Using Robustness Testing to Handle Incomplete Verification Results When Combining Verification and Testing Techniques

Stefan Huster, Jonas Ströbele, Jürgen Ruf, Thomas Kropf, Wolfgang Rosenstiel

Using Robustness Testing to Handle Incomplete Verification Results when Combining Verification and Testing Techniques

Stefan Huster, Jonas Ströbele, Jürgen Ruf,
Thomas Kropf and Wolfgang Rosenstiel

University of Tübingen, Department of Computer Science,
Sand 14, 72076 Tübingen, Germany
{huster,stroebele,ruf,kropf,rosenstiel}@informatik.uni-tuebingen.de

Abstract. Modular verification and dynamic testing techniques are often combined to validate complex software systems. Formal verification is used to cover all input spaces and program paths. However, due to the high complexity of modern software systems, they might not achieve complete verification results. Dynamic testing techniques can easily be applied to any type of software. Current approaches use them to handle incomplete verification results by validating unverified sections. This way of combining verification and testing ignores the fact that tests can only be used to show the presence of errors, but not their absence. Undiscovered errors pose the risk to trigger further errors in vulnerable code sections. Vulnerable sections are modularly verified, but depend on the guarantees of the tested code. We include robustness testing to analyse the influence of undiscovered errors. The generated robustness tests simulate failed guarantees within the tested code. The triggered response to those simulated errors helps the developer in adding additional error handling code. This makes the system more robust against undiscovered errors and guards it against uncontrolled crashes and unexpected behaviour in case of software failures. In the second part of this paper, we introduce a reference-architecture to generate and apply robustness tests. This architecture has been applied to multiple case studies and helped to identify potential errors yet undiscovered by generated test cases.

Key words: Software verification, Robustness Testing, Test Vector Generation

1 Introduction

Modular verification and dynamic testing techniques are often combined to validate complex software systems. Verification techniques are used to guarantee that an implementation matches its formal specification. For object oriented programs (OOPs), the specification is often defined as a set of conditions such as pre- and post-conditions and invariants. Modular verification techniques [1] analyse this type of specification based on a generated set of proof obligations (also

known as verification goals). A proof obligation (POG) is similar to a Hoare-Triple $\{P\}\ S\ \{Q\}$ [2]. The POG contains a program segment (Hoare: S), a set of assumptions (Hoare: $\{P\}$), and a guarantee (Hoare: $\{Q\}$) [3]. While considering all assumptions, the verification framework has to verify whether each possible execution of the embedded program section fulfils the defined guarantee. Assumptions made by one proof obligation must be ensured by another one. Only the validity of all proof obligations implies the correctness of the entire software system. Especially OOP concepts such as inheritance and (recursive) aggregation cause an infinite number of feasible control flows and thereby a high level of complexity. Due to this complexity, formal verification techniques are rarely capable of achieving complete verification results.
We use Listing 1.1 as running example to introduce our methodology. The listed method cannot be verified using the verification framework Microsoft Code Contracts [1]. It is part of a program to solve the Cutting Stock problem [4]. This problem is about cutting standard-sized pieces of material into pieces of specified sizes. The listed method is used to add new cutting lengths (Cut) to the current cutting layout (Bar). It checks whether the available material length is long enough to add the given piece length. It analyses the summarised lengths of all added cuts plus the required minimum space between two cuts (line 14). The value of `usedLength` must always be smaller than the total material length. This is required by the invariant in line 7. Figure 1 shows how Code Contracts claims that this invariant is not guaranteed on exit. This illustrates how specifications remain unverified.

```
public bool AddCut(double cutLength)
{
    Contract.Requires(cutLength > 0);
    if ((Length - usedLength - (Cuts.Count * minspace)) < cutLength)
    {
        return false;
    }
    usedLength += cutLength;
    Cuts.Add(cutLength);
    return true;
}
```

