Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science 253 (2009) 71–86
Electronic Notes in Theoretical Computer Science
www.elsevier.com/locate/entcs

Test Case Generation by Contract Mutation in Spec#

Willibald Krenn, Bernhard K. Aichernig

Institute for Software Technology
Graz University of Technology
Inffeldgasse 16b/2
A-8010 Graz, Austria

Abstract

Mutation testing is a well known fault-based testing technique that is normally used to assess the quality of a test suite. In this paper we use the mutation operation to derive test cases that demonstrate the absence of certain faults in an implementation: In difference to conventional mutation testing, which mutates program code, we mutate program contracts and generate test-input data that is able to distinguish the mutated contract from the original one.
We show how existing development tools can be used as a foundation for the presented methodology: In particular we rely on the counter-example generation capabilities of the Spec#/Boogie/Z3 system.

Keywords: test case generation, mutation testing, contract mutation, Spec#, Boogie, Z3

1 Introduction

Recent developments (e.g., [6]) within the software market indicate that contract-based specifications finally arrive at the normal developer's desk. This is very fortunate, as contracts over functions are an excellent starting point for all sorts of program verification techniques. Key to these techniques is the availability of an oracle that allows to automatically reason whether code is correct. Contracts, expressing a partial relation between pre- and post-states of functions, provide this capability in a very natural way. Hence, it comes as no surprise that there are many verification tools depending on contracts.

Based on C#, Microsoft has built the Spec# [5] system. Spec# extends C# with preconditions, postconditions, invariants, and a notion of ownership. In combination with Boogie, a verification condition generator that uses Common Intermediate Language (CIL, part of [14]) code as input, and Z3 [12] as SMT-solver, capable of producing error models, a framework for automatically verifying code has been published. From contracts and implementation code, Spec#/Boogie is able to generate

doi:10.1016/j.entcs.2009.09.052

Fig. 1. The triangle specification, taken from [2]

```
context TriangleType(j:int,k:int,l:int):String
pre:  j>=1 and k>=1 and l>=1 and
      j<(k+l) and k<(j+l) and l<(j+k)
post: if ((j=k) and (k=l)) then result = "equilateral"
      else if ((j=k) or (j=l) or (k=l)) then result = "isosceles"
      else result = "scalene" endif
      endif
```

Fig. 2. Mutated triangle specification, taken from [2]

```
context TriangleType(j:int,k:int,l:int):String
pre:  j>=1 and k>=1 and l>=1 and
      j<(k+l) and k<(j+l) and l<(j+k)
post: if ((j=k) and (k=1)) then result = "equilateral"
      else if ((j=k) or (j=l) or (k=l)) then result = "isosceles"
      else result = "scalene" endif
      endif
```

counter-examples when, e.g., a postcondition can not be proved. (Due to the workings of Boogie the reported example may be spurious.) One obvious advantage of having a counter-example is that it can be shown to the developer, so he/she understands why the contract can be broken. This approach has been pursued by the author of [7] and resulted in a tool that enables counter-example execution.

This paper focuses on another usage of counter-examples: Based on a variant of mutation based testing, we use reported counter-examples to construct test cases. In particular, we start with the contracts and mutate them according to our theory. From the mutated and the original contract we then construct test data that can distinguish between an implementation that adheres to the mutated contract instead of the correct one. So the main idea of the approach is to construct test cases that prevent a developer from implementing the "wrong" contract.

Figure 1 shows a simple specification that has been mutated in Figure 2. Note that the postcondition is slightly altered ($k = 1$ instead of $k = l$). This particular mutation, taken from [1], was motivated by the fact that DNF based test cases can miss the fault, as demonstrated in [1]. Our aim is to use the counter-example capabilities of the Spec# system to generate the relevant test case that distinguishes between the mutant and the correct specification. Since we are not interested in the implementation for the purpose of test case generation, the presented approach is black-box.

Before going into further detail, we briefly cover related testing techniques that work with contracts on the .NET platform. For a more detailed discussion of related research, see Section 6.

The PEX [25] tool can also be used to generate a test case for discovering a contract violation. The difference to our approach is that we do not need an implementation to construct test cases, and that we use the SMT solver to compute a specific counter-example for a particular mutated contract. PEX, in contrast, finds counter-examples by exploring all code-paths within a given implementation.

The presented approach also relates to model programs, as used in the SpecExplorer [26] tool. The idea of a model program is to make a specification executable and works similar to an event based system (e.g., Event-B [20]) where each action (function) has a guard and an update statement. Within model programs, the update statement (i.e. the post-condition of the event) is the method body, whereas the guard is built by a slight abuse of the notion of a precondition. (Strictly speaking, a violation of a precondition does not forbid calling the method.) Using these guards and the postconditions in the method body, SpecExplorer does a state-space exploration to discover the transition system. Note that these model programs lack a contract-like-specified postcondition.

Our approach also could be applied to these model programs, as we can "abuse" Boogie to construct a counter-example from the mutated method's code and an inserted, calculated post-condition. Having said that, it is difficult to automatically derive a postcondition from arbitrary complex method code.

