example the type is `int` with a default value of 0). If the initial value of a static field should be different from its default value, a piece of code must run before the field is accessed that assigns the correct value first. This piece of code is usually referred to as a *type initializer*, *static initializer* or *static constructor*. For this thesis *static initializer* will denote the code that is executed while *static constructor* is the name used for the method that contains the *static initializer* for a type.

Depending on the language, static fields can be initialized in two ways: inline and by writing an explicit static constructor. The C# language supports both inline and explicit initializers as illustrated by Listing 1.2. Inline initializers are simply translated to assignments that are prepended to the static constructor in declaration order (Listing 1.3). If no static constructor exists, an empty one will be created in that case. If neither an inline nor an explicit initializer exist, the type will have no static constructor. While both ways, inline and explicit, are equivalent in assigning a value to a field, an explicit static constructor changes the semantics of when the declaring type is initialized. This is discussed in depth in Section 1.2.

```
class A {
  static int Foo = 42; // Inline
      // Inline computation
  static int Bar = Foo + 10;
  static int Baz;

  static A() { // Explicit
    Baz = 256;
  }
}
```

Listing 1.2: Static Initializers

```
class A {
  static int Foo;
  static int Bar;
  static int Baz;
  static A() {
    Foo = 42;
    Bar = Foo + 10
    Baz = 256;
  }
}
```

Listing 1.3: Output produced by compiler

From the compiler transformation seen in Listing 1.3 follows automatically that the finest granularity in which fields can be initialized is per type. If one field of a type has to be initialized the runtime calls the *static constructor* if it exists. In turn the static constructor initializes all the static fields of its declaring type and returns the control to the runtime.

```
class A {
  public static int Foo = 42;
  public static int Bar = 101;

  static A() {
    Console.WriteLine("A initialized.");
  }
}

Console.WriteLine("Start");
Console.WriteLine(A.Foo); // The initializer is called before this line.
Console.WriteLine(A.Bar); // A is initialized, only Bar is printed.
```

Listing 1.4: Static initializer runs only once

The static initializer of a type is only executed once in the lifetime of the fields that it initializes. To ensure that, the runtime internally stores a flag for each type that indicates whether is has been initialized or not. Additionally the access to this flag is locked with a monitor to provide this behavior even in multi-threaded applications. Therefore the code in Listing 1.4 has the following lines as output “Start”, “A initialized.”, “42”, “101”.

## 1.2 Type Initialization Semantics in the CLI

Consider the class definitions in Listings 1.5, 1.6 and 1.7. Despite looking nearly identical the type initialization semantics of class `A` is different from the semantics of `B` and `C`.

```
class A {
  static int Foo = 42;
}
```

Listing 1.5: Inline initializer

```
class B {
  static int Foo;
  static B() {
    Foo = 42;
  }
}
```

Listing 1.6: Explicit initializer

```
class C {
  static int Foo = 42;
  static C() { }
}
```

Listing 1.7: Inline initializer with empty static constructor

While classes `B` and `C` share the default type initialization semantics, class `A` has *BeforeFieldInit* semantics. The exact differences can be found by consulting the ECMA Standard for the Common Language Infrastructure (CLI)[5, I.8.9.5 p. 43]:

- **Default semantics:** The type initializer is executed exactly at the first access to any member of that type.
- **BeforeFieldInit semantics:** The type initializer is executed at, or some time before, the first access to any static field defined for that type.

Noteworthy is, that with the default semantics the type initialization is triggered by an instruction while the BeforeFieldInit semantics allow the runtime to initialize the type at any point in time before the first field access making it impossible to predict the location where the static initializer will run.

To determine the actual characteristics of the CLR[2] the test program in Listing A.1 was used. The results can be seen in Table 1.1. The right half of the table is not surprising, since this behavior is dictated by the standard. The eager initialization of a field in a type with *BeforeFieldInit* semantics can be explained by performance reasons. By eagerly initializing the type, the locations in code where the field is accessed must not be guarded to call the static initializer beforehand resulting in less and better optimizable code. Looking at the case where a static method of a BeforeFieldInit type is called, we can see that the initialization behavior has indeed changed between version 2.0 and 4.0 of the CLR. While version 2.0 initialized the type even if no field was accessed, the CLR4.0 is now truly lazy by not running the initializer at all if not necessary.

As a consequence of this, applications that run on different implementations of the CLI may experience a different runtime behavior that can in the worst-case lead to program crashes even if all implementations adhere to the specification. An example of this can be found in Section 3.5. In practice however, static fields and initializers are rarely used in a way that makes them susceptible to these effects.

[2]The Common Language Runtime (CLR) is the specific implementation of the CLI standard that is used in the Microsoft .NET Framework

| | BeforeFieldInit | | Default | |
|---|---|---|---|---|
| | CLR2.0 | CLR4.0 | CLR2.0 | CLR4.0 |
| Field | | | | |
| not accessed | Eager | Eager | None | None |
| accessed | Eager | Eager | Lazy | Lazy |
| Method | | | | |
| not called | None | None | None | None |
| called | Lazy | None | Lazy | Lazy |

Table 1.1: Initialization behavior in release mode

## 1.3 Dynamic Symbolic Execution

The roots of *Dynamic Symbolic Execution* lie in a technique called "fuzzing" where the inputs of a program (e.g. a file or the parameters of a function) are modified randomly or with heuristics to provoke software errors[11, 2, 29]. This is also called "black-box fuzzing" since it does not take the code of the program being tested into account. *Dynamic Symbolic Execution* uses the same idea but takes a different approach to generating input values. The code is executed with a set of concrete values but simultaneously these values and their usage are tracked symbolically, indicating to the testing tool how to change the values to provoke different execution paths in the code under test.

As a concrete example consider Listing 1.8. Assume the first execution uses parameter values `a = 0; b = 0;`. The test `a > 10` will return `false` and the method returns. The symbolic execution engine observes all branch conditions and generates a *path condition* that constrains the input values, so that for all input values that satisfy the constraint the same execution path will be taken. For our concrete example the path condition for the first run is $a \leq 10$.

In order to explore more parts of the method, the path condition is modified (normally the last conjunction is inverted) and by using a constraint solver, new values for the inputs are found. In our example the new path condition would be $a > 10$ and we assume that the constraint solver returns `a = 11`. `b` is not modified and still uses its start value of 0. For the next run, the first test holds and the *then branch* is executed, containing a second condition. The second condition will evaluate to `false` for this run and the method returns. The symbolic execution engine provides the new path condition $a > 10 \wedge b \leq a * 2$. We invert the last part of the path condition resulting in $a > 10 \wedge b > a * 2$ for which the constraint solver returns the values `a = 11; b = 23;`. We run the method with this input and the exception is thrown.

```
void Test(int a, int b){
  if (a > 10) {
    if (b > a*2)
      throw new Exception("Unexpected Error");
  }
}
```

Listing 1.8: Example code for Dynamic Symbolic Execution

Using this technique, the testing tool tries to maximize branch coverage and find input values for which the method behaves unexpectedly.

Listing 1.9 contains the basic algorithm in pseudo-code. `solve` runs the constraint solver to find concrete values that satisfy the given condition. `execute` runs the method under test with the given input values and returns the path condition that was generated in this run.

```
void explore(IEnumerable<Term> pathCondition) {
  values = solve(pathCondition);
  if (pathCondition is satisfiable) {
    newPathCondition = execute(values);
    for(int i = |pathCondition|; i <= |newPathCondition|; i++) {
      var prefix = newPathCondition[1..i];
      prefix[i] = ¬prefix[i];
      explore(prefix);
    }
  }
}
```

Listing 1.9: Dynamic Symbolic Execution Algorithm (Pseudo-Code)

# Chapter 2

# Core Analysis

## 2.1 Introduction

The default behavior of Pex provides very little support for testing static fields and initializers. The root cause is that Pex does not control the type initialization manually. Therefore the static initializers will run only once for an entire test suite and the timing of the initialization is left to the CLR. This introduces a conflict with the requirement of the dynamic symbolic execution engine to be able to run a piece of code multiple times. As a result the instrumentation of type initialization code would not provide any meaningful data but in some cases even lead to wrong results. To avoid this problem, Pex does not instrument type initialization code by default making it invisible for the dynamic symbolic execution engine. The only option that Pex currently provides to deal with type initialization code is to ignore it completely (i.e. never execute it), which is also not a satisfactory solution.

A second major problem area is found when combining the current handling of type initialization with static fields. Pex currently treats the contents of static fields before a test runs as literal values. They are not considered as input and therefore cannot be part of the symbolic path condition. Additionally, static fields keep their values between test runs making the behavior of the test method dependent on the order the test methods are run as well as the order that Pex exercises the different execution paths. The effects of this are undesirable resulting in low branch coverage or, in a worst-case, the generation of invalid tests.

The objective of the core part is to extend the Pex default behavior, take type initializers and static fields into account and to provide solutions for the concrete problems described in the next section.