CodeContracts: invariant unproven: usedLength <= Length

Fig. 1: Code Contracts marks unverified invariant

In such cases, where some proof obligations remain unverified, current approaches ensure the correctness of those proof obligations by exhaustive testing. This use of testing can be shown to have residual risks. As Dijkstra put it, program testing can only be used to show the presence of bugs, but not their absence. Program sections which require the correctness of tested guarantees remain vulnerable,

[1] https://www.microsoft.com/en-us/research/project/code-contracts/, Last visit June 2017

```
1 public class Bar {
2   public double Length, UsedLength, MinSpace;
3   public List<Double> Cuts;
4   [ContractInvariantMethod]
5   private void ObjectInvariant() {
6     Contract.Invariant(UsedLength >= 0);
7     Contract.Invariant(UsedLength <= Length);
8     Contract.Invariant(Cuts != null);
9   }
10  public Bar(double length, double minSpace) {[...]}
11  public bool AddCut(double cutLength) {
12    Contract.Requires(cutLength > 0);
13    double usedSpace = Cuts.Count * MinSpace;
14    if ((Length - UsedLength - usedSpace) < cutLength) {
15      return false;
16    }
17    UsedLength += cutLength;
18    Cuts.Add(cutLength);
19    return true;
20  }
21 }
```

Listing 1.1: Code Contracts: Unverified Invariant

```
1 public void TestAddCut() {
2   Bar bar = new Bar(5000, 5);
3   bool couldAdd = bar.AddCut(1500);
4   Assert.IsTrue(couldAdd && bar.UsedLength == 1500);
5   couldAdd = bar.AddCut(5500);
6   Assert.IsTrue(!couldAdd && bar.UsedLength == 1500);
7 }
```

Listing 1.2: Testing unverified method

because undiscovered errors regarding failed guarantees produce further failures. Let's come back to our running example. The unverified method in Listing 1.1 can be tested using the unit test listed in 1.2, achieving full branch, path and condition coverage. In view of testing, this test case covers all major coverage rates and the method can be seen as validated. However, we achieve the same testing results when replacing line 14 by `if ((Length - usedSpace)< cutLength)`. This would be a major bug, because this line ignores the used material length. This bug allows to add more cuts to the bar than available material space. This could be tested when executing line 4 of our test case multiple times in a row. This simple example illustrates how testing can achieve good coverage rates while missing important defects.

This paper introduces a new approach that uses robustness testing to analyse the influence of such undiscovered errors. We initialise invalid program states to simulate failed guarantees and inspect the corresponding behaviour of vulnerable program sections. Our goal is to support the developer in adding additional

error handling code on critical locations in order to secure vulnerable sections against potential failures.
The remainder of this paper is structured as follows: Section 2 describes related work and current tools. Section 3 describes how we use robustness testing to analyse the influence of undiscovered errors. Section 4 defines one reference implementation to generate the defined type of robustness tests. Section 5 presents our results in comparison to current tools. Section 6 concludes the paper and presents future work.

2 Related Work

Several methodologies and tools already exist which combine formal verification and dynamic testing.
Christakis et al. [5–7] present a methodology that combines verification and semantic testing. Different static verification models are used together to verify the software under test in a sound way. Assumptions made by one prover, e.g. regarding numerical overflows, are ensured by another. Unverified assumptions are subsequently tested. The symbolic testing is guided to cover specifically those properties that could not been verified.
Czech et al. [8] present a method to create residual programs based on failed proof obligations. They reduce the number of required test cases, by testing only those control flows that have not been verified.
Kanig et al. [9] present an approach that uses explicit assumptions to verify ADA programs. Unverified assumptions are tested by generated test suites.
Code Contracts [10], Pex [11] and Moles [12] is the current Microsoft tool chain for software verification and symbolic test case execution. Code Contracts can be used to verify C# programs and supports contracts such as pre- and postconditions. Pex and Moles have been integrated into Visual Studio 2015 under the names IntelliTest and Fakes. Moles/Fakes is used to isolate test cases and can replace any method with a delegate. Pex iteratively applies symbolic execution to create test vectors in order to cover all branches of the method under test. The Microsoft tool chain does not provide any standard methodology to combine both tools.
In summary, all mentioned approaches try to reduce the number of required test cases by testing only unverified control flows. They try to handle incomplete verification results by achieving high test coverage on the unverified software components. No mentioned approach handled the residual risk of tested source code on vulnerable code sections. Therefore, they mark code sections as formally correct, even when those sections may contain serious errors caused by failed guarantees in tested code.