This paper is organized as follows. In Section 2 we repeat the most important definitions of the underlying theory and present a formula for mutation-based test case generation. In Section 3 we introduce Spec# and Boogie and show how to link the refinement check of Boogie with the test-case generation formula presented before. Next, we discuss the proposed methodology, give an example in Section 4 and evaluate difficulties arising in more complex environments in Section 5. Before concluding, we present related research.

2 Preliminaries

We adopt the notion of [1,2] in that a test case is an abstraction of the system specification. Within fault-based testing, one tries to cover anticipated faults, hence we are searching for test cases that can distinguish between the correct and an incorrect specification. We adopt the standard notion that faults are results of errors (bugs), while a failure is a wrong behavior caused by a fault.

In order to generate faulty specifications from the correct one, we rely on mutation operations, similar to the work presented in [18]. The underlying assumption is that the developer will create an almost correct implementation and that errors will be minor deviations from the correct specification. Thus, by slightly mutating the correct specification and calculating test cases to distinguish the two versions, we cover most of the errors.

In order to be self contained, we repeat the most important definitions from [1,2]. In our theory, that is based on UTP (Unifying Theories of Programming), we reason about programs in terms of predicates. Program variables, conceptual variables representing a system's state, and observable input-output streams are represented by free variables in those predicates. The set of names of free variables builds the alphabet. We are only interested in a particular form of predicates, called *designs* in UTP, that represent a pre-postcondition specification ($precondition \Rightarrow postcondition$). We express test cases, programs, and specifications as designs, in other words, pre-postcondition specifications.

Definition 2.1 Let T be a set of test cases, S a specification and I an implementation, and

$$T \sqsubseteq S \sqsubseteq I$$

we define

(i) T as a correct test set with respect to S,

(ii) all test cases in T as correct test cases with respect to S,

(iii) implementation I passes the test cases in T,

(iv) implementation I conforms to specification S.

Note that $S \sqsubseteq I$ has the standard notion of I being a refinement of S (S is refined by I) [22] and that given the alphabet of T, S, and I we can use implication and universal quantification over all variables in the alphabet to write the refinement as $[T \Leftarrow S]$ and $[S \Leftarrow I]$. (Similar to [13] we are using square brackets to denote universal quantification over all variables in the alphabet.)

Theorem 2.2 *Under the observation of program start (ok) and termination (ok′), and given preconditions S_{pre}, I_{pre}, and postconditions S_{post}, I_{post}, we can define the refinement of S by I ($[I \Rightarrow S]$) as*

$$[((ok \wedge I_{pre}) \Rightarrow (ok' \wedge I_{post})) \Rightarrow ((ok \wedge S_{pre}) \Rightarrow (ok' \wedge S_{post}))] \textbf{ iff}$$
$$[S_{pre} \Rightarrow I_{pre}] \text{ and } [(S_{pre} \wedge I_{post}) \Rightarrow S_{post}]$$

It follows from the above that I is *stronger*, since it has the more liberal precondition and the stronger postcondition: Commonly this property is known as that under refinement preconditions are weakened and postconditions strengthened.

Theorem 2.3 (Fault Discriminating Test Case) *Let t be an input-output test case. Furthermore, given a (input-output) specification S and a faulty version S^m where $S \not\sqsubseteq S^m$. Then there is a discriminating test case t, such that*

$$(t \sqsubseteq S) \wedge (t \not\sqsubseteq S^m)$$

Given that $S \sqsubseteq I$ and $S^m \sqsubseteq I^m$ and $S \not\sqsubseteq I^m$, the following property then holds automatically

$$(t \sqsubseteq I) \wedge (t \not\sqsubseteq I^m)$$

In other words, the discriminating test case will distinguish a faulty implementation that is a refinement of the faulty specification S^m from an implementation refining the non-faulty specification S.

For a proof of this theorem we refer the interested reader to [1]. Note that $S^m \sqsubseteq S \sqsubseteq I^m$ is possible: Differently put, an implementation of a mutated specification can be a refinement of the specification S and, hence, "repair" the mutation. This is the reason for demanding $S \not\sqsubseteq I^m$ in the theorem.

Taking everything together, we are looking for test input data that fulfills

$$(S_{pre} \wedge S^m_{post} \wedge \neg S_{post}) \vee (\neg S^m_{pre} \wedge S_{pre} \wedge S_{post})$$

and we expect output according to the solution of $S_{pre} \wedge S_{post}$ for the found input data. (For more details, we refer the interested reader to [1].)

In the next section we discuss Boogie and the properties we rely on in order to generate counter-examples that give us the necessary information for constructing test cases.

3 Spec#, Boogie

Boogie is a static program verifier that relies on the SMT-solver Z3 [12] in order to discharge the proof obligations. Boogie defines its own input language (Boogie), so different front-end systems can use Boogie for automated reasoning. Spec# is an extension to C# and adds features for contract based program verification. Internally, Spec# relies on Boogie.

