Synthesizing Method Sequences to Detect Object Invariant Violations

Timon Gehr

Bachelor's thesis

Chair of Programming Methodology
Department of Computer Science
ETH Zurich

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

Spring 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

Background Object invariants are predicates specifying valid object states. In case a specified object invariant is not actually an invariant condition under the possible mutating operations on a constructible object graph, it is breakable. Breakable object invariants can be tested in order to show that they are indeed breakable. Ideally, in the presence of breakable object invariant specifications, automated testing tools should often be able to synthesize a sequence of public operations after execution of which the negation of the invariant of a live object holds.

However, this kind of program error is often not not detected by the approach implemented in most testing tools.

It has been observed that in practice breakable invariants often follow certain patterns. The goal of this thesis is the implementation of a template-based approach that generates test suites which contain those patterns in order to better test the commonly occuring kinds of breakable object invariants.

Results A tool has been developed which is able to build test suites targeted at object invariants. The implementation uses a simple form of static program analysis in order to detect which patterns might be applicable. It then generates sequences of operations. Some of the arguments to the operations within a sequence are not specified, turning the sequences into parameterized unit tests. The sequences are assigned priorities, and the concolic testing tool Pex is then used to find appropriate input values.

Conclusions An evaluation of the tool on open source projects indicates that this simple approach for testing object invariants is able to expose a significant fraction of breakable invariants occurring in real code. The breakable object invariants not detected by the tool hint at limitations of the approach and its implementation, as well as future directions to take.

Contents

Chapter 1

Introduction

1.1 Object invariants

Object invariants are predicates specifying valid object states [1]. Objects may encapsulate arbitrary object graphs, and therefore invariants may specify properties of the state of arbitrary object graphs. *Operations* can be executed on objects. For the purposes of this thesis, an operation could be an object construction, a method call, a field read, or a field update, though in general this is dependent on the programming language. An invariant that does not hold is called *broken*. An invariant that can be broken by executing a sequence of operations is called *breakable*. Otherwise it is called *robust*.

```
public class D
{
  int x = 0;
  invariant x != 5;

  public void Update(int a)
    requires a < 4
  {
    x += a;
  }
}
```

Listing 1.1: Breakable invariant

```
public class C
{
  int x = 0;
  invariant x != 5;

  public void Update(int a)
    requires a % 2 == 0
  {
    x += a;
  }
}
```

Listing 1.2: Robust invariant

The above listings provide examples of breakable and robust invariants in a C#-like language extended with contract specifications. D's invariant is breakable. The invariant can be broken, for instance, by executing the following sequence of operations:

```
D d = new D();
d.Update(2);
d.Update(3);
```

C's invariant is robust, because the x field of C instances will always contain an even integer, and hence never the number five. This shows that specified invariants may be weaker than the strongest possible predicates maintained by all sequences of allowed operations.

Formal contract specifications are useful because they allow automated program validation. Hence, the invariants should be strong enough to allow the validation tool to check the relevant properties inside all method bodies. At the same time it should not be possible to execute a sequence of operations that results in a broken invariant.

Different tools assign different semantics to invariants. For instance, since it may be convenient that updates connecting valid object states temporarily violate object invariants, some applications allow invariants to be broken at specific times during program execution. Others introduce object validity as a first-class concept into the programming language, and require all invariants of valid objects to hold at all times [2].

Object invariants can be thought of as hidden method preconditions and as hidden method postconditions [1]. Most validation tools therefore add the invariant of the receiver as an additional conjunct to one or both of method precondition and postcondition during validation of that method.

While this may appear natural, it is never sufficient in order to establish that no invariants are broken during execution of the program. Since only the invariant of the receiver object is validated, violations of invariants of objects different from the receiver object are not detected.

Assuming an infinite memory, clearly, distinguishing breakable from robust invariants is an undecidable problem.[1] Therefore, all validation tools only approximate such a distinguisher, or may require additional evidence in order to complete the task.

1.2 Testing

This work approaches the invariant validation problem from the testing side. In other words, if our approach succeeds, breakability of the invariant in question is established and a piece of evidence in form of an explicit sequence of operations that breaks the invariant is generated. Otherwise the invariant may be breakable or robust. There is, however, the information gained that the approach does not succeed in breaking the invariant, which is sometimes considered useful information.

Recently, dynamic symbolic execution has been proposed for program testing [3, 4]. This is now state-of-the-art and is often dubbed "concolic execution", or "concolic testing" [4]. The approach proceeds by running the program under test in two distinct fashions. On the one hand, the concrete program state is computed. On the other hand, the states of all executions going down the same concrete program path are collected into a symbolic representation. After the program has executed, a number of constraints on the program input corresponding to boolean predicates evaluated in the code will have been collected. Some of those constraints correspond to conditions that can be inverted in order to force a subsequent execution down another, yet unexercised path. The symbolic execution engine collects this information alongside with the constraints. A subsequent execution requires a new concrete program input. Consequently, it is necessary to create a program input that satisfies the collected constraints, some of them inverted.

A constraint solver can be used in order to solve a set of constraints for the new program input. Since the constraints may not necessarily be part of a decidable theory supported by the constraint solver, it is possible that some constraints will not be solvable. In this case, the dynamic symbolic execution engine can substitute some of the variables for their concrete values during the last execution. This generates simpler constraints, that may then be solvable. Of course, this is not guaranteed to succeed. It is also possible that the simpler constraints are no longer satisfiable and some paths are missed. The same approach can be used to compensate for limitations of the symbolic execution engine used. In general there is an infinite number of paths, but implementations of the strategy will only ever cover some finite subset of them.

Dynamic symbolic execution can be used for *parameterized unit testing*. Given some method with a number of input parameters, the strategy may be applied in order to find a sequence of input arguments that execute different paths inside the method body.

This approach relies on all executable code being present. In our case, however, part of it, namely the sequence of operations that is executed in order to break the invariant, should be generated by the testing approach itself. This suggests that dynamic symbolic execution on its

[1] To prove this, for instance implement a class whose objects simulate universal Turing machines, providing a method that performs a finite simulation step. The invariant should specify that the machine is not in the halted state.

own is inadequate for complete invariant testing. In fact, there are classes of common invariant bugs that are usually missed by implementations of concolic parameterized unit testing.

