Synthesizing Parameterized Unit Tests to Detect Object Invariant Violations

Maria Christakis, Peter Müller, and Valentin Wüstholz

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

Abstract. Automatic test case generation techniques rely on a description of the input data that the unit under test is intended to handle. For heap data structures, such a description is typically expressed as some form of object invariant. If a program may create structures that violate the invariant, the test data generated using the invariant systematically ignores possible inputs and, thus, potentially misses bugs. In this paper, we present a technique that detects violations of object invariants. We describe three scenarios in which traditional invariant checking may miss such violations. Based on templates that capture these scenarios, we synthesize parameterized unit tests that are likely to violate invariants, and use dynamic symbolic execution to generate inputs to the synthesized tests. We have implemented our technique as an extension to Pex and detected a significant number of invariant violations in real applications.

1 Introduction

Automatic test case generation techniques, such as random testing or symbolic execution, rely on a description of the input data that the unit under test (UUT) is intended to handle. Such a description acts as a filter for undesirable input data. It is usually expressed as code in the test driver or as a method precondition that specifies the valid arguments for the method under test. When the inputs are heap data structures, some test case generators use predicates that express which instances of a data structure are considered valid. In an object-oriented setting, these predicates are often called *class* or *object invariants*.

Invariants may be provided by the programmer in the form of contracts, such as in the random testing tool AutoTest [11] for Eiffel and in the dynamic symbolic execution tool Pex [19] for .NET, or by the tester, like in the Korat [1] tool for Java, which exhaustively enumerates data structures that satisfy a given predicate up to a bound. Invariants may also be inferred by tools like the Daikon invariant detector [4], which is used by the symbolic execution tool Symbolic Java PathFinder [15] for obtaining input constraints on a UUT.

Using object invariants to generate test data requires the invariants to accurately describe the data structures a program may create. When an invariant is too weak, i.e., admits more data structures than the program may create, the test case generator may *produce undesirable* inputs, which are however easily detected when inspecting failing tests. A more severe problem occurs when an invariant is too strong,

D. Giannakopoulou and G. Salaün (Eds.): SEFM 2014, LNCS 8702, pp. 65–80, 2014.

i.e., admits only a subset of the data structures the program might actually create. The test case generator may then *not produce desirable* inputs since they are filtered out due to the overly strong invariant. Consequently, the UUT is executed with a restricted set of inputs, which potentially fail to exercise certain execution paths and may miss bugs. Too strong invariants occur, for instance, when programmers specify invariants they intend to maintain but fail to do so due to a bug, when they fail to capture all intended program behaviors in the invariant, or when invariants are inferred from program runs that do not exercise all relevant paths. Therefore, it is essential that invariants are not only used to filter test inputs but are also *checked* as part of test oracles. However, checking object invariants is very difficult as shown by work on program verification [3,12]. In particular, it is generally not sufficient to check at the end of each method that the invariant of its receiver is maintained. This traditional approach [10], which is for instance implemented in Pex and AutoTest, may miss invariant violations when programs use common idioms such as direct field updates, inheritance, or aggregate structures (see Sect. 2).

To address this issue, we propose a technique for detecting previously missed invariant violations by synthesizing parameterized unit tests (PUTs) [20] that are likely to create *broken objects*, i.e., class instances that do not satisfy their invariants. The synthesis is based on templates that capture the situations in which traditional invariant checking is insufficient. We use dynamic symbolic execution (DSE) [7], also called concolic testing [16], to find inputs to the synthesized PUTs that actually violate an invariant.

Whenever our approach detects an invariant violation, the programmer has to inspect the situation to decide which of the following three cases applies: (1) The object invariant is stronger than intended. In this case, one should weaken the invariant. (2) The invariant expresses the intended properties, but the program does not maintain it. This case constitutes a bug that should be fixed. (3) The invariant expresses the intended properties and can, in principle, be violated by clients of the class, but the entire program does not exhibit such violations. For instance, the class might provide a setter that violates an invariant when called with a negative argument, but the program does not contain such a call. In such cases, one should nevertheless adapt the implementation of the class to make the invariant robust against violations for future program changes during maintenance and for other clients of the class during code reuse.

The contributions of this paper are as follows:

- It identifies an important limitation of current test case generation approaches in the treatment of object invariants. In particular, existing approaches that use invariants as filters on input data do not sufficiently check them, if at all.
- It presents a technique that detects invariant violations by synthesizing PUTs based on templates and exploring them via DSE.
- It demonstrates the effectiveness of this technique by implementing it as an extension to Pex and using it on a suite of open source C# applications.

Outline. Sect. 2 illustrates the situations in which the traditional checks for object invariants are insufficient. Sect. 3 gives an overview of our approach.

Sect. 4 explains how we select the operations to be applied in a synthesized test, and Sect. 5 describes the templates used for the synthesis. We discuss our implementation in Sect. 6 and present the experimental evaluation in Sect. 7. We review related work in Sect. 8 and conclude in Sect. 9.