## 2.2 Problem Statement

The decision not to interfere with the type initialization behavior and its consequences lead to a number of scenarios where the outcome of a test suite is non-deterministic, bugs are missed completely or in the worst case wrong tests are generated by Pex. The following paragraphs illustrate the previously described problems with concrete examples.

- **Initialization order matters.** The code in Listing 2.1 illustrates the case where the outcome of the test suite depends on the order in which Pex runs the two test methods. There

are of course more compact ways to provoke this error, however the code for this example is taken from a real-world project and then simplified.

```
class SerializerRegistry {
  public static readonly Dictionary<Type, ISerializer> Serializers = new
      Dictionary<Type, ISerializer>();
}

class PrimitiveTypeSerializers {
  static PrimitiveTypeSerializers() {
    SerializerRegistry.Serializers[typeof(int)] = new Int32Serializer();
  }
  public static ISerializer Int32Serializer {
    get { return SerializerRegistry.Serializers[typeof(int)]; }
  }
  /* ... */
}

class MyCustomType {
  static MyCustomType {
    SerializerRegistry.Serializers[typeof(MyCustomType)] = new
        MyCustomTypeSerializer(SerializerRegistry.Serializers[typeof(int)
        ]);
  }
  public static ISerializer MyCustomTypeSerializer {
    get { return SerializerRegistry.Serializers[typeof(MyCustomType)]; }
  }
  /* ... */
}

[PexClass]
class Tests {
  [PexMethod]
  public void TestIntSerializer() {
    var s = PrimitiveTypeSerializers.Int32Serializer;
    /* ... */
  }

  [PexMethod]
  public void TestMyCustomSerializer() {
    var s = MyCustomType.MyCustomTypeSerializer;
    /* ... */
  }
}
```

Listing 2.1: Example where initialization order matters

We look at two possible cases:

1. If Pex decides to run the `TestIntSerializer` test first, the initializer of the type `PrimitiveTypeSerializers` will run adding the serializer for the type `int` to the `SerializerRegistry.Serializers`. When later the `TestMyCustomSerializer` methods runs, the static initializer of `MyCustomType` finds the serializer for `int` in the dictionary and all tests pass.

2. In the case where the tests are executed in the reverse order (`TestMyCustomSerializer` runs first), the lookup for the type `int` in the static initializers of `MyCustomType` will fail.

If the second test fails, the underlying cause of the problem is revealed: While semantically the class `MyCustomType` depends on the class `PrimitiveTypeSerializers` this is not expressed directly in the code. Depending on the test execution order, this dependency is resolved in the correct way (first case) or not (second case). If this code went out into production the occurrence of this bug could depend on user input, making this a hard error to find.

- **Static fields are not considered input.** The contents of static fields is not considered input to a test method like argument values and instance fields are. As a consequence static fields are not part of the path condition and therefore no automatic values will be generated. This can lead to very low branch coverage and essentially false negatives as the Listing 2.2 shows. Even though the condition is easily flipped, Pex does not attempt it because Active is a static field.

```
[PexClass]
class GuardedMethod {
  public static bool Active = false;

  [PexMethod]
  public static void Test() {
    if (Active)
      PexAssert.IsTrue(false);
  }
}
```

Listing 2.2: False negative by guarding a method with a static field

- **Retaining state between test runs with static fields.** The possibility to keep state between test runs has multiple disadvantageous effects. One case where this plays a role was the first example in this list. Another effect that can be triggered is the generation of invalid test cases as seen in the Listing 2.3. The root cause is the assumption of the dynamic symbolic execution engine that running a method twice with the same input will result in the same execution path. This is based on the premise that, when running deterministic code from the same state twice it will result in the same end state. This premise is violated by not considering the static fields as part of the state and therefore not being able to run the method from exactly the same state (i.e. there exists a subset of state information that is invisible to the dynamic symbolic execution engine). By exploiting the former assumption Pex can be tricked into generating an invalid test case. In the example, Pex assumes that the exception was caused by the modified parameter values. Looking at the method we can see that the exception and the parameter are completely unrelated, but the exception was indeed caused by the part of the state information that is not visible to Pex.

  As a note, Pex will emit a warning for this example that a static field has been modified, hinting that the outcome of the test might not be correct. However the possibility to generate a correct test case would still be desirable.

```
[PexClass]
class StatePreservation {
  static int x = 0;

  [PexMethod]
  public static void Test(bool arg
      ) {
    x++;
    if (x == 2)
      throw new Exception();

    if (arg)
    { /* ... */ }
  }
}
```

Listing 2.3: Example that provokes a wrong test to be generated

```
[TestMethod]
[PexGeneratedBy(typeof(
    StatePreservation))]
[PexRaisedException(typeof(
    Exception))]
public void TestThrowsException948
    ()
{
  StatePreservation.Test(
      PexSafeHelpers.ByteToBoolean
      ((byte)2));
}
```

Listing 2.4: Generated testcase

From the introduction and the examples the problem statement can be summarized by the following subproblems:

1. Multiple type initialization orders should be exercised.
2. Static fields should be considered part of the input, part of the path condition and automatic value generation process.
3. The static initialization code should be a normal part of a test run. Particularly, this requires the ability to execute a static initializer multiple times.
4. It should not be possible to keep state between test runs. The contents of static fields should be considered part of the state and be properly reset between runs.

## 2.3 Approach

To consider all type initializers and static fields in the application domain is not feasible and also not beneficial because only a very limited subset of types and static fields will be used for a specific test method. Therefore a way of detecting the set of type initializers that potentially run and static fields that may be accessed had to be found. A naive, static algorithm that would have been based on analysis of the IL instructions was considered but dismissed because of its inaccuracy. The static analysis would have to be over-approximating and therefore include all potential callees in case of dynamic dispatching. This would yield too large result sets that could not have been handled.

The taken approach uses the dynamic symbolic execution engine that Pex already provides to dynamically discover types and fields while Pex is trying to maximize branch coverage. Using this algorithm the discovered sets are very accurate but because of the mere number of types and static fields in the BCL[1] still to large. As a solution the user can provide inclusion and exclusion rules based on the name and namespace of a class for a specific test. The default settings are to include all types and fields except from the ones in the System namespace. The evaluation has shown that this default is sufficient for nearly all evaluated projects. The justification for excluding the

[1] The BCL (Base Class Library) is the collection of libraries that is installed by default when the .NET Framework is available on a computer

System namespace can be found in the fact that the BCL is not the code under test and we can assume it to be correct. Along with the in-/exclusion of types and fields, the additional analysis can be disabled on a per-method basis or restricted to only simulating the static initializers but not generating values for static fields.

As the number of possible permutations to run the static initializers grows exponentially it is again not feasible to do an exhaustive enumeration of all possible initialization orders. The tool will try to find initialization orders that include the minimal and the maximal numbers of types that need/can be initialized for a certain execution path to be taken. In particular it exercises the test under the condition that none and all of the potentially involved types are initialized.

Listing 2.5 describes the algorithm used by our approach to find all the type initializers and static fields involved in the execution of a method and the different orders in which to run the static initializers in pseudo-code. The approach uses a fix-point algorithm that stops when no new types (and therefore no new fields) can be detected anymore. The lines marked with $*$ can be ignored for now, their importance is explained in Section 3.6.

```
1  todo = {{}}
2  done = {}
3  typesToModifyFields = {}
4   *  swapList = [initialized from attribute]
5
6  do
7  {
8      types = pickAndRemove(todo)
9
10     foreach(type in types)
11     {
12         runInitializer(type)
13
14         if (type in typesToModifyFields)
15             setGeneratedValuesForStaticFields(type)
16
17         typesToModifyFields += type
18     }
19
20     newlyDetectedTypes = run()
21
22     for(newType in newlyDetectedTypes)
23         todo += { types + { newType } }
24
25  *  foreach((first, second) in swapList)
26  *  {
27  *      foreach(seq in [duplicate of todo])
28  *      {
29  *          if (seq.Contains(first) && seq.Contains(second))
30  *              todo += [seq with the position of first and second swapped]
31  *      }
32  *  }
33
34   done += todo
35 } while [done or typesToModifyFields changed in this iteration]
```

Listing 2.5: Pseudo-code for detecting all fields and exercising different initialization orders

The method `runInitializer` resets the contents of the static fields of the given type to their respective default values and subsequently calls the static constructor if it exists. The `setGeneratedValuesForStaticFields` requests values for the fields in the given type from the value provider that is also used to find values for method arguments and assigns the values to the fields. The main activity is inside the `run` method where Pex completes one concrete execution of the method under test and as a result returns the newly detected types that were used by the method. `{}` is an empty ordered set and the + `operator` the union of two sets where the new elements of the right hand set are appended to a copy of the left hand set.

## 2.4 Implementation

This section explains the concrete implementation details for the subproblems described at the end of Section 2.3.