1.3 Invariant-breaking patterns

Work on modular verification [5, 6] has identified multiple recurring patterns that may allow an object invariant to be broken if only invariants of receiver objects are validated. For instance:

- **Direct field assignments**
  Widely-used object-oriented programming languages allow objects to fully expose their fields. If the truth value of an invariant depends on the state of such a field, it may be possible to break it by assigning to the field directly.

```
public class D
{
  public C c = new C();
  invariant c != null;
  /* ... */
}
```

Listing 1.3: Breakable invariant

```
var d = new D();
d.c = null;
```

Listing 1.4: Direct field assignment used to break the invariant

- **Subclasses with stronger invariants**
  Subclasses refine their superclasses. Since inherited superclass methods may rely on the superclass invariant, the invariant is usually inherited. Additionally, the subclass may need to add additional conjuncts to the invariant in order to support its own methods. It is then possible that the subclass invariant's truth value depends on state under control of the superclass. In general, superclass methods may then break subclass invariants. For instance, a subclass may inherit a super class field and miss to override a mutator of it as follows:

```
public class C
{
  int x = 0;
  public int Value(){ return x; }
  invariant 0 <= Value() &&
            Value() < 10;

  public virtual
  void Mutate(int x)
  {
    if (0 <= x && x < 10)
      this.x = x;
  }
  /* ... */
}

public class D : C
{
  invariant Value() != 5;
  /* ... */
  /* (no override of Mutate) */
}
```

Listing 1.5: Breakable invariant

```
var d = new D();
d.Mutate(5);
```

Listing 1.6: Superclass method used to break strenghthened subclass invariant

- **Call-back scenarios**
  A call-back scenario is a situation where a method calls another method on a different receiver, and that method in turn invokes a method on the original receiver. If invariants are allowed to be broken at specific times, and if method calls can be performed during that time, it may then be possible that an object is encountered in an invalid state by a method invocation. When this occurs, a program might get stuck, compute wrong results or permanently break the invariant.

```
public class C
{
  int x = 0;
  invariant -5 < x && x < 5;

  D d = new D(this);

  public void Set(int x)
    requires -5 < x && x < 5;
  {
    this.x = x;
  }

  public void Method()
  {
    x++;
    d.Method();
    x--;
  }

  public void Fizzle()
  {
    x = 2 * x + (0 < x ? -4 : 4);
  }

  /* ... */
}

public class D
{
  C c;
  public D(C c)
  { this.c = c; }

  public void Method()
  {
    c.Fizzle();
  }

  /* ... */
}
```

Listing 1.7: Breakable invariant

```
var c = new C();
c.Set(4);
c.Method();
```

Listing 1.8: Invariant broken because of (unanticipated) call-back.

- **Multi-object invariants and mutable aliasing**
  An invariant can specify properties of multiple objects. If additional references to some of those objects are available, the invariant may be broken by mutating thos objects. Such references could be obtained, for instance, by *leaking* or *capturing*. Leaking occurs when a reference stored in an object field is obtained by invoking some method on the object. Capturing occurs, when a reference passed to a method or constructor is stored in an object field. If certain fields read in the invariant are leaked or captured, the invariant may become breakable.

```
class C
{
  D d;
  invariant d.x == 0;

  public C(){
    d = new D();
  }

  public C(D d)
    requires d.x == 0;
  {
    this.d = d;
  }

  public D LeakD()
  {
    return d;
  }
}

class D
{
  public int x = 0;
}
```

Listing 1.9: Breakable invariant

```
var c = new C();
var d = c.LeakD();
d.x = 2;
```

Listing 1.10: Invariant broken by leaking and direct field assignment.

```
var d = new D();
var c = new C(d);
d.x = 2;
```

Listing 1.11: Invariant broken by capturing and direct field assignment.

1.4 Approach

In this section, we give an informal introduction to some of the specifics of the approach implemented during the thesis. It was developed by the supervisors.

Our goal is to *generate test suites* consisting of sequences of operations built from invariant-breaking patterns. We build sequences parameterized by input values or receivers to some of the operations occurring in the sequence. A sequence parameterized in this way is called a skeleton. An existing test generation tool supporting parameterized unit tests can then be run on those skeletons in order to determine suitable values for the parameters.

```
void Sk(C rcvr, int i, C c,
        bool b, int j, string s)
{
  rcvr.AddToCounter(i);
  rcvr.Append(c);
  var x = rcvr.Read(b);
  x.Buzz(j,s);
}
```

Listing 1.12: A skeleton usable as a parameterized unit test

```
C rcvr = new C();
C c = new C();

rcvr.AddToCounter(5);
rcvr.Append(c);
var x = rcvr.Read(false);
x.Buzz(0,"123");
```

Listing 1.13: Finished sequence after parameter substitution

Our approach generates client code accessing the public interface of parts of the tested program. We test the invariant of one public class at a time. More specifically, a sequence of operations should be generated, and the invariant of an instance of the respective type should be violated after that sequence of operations has run to completion. Classes are assumed to specify sufficient preconditions for all public methods. Any way to use the public interface in a correct way such that some object invariant is broken in the final state is considered a bug. Therefore our approach not only detects object invariant violations that may occur in practice during execution of the complete program that the classes are parts of. It will additionally expose public classes whose implementations rely on non-modular implicit properties of the program execution for correctness. We do not attempt to infer the object invariant, and instead rely on explicit object invariant specifications on behalf of the programmer. We assume that the invariant is given as a boolean predicate implemented as an instance method.

A *template* is a strategy for generation of skeletons. We use templates that capture scenarios in which invariants may be violated. Our approach combines multiple templates, some of them recursively referring to other templates. We bound the length of skeletons by a small constant. Likely irrelevant or redundant sequences can be pruned using additional strategies on a by-template basis. If a skeleton contains operations that do never operate on related parts of the state, or on parts of the state not connected to the targeted object invariant, they are not generated. In this case, additional heuristics can be applied. The sequences can be assigned a priority, and higher-priority sequences can be processed first. Sequences that share a priority should be generated in a randomized order.

In order to assess whether two operations may operate on the same object fields, results of a static analysis can be used, such as conservative read and write effects. If two skeletons operate on non-overapping sets of object fields, they never operate on related parts of the state.

Our approach makes use of the following templates:

- **Default template**
  This template is the union of all other templates. It can be used to generate sequences for building object states, for instance states that violate an object invariant.

- **Fall-back template**
  This template is implementation-specific. It is used to mutate a single receiver object. This can be required as a last step, after other templates have been applied, or when none of the

templates are applicable, but the invariant is still breakable. Any conceivable approach for synthesizesing sequences of operations can be used to implement the fall-back template.

- **Public fields template**
  This template describes sequences that consist of assignments to public fields of a single receiver. This exploits the direct field assignments pattern. Public field assignments are particularly likely to lead to violations because no preconditions can be specified for them.