Static verification of methods in Spec# is done method-wise where the tool distinguishes between an internal and an external sight of each method. In particular, any method call is replaced by a check of the precondition of the to-be-called method and the output values are set according to given postconditions. Due to possible underspecification of callees, Spec#/Boogie might come up with spurious counter examples. Loops are also treated specially as Boogie analyzes a single loop iteration and then randomly chooses ("havocs") values according to the invariants. Thereafter, Boogie executes the loop body and checks that the invariants hold. Following this approach, Boogie does not need to unroll loops, as the havoc stands for arbitrary iterations of the loop.

Boogie relies on weakest-precondition calculation for verification condition generation. Before doing the weakest-precondition calculation, Boogie transforms the input program: First it desugars commands like procedure calls, Boogie then converts the method into a DAG (directed acyclic graph), adds a common, unified exit block, inserts pre- and postconditions, converts it to passive commands (code without state changes), does an optimization run, and finally generates the first order formulae (using precondition calculation) that are put forward to the Z3 solver.

The following description is taken from [3]. For each block within the passive program, Boogie introduces a boolean variable that is true if every execution starting from A is correct. So for a block

$$\text{A: } \mathit{PassiveCommands};\ \textbf{goto}\ \mathit{B,C};$$

the block equation "*BlockEq*" that defines A_{ok} is:

$$A_{ok} \Leftarrow \mathrm{wp}(\mathit{PassiveCommands},\ B_{ok} \wedge C_{ok})$$

where *wp* is the weakest precondition of *PassiveCommands* with respect to the postcondition $B_{ok} \wedge C_{ok}$. Together with the axioms of the Boogie program, the verification condition then is:

$$\mathit{Axioms} \wedge \mathit{BlockEqs} \Rightarrow \mathit{Start}_{ok}$$

Listing 1: Refinement in Spec#

```
public static void Refinement(int a, out int result)
  requires a > 0;
  ensures result > 1;
{
  result = a + 2;
}

public static void Original(int a, out int result)
  requires a > 1;
  ensures result > 0;
{
  Refinement(a, out result);
}
```

where *Axioms* is the conjunction of the axioms in the Boogie program, *BlockEqs* is the conjunction of the block equations, and *Start* is the implementation's start block. For more details regarding Spec# and Boogie we refer the interested reader to [3,5,7].

3.1 Linking Refinement with Mutations and Test Case Generation

Listing 1 shows a program demonstrating that the theory of refinement also is fundamental to Spec#/Boogie. The example passes static verification without error. It is important to know that when verifying *Original* Boogie *replaces* the call to *Refinement* by the pre- and postconditions of *Refinement*. In other words, Boogie does not look at the implementation part of the method *Refinement*.

Also, observe that the precondition of the method *Refinement* has been weakened, while the postcondition has been strengthened with respect to the conditions given in the method *Original*, hence $Original \sqsubseteq Refinement$ is valid. Furthermore, if one looks at the body of *Refinement*, one can observe that the code again is a refinement of the contracts given. This is because the result will always be greater than two, which is a stronger condition than the one given in the postcondition.

As mentioned in the last section, we are interested in a test input that fulfills $(S_{pre} \wedge S^m_{post} \wedge \neg S_{post}) \vee (\neg S^m_{pre} \wedge S_{pre} \wedge S_{post})$ when $S \not\sqsubseteq S^m$.

The ordering $S \not\sqsubseteq S^m$ is important: It says that the mutant must not be a valid refinement of S. Turning it upside-down to $S^m \not\sqsubseteq S$ means that the mutant could be a valid refinement of S. Generating mutants that are valid refinements is covered in [18] and used to asses the quality of a given test suite. The idea there is to use the mutants aside the original contract on some fixed test suite to see how well the test inputs cover the original contract. Mutations are created by, e.g., precondition weakening and postcondition strengthening.

We give an example to further clarify the difference. Suppose the contract of method *Refinement* in Listing 1 resembles S^m and S stands for the contract of method *Original*. When only looking at the postcondition, we can argue that *Original* does not refine *Refinement* (since it is the other way), so $S^m \not\sqsubseteq S$ holds. One test case, distinguishing the two, would be to generate a function-result of one. However, according to the specification given by S this result is perfectly valid. Hence, the value of such a test case remains limited with respect to uncovering implementations that do *not* adhere to the correct contract. It might, however, be used to create valid input data for the normal testing process. Since we want to detect contract violations, mutants based on $S \sqsubseteq S^m$ are of no interest to us. If, however, we happen to create such a mutant, it won't harm either. Note that by using proper mutation operators, e.g., precondition strengthening and postcondition weakening, we are able to generate mutants with the property of $S \not\sqsubseteq S^m$.

Within the given formula above, the second disjunct $(\neg S^m_{pre} \wedge S_{pre} \wedge S_{post})$ describes the case when we want to generate test cases for mutated preconditions. The meaning is quite simple, as we are searching for a test case that satisfies the original specification but fails at the mutated precondition. This means that the mutant does not fulfill the refinement property $S_{pre} \Rightarrow S^m_{pre}$ and, hence, would be an improper implementation of the given specification.