The solution for subproblem 1 is less of an implementation problem and more a conceptual one of finding a good way to prune the huge space of possible initialization orders. This was already discussed in Section 2.3.

The implementation of subproblem 2 relies heavily on the value finding algorithm that Pex already provides. The value finder allows to register *Additional Root Symbols* that can be part of the path condition and the value finding process. For each detected static field a *Root Symbol* is added and symbolically assigned to the respective field before the test runs. Additional modifications were not necessary.

The technically most challenging part was subproblem 3 that requires to ability to replace the type initialization, a task that normally the runtime system performs, with our custom implementation. Replacing the default type initialization implementation consists of three tasks: (1) Intercepting the execution when a type initialization is necessary, (2) then running the static initializer at that moment, (3) and preventing the default implementation of doing the same.

The idea how to run the initialization code at the correct points in the test method is straight forward: We intercept the execution before a type is potentially initialized and keep an internal table with which types are initialized. If we decide after a lookup in this table that an initialization is necessary we call the static constructor with active instrumentation. The initialization is then a normal method call and Pex takes the path condition generated by the static constructor into account. Two problems arose when we implemented this: First, Pex does not instrument the static constructors by default. However, this can be changed by modifying a flag that is passed to the instrumentation engine. Second, normal callbacks cannot be used for that purpose (i.e. before a static field is loaded) because they do not provide a way to inject the call to the static constructor. Inside the callback the instrumentation is disabled, so we need to leave the callback before calling the static initializers. The solution was to modify the generated instrumentation code to insert a new type of callback that is invoked before a type is potentially initialized and allows to inject an instrumented method call. The Listings 2.6 and 2.7 illustrate how the methods are instrumented with the new callback:

```
public class Foo
{
  public static Foo StaticField;

  public static void StaticMethod(
      Foo f) { }
  public void InstanceMethod() { }
}

[PexClass]
public class Bar
{
  [PexMethod]
  public static void Baz()
  {
    Foo f = Foo.StaticField;
    f.InstanceMethod();
    Foo.StaticMethod(f);
  }
}
```

Listing 2.6: Marking a type as initialized

```
[PexClass]
public class Bar()
{
  [PexMethod]
  public static void Baz()
  {
    ITypeInitializer i;

    i = BeforeInit(typeof(Foo));
    if (i != null)
      i.Run();

    Foo f = Foo.StaticField;
    f.InstanceMethod();

    i = BeforeInit(typeof(Foo));
    if (i != null)
      i.Run();
    Foo.StaticMethod(f);
  }
}
```

Listing 2.7: Rewritten method

The static constructor is a normal method with the following restrictions: The name must be ".cctor" with the "specialname" flag set, have zero parameters and no return value. [5, II.10.5.3 p. 151]. The C# language does not provide a syntax to call the static constructor manually. However by using the DynamicMethod[20] class it is possible to emit IL code at runtime to call the static constructor. In contrast to the normal MethodBuilder, the DynamicMethod class allows us to circumvent the visibility checks, which is required because static constructors created by the C# compiler are always `private`. Listing 2.8 shows the method that creates a delegate for invoking the static constructor of a type:

```
public static Action GetCctorDelegate(Type type)
{
  Contract.Requires(type != null);
  if (type.TypeInitializer != null)
  {
    var dynamicMethod = new DynamicMethod("[Type Initialization]", typeof(
        void), Type.EmptyTypes, type.Module, true);
    var il = dynamicMethod.GetILGenerator();
    il.Emit(OpCodes.Call, type.TypeInitializer);
    il.Emit(OpCodes.Ret);

    return (Action)dynamicMethod.CreateDelegate(typeof(Action));
  }
  else
    return new Action(() => {});
}
```

Listing 2.8: Method for creating a delegate for a static constructor

To fully control the times when a static constructor is called, a way to prevent the CLI from calling it had to be found. The implemented solution inserts an additional instrumentation callback

that allows to skip the body of a static constructor as described by Listing 2.10. If the CLI initializes the type while it is in the `SkipInitializerList`, no real action is performed but the CLR-internal flag that the type is initialized will still be set. However, the skipping mechanism cannot distinguish a call from the runtime and a manual call which should really execute the body of the static constructor. The solution is to manually trigger the initialization of the type (and therefore prevent the CLI from calling the static constructor ever again later) while skipping is active and then remove the skipping flag so that manual calls become possible. The initialization is triggered by calling the static constructor manually as shown in Listing 2.9. The single call instruction to the static constructor may result in two actual calls: The first call by the runtime to initialize the type and second because we called the static constructor manually. However, both times the body of the static constructor is skipped and from this point in time the runtime considers the type to be initialized and does not automatically call its initializer anymore. The type is then removed from the internal list to allow manual calls.

```
public static Action
    MarkTypeAsInitialized(Type type)
{
  Contract.Requires(type != null);
  SkipInitializerList.Add(type);
  Action cctor = GetCctorDelegate(
      type);
  cctor();
  SkipInitializerList.Remove(type);
}
```

Listing 2.9: Marking a type as initialized

```
class RewrittenCctor
{
  static RewrittenCctor()
  {
    if (SkipInitializerList.Contains
        (typeof(RewrittenCctor)))
      return;

    /* Code of the original static
    initializer is inserted here */
  }
}
```

Listing 2.10: Rewritten static constructor

A limitation of this approach is that if the type has BeforeFieldInit semantics, it can be initialized at any point before its first use, so we cannot be certain that the static initializer did not already run. The changes to the type initialization behavior in version 4.0 of the CLR alleviates the problem but does not completely remove it because direct field accesses still use eager initialization.

To solve subproblem 4 again the DynamicMethod class was used. The Listing 2.11 shows how to implement a method that resets the static fields of a type to their default value. The approach is straight forward: Using reflection the list of static fields of a type is enumerated. A DynamicMethod is created that, for each field, assigns the return value of the helper method `Default<T>` to the field. The `Default<T>` method is generically instantiated with the type of the field value. The helper method simply returns the default value for the generic parameter `T`. Using the helper method simplifies the generated IL because generating the default value of a type depends on if the type is a value or reference type. However assigning the return value of a method is uniform for all types.

```
public static Action GetTypeResetMethod(Type type)
{
  Contract.Requires(type != null);
  DynamicMethod dynamicMethod = new DynamicMethod("ResetStaticFields", typeof
      (void), Type.EmptyTypes, Type.Module, true);
  var il = dynamicMethod.GetILGenerator();
  var defaultMethod = typeof(InitializerService).GetMethod("Default",
      BindingFlags.Static | BindingFlags.NonPublic);

  foreach (var field in type.GetFields(BindingFlags.Static | BindingFlags.
      Public | BindingFlags.NonPublic))
  {
    if (!field.IsLiteral)
    {
      il.EmitCall(OpCodes.Call, defaultMethod.MakeGenericMethod(field.
          FieldType), null);
      il.Emit(OpCodes.Stsfld, field);
    }
  }

  il.Emit(OpCodes.Ret);

  return (Action)dynamicMethod.CreateDelegate(typeof(Action));
}

private static T Default<T>() {
  return default(T);
}
```

Listing 2.11: Method for creating a method that resets the static fields of a type

## 2.5 Compatibility with existing test suites

A possible manual solution to test static fields is to use a test setup and cleanup method. The setup method would initialize the static fields to a specific state and the cleanup method can ensure that no state information is leaked between test runs. If one would use this technique together with the extension presented in this chapter, by default, it would overwrite the manually prepared values of the static fields and the test might stop working. To prevent this a compatibility mode has been included that disables the automatic type initialization and value generation for the fields that are being set up manually. Additionally it detects if the setup and cleanup method do not completely prevent the keeping of state between test runs and reports a warning. The Listing 2.12 illustrates this.

This test class uses a test initializer that sets the value of `i` to 10 and thus prevents the exception in the `Test` method. Without the compatibility mode, the extension would simply replace the contents of `i` and Pex would find a way to raise the exception, therefore signaling a false positive. However, the current setup also keeps the value of the `j` field over multiple test runs. This is detected by the extension and a warning for `j` is reported. By adding the cleanup method from Listing 2.13 the problem is fixed and the warning disappears. The set of fields that possibly keep state between two test runs $F_{Leak}$ is calculated using the formula:

$$F_{Leak} = \left( \bigcup_{t \in T_{Init}} \text{StaticFieldsOfType}(t) \right) \cap F_{Test} \setminus F_{Cleanup}$$

where $T_{Init}$ is the set of types which are potentially initialized by the `TestInitialize` method, $F_{Test}$ the set of fields modified by the test method itself and $F_{Cleanup}$ the fields modified by the `TestCleanup` method.

```
[PexClass]
[TestClass]
public class ManualSetupCleanup {
    [TestInitialize]
    public void Initialize() {
        Foo.i = 10;
    }

    [PexMethod]
    public void Test() {
        if (Foo.i != 10)
            throw new Exception();

        Foo.j++;
    }

    private class Foo {
        internal static int i;
        internal static int j;
    }
}
```

Listing 2.12: Manual test setup and missing cleanup method

```
[TestCleanup]
public void Cleanup()
{
    Foo.j = 0;
}
```

Listing 2.13: Correct cleanup method

## 2.6 Limitations

In the process of reinitializing a class only managed resources are taken into consideration. If the static constructor allocates unmanaged resources (e.g. an unmanaged memory buffer) the handle to this buffer will be lost in the reset process and a new buffer is allocated when the initializer runs again. Because this operation is potentially repeated several times the process may run out of resources. A possible solution is to wrap the unmanaged resource in a managed object with the corresponding finalizer that will release the resource when the object is garbage collected.