2 Violating Object Invariants

We present three scenarios in which the traditional approach of checking at the end of each method whether it maintains the invariant of its receiver is insufficient. These scenarios have been identified by work on formal verification and together with a fourth scenario—callbacks, which are not relevant here as explained in Sect. 8—have been shown to cover *all* cases in which traditional invariant checking does not suffice [3]. We assume that invariants are specified explicitly in the code as contracts. However, our technique applies equally to predicates that are provided as separate input to the test case generator or invariants that have been inferred from program runs.

```
1  public class Person {
2    Account account;
3    public int salary;
4
5    inv 0 < account.balance + salary;
6
7    public void Spend1(int amount) {
8      account.Withdraw(amount);
9    }
10
11   public void Spend2(int amount) {
12     account.balance -= amount;
13   }
14 }
15
16 public class Account {
17   public int balance;
18
19   public void Withdraw(int amount) {
20     balance -= amount;
21   }
22 }
23
24 public class SavingsAccount : Account {
25   inv 0 <= balance;
26 }
```

Fig. 1. A C# example on invariant violations. We declare invariants using a special `inv` keyword and assume that fields hold non-null values.

We illustrate the scenarios using the C# example in Fig. 1. For simplicity, we assume that all fields hold non-null values. A `Person` holds an `Account` and has a `salary`. An `Account` has a `balance`. `Person`'s invariant (line 5) requires that the sum of the `account`'s `balance` and the person's `salary` is positive. A `SavingsAccount` is a special `Account` whose `balance` is non-negative (line 25). In each of the following scenarios, we consider an object p of class `Person` that holds an `Account` a.

Direct field updates: In most object-oriented languages, such as C++, C#, and Java, a method may update not only fields of its receiver but of any object as long as the fields are accessible. For instance, method `Spend2` (which is an alternative implementation of `Spend1`) subtracts `amount` from the `account`'s `balance` through a direct field update instead of calling method `Withdraw`. Such direct field updates are common among objects of the same class (say, nodes of a list) or of closely connected classes (say, a collection and its iterator). If a is a `SavingsAccount`, method `Spend2` might violate a's invariant by setting `balance` to a negative value. A check of the receiver's invariant at the end of method `Spend2` (here, `Person` object p) does not reveal this violation. In order to detect violations through direct field updates, one would have to check the invariants of all objects whose fields are assigned to directly. However, these objects are not statically known (for instance, when the direct field update occurs within a loop), which makes it difficult to impose such checks.

Subclassing: Subclasses may restrict the possible values of a field inherited from a superclass, i.e., they strengthen the invariant for this field, as shown by class `SavingsAccount`. Methods declared in the superclass are typically designed for and tested with instances of the superclass as their receiver, and thus the tests check only the weaker superclass invariant. When such methods are inherited by the subclass and called on subclass instances, they may violate the stronger subclass invariant. In our example, in case a is a `SavingsAccount`, calling the inherited method `Withdraw` on a might set `balance` to a negative value and violate the invariant of the subclass. To detect such violations, one would have to re-test every inherited method whenever a new subclass is declared. Moreover, subclassing makes the invariant checks for direct field updates even more difficult because one would have to consider all subclasses for the objects whose fields are updated. For instance, when introducing `SavingsAccount`, testing `Withdraw` on a subclass instance is not sufficient; one has to also re-test method `Spend2` to detect the invariant violation described in the previous scenario.

Multi-object invariants: Most data structures are implemented as aggregations of several objects. For such aggregate structures, it is common that an object invariant constrains and relates the states of several objects. In our example, the invariant of class `Person` relates the state of a `Person` object to the state of its `Account`. For such *multi-object invariants*, modifying the state of one object might break the invariant of another. For instance, when `Account` a executes method `Withdraw`, it might reduce the `balance` by an amount such that it violates the invariant of `Person` p. To detect such violations, one would have to check the invariants of all objects that potentially reference a, e.g., the invariants of `Person` objects sharing the account, of collections storing the account, etc. These objects are not statically known and cannot even be approximated without inspecting the entire program, which defeats the purpose of unit testing.

These scenarios demonstrate that the traditional way of checking object invariants may miss violations in common situations and that the checks cannot be strengthened in any practical way. Therefore, simply including all necessary invariant checks in the test oracle is not feasible; other techniques are required to detect invariant violations.

3 Approach

For a given UUT we synthesize client code in the form of PUTs to detect invariant violations. The synthesis is based on a set of four fixed templates that capture the three scenarios of Sect. 2. Each template consists of a sequence of *candidate operations*, i.e., updates of public fields and calls to public methods. These operations are applied to the object whose invariant is under test or, in the case of aggregate structures, its sub-objects. (Note that since our approach synthesizes client code, it uses public candidate operations. To also synthesize code of possible subclasses, one would analogously include protected fields and methods.) The candidate operations are selected conservatively from the UUT based on whether they potentially lead to a violation of the object invariant. Depending on the template, additional restrictions are imposed on the candidate operations, e.g., that they are inherited from a superclass. By instantiating the templates with candidate operations, the synthesized PUTs become snippets of client code that potentially violate the object invariant.