The first disjunct $(S_{pre} \wedge S^m_{post} \wedge \neg S_{post})$ deals with mutations in postconditions: We are searching for a test case that fulfills the precondition and the mutated postcondition but not the original postcondition. This is, again, a violation of the refinement relation: $\neg((S_{pre} \wedge S^m_{post}) \Rightarrow S_{post}) \equiv (S_{pre} \wedge S^m_{post} \wedge \neg S_{post})$

Translated to Spec# and to the example of Listing 1 this means that we have to insert the mutated pre- and postconditions at *Refinement*. We then employ Boogie to do a refinement check for us: If the mutant is a refinement, then Boogie will not report a contract violation. Otherwise, Boogie will generate a counter-example that gives us the information we are searching for, a discriminating test case.

We now present our methodology for contract-mutation based test-case generation with Spec#.

3.2 Methodology

The Spec#/Boogie system can be used for automated test case generation of mutated contracts as follows:

(i) Let the correct (original, not mutated) contract be C^{ok}, the mutated version C^m, the precondition of a contract C_{pre}, and the postcondition C_{post}. Also, C^{ok} needs to pass the static verifier without any error.

(ii) Create a method O with C^{ok}.

(iii) Create a method M without any contract.

(iv) If a postcondition was mutated, add C^m_{post} to M. Otherwise add C^{ok}_{post}.

(v) If a precondition was mutated, add C^m_{pre} to M. Otherwise add C^{ok}_{pre} (or say true).

(vi) Add the *[Verify(false)]* attribute to M that tells Spec# to not statically verify M. (Since M has an empty body, a static check of the contract will fail.)

(vii) Within O place a call to M as sole element of the method body.

(viii) If O has a non-void return type, assign an arbitrary, but valid, return value within the body of M and place a *return* statement before the call of M in the body of O.

Fig. 3. A more abstract, non-deterministic version of the postcondition of Figure 1.

```
context TriangleType(j:int,k:int,l:int):String
pre:  j>=1 and k>=1 and l>=1 and
      j<(k+l) and k<(j+l) and l<(j+k)
post: if ((j=k) or (j=l) or (k=l)) then
        ((result = "isosceles") or (result = "equilateral"))
      else result = "scalene"
      endif
```

(ix) Call the static verifier:
- If one or more counter-examples can be found, create test cases.
- If no counter-example can be found, $C^{ok} \sqsubseteq C^m$ holds and the mutation generates valid behavior. So no test is needed.

Boogie reports several counter-examples when more than one pre/postcondition does not hold. This might be, e.g., due to non-deterministic behavior of C^m that potentially violates several conditions, or in the case both, pre- and postcondition, were mutated. Adding mutated pre- and postconditions at the same time, however, is not recommended, as a mutated precondition may hide contract violations of the postcondition.

Non-determinism can be expressed to some degree within the contract: Given the postcondition of Figure 1, a non-deterministic (more abstract) mutant can be seen in Figure 3. In the following we assume that the implementation of the contract shown in Figure 3 itself is deterministic. Then we need two test cases to be able to distinguish the mutant from the original specification: One test case with $j = k = l$, and one for, e.g., $(j \neq k) \wedge (k = l)$. This implies that we need to be provided with two counter-examples. However, because Z3 is a SAT solver, it will only report one error model per formula even if several different models exist, as can be the case in non-determinism. This can, to some degree, be moderated by splitting combined conditions, e.g., lengthy If-Then-Else constructs, into separate conditions. In this case Boogie will report a separate counter-example for each failing condition instead of reporting one counter-example for the combined condition. In general, however, non determinism is a tricky issue.

We discuss test-case generation from contract mutation of the triangle example (see Figures 1 and 2) in the next section.

4 Example

Listing 2 shows a Spec# translation of the triangle example introduced in Figures 1 and 2. As described in the last section, the contract of method *TriangleType_O* is a direct copy of the specification as shown in Figure 1. The contract of *Triangle-Type_M*, on the other hand, represents the mutated specification, as can verified by looking at the condition for equilateral triangles: The original specification treats triangles with the property of $j = k = l$ as equilateral. The mutated contract treats triangles with the property $j = k = 1$ as equilateral. Variables j, k, l encode the lengths of the sides of a triangle.