- **Subclassing template**
  This template applies if a subclass may strenghthen a superclass invariant. It describes sequences that first apply a number of operations to build an appropriate object state and then call an operation of the superclass. The superclass methods are designed to maintain a potentially weaker invariant, often without knowledge of the subclass invariant, which they may violate. The violation occurring in such a method may be conditioned on the object state, which is why the prefix is also generated.

- **Call-back template**
  The call-back template describes sequences that apply operations to build an object state and then call into a method that might call a method on a different receiver and back into a public method of the same receiver. This is our treatment of call-back scenarios.

- **Multi-object invariants template**
  We use the following templates to attempt obtaining references to data read in the targeted object invariant.

  - **Leaking template**
    References may be obtained by operations returning them. The leaking template describes sequences which apply operations to build an object state and end in an operation that may leak an object reference.
  - **Capturing template**
    Another way to obtain a reference to an object read in the invariant is by updating some field to reference an existing object. Sequences which end in an operation that may capture an object reference are captured by the capturing template. The prefix of the sequence consists of one subsequence for building a receiver state, and one subsequence for building an object state for the object to be captured. This makes sense because whether something is captured may depend on both the receiver state and the state of the captured object.

  The multi-object invariants template applies the leaking and capturing templates and then mutates the leaked or captured object using further operations. In this way, improper encapsulation of an object read in the invariant may be detected.

Chapter 2

Implementation

2.1 Basis

2.1.1 Code contracts

We have concentrated on testing C# code. The C# language does not have native support for contract specifications. We have used specifications in Code Contracts, which is the standard tool for contract-based programming in C# [8]. In order to get the specifications into a more convenient form we have slightly rewritten the assemblies under test before generating tests using a custom rewriter developed by the supervisor.

2.1.2 Pex

An academic licence for the source code of the concolic testing tool Pex [7], developed at Microsoft Research, has been made available to us. However, we have not been provided any source code documentation.

Pex is a tool supporting parameterized unit testing for .NET using dynamic symbolc execution. It is based on Microsoft's implementation of the CLI. Conjuncts of the path constraint are inverted using a custom strategy. The constraint to invert is chosen using heuristics, to the end of quickly establishing a reasonably large branch coverage [7].

Pex supports ignoring certain paths using the PexAssume static class.

Pex may generate short sequences of operations to bring some objects into the wanted initial state. However, this approach did not seem extensible to support our purposes in a straightforward way. Therefore, our implementation generates .NET assemblies containing static classes with methods containing the sequences generated using the System.Reflection.Emit API.

Pex has built-in support for Code Contracts.

2.2 Effect analysis

In order to support sequence generation, our implementation applies some static analysis. The current implementation uses a simplistic field-level read and write effect analysis.

The analysis builds a call graph. In order to support method overrides, subtypes of the static type of the invocation target are determined. The analysis considers all classes in loaded assemblies. The call graph then has an edge for every potential dynamic invocation target of a virtual method.

The call graph is traversed, and field reads and writes are collected. This approach is quite imprecise: The analysis is flow-insensitive and it does not use any form of alias analysis beyond type information. Also note that field writes through pointers are not recorded.[1]

[1] Pointers may occur in safe CIL bytecode. They correspond to `ref` and `out` arguments in C#.

The analysis depth is limited by a small constant. During analysis of the System namespace, it is cut off sooner, in order to improve performance. The System.Diagnostics namespace is ignored completely, as state changes inside that namespace are not supposed to affect normal control flow.

Alternatively, we could have used existing research, for instance SEAL [9], a read and write effect analysis tool for the CLI.

2.3 Skeleton generation

First some general remarks: In order to prevent spurious NullReferenceExceptions in the generated tests, all object references that are used as receivers by the tool are assumed to be not null by inserting calls to PexAssume.IsNotNull into the operation sequence. The invariant is checked using PexAssert.IsTrue. Public fields are generally treated like pairs of setter and getter methods, but special cased in certain contexts. As mentioned in the introduction, method and constructor calls, field assignments and field reads are referred to as *operations*.

```
void PFLLeakFieldFCellRcvrSk0OpSk0Pat1(PublicFields rcvr, int arg1) {
  PexAssume.IsNotNull(rcvr);
  rcvr.LeakField();
  var tmp1 = rcvr.Cell;
  PexAssume.IsNotNull(tmp1);
  tmp1.V = arg1;
  PexAssert.IsTrue(rcvr.$InvariantMethod$());
}
```

Listing 2.1: Example of a generated skeleton

We have implemented skeleton generator classes according to the different templates discussed in the introduction. In order to only generate relevant skeletons, every generator is parameterized by a method to prepare invocation of, the *target method*, as well as a number of generated local variables holding receiver objects (possibly zero), to be used in specified ways. The generator will attempt to mutate the fields that are read by the target method using the receiver objects. Some templates may also create all the receivers on their own, for instance the leaking and capturing templates applied to constructors may not require any receivers from outside.

For the top-level skeleton generator, the target method is exactly the invariant method of the receiver that is attempted to be broken. Otherwise, the target method might refer to the invariant method of some class of an object a receiver object was leaked from. Alternatively the target method could refer to some method that should be called next. This can be required because at runtime, the state of the client heap may have to be changed such that execution goes down the correct path inside that method.

To this end, the generators usually will compute the read effect of the target method using the simple analysis described above.

All skeletons are generated lazily using the iterator pattern. Streams of sequence skeletons are manipulated using custom combinators in order to facilitate the implementation. This includes a combinator that lazily merges a stream of streams into a single stream in a randomized fashion. At each step, a stream is picked uniformly at random. Its first element then is the next element of the randomized stream. This randomized join combinator is used wherever possible and reasonable in order to randomize the order the skeletons are generated. This helps to cover a wider spectrum of sequences in case not all sequences can be generated or explored in the given time frame. Without some form of interleaving, similar skeletons would usually be generated in sequence.

The default template is implemented as a multiplexer that uses this randomized join combinator in order to combine the results of all the other skeleton generators.

2.3.1 Operation inputs and results

Every operation has zero or more input arguments and zero or more results. A field write has one input argument, the written field value. A field read has one output argument, the read field value. The types of both values correspond to the types of the respective fields.

Methods and constructors have one input argument for each parameter of the respective method or constructor declaration which is not qualified `out`. Note that there is an input argument for `ref`-qualified parameters.

Methods and constructors have one result for each parameter of the respective method or constructor declaration which is qualified `out` or `ref`. Their types correspond to the declared parameter types of the `out`- or `ref`-qualified parameters.