Alg. 1. Synthesis of parameterized unit tests.

```
function Synthesize(class, inv, len)
  candOps ← ComputeCandOps(class, inv)
  puts ← GenFieldCombs(candOps, len)
  puts ← puts + GenMultiCombs(candOps, len, inv)
  puts ← puts + GenSubCombs(candOps, len)
  puts ← puts + GenAllCombs(candOps, len)
  return AddSpecs(puts)
```

Alg. 1 shows the general strategy for the PUT synthesis. Function SYNTHESIZE takes the class of the object to which candidate operations should be applied (*class*), the object invariant under test (*inv*), and the desired length of the PUTs to be synthesized (*len*). The last argument prevents a combinatorial explosion by bounding the number of operations in each synthesized PUT. SYNTHESIZE returns a list of PUTs. Each PUT consists of a sequence of candidate operations and additional specifications, such as invariant checks, which are inserted by ADDSPECS and explained in Sect. 4. The algorithm first determines the set of candidate operations (*candOps*) of *class* that could potentially violate the object invariant *inv*. It then synthesizes the PUTs for each of the three scenarios using the corresponding templates. We discuss the selection of candidate operations as well as these templates in detail in the next sections.

We complement the templates, which cover specific scenarios for violating invariants, by an *exhaustive enumeration* of combinations (of length *len*) of candidate operations. In the algorithm, these combinations are computed by function GENALLCOMBS. As we will see in Sect. 5, this exhaustive exploration is useful for multi-object invariants where the actual violation may happen by calling a method on a sub-object of an aggregate structure.

For a given UUT, we apply function SYNTHESIZE for each class in the unit and the invariant it declares or inherits. We perform this application repeatedly for increasing values of *len*. All operations in the resulting PUTs have arguments that are either parameters of the enclosing PUT or results of preceding operations; all such combinations are tried exhaustively, which, in particular, includes aliasing among the arguments. This makes the PUTs sufficiently general to capture the scenarios of the previous section, i.e., to detect invariant violations caused by these scenarios. We employ dynamic symbolic execution (DSE) [7,16] to supply the arguments to the PUTs.

4 Candidate Operations

To synthesize client code that violates object invariants, we select candidate operations from the public fields and methods of the UUT. To reduce the number of synthesized PUTs, we restrict the operations to those that might violate a given invariant. Such operations are determined by intersecting the read effect of the invariant with the write effect of the operation. The *read effect* of an invariant is the set of fields read in the invariant. If the invariant contains calls to side-effect free methods, the fields (transitively) read by these methods are also in its read effect. The *write effect* of a method is the set of fields updated during an execution of the method including updates performed through method calls. The write effect of a field update is the field itself. Note that the effects are sets of (fully-qualified) field names, not concrete instance fields of objects. This allows us to use a simple, whole-program static analysis that conservatively approximates read and write effects without requiring alias information (see Sect. 6).

```
public class C {
  public int x;
  int y;

  inv x == 42;

  public void SetX() { x = y; }

  public void SetY(int v) { y = v; }
}
```

To illustrate these concepts, consider the example on the right. The read effect of the invariant is {`C.x`} indicating that only the value of `C`'s field `x` determines whether the invariant holds. The write effect of an update to the public field `x` and of method `SetX` is {`C.x`}, while method `SetY` has write effect {`C.y`}. By intersecting these read and write effects, we determine that field updates of `x` and calls to `SetX` must be included in the candidate operations.

```
void PUT_0(C o, int v) {
  assume o != null && o.Invariant();
  o.x = v;
  assert o.Invariant();
}
void PUT_1(C o) {
  assume o != null && o.Invariant();
  o.SetX();
  assert o.Invariant();
}
```

With these operations, the exhaustive enumeration of sequences of length 1 (function GenAllCombs in Alg. 1) produces the two PUTs on the right (`PUT_0`, `PUT_1`). As shown here, each synthesized test expects as argument a non-null object `o` whose invariant holds, applies the synthesized sequence of candidate operations to `o`, and then asserts that `o`'s invariant still holds. We encode the invariant via a side-effect free boolean method `Invariant` and use `assume` statements to introduce constraints for the symbolic execution. The `assume` and `assert` statements are inserted into the PUTs by function AddSpecs of Alg. 1. The input object `o` is constructed using operations from the UUT, for instance, a suitable constructor. As explained above, the arguments of candidate operations (like the value `v` for the assignment to `o.x` in the first test) are either parameters of the PUT and supplied later via DSE, or results of preceding operations.

Whether a method call violates an invariant may not only depend on its arguments but also on the state in which it is called. For instance, a call to `SetX` violates the invariant only if `y` has a value different from 42. Therefore, tests that apply more than one candidate operation must take into account the possible interactions between operations. Consequently, for each candidate operation op_d that might directly violate a given object invariant, we compute its read effect and include in the set of candidate operations each operation op_i whose write effect overlaps with this read effect and might, therefore, indirectly violate the invariant. To prune the search space, we record that op_i should be executed before op_d. This process iterates until a fixed point is reached.

In our example, method `SetX` has read effect {`C.y`}. As a result, method `SetY` is used in the PUTs as a candidate operation that should be called before `SetX`. Therefore, the exhaustive enumeration of sequences of length 2 includes the PUT above (`PUT_2`). Note that, by assuming `o`'s invariant before the call to `SetX`, we suppress execution paths that have already been tested in a shorter PUT, i.e., paths that violate `o`'s invariant before reaching the final operation.

```
void PUT_2(C o, int v) {
  assume o != null && o.Invariant();
  o.SetY(v);
  assume o.Invariant();
  o.SetX();
  assert o.Invariant();
}
```

5 Synthesis Templates

We now present the templates that capture the three scenarios of Sect. 2. Besides other arguments, each template expects an object `r` to which candidate operations are applied, and an object `o` whose invariant is under test. When the templates are used to synthesize an entire test, these two objects coincide and we include only one of them in the PUT. The templates are also used to synthesize portions of larger PUTs, and then `r` and `o` may refer to different objects.