3 Methodology

The presented methodology integrates into existing workflows combining formal verification and dynamic testing techniques. Figure 2 shows an abstract illus-

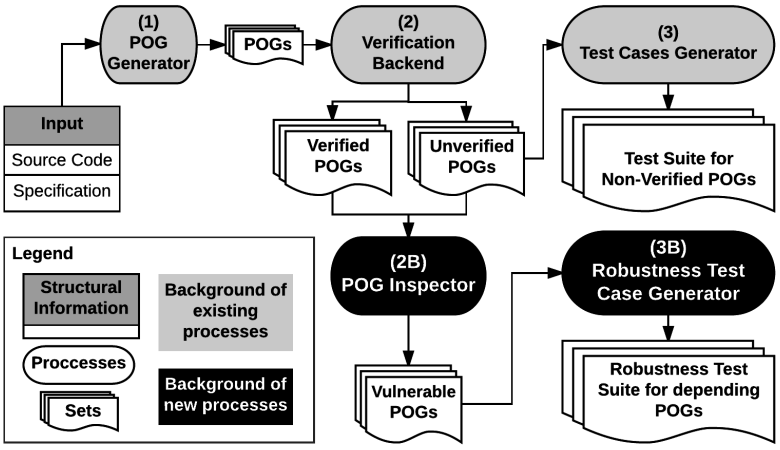

Fig. 2: Abstract workflow to combine formal verification and testing techniques

tration on how current approaches combine verification and testing. The input to those workflows is the program source code and its specification. The proof-obligation-generator analyses the source code and the specification to generate a set of proof obligations. In the second step, those proof obligations are verified by the verification backend. This step divides the set of proof obligations into a verified and unverified subset. In step 3, unverified proof obligations are further analysed by a test case generation framework. Those frameworks use symbolic execution (also known as concolic testing) to automatically create test cases and test vectors to explicitly cover control flows of unverified proof obligations.
The presented methodology adds two new steps to existing workflows. In step 2B, the POG inspector analyses the dependencies between verified and unverified proof obligations in order to identify vulnerable code sections. Step 3B generates robustness tests for those vulnerable POGs. These tests simulate errors within the tested code and uncover locations where additional error handling and sanity checks are required.
This section starts by defining proof obligations, then describes how to identify vulnerable code sections, illustrates their risks and finally explains how to generate corresponding robustness tests.

3.1 Proof Obligations

We start by defining the input to our methodology and corresponding symbols to refer to the different components of object oriented programs. The input to the proof obligation inspector is a set of verified and unverified proof obligations (POG). The POGs are generated based on the source code of an object-oriented program **Prog** and its specification set $\Gamma_{\mathbf{Prog}}$. This program contains classes $c \in \mathcal{C}_{\mathbf{Prog}}$. Each class can contain methods $m \in \mathcal{M}_c$ and fields $f \in \mathcal{F}_c$. A method

consists of an ordered set of statements $s \in \langle S \rangle_m$. The list of method parameters is $\vec{m}$. The specification can contain preconditions Γ_m^{pre}, postconditions Γ_m^{post}, and object invariants Γ_c^{inv}.