Constructors have one additional result, which is the constructed value of the appropriate type. Methods may have one additional result, which is their return value, of their declared return type.

Results from earlier operations may be freely re-used as input arguments to further operations, given that the results are assignable to variables of the input argument types. The tool supports exhaustive enumeration of all ways that results may be re-used.

2.3.2 Exhaustive generator

This generator implements the fall-back and public field assignments templates. We have chosen and implemented a straightforward fall-back strategy with some optimizations, but existing approaches could have been used alternatively.

The fall-back strategy is to generate sequences on the receiver object in an exhaustive way, pruning some uninteresting or redundant sequences:

- Only operations whose side effects may affect the execution of later operations are added into the sequence skeleton.

  We consider operations that may write to some field read in some later operation, according to the read and write effects computed by the strategy described in section 2.2, as well as operations that generate a result that can be used as an argument later in the sequence. In the current implementation, only the first kind of dependency is taken into account.

- There are pairs of operations that can be executed in any order to the same effect. In other words, operation sequencing is commutative restricted to such a pair.

  We can divide the set of all operation sequence skeletons into multiple equivalence classes. Two sequences are considered equivalent iff they differ only in the ordering of operations with commutative sequencing. Since all skeletons inside an equivalence class by definition have the same effect on the heap, only one of their elements has to be generated.

  We detect a subset of the commutative operations:

  Let two operations be called *independent* if for any field in the intersection of the unions of their respective read and write effects has an empty intersection with the union of their respective write effects. This means that any field written in one operation is neither read nor written in the other operation. All effect overlap is constrained to the read effect of the operations. Clearly, sequencing of independent operations is commutative.

  We consider the following order on sequence skeletons: Assign a priority to every operation under consideration. Then sequence skeletons can be ordered lexicographically by using the ordering implied by the priorities. Our implementation uses the names of the respective fields and methods as the priorities. (The astute reader has noticed that this does not discriminate field reads and writes on the same field. This will not be a problem because operations of those two kinds are never independent.)

The generator computes an approximation of the read and write effects of all methods of the given receiver object type. It computes the skeletons in ascending order of length. The public fields

template is implemented by giving precedence to public field assignments. With this exception, the generation order is completely randomized. For every length, the generator computes the sequences using a recursive helper procedure. Arguments to the helper procedure are the length l of the sequences that should be generated, a field read effect R and a set of reusable types T. It returns a stream of sequence skeletons. The read effect is used to keep track of fields whose mutation may change some behaviour we are interested in. The set of reusable types is used to additionally select methods that may generate results which can be re-used later in the sequence. Initially, the procedure is called with the read effect of the target method. For l nonzero, the helper procedure first selects a subset of operations O on the receiver object that either have a write effect whose intersection with R is non-empty or have some result type within T. For each operation $o \in O$, it computes the union of o's read effect with R as R', and the union of the set of o's argument types with T as T'. l' is obtained as $l - 1$. Those are used as arguments to a recursive invocation of the helper procedure in order to obtain the remaining prefix of the generated sequence skeletons. In case o is an assignment to a public field, the respective field is first removed from R'.

This is sensible, since no write to that field further up the sequence skeleton prefix may possibly affect any operation after the direct field assignment: it is fully overwritten. The same optimization cannot be applied to operations in general, because they might not write to every field in the conservative write effect approximation during a single method invocation. In order to get skeletons of the correct length, o is lazily appended to some of the obtained sequence skeletons. In order to determine whether or not a recursively obtained sequence skeleton is chosen and extended, the longest suffix consisting of only operations independent of o is considered. The sequence skeleton is chosen iff all operations occurring in this suffix have a priority lower than or equal to o.

The streams of sequences obtained this way for each operation in O are then combined using the randomized join combinator. However, in order to support the public fields template, sequences ending in public field writes are randomized separately and generated first.

2.3.3 Leaking and capturing generator

This generator implements the leaking and capturing templates. Every possible operation on some receiver object is a candidate for an operation that may leak or capture. Furthermore, any constructor may also leak or capture. We use a simple analysis in order to determine which operations may leak or capture. To take into account the fields relevant for leaking and capturing, the computed approximate read effect of the target method and effects of each candidate operation are taken into account.

We say that two types are *compatible* if one is a subtype of the other and *incompatible* otherwise. Clearly, if two types are incompatible, references of the two types cannot alias.

Further note that a field in the target method's read effect is only relevant for leaking and capturing if there are other fields in the read effect that are immediately reachable from an object of a dynamic type compatibe with the type of the field. Otherwise there is no possible way to change state read by the target method. Furthermore, any immutable type is not relevant for analysis. However, immutable types may still contribute to the computed field write effects, in case they have some fully encapsulated internal state. As this seems to be indeed the case for Microsoft's System.String implementation, we have built a simple analysis to exclude fields of immutable types, where System.String is special cased. The same is done for value types without fields of mutable reference types. With the exception of System.String, the last two optimizations do not contribute to the precision of our approach, because types whose fields can never be written to using the public interface are excluded in later, and more costly stages of sequence skeleton generation as described below:

- **For leaking**
  The read effect of the candidate operation is intersected with the read effect of the target method. The resulting set of fields determines a conservative approximation of fields that

may be leaked. Sequence skeletons are generated separately for every result of the candidate operation, excluding newly created objects returned from constructors.

Sequence skeletons are only generated for a given result if its type is compatible with at least one type of a field field that may be leaked by the operation.

We have added an improvement that may detect more cases of bugs: For every result, we first determine which potentially leaked fields have types that are compatible with the result type. From these types, we compute the most general type. In case it is below the result type, an additional assumption is added into each of the sequences using PexAssume. Namely, it is assumed that the type will actually conform to the more specific type we have determined. Otherwise, clearly no leaking has taken place. Furthermore, the result type is immediately down-cast to the more specific type, which potentially supports more operations. While this has proven to be a benefit in practice, it is not a complete treatment of the down-cast issue.

Note that by our definition of operation results, this approach already includes leaking by `out` and `ref` parameters.

Because a method that is not a constructor may leak depending on state, a simple heuristic is used: The receiver object is mutated using the default template prior to applying the candidate leaking operation. The target method is given by the candidate leaking operation.

After an object has been obtained using the leaking operation, it is mutated using the default template and the original target method.