5.1 Direct Field Updates

The direct-field-update template tries to violate the invariant of an object `o` by assigning to a field of `r` (or to an element of `r` when `r` is an array). The template has the form shown on the right. It applies a sequence of operations (`Op0` to `OpM`) to `r` to create a state in which the subsequent update of `r.f` may violate `o`'s invariant. For instance, if the invariant relates `f` to private fields of the same object, these operations may be method calls that update these private fields. The operations `Op0` to `OpM` are selected from the set of candidate operations and may include a method call or field update more than once. Their arguments as well as the right-hand side `v` of the last field update are either parameters of the template (`a0` to `aN`) or results of preceding operations; all such combinations are tried exhaustively.

```
void DFU(r, o, a0..aN) {
  assume r != null;
  assume o != null && o.Invariant();
  Op0(r, ...); ... OpM(r, ...);
  assume o.Invariant();
  r.f = v;
  assert o.Invariant();
}
```

The synthesis of PUTs from this template is performed by function GenFieldCombs in Alg. 2, which is invoked from Alg. 1. Line 3 generates all possible sequences of length $len - 1$ from the set of candidate operations. Line 4 selects the set $fieldOps$ of all field updates from the set of candidate operations, *candOps*. Lines 5–7 append each field update $fieldOp$ to each of the sequences of operations computed earlier.

Alg. 2. Synthesis of parameterized unit tests from the direct-field-update template.

1 **function** GenFieldCombs(*candOps*, *len*)
2   *puts* ← []
3   *combs* ← GenAllCombs(*candOps*, $len-1$)
4   $fieldOps$ ← FieldOps(*candOps*)
5   **foreach** *comb* **in** *combs*
6   **foreach** $fieldOp$ **in** $fieldOps$
7   *puts* ← *puts* + [*comb* + [$fieldOp$]]
8   **return** *puts*

```
void PUT_DFU(Person o, int a, int s) {
  assume o != null && o.Invariant();
  o.Spend1(a);
  assume o.Invariant();
  o.salary = s;
  assert o.Invariant();
}
```

Consider an invocation of the synthesis with this template, where the object to which operations are applied and the object whose invariant is being tested are the same instance of class `Person` from Fig. 1. The synthesized PUTs of length 2 include the test above (`PUT_DFU`). Symbolically executing this PUT produces input data that causes the assertion of the invariant to fail; for instance, a `Person` object with `salary` 100 and whose `Account` has `balance` 100 for `o`, the value 150 for `a`, and any value less than or equal to 50 for `s`.

5.2 Subclassing

```
void S(r, o, a0..aN) {
  assume r != null;
  assume o != null && o.Invariant();
  Op0(r, ...); ... OpM(r, ...);
  assume o.Invariant();
  Op_super(r, ...);
  assert o.Invariant();
}
```

The template for the subclassing scenario (on the right) aims at breaking the invariant of an object by invoking inherited operations. It exhaustively applies a number of operations to an object of the subclass, including any operations inherited from a superclass, and requires that the last operation is an inherited one (i.e., an update of an inherited field or a call to an inherited method) to reflect the subclassing scenario described in Sect. 2. Like in the template for direct field updates, the first `M + 1` operations (`Op0` to `OpM`) construct a state in which the final inherited operation may violate `o`'s invariant as this operation was designed to maintain the weaker invariant of a superclass. This template is useful only when a subclass strengthens the invariant of a superclass with respect to any inherited fields. We identify such subclasses using a simple syntactic check: if the read effect of the invariant declared in the subclass includes inherited fields, we conservatively assume that the invariant is strengthened with respect to those.

```
void PUT_S(SavingsAccount o, int a) {
  assume o != null && o.Invariant();
  o.Withdraw(a);
  assert o.Invariant();
}
```

The synthesis of PUTs based on this template is performed by function GenSubCombs, which is invoked from Alg. 1. GenSubCombs is analogous to GenFieldCombs (Alg. 2) except that on line 4 it selects the candidate operations that are inherited from a superclass. Consider an invocation of the synthesis with this template, where the object to which operations are applied and the object whose invariant is being tested are the same instance of class `SavingsAccount` from Fig. 1. This class strengthens the invariant of its superclass `Account` for the inherited field `balance`. The synthesized PUTs of length 1 include the test above (`PUT_S`). The symbolic execution of this PUT produces input data that causes the assertion of the invariant to fail; for instance, a `SavingsAccount` object with a `balance` of 0 for `o` and any positive value for `a`.

5.3 Multi-object Invariants

Multi-object invariants describe properties of aggregate structures. The invariant of such a structure may be violated by modifying its sub-objects. For instance, one might be able to violate a `Person`'s invariant by reducing the balance of its account. Such violations are possible when sub-objects of the aggregate structure are not properly encapsulated [12] such that clients are able to obtain references to them: when a client obtains a direct reference to the `Account` sub-object, it can by-pass the `Person` object and modify the account in ways that violate the `Person`'s invariant. To reflect this observation, we use two templates that allow clients to obtain references to sub-objects of aggregate structures. One template uses *leaking*, i.e., it passes a sub-object from the aggregate structure to its client. The other one uses *capturing*, i.e., it passes an object from the client to the aggregate structure and stores it there as a sub-object. Leaking and capturing are the only ways in which clients may obtain a reference to a sub-object of an aggregate structure.