For demonstration purposes we have also included an out-commented, mutated

Listing 2: The familiar triangle-example in Spec#

```
using System;
using Microsoft.Contracts;

public class Program
{
  public enum TriangleEnum {Scalene, Isosceles, Equilateral};


  // This method implements the original contract.
  public static TriangleEnum TriangleType_O(int j, int k, int l)
    // unmodified precondition
    requires (j >= 1) && (k >= 1) && (l >= 1) && (j < (k+l)) && (k < (j+l))
              && (l < (j+k));
    // unmodified postcondition
    ensures  ((j == k) && (k == l)) ? (result == TriangleEnum.Equilateral) :
             (((j == k) || (j == l) || (k == l)) ? (result ==
               TriangleEnum.Isosceles):(result == TriangleEnum.Scalene));
  {
    // constrain the result so as to lie within incorrect "bounds"
    return TriangleType_M(j,k,l);

    /* if we have a real implementation here, we could use a mutated
     version of the implementation!

     However, as we are not working on model programs this time, we
     ignore the implementation.

      TriangleEnum res;
      if ((j == k) && (k ==  1)) res = TriangleEnum.Equilateral;
      else if ((j == k) || (j == l) || (k == l)) res = TriangleEnum.Isosceles;
      else res = TriangleEnum.Scalene;
      return res;
    */
  }


  // We use this method for the sole purpose of constructing the counter
  // example, therefore we are not interested in the implementation and
  // say:
  [Verify(false)]
  public static TriangleEnum TriangleType_M(int j, int k, int l)
    // copy original precondition (not necessary)
    requires (j >= 1) && (k >= 1) && (l >= 1) && (j < (k+l)) && (k < (j+l))
              && (l < (j+k));

    /* If the precondition is to be mutated, then it has to be
       included here instead of the original precondition:
       requires ((j >= 1) && (k >= 1) && (l < 1) && (j < (k+l)) && (k < (j+l))
              && (l < (j+k)));
    */

    // mutate postcondition
    ensures  ((j == k) && (k == 1)) ? (result == TriangleEnum.Equilateral) :
             (((j == k) || (j == l) || (k == l)) ? (result ==
               TriangleEnum.Isosceles):(result == TriangleEnum.Scalene));
  {
    // This body is ignored when checking TriangleType_O!
    return TriangleEnum.Equilateral; // keep compiler from complaining..
  }


  static void Main(string![]! args)
  {
  }
}
```

precondition, as can be seen in line 48. The mutated precondition says that variable l must be smaller than one. When uncommented, Boogie will report a counter-example ($j = k = l = 1$) for the precondition. As already mentioned in the last section, we only mutate either pre- or postcondition – never both at the same time. In the remainder of this section, we look at the mutated postcondition.

In order to understand the following counter-example, it is useful to know that *TriangleEnum.Scalene* is represented by the value 0, *TriangleEnum.Isosceles* by 1, and *TriangleEnum.Equilateral* by 2.

Since we can argue after manual inspection of the postcondition that the contract of *TriangleType_O* is not refined ($\not\sqsubseteq$) by the contract of *TriangleType_M* and because we call *TriangleType_M* within the body of *TriangleType_O*, we expect Boogie to calculate a counter-example. Indeed, after static verification, Boogie presents us a counter-example. Within the following excerpt of the value-part of the counter-example, the first column gives a partition number, the second column the value of the partition, and the last column lists all variables that have assigned the value of the partition.

```
[PartitionID]: [Value] {[Variables]}
  ...
  *68: 1  {call7885formal@$result@0}
  ...
  *73: 2  {k$in j$in l$in}
  ...
```

The presented snippet is only an excerpt of the rather lengthy example delivered by Boogie but it contains the relevant information: We can see that the input values of j, k, and l were determined to be equal to 2. Relating *call7885formal@$result@0* to the call of *TriangleType_M* needs a little more work, but can be done.

We first look at the Boogie program that has been built from the Spec# program to determine the block number containing the call to *TriangleType_M*:

```
block2397:
  assume true;
  // ----- nop
  // ----- copy  ----- Program.ssc(19,3)
  stack0i := j;
  // ----- copy  ----- Program.ssc(19,3)
  stack1i := k;
  // ----- copy  ----- Program.ssc(19,3)
  stack2i := l;
  // ----- call  ----- Program.ssc(19,3)
  call return.value := Program.TriangleType_M$System.
    Int32$System.Int32$System.Int32(stack0i, stack1i, stack2i);
  // ----- branch
  assume true;
  goto block2091;
```

We can see that within *block2397* the method gets ”called”. Remember, that Boogie will replace the statement with the contract of *TriangleType_M*. Boogie's */traceverify* option allows us to look at the transformation process. In particular, Boogie prints out converted versions of the method under investigation. From the output of the passive form (see [3,19,4]), we can observe the name of the variable where the “return” value of the “method call” is being saved. The following is an excerpt, most of the additional assumptions and assertions have been omitted.

```
block2397:
  ...
  assume InRange(call7885formal@$result, Program.TriangleEnum);
  assert j$in >= 1;
  assert k$in >= 1;
  assert l$in >= 1;
  assert j$in < k$in + l$in;
  assert k$in < j$in + l$in;
  assert l$in < j$in + k$in;
  assume IsHeap($Heap@0);
  assume InRange(call7885formal@$result@0, Program.TriangleEnum);
  assume cast($IfThenElse(j$in == k$in && k$in == 1, call7885formal@$result@0 == 2,
         cast($IfThenElse(j$in == k$in || j$in == l$in || k$in == l$in,
           call7885formal@$result@0 == 1, call7885formal@$result@0 == 0),bool)),bool);
  ...
  assume $HeapSucc($Heap, $Heap@0);
  ...
  assume true;
  goto block2091;
```

Finally, we can confirm that *call7885formal@$result@0* stores the result of the "call" to *TriangleType_M*. In addition, we can see that Boogie switches the heap generation before evaluating the postcondition of *TriangleType_M*.