The leaking approach may be illustrated as follows:

```
/* operations on r, using candidate_leaking_operation as the
   target method */
tmp = r.candidate_leaking_operation();
PexAssume.IsNotNull(tmp as T); /* where T is a potentially more
   specific type */
T l = (T)tmp; // down-cast
/* operations on l, using the current target method, for
   instance, r's invariant */
// i.e. we do not try to break l's invariant, if there is any.
```

In case the candidate operation is a public field, the generator also attempts to leak a field read in the invariant into the public field. If such a sequence is found, the public field is considered even if it does not occur directly inside the target method's read effect.

- **For capturing**
  The capturing approach is mostly dual to the leaking approach. The write effect of the candidate operation is intersected with the write effect of the target method. The resulting set of fields describes a conservative approximation of fields that may capture some input object. The sequence skeletons are then generated separately for every input argument of the candidate operation. Sequences are only generated in case the argument type is compatible with at least one field that may be captured.

  Similar to above, the receiver object is mutated using the default template prior to applying the candidate capturing operation. Additionally, the object that should be captured is mutated by using the default template. In both cases, the candidate capturing operation is used as the target method.

  In case the candidate operation is a public field, additionally, sequences are generated to capture state written into the public field.

  After capturing of an object, it is again mutated using the default template. However, in this case the original target method is passed on as the target method.

Note that the case when an input argument is a ref argument is already covered: It is important that the object that was passed in will be used to mutate the captured object, not the reference we obtain from potential assignments to the ref argument.

The capturing approach may be illustrated as follows:

```
/* operations on r, using candidate_capturing_operation as the
   target method */
/* operations on c, using candidate_capturing_operation as the
   target method */
r.candidate_capturing_operation(c);
/* operations on c, using the current target method, for
   instance, r's invariant */
// i.e. we do not try to break c's invariant, if there is any.
```

The leaking and capturing generators do not cover all the cases how an object may be broken by leaking and capturing.

The sequence skeletons are built by exhaustively combining the possible subsequences, in a randomized order. To improve performance, the implementation sometimes checks whether some of the subsequences are empty. This can avoid iterating over the other subsequences. Since there are potentially many subsequences, the resulting improvement is quite large.

2.4 Prioritization

The public fields, subclassing and call-back templates describe skeletons that are already generated by the above implementation. However, in practice the number of generated skeletons can be huge and the execution times of testing tools are often bounded by timeouts. In such a constrained environment, it is useful to first consider those sequences that are more likely to provide a result. Therefore, our implementation prioritizes skeletons satisfying the patterns of the public fields, subclassing and call-back templates. In case of the public fields template, there is no check for the exact pattern, which would exclude all skeletons that contain anything but public field assignments, but instead prioritization depends on the number of public fields assigned to in the generated sequence.

2.5 Practicability improvements

In order to make the approach run satisfactorily on real projects, we have implemented some extensions dealing with the most pressing issues.

2.5.1 Arrays

Since arrays are handled specially in the CLI bytecode and contain implicit contracts, they require manual support. At the time of writing, we only support single-dimensional arrays. Arrays are seen as objects supporting three operations:

- Getting the length.
- Reading an array slot using an index.
- Storing to an array slot using an index and a new value.

The implementation uses the metadata of dummy declarations in order to treat the array operations like other operations throughout the code base. Arrays are then special cased during byte code generation. PexAssume.IsInBounds is used to constrain the indices to valid ones. An example of a generated skeleton that contains array operations follows:

```
void LLeakP0RcvrSk0OpSk5Pat1(C rcvr, int arg1, int arg2, int arg3) {
  PexAssume.IsNotNull(rcvr);
  var tmp0 = rcvr.Leak();
  PexAssume.IsNotNull(tmp0);
  PexAssume.InRange(arg1, 0, tmp0.Length);
  var tmp1 = tmp0[arg1];
  PexAssume.IsNotNull(tmp1);
  PexAssume.InRange(arg2, 0, tmp1.Length);
  var tmp2 = tmp1[arg2];
  PexAssume.IsNotNull(tmp2);
  tmp2.SetX(arg3);
  PexAssert.IsTrue(rcvr.$InvariantMethod$());
}
```

Listing 2.2: Example of a generated skeleton containing array operations. The code updates a field of an object referenced inside a strided two-dimensional array.

2.5.2 Genericity

C# supports parametric polymorphism in the form of generic types and generic methods. The generic argument types can be constrained using subtyping constraints and default construction constraints. A subtyping constraint restricts the valid argument types to subtypes of a given type. Subtyping constraints may depend on other generic arguments, though circular dependencies are illegal. A default construction constraint restricts the valid argument types to types that can be constructed without providing constructor arguments. The constraints are taken into account during modular type checking of the generic definition.

Our implementation also takes into account generic types and methods. Arguments satisfying the constraints are deduced using a naive heuristics. This sequence of arguments is further referred to as a *satisfying assignment*. The heuristics uses types defined in loaded assemblies. The subtyping constraints are processed in topological order regarding their dependencies. The heuristics chooses maximally general types. In case there are multiple maximally general types, the first one found is chosen. In case there is none, the heuristics fails.

Parameters occuring in constraints of other parameters are substituted as soon as an argument is chosen. In case a constraint becomes malformed during substitution, the heuristics fails. Note that if there is one single subtyping constraint, the heuristics will choose the upper bound for the argument. This is comparable to type erasure as present in Java.

During subtype search as described in section 2.2, a different heuristics is used. Generic types are instantiated with the same generic arguments the given super type was instantiated with, if any. This works remarkably well with the standard collection types.

We currently use a slighly modified version of the heuristics: Since Pex appears to have better support for strings and integers than for object identities, every parameter is first checked for compatibility with `string` and then `int`. The first one that satisfies the generic parameter constraints is chosen. If none of them satisfies the constraints, the algorithm continues in the usual fashion.

For the examples below, assume that the given classes are the only classes occuring in each program.

```
public class C<R,S,T>
  where R : T, IBar      // two subtyping constraints
  where S : IFoo, new() // subtyping and construction constraints
  where T : S
{ /* ... */ }
```

```
public interface IFoo
{ /* ... */ }

public class D : IFoo
{
  public D()
  { /* ... */ }
  /* ... */
}

public class E : D, IBar
{ /* ... */ }

public class F : D
{ /* ... */ }
```

We run the heuristics in order to find valid inputs for class `C` in the above code. The only type parameter whose constraint does not depend on another type parameter is type parameter `S`. The most general type satisfying the `IFoo` subtyping constraint and the default construction constraint is `D`. Therefore we substitute `S` for `D`. The most general type satisfying the constraint for `T` is now `D`. After substituting `T` for `D`, `R` can be processed. The most general type that is a subtype of both `D` and `IBar` is `E`, therefore we substitute `R` for `E`. The heuristics succeeds in this case and instantiates the type `C<E,D,D>`

We now give examples demonstrating that this naive greedy heuristics does not always find a possible satisfying assignment.

```
public class C
{
  void M<T,S>(T t, S s)
    where T : D, J
    where S : T, I
  {
    /* ... */
  }

  /* ... */
}

public interface I
{ /* ... */ }

public interface J
{ /* ... */ }

public abstract class D
{ /* ... */ }

public class A : D, J
{ /* ... */ }

public class B : D, J
{ /* ... */ }

public class X : B, I
{ /* ... */ }
```

We run the algorithm on the generic method `C.M`. The method can be instantiated using the argument types `<B,X>`.