The CLR gives strong guarantees about the thread-safety of static initializers in multi-threaded applications. If a test starts multiple threads the correctness of the results cannot be guaranteed anymore.

Finally, the simulated initialization behavior of BeforeFieldInit classes, while still being inside of the specification boundaries, differs from behavior of CLR2.0 and CLR4.0. The CLR initializes BeforeFieldInit types eagerly during JIT compilation, however the simulation is always lazy. The parts of the code that are dependent on implementation details of the runtime can be detected by the analysis introduced in Section 3.5 and subsequently fixed.

# Chapter 3

# Enhanced Analysis

## 3.1 Overview

The extension presented in Chapter 2 allows the use of Pex to find bugs related to type initialization and static fields. However, the technique used by Pex does not allow checking large codebases to determine if the new additions would allow to find more bugs in real world code. The focus of Pex lies on detailed analysis of a small number of methods and is therefore too slow and expensive in terms of resources to run it for example on all methods in a project. Additionally it is not possible to gather statistical information about the usage and frequency of occurrence of static initializers and static fields. To find answers to these questions Chapter 3 introduces a tool that uses a fast, coarse-grained analysis to find methods where the newly developed extensions for Pex are expected to make a difference and collect statistical information about the usage static intializers and fields.

This chapter first describes the approach and implementation that is common for all kinds of analyses and then discusses the specifics of each analysis separately.

## 3.2 General Approach

As a first step towards designing such a tool, the fundamental decision on the type of input format over which the analysis will be performed had to be made. We chose to do our analysis on the IL level as opposed to directly on the source code because this allowed us to focus more on the analysis aspect and leave the parsing etc. to the compiler. To open and read the assembly files the open-source project Mono.Cecil[7] was chosen over the *ExtendedReflection* module that Pex uses because of its intuitive and comprehensive API and to avoid dependencies to proprietary code. As a result of this decision process the tool takes a list of CIL assemblies as input and generates the output by statically analysing the IL code in the input assemblies. We denote a set of input assemblies as the *Analysis Context*. The *Analysis Context* must be self-contained, i.e. contain all referenced assemblies and therefore almost always the core assemblies of the BCL.

To simplify the evaluation process of the Pex extensions, the first task was to find methods inside the *Analysis Context* that would benefit from the newly added extension. An initial naive approach was to search the body of all methods for instructions that read and write static fields. While this method is simple and fast, there are strong disadvantages in terms of accuracy and detail. This naive analysis returns only a single boolean value per method and there is no indication about the usage of the static fields. Since this kind of analysis could not be improved to provide more detail, a more elaborate approach was chosen next.

The new approach is to use a static symbolic execution engine. The symbolic execution trace can easily provide the list of read/write effects on static fields as well as their usage in form of symbolic expressions. Additionally it allows inter-procedural analysis and works in the presence of simple pointer operations, which occur in verifiable managed code when using `ref` and `out` parameters. The static symbolic execution is still fast enough and fits well with the kind of information that we wanted to gather, therefore this is also the approach in the final version of the tool.

## 3.3 Implementation

The basis for all the analyses that are performed by the tool is the *symbolic execution trace* of a method. The symbolic execution trace is the result of symbolically executing the CIL instructions of a method, simulating the execution stack, local variables, argument values etc. Concretely the output is a control flow graph where the vertices are basic code blocks (i.e. sequences of instructions that will never be interrupted with the exclusion of exceptions) and the edges are transitions between these blocks annotated with a condition that states under which circumstances the transition can be taken. The basic blocks consist of symbolic instructions that are higher-level than CIL but lower-level than C#. The symbolic instructions are in turn built of symbolic expressions (Terms).

The symbolic execution trace is built by interpreting the CIL instructions. To track the values that the instructions can reference, a state object is used that contains symbolic representations of the values of the evaluation stack, local variables, arguments, path condition, exception state etc. The single instructions are combined into increasingly complex expression trees and finally into *symbolic instructions* that look similar to C# code. The interpretation works block wise: Each basic block has a start state, which is then transformed by applying the CIL instructions. If an instruction has more than one possible result state (e.g. branch) the final state for the basic block is stored and the process restarts for all the target basic blocks.

If a basic block has multiple entry transitions, consequently it has multiple start states. These states are then merged into a single new start state for the basic block. If an expression has different values in the two entering states, these values are combined into a *multi value term*. The semantics of a multi value term is that it represents an arbitrary but fixed value out of a set of possible values. This approach deliberately loses information[1] when combining multiple values, but by doing that circumvents the exponential state space explosion when code contains many branches. To avoid infinite simulation when interpreting loops, the state is only recalculated once for every entry transition on the basic block.

A shortcoming of this approach is that inter-procedural analysis is not provided by the symbolic execution engine but must be handled by each subsequent analysis separately. A limitation specific to this implementation is that complex constructs like arrays or object instances are not simulated symbolically by the engine to reduce complexity and increase speed. However since the main focus is on static fields, the analysis results are rarely affected by this. Futhermore some instructions and prefixes (tailcall, TypedReference, unsafe code) are not supported by the engine. However, these features are unused by the current C# compiler (tailcall), only used by mscorlib (a hidden C# language feature: TypedReference) or not very common (unsafe code).

Listing 3.1 contains a C# version of the classical iterative Euclidean algorithm to compute the greatest common divisor and Figure 3.1 a visualization of the symbolic execution trace.

[1] As an example: If two variables a and b contain $a = 1$ and $b = 2$ in one input state and $a = 3$ and $b = 4$ in another input state, combinations like $a = 1$ and $b = 4$ are not possible, however the model does not reflect this.

```
public int Euclid(int a, int b) {
    if (a == 0)
        return b;
    else {
        while (b != 0) {
            if (a > b)
                a -= b;
            else
                b -= a;
        }
    }
    return a;
}
```

Listing 3.1: Classical Euclidean Algorithm (iterative)

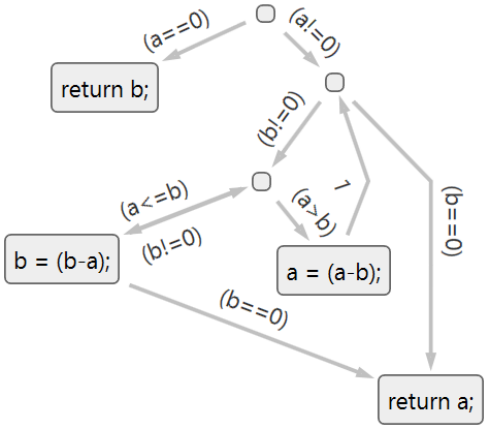


Figure 3.1: Graphical visualization of the *Symbolic Execution Trace*

## 3.4 Finding methods for the evaluation

The first objective of the tool was to generate a list of methods that would be suitable for evaluating the extensions for Pex developed in Chapter 2. The taken approach assigns a score to each method based on the information that can be retrieved from the symbolic execution trace. The score is a number that indicates the likeliness of there being a bug related to static fields in a method. The output of the analysis is a list of methods, ordered by the score, on which then Pex can be applied for evaluation purposes. To keep the number of methods feasible minimal score threshold can be specified.

The assignment of a method score is done in two steps. In the *vulnerability analysis* the body of a method is inspected for usages of static fields that can result in unexpected exception or assertion errors. The second step, internally referred to as *accessibility analysis* determines whether the static fields that have a potential error are modifiable from the public API to reduce the number of false positives.

### 3.4.1 Vulnerability analysis

The vulnerability analysis is carried out by scanning the symbolic execution trace of a method for expressions that could result in a runtime exception if the code were executed. The following types of expressions are considered in the scan:

- Dereferences because of their potential for a `NullReferenceException`.
- The arguments in calls to the assertion methods `Debug.Assert`, `Contract.Assert`, `Contract.Ensures`, `Contract.Invariant`, `Contract.Requires` (only for nested method calls)
- In case of a `throw` instruction, the path condition.
- Expressions that could result in one of the exceptions that are thrown by the execution engine like `DivisionByZero`, `ArithmeticException`, `OverflowException`

Each of the matching expressions is searched for static field references to determine if a static field could be the cause of the potential exception. For each occurrence of a static field reference in an expression a *Vulnerability* is reported. A *Vulnerability* is defined as a triple consisting of

the instruction that potentially fails, the static field that is responsible for the failure and a score between 0 and 100 that describes how close the relation between the field and the potential failure is. The score is calculated as the inverse of the depth of the static field reference in the offending expression and then scaled to a value between 0 and 100. This is based on the observation that in general the influence of an subexpression to the total result decreases with its nesting level.