From the counter-example we now have observed the input values, and the return value of the mutated method. We still lack the expected return value according to the specification. Unfortunately, we can not observe this value directly from the error-model provided. This is because the postcondition is checked within following statement:

```
ReallyLastGeneratedExit:
  ...
  assert cast($IfThenElse(j$in == k$in && k$in == l$in, call7885formal@$result@0 == 2,
         cast($IfThenElse(j$in == k$in || j$in == l$in || k$in == l$in,
         call7885formal@$result@0 == 1, call7885formal@$result@0 == 0),bool)),bool);
  return;
```

Since *IfThenElse* is treated as function, the above statement requires us to look at the function interpretations of the error-model, which is tedious task. However, within the given counter example, we can observe following interpretation that could not be evaluated to *true*:

```
$IfThenElse(@true, anyEqual(1, 2), @true) = anyEqual(1, 2)
```

Since 1 is not equal to 2, that is the correct result, the proof fails. This completes the discussion of the reported counter-example.

For this simple example, we have determined the input values of j, k, and l. In addition, Boogie has us also supplied with the output of the mutant which suffices to detect the mutant. Recovering the result of the original specification requires more work. However, since we can use a compiled version of the original postcondition as oracle at runtime, we do not depend on the exact value of the original specification's result.

5 Discussion

We understand that the presented example is a rather simple one. It was mainly given for motivational reasons. In this section, we discuss a more advanced exam-

Listing 3: Example operating on objects (calculator)

```
static void Add_O (MyStack Input)
  requires (Input != null) && (Input.Count >= 2);
  ensures  (Input.Count == old(Input.Count) - 1);
  ensures   (Input.Peek() == old(Input.Peek()) + old(Input.Peek2()));
  modifies Input.*;
{
  Add_M(Input);
}

// mutated contract
[Verify(false)]
static void Add_M (MyStack Input)
  requires (Input != null) && (Input.Count >= 2);
  ensures  (Input.Count == old(Input.Count) - 1);
  ensures  (Input.Peek() == old(Input.Peek()) - old(Input.Peek2()));
  modifies Input.*;
{
}
```

ple and highlight arising issues for test-case generation. Finally, we also mention limitations that are inherent to the approach and evaluate their impact on the methodology.

One of the main features of Spec# is that it is able to work with all data types offered by C#. This includes reference types, value types, as well as generic types. Spec# also comes with annotations for some parts of the .NET core library. Because Microsoft seems serious about bringing contracts to the every-day-developer, it is reasonable to expect even better contract coverage of core-libraries in future. Taking all this together makes Spec# a promising target for automated test case generation. However, the same reasons also make test case generation a complex task, which is also reflected to some part within the reported counter-example.

We give the example of an *add* method of a stack-based calculator, as seen in Listing 3. Since we now left the world of "ints" and work on objects, we need to call functions within the contract. The requirement for functions (methods) that can be called from within contracts is the absence of any side-effects: They are not allowed to alter the state. Methods fulfilling this requirement may be marked with the *Pure* attribute. Within the counter-example, pure methods are represented by uninterpreted functions [11], so as soon as we want to have the capability to treat objects for test-case generation, we have to evaluate uninterpreted functions within the error-model.

For the given example in Listing 3, a counter-example looks like the following:

```
function interpretations:
...
#MyStack.get_Count($Heap, Input$in) = 2
#MyStack.get_Count($Heap@0, Input$in) = 1

#MyStack.Peek($Heap, Input$in) = 0
#MyStack.Peek($Heap@0, Input$in) = -1

#MyStack.Peek2($Heap, Input$in) = 1
```

At different times within the computation process, these functions will return different values. This is reflected by different heap generations in the counter-example: *$Heap* means the initial generation that is followed by *$Heap@0*. (The successorship is encoded elsewhere in the counter-example.)

In order to generate a test case that reproduces this behavior, we have to "mock" the MyStack object. As we do not know what methods an eventual implementation will call on the object, we need to reconstruct the object state as good as possible.

Under the assumption that the methods of the class MyStack also have contracts assigned, we can search for a sequence of method calls ("actions") to get from an initial state to the one described by the counter-example. This is very similar to the classical AI planning problem [23], where an initial state, a goal state, and a set of actions are given. A solution to this problem is a sequence of actions that transforms the initial state to the goal state.

Besides these object-creation issues, there are more fundamental ones. For test-case generation we rely on the solving capabilities of Z3: If the solver can not prove a given verification condition, even if it is true, we will try and create a test case. However, given that we start from a proved specification and also have control over the mutation operations being performed, this case seems controllable.

Another issue for test-case generation is that of bugs in error models that might occur due to implementation errors. We can not do much about this, other than adding some validation step for created test cases.

After discussing the presented approach, we set it in context to related research in the next section.