Leaking. A method is said to *leak* an object `l` if the following three conditions hold: (1) the method takes as an argument (or receiver) an object `o` that (directly or transitively) references `l`, (2) the method returns the reference to `l` or assigns it to shared state, and (3) a field of `l` is dereferenced in `o`'s invariant. We use a static analysis to approximate the operations that might leak a sub-object (see Sect. 6). These operations include reading public fields with reference types.

For example, assume that class `Person` from Fig. 1 provides a public getter `GetAccount` for field `account`. This method leaks the `account` sub-object of its receiver since (1) its receiver directly references the account, (2) it returns the account, and (3) `account` is dereferenced in the invariant of `Person`. Consequently, this getter enables clients to obtain a reference to the `account` sub-object and violate `Person`'s invariant, for instance by invoking `Withdraw` on the account.

```
void L(r, o, a0..aN) {
  assume r != null;
  assume o != null && o.Invariant();
  Op0(r, ...); ... OpM(r, ...);
  var l = Op_leaking(r, ...);
  ... // operations on leaked 'l'
  assert o.Invariant();
}
```

In the template for leaking (on the right), we first apply a number of operations to create a state in which a sub-object `l` may be leaked via the operation `Op_leaking`. Once the object has been leaked, we try to violate `o`'s invariant by applying operations to the leaked object `l` (indicated by the ellipsis with the corresponding comment in the above template). To obtain a suitable sequence of operations on `l`, we apply function Synthesize (Alg. 1) recursively with the class of the leaked object `l` and the invariant of `o`. This recursive call selects candidate operations on `l` that may break `o`'s invariant, for instance by updating a public field of `l` or via complex combinations of scenarios such as repeated leaking. Note that this template attempts to violate `o`'s invariant; whether `l`'s invariant holds is an orthogonal issue.

```
void PUT_L(Person o, int a) {
  assume o != null && o.Invariant();
  var l = o.GetAccount();
  // exhaustive enumeration
  assume l != null && o.Invariant();
  l.Withdraw(a);
  assert o.Invariant();
}
```

Based on this template, we obtain the PUT on the right (`PUT_L`) for objects of class `Person` from Fig. 1. In this test, method `GetAccount` leaks the `Person`'s `account` object. The recursive application of function Synthesize determines method `Withdraw` as a candidate operation because its write effect includes `balance`, which is also in the read effect of `Person`'s invariant. `Withdraw` is selected by the exhaustive enumeration (function GenAllCombs of Alg. 1) and would not be selected by any of the other templates. Symbolically executing this PUT produces input data that causes the assertion of the invariant to fail; for instance, a `Person` object with `salary` 100 and whose `Account` has `balance` 100 for `o`, and a value of at least 200 for `a`.

Capturing. A method is said to *capture* an object `c` if: (1) the method takes as arguments two objects `o` and `c` (`o` or `c` could also be the receiver), (2) the method stores a reference to `c` in a location reachable from `o`, and (3) the field in which `c` is stored is dereferenced in `o`'s invariant. Updating a field `f` that has a reference type is also considered capturing if `f` is dereferenced in `o`'s invariant.

The template for capturing (on the right) is analogous to leaking. In particular, it also uses a recursive application of function Synthesize to determine the operations to be applied to the captured object. In the common case that the capturing operation is a constructor of object `o`, the template is adjusted as shown on the right (`Cctor`). This adjustment ensures that `o` is actually created with a constructor that captures `c` instead of a constructor selected by the symbolic execution. Note that before the capturing operation we could also allow a number of operations on `c` with the goal of bringing it to a state such that, for instance, the precondition of `Op_capturing` is satisfied or the capturing execution path is taken. We omit such operations to simplify the presentation.

```
void C(r, o, c, a0..aN) {
  assume r != null;
  assume o != null && o.Invariant();
  Op0(r, ...); ... OpM(r, ...);
  Op_capturing(r, c, ...);
  ... // operations on captured 'c'
  assert o.Invariant();
}
void Cctor(c, a0..aN) {
  assume c != null;
  Op0(c, ...); ... OpM(c, ...);
  var o = new ctor(c, ...);
  ... // operations on captured 'c'
  assert o.Invariant();
}
```

Synthesis. The synthesis of PUTs based on these templates is performed by the GenMultiCombs function in Alg. 3. On line 3, a new set of candidate operations, *multiOps*, is created by selecting from *candOps* the operations that leak or capture objects according to the above criteria. Since the synthesis for these templates includes a recursive application of Synthesize, we must split the overall length of the PUT between the operations occurring before the leaking or capturing operation and the operations on the leaked or captured object occurring after. To explore all possible splits, we generate all combinations of candidate operations of length up to $len - 2$ to be applied before the