The list of vulnerabilities is generated on a per method basis. However direct and indirect callees of the method are taken into account when running the analysis. As an example, if a static field is used as an argument to a method and the callee does a null check on its parameter, this is reported as a potential vulnerability. An exception from this are calls into mscorlib.dll and System.dll where only the direct callee is taken into account, but nested calls are ignored. This is done to avoid the great number of non-analysable methods in the internals of these libraries and to improve performance. The impact of this limitation is relatively small because most of the public methods immediately validate their arguments before passing them on to internal methods. Combined with the fact that not the BCL but the project itself should be tested puts the effects of this limitation in perspective. A second limitation are recursive calls, which are ignored by the vulnerability analysis to avoid infinite recursion.

### 3.4.2 Accessibility analysis

The vulnerability analysis yields many false positives, because all static fields are treated equally, even if they are private or only written once. To account for this, a per field-analysis is performed to determine how easy or difficult it is to modify a static field from the public API. This idea is based on the observation that most of the time the public API of a software component is tested, not the internals and therefore reduces the number of false positives.

The result of the analysis is a score between 0 and 100 that describes how accessible the field is from the public API where 0 indicates that no way to change the field was found and 100 that the field is directly writeable because it is public or there exists an accessor method for it (e.g. property setter). Other values indicate that the value cannot be directly set but somehow be influenced by using methods from the public API.

The score is determined by inspecting all write effects that were found on this static field. For each write effect a score is determined that expresses the accessibility of the field through that write effect. The maximum score across all write effects is then used as the accessibility score of the field.

### 3.4.3 Method score calculation

After the results of both the vulnerability and accessibility analysis are available, each method receives a score between 0 and 100 where 0 means that the analysis could not find a way to provoke an exception using static fields in a method and 100 that there exists the possibility to provoke an exception by writing certain values into static fields before calling the method. Other values express the certainty of there being a potential bug.

The score is calculated by multiplying the score of each vulnerability with the accessibility score of the offending field, normalizing the values again to a range between 0 and 100 and taking the maximum.

$$\forall m \in Methods.\ \text{score}(m) = \max_{v \in vuln(m)} \frac{\text{score}(v) * \text{score}(\text{field}(v))}{10000}$$

Using this formula the method receives the score of the most exploitable vulnerability scaled with accessibility score of the static field that is required for exploiting the vulnerability.

## 3.5 Testing the effects of eager initialization

As stated in Section 1.2 the location where a static initializer is run in case of eager initialization cannot be predicted. As a result, the initialization behavior of the type initializer system developed in Chapter 2 is different from the normal behavior. Hence, static initializers might run at different points during testing and production or even depend on the version of the CLR.

Consider the code in Listing 3.2. Running the test method once will work on the CLR 2.0 and 4.0, however always fail when running under Pex with the extensions. When the `Test` method is called running on the CLR it will set `i` to 0. To execute the call to the `Trigger` method, the JIT compiler first has to translate it. Since the field `j` is referenced in that method and `Foo` has *BeforeFieldInit* semantics the runtime will eagerly initialize `Foo` which results in `i` being incremented. The condition in the if statement evaluates to false, leaving the field `j` unchanged. The assertion holds because the JIT compilation of `Trigger` eagerly initialized type `Foo`. A second call to `Test` will fail because `Foo` can only be initialized once. The Pex type initializer however is truly lazy in all cases, because it is unaware of the JIT process. Since the field `j` is never actually accessed, Foo would never be initialized and the assertion always fails. The code in Listing 3.3 works on CLR 2.0 but the assertion fails under CLR 4.0 or Pex.

```
public static class Eager {
  static int i;

  public static void Test() {
    i = 0;
    Trigger();
    Debug.Assert(i == 1);
  }

  static void Trigger() {
    if (i == 1000)
      Foo.j++;
  }

  static class Foo {
    public static int j = i++;
  }
}
```

Listing 3.2: Eager initialization in the CLR

```
public class Clr20vs40 {
  static int i;

  public static void Test() {
    i = 0;
    Foo.Greet();
    Trace.Assert(i == 1);
  }

  static class Foo {
    static int j = i++;
    public static void Greet() {
      Console.WriteLine("Hi!");
    }
  }
}
```

Listing 3.3: Lazy or truly lazy?

Bugs of this kind are very hard to find because they depend on the runtime and build configuration. Because the initialization behavior and performance are related, these differences can only be detected when running in release mode when its already to late. To determine whether real-world projects are susceptible to these bugs, the existing analysis tool was extended to cover this problem area.

Listing 3.4 shows the sample that will be used to explain the approach and implementation of the eager initialization effect analysis. In the example the assertion will fail if the static initializer runs between the assignment of 0 to `i` and the call to `Trace.Assert`. Note that it is never the case if the code was run on the CLR 2.0, 4.0 or Pex but the assertion is breakable in theory according to the specification.

```
public class EagerInitialization {
  static int i;

  public static int Test() {
    i = 0; // If Foo is initialized after this line
    Trace.Assert(i == 0); // this assertion fails.
    return Foo.j; // The reference to Foo.j is required to have a connection
        between the Test method and class Foo
  }

  static class Foo {
    public static int j = i++;
  }
}
```

Listing 3.4: Eager initialization example used as illustration

### 3.5.1 Approach

The basic concept to detect eager initialization effects is to find the instructions in a method, where the outcome of the method differs if a static initializer runs before or after the instruction. A naive approach would be to generate two test cases for each instruction and each type with BeforeFieldInit semantics in the AppDomain. In first test case the initializer would run before and in the second after the instruction. While this brute-force technique is simple to implement, the number of test cases would be far from realistic even for short methods. In order to reduce the number of test cases, both the list of instructions and types must be shortened drastically.

The idea is to build a list of *critical points* where a critical point is defined as a triple of an instruction $i$, a field $f$ and a type $t$. A critical point is evidence that the field $f$ is accessed by both, the instruction $i$ and the static initializer of $t$ in a conflicting way (i.e. at least one of them writes to the field). Each critical point describes the potential for a different outcome based on the order in which instruction $i$ and the static initializer of $t$ are executed.

The approach to shorten the list of types to consider is based on the assumption that the runtime will not eagerly initialize arbitrary types that are completely unrelated to the method, but only types that the method would potentially initialize anyway. Therefore the set of types that needs to be considered ($prospect(i)$) for an instruction $i$ is the set of types that are potentially initialized by the method minus the set of types that must have been initialized to reach the instruction $i$. Formally expressed:

Let $T$ be the set of all types in the AppDomain, $m$ the sequence of instructions of the method being analyzed and $E$ the set of all finite execution paths in method $m$ where an execution path is a sequence of instructions. We define the set $force(i)$ to contain all types that must have been initialized after instruction $i$ has been executed:

$$\forall i \in m, t \in T : t \in force(i) \Leftrightarrow \text{Instruction } i \text{ forces the initialization type } t$$

Now we define $init_k(e)$ which contains the types that must have been initialized after executing $k$ instructions in execution path $e$ recursively over the number of steps:

$$\forall e \in E, init_0(e) := \varnothing$$

$$\forall e \in E, 0 < k \leq |e| : init_k(e) := init_{k-1}(e) \cup \text{force}(i_k^e)$$

where $i_k^e$ is the $k$th instruction in execution path $e$.

Based on this we define $init(i)$ to be the set of types that must have been initialized on every path that reaches instruction i:

$$\forall i \in m : init(i) := \bigcap_{\forall e \in E, k \in \mathbb{N} : i_k^e = i} init_k(e)$$

To get the set $init(m)$ of all types that are potentially initialized by a method we build the union over the types that must be initialized to reach an arbitrary instruction in $m$

$$init(m) := \bigcup_{i \in m} init(i)$$

And finally we can define the set of prospect types for an instruction $i$ as

$$\forall i \in m.prospect(i) = (init(m) \setminus init(i)) \cup force(i)$$

For each instruction $i$ that reads a static field $f$, we calculate the union of all write effects of the static initializers of the types in $prospect(i)$. For each write effect of type $t$ to field $f$, add a *critical point*$(i, f, t)$ to the list. For each instruction $i$ that writes static field $f$, calculate the union of all read and write effects of the static initializers of the types in $prospect(i)$. Again add for each read or write effect of type $t$ to field $f$, *critical point*$(i, f, t)$ to the list.

A limitation of this approach is that the analysis only takes accesses to static fields into account. However, because of aliasing other instructions could be critical as well and those are currently missed by the tool. Trying to solve this problem would increase the complexity drastically, especially when considering access paths that include multiple indirections.