In case the heuristics chooses `A` for `T`, the constraint for `S` becomes unsatisfiable.

```
public class C<A,B>
  where A : D
  where B : F<A>
{ /* ... */ }

public class F<A> // (helper definition)
  where A : E
{ /* ... */ }

public class D
{ /* ... */ }
public class E : D
{ /* ... */ }
```

The constraint on parameter `A` of `F` is not taken into account when choosing an argument for `A` during argument search for `C`. The heuristics will therefore assign `D` to `A` and the constraint on `B` becomes malformed.

There are further complications related to generic parameter variance. Devising a better method for satisfying generic parameter constraints is out of the scope of this thesis.

Chapter 3

Experimental Evaluation

In order to investigate the effectiveness of the approach and the implementation, we have run the tool on a suite of large open source C# applications that use Code Contracts and contain object invariant specifications. We have investigated all applications on GitHub that could be found by using the query "ContractInvariantMethod" in early January 2013. The applications Yaml and Boogie & Dafny were taken from CodePlex. Boogie & Dafny were choosen for their large number of specified object invariants.

A total of 372 object invariants are included in this investigation, disregarding classes that fully inherit their invariant specification from their super classes.

For our purposes, we have extended the Code Contract support of Pex such that precondition failures that are caused by the tool calling into the public API are never reported. At the same time, precondition failures caused by the code under test are still detected.

3.1 Manual inspection

While it makes sense to base the evaluation of a testing tool on real data, it is not obvious how to interpret the results in the best possible fashion. While in this case it is clear that the tool should perform at least as well as the dynamic symbolic execution it is based on alone, an upper bound is missing.

In order to render the found numbers of breakable object invariants more meaningful, we have manually analyzed the code under test in a best-effort fashion. This enables better reasoning about the evaluation results, and insights about limitations of the approach or the implementation can be gained.

Of the 372 object invariants, 129 are breakable, and one is not well formed (meaning its evaluation may terminate with an exception). The largest number of invariants and breakable invariants was obtained from the Boogie & Dafny source code.

During manual inspection, it became clear that many invariants are non-null constraints. Namely, 273 invariant specifications would be completely unnecessary in a language that allows directly declaring class types that do not include null. Out of those, 54 are breakable.

There are 62 invariants that read the state of more than one object. Out of those, 52 are breakable.

There are 181 invariants that have constructor preconditions that are equivalent to the invariant of the fields initialized to constructor arguments after construction and it can be very easily verified (made precise below) that the invariants cannot be broken. In some cases readonly fields with non-null constraints are initialized to a newly created object in the construcor. We call those invariants *trivially robust*. The state initialized by constructors of a class with a trivially robust invariant may either be readonly, or a property field with auto-generated accessors. Code Contracts has a feature that automatically adds pre- and postconditions to those accessors such that

they cannot violate the invariant.[1]

```
public class NonEmptyArrayWrapper
{
  private readonly int[] array;

  public NonEmptyArrayWrapper(int[] array)
  {
    Contract.Requires(array != null && 0 < array.Length);

    this.array = array;
  }

  [ContractInvariantMethod]
  void ObjectInvariant()
  {
    Contract.Invariant(array != null && 0 < array.Length);
  }

  /* ... */
}
```

Listing 3.1: Example of a trivially robust invariant.

Conversely, there are 10 invariants that can be broken during construction of the receiver object.

There are 65 invariants that can be broken by assigning to a single public field. 64 of them are inside the Boogie & Dafny source code. Note that this does not necessarily indicate a low-quality code base. Most detected violations circumvent unchecked conventions taken in the code, most often non-null specifications given in comments.

[1] The author considers this feature questionable, because it makes the object invariant part of the public API

Application	Invariants	Breakable	Generated	Broken
Boogie & Dafny	258	111	58	53
ClueBuddy	4	1	1	1
Draugen	8	3	3	3
goalstracker	5	1	1	1
Griffin Framework	3	1	1	1
Love-Studio	8	3	3	1
ncqrs	3	0	0	0
neostructure	3	0	0	0
Trinity-Encore	73	7	1	1
YAML	7	2	2	2

Table 3.1: Summary of experimental evaluation results. The first column indicates the number of public classes with an invariant specification within the given application. The second column shows the number of breakable invariants as determined by manual inspection. The third column shows for how many of the breakable invariants, tests were successfully generated. The last column shows the number of invariants that were successfully broken by the tool.

3.2 Experiments

We have run the tests using a timeout of 4 minutes, a maximum sequence length of 15, and we have limited the generated skeletons to the first 300. The results of the experimental evaluation are contained in the table above. The violations in applications other than Boogie & Dafny were first detected using the tool by Valentin Wüstholz, who currently is a Ph.D. Student at ETH.

We have tested the approach on all invariants in each application. Not all generated skeletons have produced runnable tests, for reasons discussed in the next chapter.

In total, tests for 201 invariants have been generated, 71 of them breakable. 63 of those invariants have been successfully broken by the tool. The remaining seven cases will be discussed in the limitations section. The one invariant that is not well formed is successfully detected. This information is not included in the table. The application containing this invariant is Trinity.Encore.

35 invariant violations were due to assignments to public fields. 21 violations were due to leaking or capturing. One invariant violation was caused by a call-back. The remaining violations were detected purely by the fall-back template.

All seven missed violations require some form of leaking or capturing in order to occur.

After some practical experience from experimental evaluation, support for genericity and arrays was introduced. Furthermore, the decision to analyze the “System” namespace less thoroughly was taken for performance reasons.

Chapter 4

Limitations

The implementation relies on the support for automatic input object construction that has been implemented in Pex. One cause for missed invariant violations is failure of automatically constructing an object of the class which specifies the breakable invariant. Input object construction is a very important issue for our approach, but building our own was outside the scope of this thesis.