Alg. 3. Synthesis of parameterized unit tests from the leaking and capturing templates.

```
1  function GenMultiCombs(candOps, len, inv)
2    puts ← []
3    multiOps ← MultiOps(candOps)
4    for i = 0 to len − 2 do
5      prefixes ← GenAllCombs(candOps, i)
6      foreach multiOp in multiOps
7        class ← GetClass(multiOp)
8        suffixes ← Synthesize(class, inv, len − 1 − i)
9        foreach prefix in prefixes
10         foreach suffix in suffixes
11           put ← prefix + [multiOp] + suffix
12           puts ← puts + [put]
13   return puts
```

leaking or capturing operation (lines 4–5). These operations create a state in which the next operation can leak or capture an object. After invoking any such operation, we recursively apply function SYNTHESIZE of Alg. 1 by taking into account the class of the leaked or captured object ($class$) and the original object invariant under test, inv (lines 6–8). Therefore, $suffixes$ is a list of sequences of operations to be applied to the leaked or captured object. On lines 9–12, we combine the synthesized sub-sequences of lengths i, 1, and $len - 1 - i$.

6 Implementation

We have implemented our technique as an extension to Pex. Our implementation builds a static call-graph for the entire UUT that includes information about dynamically-bound calls. The call-graph is used to compute the read and write effects of all methods in the UUT with a conservative, inter-procedural, control-flow insensitive static analysis on the .NET bytecode. The effects determine the candidate operations that might, directly or indirectly, lead to an invariant violation (see Sect. 4). Our effect analysis is extended to also approximate the sets of leaking and capturing operations using their read and write effects, respectively, in addition to the read effect of the invariant under test. For simplicity, we only consider leaking operations that return the leaked object or store it in a public field of their receiver.

To detect invariant violations more efficiently, we carefully chose the order in which the synthesis (Alg. 1) applies the templates of Sect. 5 and exhaustive enumeration. The templates for direct field updates and multi-object invariants have proven to most effectively detect invariant violations and are therefore explored first. The exhaustive enumeration comes last as it produces the largest number of PUTs and requires the most effort in the symbolic execution.

7 Experimental Evaluation

We have evaluated the effectiveness of our technique using ten C# applications, which were selected from applications on Bitbucket, CodePlex, and GitHub containing invariants specified with Code Contracts [5]. This section focuses on our experiments with the nine applications for which invariant violations were detected.

Tab. 1 summarizes the results of our experiments. The third and fourth columns show the total number of classes and the number of classes with invariants for each application, respectively. We have tested the robustness of all invariants in these applications. Note that the total number of classes refers only to the classes that were included in the evaluation and not to all classes of each application. We have left out only classes that were defined in dynamic-link libraries (DLLs) containing no object invariants. The two rightmost columns of Tab. 1 show the unique and total numbers of invariant violations detected with our technique. The unique number of violations refers to the number of invariants that were violated at least once. The total number of violations refers to

Table 1. Summary of results. The third and fourth columns show the total number of classes and the number of classes with invariants for each application. The two rightmost columns show the unique and total numbers of invariant violations detected with our technique.

Application	Description	Classes	Classes w/ invariants	Invariant violations unique	Invariant violations total
Boogie	Intermediate verification engine[1]	355	144	21	64
ClueBuddy	GUI application for board game[2]	44	4	1	2
Dafny	Programming language/verifier[3]	310	113	15	53
Draugen	Web application for fishermen[4]	36	5	3	3
GoalsTracker	Various web applications[5]	63	5	1	1
Griffin	.NET and jQuery libraries[6]	31	3	1	1
LoveStudio	IDE for the LÖVE framework[7]	66	7	2	2
Encore	'World of Warcraft' emulator[8]	186	30	1	4
YAML	YAML library[9]	76	6	1	2

the number of distinct PUTs that led to invariant violations and may include violations of the same object invariant multiple times.

When running Pex with our technique, we imposed an upper bound of 3 on the number of operations per PUT, and an upper bound of 300 on the number of synthesized PUTs per object invariant. It turned out that all unique invariant violations were detected already with 2 operations per PUT; increasing the bound to 4 for some projects did not uncover previously undetected invariant violations. On average, 14.7 PUTs were synthesized per second. We then applied Pex to generate input data for the synthesized PUTs forcing Pex to use only public operations of the UUT (to guarantee that all inputs are constructible in practice). We counted the number of unique invariant violations and of distinct PUTs that led to invariant violations. We imposed a timeout of 3 minutes for the DSE in Pex to generate inputs for and run the synthesized PUTs. Here, we report the number of invariant violations that were detected within this time limit. Within this time limit, the first invariant violation was detected within 4–47 seconds (12.8 seconds on average) for all object invariants in all applications.