### 3.5.2 Implementation

The list of critical points is generated according to the definition given in Subsection 3.5.1. Using an IL rewriter, markers are inserted into the code before and after each *critical* instruction for each *critical* type. Listing 3.5.2 shows the rewritten method after the markers have been inserted.

```
  public static int Test() {
    Critical(typeof(Foo));
    i = 0;
    Critical(typeof(Foo));

    Critical(typeof(Foo));
    Trace.Assert(i == 0);
    Critical(typeof(Foo));

    return Foo.j;
  }
}
```

After that, Pex runs over the method and injects the call to the static constructor at a different marker on each run. If a change in program behavior (i.e. control flow or throwing an exception) is detected, Pex will automatically explore these different execution paths. For the concrete example, the `Test` method is run four times. In the first run `Foo` will be initialized before the assignment to `i`, on runs two and three between the assignment and the assertion and on run four after the assertion. The tool detects the assertion errors in run two and three and reports to the user the exact IL instruction offset where an invocation of the static initializer would be problematic.

## 3.6 Dependencies between static initializers

A common property of many of the introduced examples of unstable static field code is the use of two classes with static initializers that reference the same static fields in a conflicting way. Conflicts can be introduced if at least one of the static initializers writes to a static field and another static initializer reads or writes to the same static field. In that case the program behavior may be different based on the initialization order of these types. To build statistics on how common this situation in real world code is, an analysis was included that calculates all these conflicts based on the read and write effects of the static initializers of the types whose initialization can be forced by a method.

The Pex extension implemented in Chapter 2 supports the manual specification of a list of type pairs $(t_1, t_2)$, that ensures that there is at least one test which initializes first $t_1$ and then $t_2$ and one test in which the initialization order is reversed. The lines of the algorithm in Listing 2.5 that provide this functionality are marked with $*$. The list of conflicting types can be used to find pairs of types where a conflict between the static initializers of the two types exists and therefore make it interesting to test both initialization orders. When Pex is used in conjunction with the tool introduced in this chapter, the user can annotate methods with these type pairs. An example snippet that contains a static initialization conflict can be found in Listing 2.1.

A generalized version of the former problem is the following: If there is a conflict between any method and a static initializer (i.e. is there a method or field that can change the behavior of a static initializer) which was internally dubbed *dependency race*. The reasoning behind being, that similarly to a race conflict in multi-threaded applications, because of eager initialization, the outcome of a dependency race is nearly unpredictable and the correctness of an application should not depend on the concrete execution order. An example that contains a *dependency race* where the outcome depends on the version of the CLR was already introduced in Listing 3.3.

Similarly to the static initializer conflicts analysis, a check for *dependency races* is included in the tool, allowing to determine for each type the set of methods where a dependency race is present between them.

# Chapter 4

# Evaluation

## 4.1 Quantitative Evaluation

To measure the effectiveness of the presented techniques for detecting bugs related to static fields a number of open source projects were analysed. To compare the accuracy and impact of each of the approaches, the analysis was run multiple times using different sets of tools.

For each method the following metrics where determined:

- **# Static Field Referencing Instructions (SFRI):** The number of instructions in the method that reference static fields. For more information see Section 3.2.
- **# Maximum Forced Type Initializations (MFTI):** The number of distinct types that can be forced to be initialized by a method.
- **Vulnerability Score (VS):** The vulnerability score as described in Section 3.4.
- **# Critical Points (CP):** The number of critical points as described in Section 3.5.1.
- **# Static Initializer Conflicts (SIC):** The number of conflicts between static initializers of types that can potentially be initialized by the method as described in Section 3.6.
- **# Dependency Races (DR):** The number of dependency races of the method as described in Section 3.6.
- **Pex without the extension (BPR):** The number of bugs related to static fields detected by Pex without the extension introduced in Chapter 2.
- **Pex, type initialization only (BPTIO):** The number of bugs related to static fields detected by Pex with the extension used to include static initializers in the dynamic symbolic execution.
- **Pex, with modification (BPFM):** The number of bugs related to static fields detected by Pex with the extension used to generate values for static fields.
- **Pex, with modification and swap list (BPSL):** The number of bugs related to static fields detected by Pex with the extension used to generate values for static fields and using the list of conflicts between static initializers to exercise different initialization orders.

To gather further information on how widespread the use of static fields is the total number of static fields **(SF)**, read-only static fields **(IOSF)**, total fields **(TF)**, methods that use static fields (i.e. methods where #SFRI > 0) **(SFRM)** and methods **(MC)** in general was calculated. Finally **FA** contains the number of analyses that failed and **TA** the number of total analyses performed.

To determine the accuracy of the various analysis techniques the normalized correlation between the metric that the analysis provides and the real bug count was calculated. The numerical results grouped by project can be found in Table 4.1 and Table 4.2.

The table indicates that only **9%** of all fields are static and about **11%** of all methods reference static fields. These values are fairly low and a partial explanation why the number of bugs that involve static fields is fairly low.

## 4.2 Usage patterns for static initializers

During the study of the open source projects a number of common usage patterns emerged. This section describes four of the these patterns and discusses the benefits, weaknesses and dangers.

### 4.2.1 Static initializers to provide default values

By far the most common use of static initializers is to provide default values for static fields. This is not surprising as it is the main purpose of static initializers as noted in [5, II.10.5.3 p. 151]. If a static initializer only accesses static fields that belong to the same type as the static initializer itself, this use is free of any danger to fall subject to any of the hard to find bugs introduced in the last chapters. If a type only contains fields with primitive types this property is easily verifiable. More complex are the cases where the value is a newly created instance, since the constructor of that instance could contain complex code or its source code may be unavailable.

Noteworthy is the use of P/Invoke in a static initializer. If a P/Invoke method is called in a static initializer this has many side effects that are not directly apparent. If this is the first call to a method of a specific unmanaged library, the call will load the library from disk into the address space of the process. On Windows systems, this leads to a call to the DllInit method which can contain arbitrary code. Another side effect is that the value of Marshal.GetLastError() can change, possibly masking an error. Additional information can be found at [3].

### 4.2.2 Public static read-only fields

In general the Microsoft Framework Design Guidelines state that instance fields should not be public or protected[19]. No statement is made about static fields, however the use of static read-only fields for *predefined objects* is emphasised[4, p. 161]. In fact, public static read-only fields are heavily used by the BCL itself. An example of this are Dependency Properties[17] that are widely used by the Windows Presentation Foundation[27]. Dependency Properties augment the existing concept of properties in .NET to support styling, animation and data binding. In particular every Dependency Property owns a set of metadata information that is stored in a public static read-only field. In Listing 4.1 a property `Text` on an object `TextBox` is declared. The public static read-only field `TextProperty` contains the metadata (in this example the default value of `string.Empty` as well as name, value type and declaring type). The values of Dependency Properties are stored in an internal dictionary where the DependencyProperty instance acts as a key when reading and modifying the value of this property over the normal getter and setter of the `Text` property.

Table 4.1: Statistical evaluation results

| Project | SFRI | MTFI | VS | SIC | DR | CP | BPR | BPTIO | BPFM | BPSL |
|---|---|---|---|---|---|---|---|---|---|---|
| Umbarco | 18.84 | 0.11 | 6.58 | 0.04 | 15.28 | 0.03 | n/A | n/A | n/A | n/A |
| Ncqrs | 9.15 | 0.10 | 5.30 | 0.00 | 8.39 | 0.02 | 0.01 | 0.02 | -0.01 | -0.01 |
| Ncolony | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | n/A | n/A | n/A | n/A |
| FacebookClient | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | n/A | n/A | n/A | n/A |
| NRefactory | 0.50 | 0.13 | 16.44 | 0.00 | 0.21 | 0.00 | -0.06 | -0.06 | 0.08 | 0.08 |
| Bot | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | n/A | n/A | n/A | n/A |
| MishaReader | 0.01 | 0.03 | 2.14 | 0.00 | 0.00 | 0.00 | -0.02 | -0.02 | -0.03 | -0.03 |
| Dsa | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | n/A | n/A | n/A | n/A |
| DbExecutor | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | n/A | n/A | n/A | n/A |
| NSynth | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | n/A | n/A | n/A | n/A |
| Boogie | 1.67 | 0.80 | 35.40 | 0.00 | 0.04 | -0.01 | 0.01 | 0.04 | 0.93 | 0.93 |
| Rxx | 0.64 | 0.00 | 0.11 | 0.19 | 0.16 | 0.01 | n/A | n/A | n/A | n/A |
| NitoAsync | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | n/A | n/A | n/A | n/A |
| OmegaNet | 0.04 | 0.04 | 4.40 | 0.00 | 0.00 | 0.00 | 0.00 | -0.01 | -0.03 | -0.03 |
| Frost | 9.88 | 0.03 | 0.00 | 0.00 | 8.84 | 0.05 | n/A | n/A | n/A | n/A |
| AutoDiff | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | n/A | n/A | n/A | n/A |
| iMoreThanReader | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | n/A | n/A | n/A | n/A |
| IDComLog | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | n/A | n/A | n/A | n/A |
| EnterpriseLibrary | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | n/A | n/A | n/A | n/A |
| AegisVoterList.UI | 11.13 | 0.14 | -0.01 | 0.50 | 7.23 | 0.16 | n/A | n/A | n/A | n/A |
| ChaosUtil | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | n/A | n/A | n/A | n/A |
| CciAst | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | n/A | n/A | n/A | n/A |
| AegisVoterList.DigitalVoterList | 11.38 | 0.15 | -0.01 | 0.51 | 7.40 | 0.17 | n/A | n/A | n/A | n/A |
| DarkHeresy | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | n/A | n/A | n/A | n/A |
| Boggle | 55.85 | 29.46 | 870.68 | 0.02 | 7.58 | 0.16 | 44.51 | 42.18 | 64.10 | 64.10 |
| Neovolve | 0.06 | 0.15 | 4.97 | 0.00 | 0.00 | 0.00 | -0.05 | -0.05 | -0.06 | -0.06 |
| Linq2Rest | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | n/A | n/A | n/A | n/A |
| Mono.Cecil | 29.34 | 0.06 | -0.01 | 0.00 | 29.38 | 0.13 | n/A | n/A | n/A | n/A |
| Scrabble | 6.47 | 6.00 | 142.00 | 0.01 | 1.37 | 0.06 | 7.57 | 7.45 | 7.62 | 7.62 |
| QuickGraph | 0.06 | 0.02 | 0.64 | 0.00 | 0.00 | 0.00 | -0.01 | -0.01 | -0.01 | -0.01 |
| | 5.17 | 1.24 | 36.29 | 0.04 | 2.86 | 0.03 | 5.77 | 5.50 | 8.07 | 8.07 |