For nine classes, we were unable to get a result because there were errors in the native part of Pex. We now go on to discuss the missed invariant violations. We consider the subset of cases for which tests were successfully generated.

4.1 Missed violations

There were seven cases of undetected invariant violations. The first one, from the Boogie project, requires an interesting combination of leaking and capturing in order to occur:

```
public class StmtList {
  // ...
  public readonly List<BigBlock/*!*/>/*!*/ BigBlocks;
  // ...
  public StmtList([Captured] List<BigBlock/*!*/>/*!*/ bigblocks,
      IToken endCurly) {
    Contract.Requires(cce.NonNullElements(bigblocks));
    // ...
    this.BigBlocks = bigblocks;
  }
}
public class StmtListBuilder {
  List<BigBlock/*!*/>/*!*/ bigBlocks = new List<BigBlock/*!*/>();
  // ...
  [ContractInvariantMethod]
  void ObjectInvariant() {
    Contract.Invariant(cce.NonNullElements(bigBlocks));
  }
  public StmtList Collect(IToken endCurlyBrace) {
    // ...
    return new StmtList(bigBlocks, endCurlyBrace);
  }
}
```

Listing 4.1: The relevant parts of the missed invariant violation of class `StmtListBuilder`

In order to break `StmtListBuilder`'s invariant, its private field bigBlocks has to be leaked by the method `Collect` in order to construct a new object of type `StmtList`, which then contains the reference inside a public field.

This kind of violation is not covered by our implementation of the leaking and capturing templates. As StmtListBuilder does not declare a field of type `StmtList`, the method `Collect` is not taken into account as a possible candidate method for leaking.

In Boogie & Dafny, there are two cases of simple capturing of a `List` inside a readonly public field with non-null constraints on the `List` elements. In one case, the necessary skeletons are not within the first 300 generated ones, and in the other case, the program exploration times out before the appropriate skeletons are taken into consideration. This hints at limitations of the simple static analysis approach: Many irrelevant skeletons are not eliminated by static analysis.

The last of five missed violations for Boogie & Dafny is due to failure of creating a necessary input object: The appropriate skeleton to exploit leaking was within the first ten skeletons that were generated.

```
public class CallStmt : Statement {
  [ContractInvariantMethod]
  void ObjectInvariant() {
    // ...
    Contract.Invariant(cce.NonNullElements(Lhs));
    // ...
  }

  public readonly List<Expression/*!*/>/*!*/ Lhs;

  public CallStmt(IToken tok, List<Expression/*!*/>/*!*/ lhs,
      Expression/*!*/ receiver,
  string/*!*/ methodName, List<Expression/*!*/>/*!*/ args)
  : base(tok) {
    // ...
    Contract.Requires(cce.NonNullElements(lhs));

    this.Lhs = lhs;
    // ...
  }
  // ...
}
```

Listing 4.2: Invariant violation in CallStmt

```
CallStmt rcvr = ...;                    // instantiate receiver
IEnumerable<Expression> arg1 = ...; // an argument to the skeleton
PexAssume.IsNotNull((object)rcvr);
List<Expression> tmp0 = rcvr.Lhs;
PexAssume.IsNotNull((object)tmp0);
tmp0.AddRange(arg1);
```

Listing 4.3: The appropriate generated skeleton taken from the test report. It is an instance of the leaking template. The dots indicate parts to be filled in by Pex.

Pex is unable to instantiate an `IEnumerable` containing a null entry without manual user interaction in this case, which would be required in order to violate the invariant of `CallStmt`. Note that the simpler skeleton using index assignment is not taken into consideration until after the time limit. This again indicates that the static analysis does not deliver strong enough results.

The remaining two cases of missed invariant violations are from the application Love-Studio. They are both similar. One example is given below.

```
class Studio {
  public Studio() {
    // ...
    if (File.Exists(projFile))
      openProject(projFile);
    // ...
  }
  // ...
  private Editor _currentEditor;
  private BiMap<string, Editor> _files;
  private Project _currentProject;
  public Editor CurrentEditor
  { get { return _currentEditor; } }

  public BiMap<string, Editor> FileEditorMap
  { get { return _files; } }
  // ...
  [ContractInvariantMethod]
  private void ObjectInvariant() {
    // ...
    Contract.Invariant(_currentEditor == null
    || _files.Contains(_currentEditor));
  }
  // ...
  void newEditor_GotFocus(object sender, EventArgs e) {
    _currentEditor = (Editor)sender;
    // ...
  }
  private Scintilla SpawnEditor(string fileName, TabControl cC) {
    // ...
    Scintilla newEditor = new Editor();
    newEditor.GotFocus += new EventHandler(newEditor_GotFocus);
    // ...
    _files[(Editor)newEditor] = fileName;
    return newEditor;
  }
  public void OpenFile(string fileName, bool autoFold = false,
     TabControl containingControl = null) {
    if (_currentProject == null) {
      // ...
      return;
    }
    Scintilla newEditor = SpawnEditor(fileName,containingControl);
  }
  private void openProject(string fileName) {
    _currentProject = new Project(fileName,this);
    // ...
  }
}
```

Listing 4.4: Some relevant parts of the code to illustrate the invariant violation.

The first roadblock is the fact that a specific file has to exist on disk in order to initialize the field `_currentProject`. Even if that constraint can be flipped by the dynamic symbolic execution engine or manually, it is still necessary to call a method of the receiver using a leaked field of the same receiver as the argument. None of our implemented generators covers this case. This is also the reason why the other violation (inside the class `TabManager`) remains undetected.

Chapter 5

Related work

We are aware of multiple other attempts to generate sequences of operations for testing. However, the approaches we are aware of follow the goal of maximizing overall branch or statement coverage, while our approach concentrates on invariant specifications.

One simple approach for generating sequences is by randomly choosing the constituent operations. RANDOOP [10] improves on the naive random approach by adding incrementalization: Known sequences are not tested again, and sequences that lead to errors are not extended by more operations. The testing tool takes feedback from tests that were already run into consideration. Furthermore, RANDOOP maintains a pool of values, both predetermined primitive values and values generated as results of operations inside already considered sequences. In this way, a large amount of valid objects suitable for testing further parts of the application are generated. Arguments to operations are then chosen randomly from this pool.

There are evolutionary approaches, like [11] that attempt to improve over other randomized approaches by exploiting evolutionary effects. A population of sequences is evolved by using some fitness criterion, like the number of covered branches. Furthermore, sequences are partly randomized (mutation) and partly combined from existing sequences (crossover). The crossover operator used in [11] creates new sequences by randomly cutting the existing ones and recombining the beginnings and endings.