The proof of the overall correctness is divided into a generated set of proof obligations. Each proof obligation covers one control flow:

Definition 1 (Control Flow). *A control flow $\tilde{S} = \langle s_0, \ldots, s_n \rangle$ is a set of statements $s_i \in S_m$, $m \in \mathcal{M}_c$. Between each pair of statements s_i and s_{i+1} exists one unique transition.*

Definition 2 (Proof Obligation). *The set of all generated proof obligations is Π. A proof obligation $\pi = (\Omega, \tilde{S}, \phi)$ is a triple, combining a set of assumptions (Ω), a control flow ($\tilde{S}$), and a verification goal (ϕ). We refer to the method which contains the control flow $\tilde{S}$ by m_π. Assumptions and goals are represented as boolean predicates. A proof obligation is verified iff one can show that each execution of $\tilde{S}$ validates ϕ while assuming Ω. The predicate $\Psi(\pi)$ is true iff π can be verified. A proof obligation is always generated based on a specification $\gamma \in \Gamma$, we write $\Pi(\gamma) \to \pi$.*

Let's apply this to our running example. Here we can extract two work flows and POGs: $\tilde{s}_1 = \langle s_{12}, s_{13}, s_{14}, s_{15} \rangle$ and $\tilde{s}_2 = \langle s_{12}, s_{13}, s_{14}, s_{17}, s_{18}, s_{19} \rangle$. The indexes s_i mark the global line number of the corresponding statement. The unverified invariant $\phi_1 = (UsedLength <= Length)$ is covered by two proof obligations: $\Omega_1 = \{(cutLength > 0), (CutLength! = null)\}$ in $\pi_1 = (\Omega_1, \tilde{s}_1, \phi_1)$ and $\pi_2 = (\Omega_1, \tilde{s}_2, \phi_1)$.

3.2 Identifying Vulnerable Proof Obligations

The verification framework (Step 2 in Fig. 2) divides the set of POGs Π into a set of verified POGs $\Pi^+ = \{\pi \in \Pi | \Psi(\pi)\}$ and a set of unverified POGs $\Pi^- = \{\pi \in \Pi | \neg\Psi(\pi)\}$. Modular verification techniques build their correctness proof upon dependencies between different POGs. Those dependencies must be considered when testing unverified POGs. In step 3B of Fig. 2, we identify POGs depending on unverified code. We call them vulnerable proof obligations.

Definition 3 (Vulnerable Proof Obligations). *One proof obligation $\pi_i = (\Omega_i, \tilde{s}_i, \phi_i)$ depends on a different proof obligation $\pi_j = (\Omega_j, \tilde{s}_j, \phi_j)$ iff the assumption list Ω_i contains the verification goal ϕ_j:*

$$\pi_i \vdash \pi_j \Leftrightarrow \exists\, \omega \in \Omega_i | \omega \equiv \phi_j \tag{1}$$

One proof obligation π_i is vulnerable iff $\pi_i \vdash \pi_j \wedge \neg\Psi(\pi_j)$. The set of all vulnerable proof obligations is defined as $\Pi^? = \{\pi_i | \pi_i \vdash \pi_j \wedge \neg\Psi(\pi_j)\}$.

```
1 public List<Bar> CreateBars(List<double> cutLengths,
  Dictionary<double, int> materials) {
2   List<Bar> cuttingLayouts = new List<Bar>();
3   foreach(double cLen in cutLengths) {
4     bool couldAdd = false;
5     foreach(Bar bar in cuttingLayouts)  {
6       if(bar.AddCut(cLen)) { couldAdd = true; break; }
7     }
8     if(!couldAdd) {
9       double bfLength = Double.MaxValue;
10      foreach(double matLength in materials.Keys) {
11        double offcut = matLength - cLen;
12        if (offcut > 0 && offcut < bfLength - cLen &&
           materials[matLength] > 0) {
13          bfLength = matLength;
14        }
15    }
16    if(bfLength < Double.MaxValue) {
17      Bar newBar = new Bar(bfLength, 5);
18      newBar.AddCut(cLen);
19      cuttingLayouts.Add(newBar);
20      materials[bfLength] -= 1;
21    }
22  }
23  return cuttingLayouts;
24 }
```