Table 4.2: Statistical evaluation results

| Project | SF | IOSF | TF | SFRM | MC | FA | TA |
|---|---|---|---|---|---|---|---|
| Umbarco | 8969 | 5381 | 39698 | 12131 | 117362 | 22802 | 1290982 |
| Ncqrs | 3389 | 2214 | 18715 | 5339 | 56547 | 60 | 622017 |
| Ncolony | 3 | 0 | 10 | 3 | 44 | 0 | 484 |
| FacebookClient | 15 | 7 | 151 | 28 | 547 | 0 | 6017 |
| NRefactory | 1758 | 600 | 7578 | 3064 | 24321 | 129 | 267531 |
| Bot | 453 | 260 | 1725 | 988 | 4442 | 0 | 48862 |
| MishaReader | 522 | 240 | 3247 | 1177 | 9328 | 0 | 102608 |
| Dsa | 2 | 0 | 49 | 20 | 259 | 0 | 2849 |
| DbExecutor | 9 | 1 | 132 | 7 | 173 | 0 | 1903 |
| NSynth | 67 | 56 | 348 | 143 | 1464 | 0 | 16104 |
| Boogie | 255 | 73 | 2417 | 783 | 5364 | 0 | 59004 |
| Rxx | 3680 | 669 | 21719 | 5627 | 44071 | 8299 | 484781 |
| NitoAsync | 216 | 56 | 3755 | 964 | 6142 | 0 | 67562 |
| OmegaNet | 1299 | 605 | 10168 | 2299 | 29519 | 10 | 324709 |
| Frost | 688 | 644 | 4766 | 1540 | 7282 | 1061 | 80102 |
| AutoDiff | 30 | 0 | 130 | 31 | 286 | 0 | 3146 |
| iMoreThanReader | 224 | 142 | 989 | 539 | 2656 | 39 | 29216 |
| IDComLog | 82 | 6 | 1399 | 103 | 1643 | 0 | 18073 |
| EnterpriseLibrary | 910 | 275 | 5307 | 4319 | 19072 | 0 | 209792 |
| AegisVoterList.UI | 1205 | 1163 | 4364 | 1024 | 9769 | 0 | 107459 |
| ChaosUtil | 9 | 2 | 148 | 13 | 715 | 0 | 7865 |
| CciAst | 182 | 38 | 5091 | 409 | 19129 | 0 | 210419 |
| AegisVoterList.DigitalVoterList | 1198 | 1163 | 4209 | 1014 | 9551 | 0 | 105061 |
| DarkHeresy | 158 | 0 | 1138 | 100 | 2900 | 0 | 31900 |
| Boggle | 318 | 78 | 936 | 398 | 1825 | 0 | 20075 |
| Neovolve | 1226 | 250 | 4494 | 2182 | 16074 | 870 | 176814 |
| Linq2Rest | 327 | 182 | 1677 | 714 | 5813 | 0 | 63943 |
| Mono.Cecil | 302 | 240 | 1122 | 190 | 3008 | 0 | 33088 |
| Scrabble | 476 | 116 | 4429 | 1300 | 8164 | 0 | 89804 |
| QuickGraph | 498 | 244 | 5721 | 1066 | 15696 | 5829 | 172656 |
| | 28470 | 14705 | 155632 | 47515 | 423166 | 39099 | 4654826 |

```
public string Text
{
  get { return (string)GetValue(TextProperty); }
  set { SetValue(TextProperty, value); }
}

public static readonly DependencyProperty TextProperty =
  DependencyProperty.Register("Text", typeof(string), typeof(TextBox), new
      UIPropertyMetadata(string.Empty));
```

Listing 4.1: Declaration of a dependency property

The `readonly` modifier only prevents to change the contents of the field. However, if the field contains a reference to a mutable object, changes that object to are still possible. This is addressed with the Code Analysis rules [18, DoNotDeclareReadOnlyMutableReferenceTypes (CA2104)] and [16, ArrayFieldsShouldNotBeReadOnly (CA2105)]. Especially in the case of arrays this can be confusing as Listing 4.2 shows. This is in contrast to for example the `immutable` keyword in the D programming language where immutability is transitive (Listing 4.3)[12]. On the other hand, the semantics of the `readonly` modifier can also be beneficial for skipping `null` checks when the constructor initializes the field with a valid reference.

```
public class Foo {
  public static readonly int[]
      MyFavoriteInts = { 2, 16, 42,
      101, 9999 };
}
public class Program {
  static int Main(string[] args) {
    Foo.MyFavoriteInts[0] = 10;
    return Foo.MyFavoriteInts[0]; //
        Returns 10
  }
}
```

Listing 4.2: Readonly arrays are not immutable

```
immutable char[] s = "foo";
s[0] = 'a'; // error, s refers to
    immutable data
```

Listing 4.3: Immutability in D

### 4.2.3 Static initialization for singletons

By using the strong guarantees given about the initialization process in combination with concurrency[5, I.8.9.5 p. 43]static fields can be used to simplify the lazy creation of the instance when using the Singleton Pattern[10, p. 144]. Listing 4.4 shows the implementation for lazily creating the singleton in a multi-threaded environment without instantiating the object in the static initializer as proposed by the Microsoft guidelines[23].

```
public class MySingleton {
  private MySingleton() { }

  private static volatile MySingleton instance;
  private static object instanceLock = new object();
  public static MySingleton Instance {
    get {
      if (instance == null) {
        lock (instanceLock) {
          if (instance == null) {
            instance = new MySingleton();
          }
        }
      }
    }
  }
}
```

Listing 4.4: Classical implementation of the singleton pattern

In contrast to that, the second version in Listing 4.5 is shorter, simpler to understand and easier to implement correctly. The locking and safeguard that only one instance will be created is handled by the CLR. The instantiation is still lazy because of the empty static initializer that prevents the type from having BeforeFieldInit semantics.

```
public class MySingleton {
  private static readonly MySingleton instance = new MySingleton();
  static MySingleton() { }
  private MySingleton() { }
  public static MySingleton Instance { get { return instance; } }
}
```

Listing 4.5: The singleton pattern leveraging static initializers

### 4.2.4 Public static mutable fields

As discussed in Section 4.2.2 the official Field Usage Guidelines[22] state that instance fields should not be public or protected. Surprisingly no statement is made about static fields, even though the same reasoning as for instance fields applies. The Static Field Naming Guidelines[25] suggest to use static properties instead of public static fields, this is however not checked by a Code Analysis rule[14].