A limitation of random testing in general is that some branches are very unlikely to be covered using the resulting input value distributions. Evolutionary testing also suffers from this limitation. Therefore, Evacon [12] combines evolutionary testing and concolic testing. The evolutionary testing approach is used in order to generate skeletons, and the concolic testing approach is used to find appropriate input values. This is similar to how we approach invariant testing. There is a feedback loop involved: The concrete sequences that had arguments filled in by concolic execution are used to generate chromosomes for further evolution of skeletons.

Seeker [13] is an approach to method sequence generation that exploits static code analysis in order to choose suitable methods for extending existing skeletons. It generates skeletons, and then uses dynamic analysis (i.e. concolic execution) in order to cover branches. In case some branch could not be covered, Seeker statically analyzes the condition that should be flipped and heuristically suggests methods that may flip the condition. Then dynamic analysis is run on skeletons that are extended by those methods appropriately. This process is repeated until it times out.

Symstra [14] and [15] are two different approaches for testing related to research on model checking. Both rely on some form of symbolic model checking, and both approaches can be used to generate sequences of operations. [15] manually rewrites Java code into code that performs symbolic execution of the same code, in order to be able to use the existing Java PathFinder Model checker to exploit, for instance, its partial order and symmetry reduction capabilities. Symstra symbolically explores all method sequences up to some bound. It manually implements state subsumption. This can then be used to prune sequences completely from consideration without sacrificing exhaustiveness.

Finally, since our approach relies on concolic testing, it is also interesting to study work that improves on the method. In [16], program verification techniques are applied in order to prune paths that can be proven to never lead to an error. The approach can be described as using the method of weakest preconditions using the symbolic state tree whose paths are concolic path explorations as the control flow graph. (The authors use a different terminology though).

Chapter 6

Conclusion

As part of this thesis, a project work has been carried out. An extension to the dynamic symbolic execution tool Pex for invariant testing has been developed according to an approach developed by the supervisors. The developed tool has been evaluated on a suite of open source projects with a significant number of invariants. It has been shown that the implementation of the approach is able to break a significant fraction of the actually breakable object invariants. However, a few limitations have been seen that would likely be more serious for more complicated object invariants than those which are present in the tested open-source projects found online.

Most missed invariant violations stem from limitations of Pex' support for automated input object construction, which our implementation relies on. Pex was designed for parameterized unit testing, which includes manual writing of unit tests, and therefore it is not absolutely crucial for Pex that input object construction is fully automatic under normal circumstances.

Another limitation of the developed implementation is that some conditions may rely on object identities, but the approach does not specifically attempt to execute operations just in order to use their results as arguments to future operations.

Since the used analysis is simple and imprecise, it is often the case that too many irrelevant skeletons are generated to explore. One possible considered improvement that would mitigate this problem in some cases is to subdivide the object invariant into conjuncts and to analyze each conjunct separately. We have not done this yet, but for the tested projects, the effects of the imprecise static analysis were seen even for some classes whose invariants contain few conjuncts. Additionally to subdividing the invariant into conjuncts, it should therefore also prove interesting to improve the static analysis approach and see which effects this has. Possible improvements include alias analysis and flow-dependency. Furthermore, some of the related approaches feed back information from dynamic symbolic execution to the skeleton generation approach. A possible future direction would be to fully integrate skeleton generation with dynamic symbolic execution in similar ways as those related approaches do it, and then investigate if invariant-breaking patterns can improve the test quality further. Alternatively, it may also be possible to replace the fall-back template by one (or multiple) of those related approaches. After some of these improvements are in place, it may be possible to open up the scope of the approach from invariant testing to general testing. Since not all relied-upon invariants are actually specified formally in practice, it is likely that the same kind of invariant-breaking patterns would apply even to code that does not contain explicit invariant specifications.

Chapter 7

References

[1] B. Meyer. *Object-Oriented Software Construction, 2nd Edition*. Prentice-Hall, 1997.

[2] K. R. M. Leino and P. Müller. Object invariants in dynamic contexts. In *ECOOP*, volume 3086 of LNCS, pages 491–516. Springer, 2004.

[3] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed automated random testing. In *PLDI*, pages 213–223. ACM, 2005.

[4] K. Sen, D. Marinov, and G. Agha. CUTE: A concolic unit testing engine for C. In *ESEC*, pages 263–272. ACM, 2005.

[5] P. Müller, A. Poetzsch-Heffter, and G. T. Leavens. Modular invariants for layered object structures. *Science of Computer Programming*, 62:253–286, 2006.

[6] S. Drossopoulou, A. Francalanza, P. Müller, and A. J. Summers. A unified framework for verification techniques for object invariants. In *ECOOP*, volume 4963 of *LNCS*, pages 412–437. Springer, 2008.

[7] N. Tillmann and J. de Halleux. Pex—White box test generation for .NET. In *TAP*, volume 4966 of *LNCS*, pages 134–153. Springer, 2008.

[8] M. Fähndrich, M. Barnett, and F. Logozzo. Embedded contract languages. In *SAC*, pages 2103–2110. ACM, 2010.

[9] R. Madhavan, G. Ramalingam, and K. Vaswani. Modular heap analysis for higher-order programs. In *SAS*, volume 7460 of *LNCS*, pages 370–387. Springer, 2012

[10] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball. Feedback-directed random test generation. In *ICSE*, pages 75–84. IEEE Computer Society, 2007.

[11] P. Tonella. Evolutionary Testing of Classes. In *ISSTA*, pages 119–128. ACM, 2004.

[12] K. Inkumsah and T. Xie. Evacon: a framework for integrating evolutionary and concolic testing for object-oriented programs. In *ASE*, pages 425–428. ACM, 2007.

[13] S. Thummalapenta, T. Xie, N. Tillmann, J. de Halleux and Z. Su. Synthesizing Method Sequences for High-Coverage Testing. In *OOPSLA*, pages 189–206. ACM, 2011.

[14] T. Xie, D. Marinov, W. Schulte and D. Notkin. Symstra: a framework for generating object-oriented unit tests using symbolic execution. In *TACAS*, volume 3440 of *LNCS*, pages 365–381. Springer, 2005

[15] W. Visser, C. S. Păsăreanu, and S. Khurshid. Test input generation with Java PathFinder. In *ISSTA*, pages 97–107. ACM, 2004.

[16] Joxan Jaffar, Vijayaraghavan Murali and Jorge A. Navas. Boosting Concolic Testing via Interpolation. *Submitted/Under review*