Listing 1.3: Implicit depending code

Let's have a look what dependencies and vulnerable code section we can identify in our running example. Listing 1.3 shows another code section of the Cutting Stock program. To conserve space, we list this example without its specification. This method creates different cutting layouts using the `AddCut` method. The above defined POG $\pi_2 = (\Omega_1, \tilde{s}_2, \phi_1)$ of `AddCut` requires a valid precondition. Therefore, this POG depends on all POGs covering this precondition. One of those POGs is generated based on the following control flow $\tilde{s}_3 = \langle s_2, s_3, s_4, s_5, s_6, \dots \rangle$ for the `CreateBars` method $\pi_3 = (\Omega_3, \tilde{s}_3, (CLen > 0))$. This control flow must guarantee that the used cut length is greater than zero before calling `AddCut`. We call this an explicit dependency. There exists another kind of dependency for all control flows calling `AddCut`. These code sections depend on the invariants of `Bar`, such as π_2, even if these invariants are not explicitly addressed. This is the case because every method requires valid object states when calling their methods. We call this an implicit dependency. Such dependencies are expressed as assumptions and are handled during the POG generation. However, we remember that the POG π_2 could not be verified. Therefore, all POGs calling `AddCut` in their control flow are considered as vulnerable.

3.3 Spreading Errors - The Risk of Vulnerable Proof Obligations

The main risk is the spreading of undiscovered errors in tested code into seemingly unrelated or previously verified code sections. In such cases, errors might be difficult to find, because the error source might be hidden in the method call stack. This is illustrated in Figure 3. The method call graph shows two methods, m_1 and m_3, both calling method m_2. The precondition γ_3 of method m_2 must be respected by both calling methods m_1 and m_3. Therefore, the precondition is covered by two POGs: π_{3a} for the control flow in m_1 and π_{3b} for the control flow in m_3. The postcondition γ_4 of m_2 is covered by the POG $\pi_4 = \Pi(\gamma_4)$. It depends on the correctness of the precondition γ_3: $\pi_4 \vdash \{\pi_{3a}, \pi_{3b}\}$. Let us assume

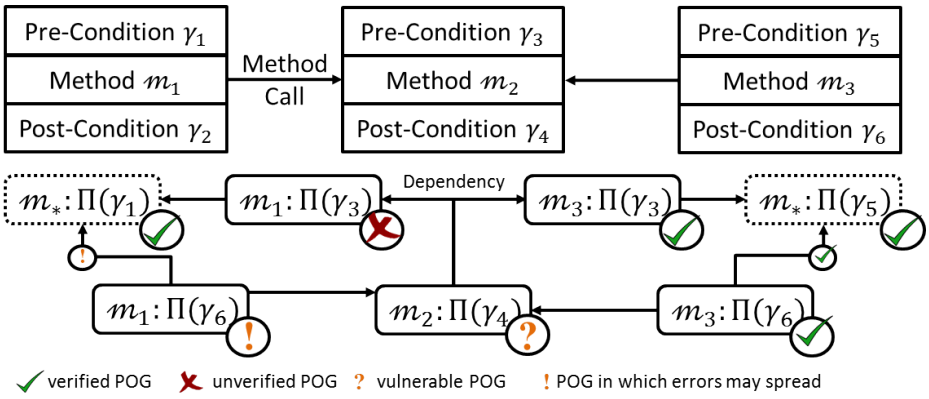

Fig. 3: Method call graph (Top) and possible POG dependency graph (Bottom)

that the POG π_{3a} can not be verified. This makes π_4 vulnerable because it now depends on an unverified POG. Errors in m_1 may cause a failed precondition γ_3, which produces errors in m_2 even though m_2 has been modularly verified. The result might be an invalid return value of m_2, which in turn may affect the code section in m_1 handling this return value. Thereby, even the postcondition γ_2 of method m_1 may fail.

The goal of our approach is to identify such risks by testing the method m_2 assuming that γ_3 has failed. This allows us to add additional checks to prevent such an error propagation.