A framework that makes use of public static fields is the Caliburn.Micro MVVM Framework[6]. The purpose is extensibility by offering the option to replace certain services of the framework with custom implementations. A public static field is initialized with a reference to the default implementation and it is open to the user to replace this reference. Listing 4.6 shows the important parts of the `ViewLocator` class to illustrate this pattern. In this class the field `NameTransformer` contains a reference to a service instance. The field is initialized with a default implementation (`NameTransformer` class) and the `AddTypeMapping` method then uses the methods provided by the NameTransformer service.

```
public static class ViewLocator {
  /* ... */
  public static NameTransformer NameTransformer = new NameTransformer();
  /* ... */

  public static void AddTypeMapping(/* ... */) {
    NameTransformer.AddRule(
      /* ... */
    );
  }
}
```

Listing 4.6: Parts of ViewLocator.cs

When static fields are used like this, an obvious way to trigger a NullReferenceException[24] is by setting the `NameTransformer` field to `null` and then calling the `AddTypeMapping` method. As stated in the design guidelines, public APIs should never throw a NullReferenceException[4, p. 237]. In a single-threaded application this problem can be averted by inserting a precondition `Contract.Require(NameTransformer != null);` . However, as stated in the Threading Design Guidelines[26] static state should be thread-safe which in this example is not the case.

# Chapter 5

# Related Work

*Pex*[28] is a white-box testing tool based on *Dynamic Symbolic Execution* that automatically explores methods and generates test-cases. Pex works on CIL level and is therefore applicable to all languages that compile into CIL like C#, VB.NET and recently F#. Internally it is built on top of the *ExtendedReflection* library[21] that allows to add callbacks to CIL code through which a method can be observed and intercepted. By using these callbacks the symbolic execution engine was realized.

*Code Contracts* is an specification language[8] for implementing the design-by-contract program design approach[13]. Contracts express the requirements on the input values of a method and a series of guarantees that the method gives on the return values as well as a set of object invariants that must hold for the whole lifetime of each instance of that class. *Code Contracts* integrates with Pex[1] allowing the use of the contract information to reduce the number of runs with invalid input data and to empirically check if the guarantees that the method gives really hold.

As a final remark, it is worth mentioning that Clousot[9], an abstract interpretation tool for .NET and Code Contracts assumes that static initializer have already run prior to a method invocation.

# Chapter 6

# Conclusion

As the examples have shown, static fields and initializers can lead to convoluted and nearly impossible to find bugs. However in practice the problems introduced by static fields are minimal. An explanation for this can be found in the following discoveries: Firstly, static fields and methods that interact with static fields are not that common and represent only a small percentage of an average code base. Secondly, most of the projects in the evaluation used static fields as proposed by the official developing guidelines that prevent hard to find bugs by design.

The evaluation has shown that the vulnerability analysis introduced in Section 3.4 is the most cost-effective analysis. The extension for Pex improves its results when the modification of fields is allowed. However, bugs that are only based on static initializers and their execution order seem to be very rare and the addition from Section 3.6 does not improve the results.

As a part of this thesis a number of different analysis techniques for testing static field and initializers has been found, implemented and evaluated. Surprisingly, even coarse-grained analysis techniques give a good starting point for detecting and eliminating bugs related to static fields. The evaluation results can guide future efforts in refining and extending testing tools for detecting this class of bugs. A possible extension would be the implementation of a Visual Studio plug-in that makes the user aware of dangers when working with static fields by interactively displaying dependency races and vulnerability points.

# Appendix A

# Additional Source Code

## A.1 Code to determine the initialization behavior of the CLR.

This program was used to determine the type initialization behavior of the CLR. The results can be found in table 1.1.

```
using System;

namespace InitBehaviorTester
{
    class Foo
    {
        public static int Field = Initialized();

#if !BeforeFieldInit
        static Foo() { }
#endif

        private static int Initialized()
        {
            Console.WriteLine("In Foo::.cctor");
            return 42;
        }

        public static void Log()
        {
            Console.WriteLine("In Foo::Log()");
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("CLR Version {0}", Environment.Version);
            Console.WriteLine("Enter Program::Main");

#if Execute
            if (args != null)
```

```
#else
            if (args == null)
#endif
            {
#if Field
                Console.WriteLine(Foo.Field);
#else
                Foo.Log();
#endif
            }

            Console.WriteLine("Exit Program::Main");
        }
    }
}
```

Listing A.1: Code to determine the initialization behavior of the runtime.

# References


[1] Michael Barnett, Manuel Fähndrich, Peli De Halleux, Francesco Logozzo, and Nikolai Tillmann. Exploiting the synergy between automated-test-generation and programming-by-contract. In *ICSE Companion*, pages 401–402, 2009.

[2] Cristian Cadar, Vijay Ganesh, Peter M Pawlowski, David L Dill, and Dawson R Engler. EXE: Automatically generating inputs of death. *ACM Transactions on Information and System Security (TISSEC)*, 12(2):10, 2008.

[3] Raymond Chen. C# static constructors are called on demand, not at startup. http://blogs.msdn.com/b/oldnewthing/archive/2007/08/15/4392538.aspx. Accessed: 2013-07-21.

[4] Krzysztof Cwalina and Brad Abrams. *Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries (2nd Edition)*. Addison-Wesley, second edition, 2008.

[5] ECMA. *ECMA-335: Common Language Infrastructure (CLI)*. ECMA (European Association for Standardizing Information and Communication Systems), fourth edition, 2006.

[6] Rob Eisenberg. Caliburn Micro - WPF, Silverlight, WP7 and WinRT Metro made easy. - Home. http://caliburnmicro.codeplex.com/. Accessed: 2013-07-21.

[7] Jb Evain. Cecil. http://www.mono-project.com/Cecil. Accessed: 2013-07-21.

[8] Manuel Fähndrich, Michael Barnett, and Francesco Logozzo. Embedded contract languages. In *Proceedings of the 2010 ACM Symposium on Applied Computing*, pages 2103–2110. ACM, 2010.

[9] Manuel Fähndrich and Francesco Logozzo. Static contract checking with abstract interpretation. In *Formal Verification of Object-Oriented Software*, volume 6528 of *Lecture Notes in Computer Science*, pages 10–30. Springer, 2011.

[10] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison Wesley, 1995.

[11] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: Directed automated random testing. In *ACM Sigplan Notices*, volume 40 (6), pages 213–223. ACM, 2005.

[12] Digital Mars. Const and Immutable. http://dlang.org/const3.html. Accessed: 2013-07-21.

[13] Richard Mitchell, Jim McKim, and Bertrand Meyer. *Design by contract, by example*. Addison Wesley Longman Publishing Co., Inc., 2002.

[14] MSDN. Analyzing Application Quality by Using Code Analysis Tools. http://msdn.microsoft.com/en-us/library/dd264897.aspx. Accessed: 2013-07-21.

[15] MSDN. AppDomain Class. http://msdn.microsoft.com/en-us/library/system.appdomain.aspx. Accessed: 2013-07-21.

[16] MSDN. Array fields should not be read only. http://msdn.microsoft.com/en-us/library/ms182302(v=vs.80).aspx. Accessed: 2013-07-21.

[17] MSDN. Dependency Properties. http://msdn.microsoft.com/en-us/library/ms752914.aspx. Accessed: 2013-07-21.

[18] MSDN. Do not declare read only mutable reference types. http://msdn.microsoft.com/en-us/library/ms182302(v=vs.80).aspx. Accessed: 2013-07-21.

[19] MSDN. Do not declare visible instance fields. http://msdn.microsoft.com/en-us/library/ms182141(v=vs.80).aspx. Accessed: 2013-07-21.

[20] MSDN. DynamicMethod Class. http://msdn.microsoft.com/en-us/library/system.reflection.emit.dynamicmethod.aspx. Accessed: 2013-07-21.

[21] MSDN. ExtendedReflection - Dynamic Analysis Framework for .NET. http://research.microsoft.com/en-us/projects/extendedreflection/. Accessed: 2013-07-28.

[22] MSDN. Field Usage Guidelines. http://msdn.microsoft.com/en-us/library/ta31s3bc(v=vs.71).aspx. Accessed: 2013-07-21.

[23] MSDN. Implementing Singleton in C#. http://msdn.microsoft.com/en-us/library/ff650316.aspx. Accessed: 2013-07-21.

[24] MSDN. NullReferenceException Class. http://msdn.microsoft.com/en-us/library/system.nullreferenceexception.aspx. Accessed: 2013-07-21.

[25] MSDN. Static Field Naming Guidelines. http://msdn.microsoft.com/en-us/library/d53b55ey(v=vs.71).aspx. Accessed: 2013-07-21.

[26] MSDN. Threading Design Guidelines. http://msdn.microsoft.com/en-us/library/f857xew0(v=vs.71).aspx. Accessed: 2013-07-21.

[27] MSDN. Windows Presentation Foundation. http://msdn.microsoft.com/en-us/library/ms754130.aspx. Accessed: 2013-07-21.

[28] Nikolai Tillmann and Jonathan Halleux. Pex - white box test generation for .net. In *Tests and Proofs*, volume 4966 of *Lecture Notes in Computer Science*, pages 134–153. Springer, 2008.

[29] Dries Vanoverberghe, Nikolaj Bjørner, Jonathan Halleux, Wolfram Schulte, and Nikolai Tillmann. Using dynamic symbolic execution to improve deductive verification. In *Model Checking Software*, volume 5156 of *Lecture Notes in Computer Science*, pages 9–25. Springer, 2008.