6 Related Research

Closest to our work is that of Billeter [7]. Starting from a Boogie generated counter-example, that was derived from an implementation possibly not adhering to it's contract, Billeter presents an approach for making the counter-example executable within the Visual-Studio Debugger. In order to reach that goal, he rewrites the CIL-byte-code so that the method under investigation shows exactly the behavior as specified in the counter-example. Since our presented work also is based on the Boogie-reported counter-example, some of the same limitations as reported in [7] apply: According to Billeter the counter-example contains incorrect values when comprehensions (sum, product, min, max, count) are used. Also, as both approaches depend on the ability of Z3 as solver, both approaches fail if Z3 is not able to prove a certain verification condition. In our approach, we would generate unnecessary test cases, while Billeter is not able to reproduce a failing state. While both approaches rely on the counter-example, the focus is very different: We intend to create test cases that are able to detect whether a certain mutated specification has been implemented, while Billeter makes the counter-example executable within the debugger.

Check 'n' Crash [10] (and the successor DND-Crasher) uses a combination of static program analysis and dynamic test cases to test Java programs. These tests uncover a set of possible failures, such as (among others) division by zero, dereferencing null, and accessing arrays outside of their domains. Check 'n' Crash first analyzes Java programs for possible errors with ESC/Java. If a counter-example is found, a constraint solver generates a test case that is then executed in order to see whether the found counter-example was spurious or not. While Check 'n' Crash also uses counter-examples for test case generation, the aim differs from that of the presented approach, as we create test cases to uncover an implementation that refines the wrong contract. Check 'n' Crash uses tests to filter out spurious counter-examples.

Based on Abstract Interpretation [9], Ferrara, Logozzo, and Fähndrich [15] have developed the static checker Clousot. Abstract Interpretation is a theory of approximations: It uses more or less precise abstract semantics to reason about properties of programs. Clousot, similar to Spec#/Boogie, is intended to statically uncover contract violations and other errors from CIL code. It has a modular architecture and allows the refinement and composition of existing abstract domains in order to create new ones. In comparison to Spec#, Clousot promises faster analysis while also requiring less help from the programmer: As stated in [15], the tool rarely fails to infer loop invariants to validate memory access. Since Clousot does not provide a counter-example in the fashion Boogie does, it can not be used to generate test cases for mutated specifications.

For contract representation, Clousot depends on FoxTrot, a CIL based annotation language for .NET. It is possible to runtime-check FoxTrot specifications: Whenever a pre- or postcondition does not hold, an exception is thrown. In combination with PEX [25] that does a path exploration of methods, this approach can be used to automatically uncover bugs. In particular PEX uses the .NET profiling API to inspect and rewrite the CIL instructions of a method prior to just-in-time compilation. The instrumented code then drives a "shadow interpreter" that constructs and maintains symbolic representations and records conditions over which the program branches. Because the postcondition is included within the method code, PEX also explores all paths within the oracle, uncovering any faults within the method body. During exploration, PEX determines all equivalence classes of the method under investigation (including the contract). It is possible to apply the presented approach with modifications to PEX: Instead of specifying the mutated postcondition within the ensures block of the inserted method, we need to make the postcondition executable and insert it as method body. Using the *Choose* function provided by PEX can thereby help modeling non-determinism. We have tested the PEX approach with the given stack-calculator, as well as the given triangle example. In both cases PEX is able to deliver test input that violates the original postcondition. However, the stack-calculator required some explicit, manually defined factory method that allows PEX to construct integer stacks with a depth of two. If such a factory method is not given, PEX can not find a test case that passes the precondition, hence it will not find a test case for distinguishing original from mutation. This means that in order for PEX to find the discriminating test case, we need to provide a constructor that allows the creation of a discriminating object state in the first place. When this is not the case, an error will be missed. Boogie, on the other hand, tells us exactly what object state we are looking for.

In contrast to concrete and symbolic (concolic) execution frameworks, such as DART [16], Cute [24], EXE [8], and PEX, that rely on code instrumentation, random testing, as, e.g., proposed in [21] can be used to generate test cases from pre- and postconditions. This work has been integrated into the latest version of the Eiffel [17] development environment. Goal is to provide the developer immediate feedback if an implementation does not adhere to the given contract. Failing test cases are stored and re-run during the next compilation cycles. Since we can modify our approach to fit the PEX methodology, we could also use random testing in order to generate discriminating test inputs. Because random testing only guarantees to find specific cases with a certain probability we did not further evaluate this combination.

7 Conclusion

Based on the theory of refinement and by leveraging the capabilities of the Spec# system, we present a methodology for automatically generating tests that can distinguish whether an implementation refines a faulty specification. The presented approach can be automatized and depends on calls to the underlying SAT solver at test-generation time. Any subsequent run of the test-suite does not incur the time-cost of calling the solver. Since the approach is goal oriented in that per erroneous specification only one test-case (which might be tree-like, if non determinism is present) has to be generated, the number of generated tests is directly dependent on the number of faults that should be covered. In addition, for each test-case it is known for which error the test was designed. An inherent property of the approach is that equivalent mutants are ignored, meaning that no test cases are generated for them. We have given an example and discussed limitations and expectations bound to the approach. We also set the approach in context of recent developments within the formal methods community. Currently, work is underway to build a mutator for Spec# and add the missing counter-example instantiation capabilities.