[1] `http://boogie.codeplex.com`, rev: `f2ffe18efee7`
[2] `https://github.com/AArnott/ClueBuddy`, rev: `c1b64ae97c01fec249b2212018f589c2d8119b59`
[3] `http://dafny.codeplex.com`, rev: `f2ffe18efee7`
[4] `https://github.com/eriksen/Draugen`, rev: `dfc84bd4dcf232d3cfa6550d737e8382ce7641cb`
[5] `https://code.google.com/p/goalstracker`, rev: `556`
[6] `https://github.com/jgauffin/griffin`, rev: `54ab75d200b516b2a8bd0a1b7cfe1b66f45da6ea`
[7] `https://bitbucket.org/kevinclancy/love-studio`, rev: `7da77fa`
[8] `https://github.com/Trinity-Encore/Encore`, rev: `0538bd611dc1bc81da15c4b10a65ac9d608dafc2`
[9] `http://yaml.codeplex.com`, rev: `96133`

The total violations found by our technique may be classified into the following categories based on the template that was instantiated: 60 due to direct field updates, 41 due to leaking, and 25 due to capturing. The remaining 6 violations were detected by the exhaustive enumeration. Out of these 6 invariant violations, 5 are also detected by the version of Pex without our technique, i.e., with the traditional approach of checking the invariant of the receiver at the end of a method. The last violation requires a sequence of two method calls and was detected only by our technique. This is because Pex could not generate appropriate input data to the second method such that the invariant check at the end of the method failed. In this case, the exhaustive enumeration served as a technique for generating more complex input data. The object invariants that were violated at least once can be classified into the following categories: 27 invariants were violated at least once due to direct field updates, 24 due to leaking, 17 due to capturing, and 5 due to the exhaustive enumeration. Note that in these applications we found no subclasses that strengthen the invariant of their superclass with respect to any inherited fields. This is why no invariant violations were detected with the subclassing template.

An example of an invariant violation detected by our technique in LoveStudio is shown on the right. A `StackPanel` object has a `LuaStackFrame` array, and its invariant holds if all array elements are non-null. In the PUT, method `SetFrames` captures `a0` depending on the value of `a1`. The last operation of the test assigns a `LuaStackFrame` object to the array at a valid index `a2`. In case `a3` is `null`, `o`'s invariant is violated.

```
void PUT(StackPanel o,
         LuaStackFrame[] a0, bool a1,
         int a2, LuaStackFrame a3) {
  assume o != null && o.Invariant();
  o.SetFrames(a0, a1);
  assume a0 != null && o.Invariant();
  assume 0 <= a2 && a2 < a0.Length;
  a0[a2] = a3;
  assert o.Invariant();
}
```

We have manually inspected all detected invariant violations. Violations detected with the direct-field-update template reveal design flaws and can be fixed by making fields private and providing setters that maintain the invariants. Violations due to leaking or capturing could be fixed either by cloning the leaked or captured objects, or by using immutable types in the interfaces of the classes whose invariants are under test. The largest number of invariant violations found with the leaking and capturing templates was detected in the Boogie and Dafny applications, which declare several multi-object invariants in their code.

The detected invariant violations indicate overly strong invariants in the sense that they may be violated by possible clients of a UUT. These clients are not necessarily present in a given application and, thus, the violations do not necessarily reveal bugs. This behavior is to be expected for unit testing, where each unit is tested independently of the rest of the application. Nevertheless, the detected violations do indicate robustness issues that might lead to bugs during maintenance or when classes are reused as libraries. We discussed the detected invariant violations in Boogie and Dafny with the lead developer, Rustan Leino. All of them seem to indicate robustness issues, which will be addressed by either weakening the invariants or changing the design of the code.

8 Related Work

Our approach to testing object invariants is inspired by static verification techniques. Poetzsch-Heffter [14] pointed out that the traditional way of checking the invariant of the receiver at the end of each method is insufficient. The checks he proposed are sufficient for sound verification, but not suitable as unit test oracles since they make heavy use of universal quantification. Some modular verification techniques for object invariants [9,12] handle the challenges mentioned in Sect. 2, but require annotation overhead that does not seem acceptable for testing.

We distilled our templates for test synthesis from a formal framework for verification techniques for object invariants [3]. This framework identifies an additional scenario (not presented in Sect. 2) involving a *call-back*. In this scenario, a method `L` violates the invariant of its receiver `r` and then calls another method `M`. If `M` performs a call-back `r.N`, method `N` finds the invariant of `r` broken, which may lead to an error in the body of `N` or a violation of the invariant check at the end of `N`. We omitted this scenario because Pex already detects such problems while testing `L`. It attempts to generate inputs for `L` that violate the assertions in method `L` and all methods it calls, in particular, the check of `r`'s invariant at the end of `N`.

There are several test case generators for object-oriented programs that rely on invariants, but miss the violations presented here. AutoTest [11], a random testing tool for Eiffel, follows the traditional approach of checking the invariant of the receiver at the end of each method. Pex [19] follows the same approach, but asserts the invariant of the receiver only at the end of public methods. Korat [1] and Symbolic Java PathFinder [15] do not check object invariants of the UUT at all; they use invariants only to filter test inputs. All such tools may miss bugs when object invariants are violated and would thus benefit from our technique.

Work on synthesizing method call sequences to generate complex input data is complementary to ours. In fact, such approaches could be applied in place of the object construction mechanism in Pex to generate input objects for our PUTs. In certain cases, this might reduce the length of the synthesized tests since fewer candidate operations may be required to generate the same objects. These approaches include a combination of bounded exhaustive search and symbolic execution [22], feedback-directed random testing [13], a combination of feedback-directed random testing with concolic testing [6,2], evolutionary testing [21], an integration of evolutionary and concolic testing [8], and source code mining [17]. Moreover, Palus [23] combines dynamic inference, static analysis, and guided random test generation to automatically create legal and behaviorally-diverse method call sequences. In contrast to existing work, our technique synthesizes code that specifically targets violations of object invariants. This allows for a significantly smaller search space restricted to three known scenarios in which invariant violations may occur.

The work on method call synthesis most closely related to ours is Seeker [18], an extension to Pex that combines static and dynamic analyses to construct input objects for a UUT. More specifically, Seeker attempts to cover branches that are missed by Pex. Even though this approach does not rely on object invariants, the negation of an invariant could be regarded as a branch to be covered. However, some of the scenarios of Sect. 2 are not captured by Seeker's static analysis. For example, a multi-object invariant violation involves leaking or capturing parts of an object's representation, and might not necessarily involve a sequence of missed branches. The same holds for subclassing.

9 Conclusion

We have presented a technique for detecting object invariant violations by synthesizing PUTs. Given one or more classes under test, our technique uses a set of templates to synthesize snippets of client code. We then symbolically execute the synthesized code to generate inputs that might lead to invariant violations. As a result, our technique may reveal critical defects in the UUT, which go undetected by existing testing tools. We have demonstrated the effectiveness of our implementation on a number of C# applications with object invariants.

Acknowledgments. We thank Nikolai Tillman and Jonathan "Peli" de Halleux for sharing the Pex source code with us. We also thank Timon Gehr for implementing and evaluating parts of this technique for his Bachelor's thesis, and the reviewers for their constructive feedback.

References

1. Boyapati, C., Khurshid, S., Marinov, D.: Korat: Automated testing based on Java predicates. In: ISSTA, pp. 123–133. ACM (2002)
2. Dimjašević, M., Rakamarić, Z.: JPF-Doop: Combining concolic and random testing for Java. In: Java Pathfinder Workshop. Extended abstract (2013)
3. Drossopoulou, S., Francalanza, A., Müller, P., Summers, A.J.: A unified framework for verification techniques for object invariants. In: Vitek, J. (ed.) ECOOP 2008. LNCS, vol. 5142, pp. 412–437. Springer, Heidelberg (2008)
4. Ernst, M.D., Perkins, J.H., Guo, P.J., McCamant, S., Pacheco, C., Tschantz, M.S., Xiao, C.: The Daikon system for dynamic detection of likely invariants. Sci. Comput. Program. 69, 35–45 (2007)
5. Fähndrich, M., Barnett, M., Logozzo, F.: Embedded contract languages. In: SAC, pp. 2103–2110. ACM (2010)
6. Garg, P., Ivančić, F., Balakrishnan, G., Maeda, N., Gupta, A.: Feedback-directed unit test generation for C/C++ using concolic execution. In: ICSE, pp. 132–141. ACM (2013)
7. Godefroid, P., Klarlund, N., Sen, K.: DART: Directed automated random testing. In: PLDI, pp. 213–223. ACM (2005)
8. Inkumsah, K., Xie, T.: Evacon: A framework for integrating evolutionary and concolic testing for object-oriented programs. In: ASE, pp. 425–428. ACM (2007)
9. Leino, K.R.M., Müller, P.: Object invariants in dynamic contexts. In: Odersky, M. (ed.) ECOOP 2004. LNCS, vol. 3086, pp. 491–515. Springer, Heidelberg (2004)
10. Meyer, B.: Object-Oriented Software Construction, 2nd edn. Prentice-Hall (1997)
11. Meyer, B., Fiva, A., Ciupa, I., Leitner, A., Wei, Y., Stapf, E.: Programs that test themselves. IEEE Computer 42(9), 46–55 (2009)

12. Müller, P., Poetzsch-Heffter, A., Leavens, G.T.: Modular invariants for layered object structures. Sci. Comput. Program. 62, 253–286 (2006)
13. Pacheco, C., Lahiri, S.K., Ernst, M.D., Ball, T.: Feedback-directed random test generation. In: ICSE, pp. 75–84. IEEE Computer Society (2007)
14. Poetzsch-Heffter, A.: Specification and verification of object-oriented programs. Habilitation thesis. Technical University of Munich (1997)
15. 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)
16. Sen, K., Marinov, D., Agha, G.: CUTE: A concolic unit testing engine for C. In: ESEC, pp. 263–272. ACM (2005)
17. Thummalapenta, S., Xie, T., Tillmann, N., de Halleux, J., Schulte, W.: MSeqGen: Object-oriented unit-test generation via mining source code. In: ESEC/SIGSOFT FSE, pp. 193–202. ACM (2009)
18. Thummalapenta, S., Xie, T., Tillmann, N., de Halleux, J., Su, Z.: Synthesizing method sequences for high-coverage testing. In: OOPSLA, pp. 189–206. ACM (2011)
19. 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)
20. Tillmann, N., Schulte, W.: Parameterized unit tests. In: ESEC/SIGSOFT FSE, pp. 119–128. ACM (2005)
21. Tonella, P.: Evolutionary testing of classes. In: ISSTA, pp. 119–128. ACM (2004)
22. Xie, T., Marinov, D., Schulte, W., Notkin, D.: Symstra: A framework for generating object-oriented unit tests using symbolic execution. In: Halbwachs, N., Zuck, L.D. (eds.) TACAS 2005. LNCS, vol. 3440, pp. 365–381. Springer, Heidelberg (2005)
23. Zhang, S., Saff, D., Bu, Y., Ernst, M.D.: Combined static and dynamic automated test generation. In: ISSTA, pp. 353–363. ACM (2011)