Acknowledgement

Research herein was funded by the EU project ICT-216679, Model-based Generation of Tests for Dependable Embedded Systems (MOGENTES).

References

[1] Aichernig, B. K. and J. He, *Mutation testing in UTP*, Formal Aspects of Computing (2008).

[2] Aichernig, B. K. and P. A. P. Salas, *Test case generation by OCL mutation and constraint solving*, in: K.-Y. Cai and A. Ohnishi, editors, *QSIC 2OO5, Fifth International Conference on Quality Software, Melbourne, Australia, September 19-21, 2005* (2005), pp. 64–71.

[3] Barnett, M., B.-Y. E. Chang, R. DeLine, B. Jacobs and K. R. M. Leino, *Boogie: A modular reusable verifier for object-oriented programs*, in: *In FMCO 2005, volume 4111 of LNCS* (2006), pp. 364–387.

[4] Barnett, M. and K. R. M. Leino, *Weakest-precondition of unstructured programs*, SIGSOFT Softw. Eng. Notes **31** (2006), pp. 82–87.

[5] Barnett, M., K. R. M. Leino and W. Schulte, *The Spec# programming system: An overview* (2004), pp. 49–69.

[6] Barnett, M. and N. Tillmann, *PDC: Contract checking and automated test generation with Pex* (2008).

[7] Billeter, J., "Counterexample Execution," Master's thesis, ETH (2008).

[8] Cadar, C., V. Ganesh, P. M. Pawlowski, D. L. Dill and D. R. Engler, *EXE: automatically generating inputs of death*, in: *ACM Conference on Computer and Communications Security*, 2006, pp. 322–335.

[9] Cousot, P. and R. Cousot, *Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints*, in: *POPL '77: Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages* (1977), pp. 238–252.

[10] Csallner, C. and Y. Smaragdakis, *Check 'n' Crash: Combining static checking and testing*, in: *Proc. 27th ACM/IEEE International Conference on Software Engineering (ICSE)* (2005), pp. 422–431.

[11] Darvas, Á. and K. R. M. Leino, *Practical reasoning about invocations and implementations of pure methods*, in: *FASE*, 2007, pp. 336–351.

[12] de Moura, L. M. and N. Bjørner, *Z3: An efficient SMT solver*, in: *TACAS*, 2008, pp. 337–340.

[13] Dijkstra, E. W. and C. S. Scholten, "Predicate calculus and program semantics," Springer-Verlag New York, Inc., New York, NY, USA, 1990.

[14] *Standard ECMA–335, common language infrastructure (CLI)* (2006).

[15] Ferrara, P., F. Logozzo and M. Fähndrich, *Safer unsafe code for .NET*, in: *OOPSLA '08: Proceedings of the 23rd ACM SIGPLAN conference on Object oriented programming systems languages and applications* (2008), pp. 329–346.

[16] Godefroid, P., N. Klarlund and K. Sen, *DART: directed automated random testing*, in: *PLDI*, 2005, pp. 213–223.

[17] ISO, *ISO/IEC 25436:2006 – Eiffel: Analysis, design and programming language* (2006).

[18] Jiang, Y., S.-S. Hou, J.-H. Shan, L. Zhang and B. Xie, *Contract-based mutation for testing components*, in: *Proceedings of the 21st IEEE International Conference on Software Maintenance* (2005), pp. 483–492.

[19] Leino, K. R. M., *This is Boogie 2* (2008), working draft.

[20] Métayer, C., J.-R. Abrial and L. Voisin, "Event-B Language," (2005).

[21] Meyer, B., I. Ciupa, A. Leitner and L. L. Liu, *Automatic testing of object-oriented software*, in: J. van Leeuwen, editor, *Proceedings of SOFSEM 2007 (Current Trends in Theory and Practice of Computer Science)*, Lecture Notes in Computer Science (2007).

[22] Morgan, C., "Programming from specifications," Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1990.

[23] Russell, S. and P. Norvig, "Artificial Intelligence: A Modern Approach (Second Edition)," Prentice Hall, 2003.

[24] Sen, K., D. Marinov and G. Agha, *CUTE: a concolic unit testing engine for c*, in: *ESEC/SIGSOFT FSE*, 2005, pp. 263–272.

[25] Vanoverberghe, D., N. Bjørner, J. de Halleux, W. Schulte and N. Tillmann, *Using dynamic symbolic execution to improve deductive verification*, in: K. Havelund, R. Majumdar and J. Palsberg, editors, *SPIN*, Lecture Notes in Computer Science **5156** (2008), pp. 9–25.

[26] Veanes, M., C. Campbell, W. Grieskamp, W. Schulte, N. Tillmann and L. Nachmanson, *Model-based testing of object-oriented reactive systems with Spec Explorer*, in: R. M. Hierons, J. P. Bowen and M. Harman, editors, *Formal Methods and Testing*, Lecture Notes in Computer Science **4949** (2008), pp. 39–76.