3.4 Generating Robustness Tests

In general, robustness tests are used to analyse the behaviour of a program under hostile circumstances. This can be done in different ways. In some cases, it is sufficient to call a method with invalid parameter values. In other cases it is required to modify the tested code in order to simulate failures. In such a scenario, we speak of "Mocked Test Cases".

Definition 4 (Mocked Test Case). *A test case $t \in T$ with test vector $\vec{t}$ executes the control flow $\tilde{s}$: $t \rightarrow \tilde{s}$. A mocked test case $t[\eta \mapsto \acute{\eta}] \rightarrow \tilde{s}$ replaces in $\tilde{s}$ the symbol η by $\acute{\eta}$ before executing $\tilde{s}$.*

Definition 5 (Robustness Tests). *A robustness tests $t \in T^?$ is used to simulate failed guarantees by injecting invalid symbol values. We use following syntax to express the requirements on a symbol's value: $[\![\gamma]\!]_v \rightarrow \bot$. When evaluating γ using value v, γ must evaluate to false. Additional test oracles are defined using following syntax: $[\![t]\!] \rightarrow \bot\phi$ The evaluation of test case t must fulfil the boolean condition ϕ. Invalid symbols must be set during the test case execution and the only location to define them is within the robustness test parameter vector. Therefore it must be possible to back trace those values to the test vector. We use the right arrow syntax $\rightarrow$ to express this trace, e.g. $\vec{t} \rightarrow \vec{\dot{m}} \rightarrow \mathcal{F}$. This syntax expresses that the original test vector $\vec{t}$ is used to fill the method parameter list $\vec{\dot{m}}$. These parameter values $\vec{\dot{m}}$ are used to set the class field values $\mathcal{F}$.*

We create one robustness test for each POG π_i depending on an unverified POG:

$$T^? = \{\forall_i \pi_i = (\Omega_i, \tilde{S}_i, \phi_i) \in \Pi^? \ : \ t \rightarrow \tilde{S}_i\} \tag{2}$$

We use the verification goals of all proof obligations covering the tested control flow as additional test oracles:

$$\forall_i (t_i \rightarrow \tilde{S}_i) \in T^? \ \forall_j \pi_j = (\Omega_j, \tilde{S}_j, \phi_j) \in \Pi \ : \ [\![t_i]\!] \rightarrow \phi_j \mid \tilde{S}_i \subseteq \tilde{S}_j \tag{3}$$

The way failed guarantees are simulated depends on the POG's origin:
If the unverified POG π_i was generated to cover the precondition γ_i^{pre} of method $\dot{m}$, we must create a robustness test which calls $\dot{m}$ with parameter values $\vec{t}$ violating γ_i^{pre}:

$$\forall_j \pi_j \vdash \pi_i \mid \neg\Psi(\pi_i) : t \rightarrow \tilde{S}_j, \vec{t} \mid [\![\gamma_i^{pre}]\!]_{\vec{t}} \rightarrow \bot \tag{4}$$

If the unverified POG π_i was generated to cover the postcondition γ_i^{post} of method $\dot{m}$, we must inject a return value of $\dot{m}$ violating γ_i^{post}. To inject the simulated return value, we must create a mocked version $\ddot{m}$ of $\dot{m}$. The mocked version uses an extended parameter list $\vec{\ddot{m}}$, which allows us to directly set the return value based on $\vec{t}$. We refer to the return value of $\ddot{m}$ by $[\![\ddot{m}]\!]$.

$$\forall_j \pi_j \vdash \pi_i \mid \neg\Psi(\pi_i) : t[\dot{m} \mapsto \ddot{m}] \rightarrow \tilde{S}_j, \vec{t} \rightarrow \vec{\ddot{m}} \rightarrow [\![\ddot{m}]\!] \mid [\![\gamma_i^{pos}]\!]_{[\![\ddot{m}]\!]} \rightarrow \bot \tag{5}$$

If the unverified POG π_i was generated to cover an invariant γ_i^{inv} of class c_i, we must inject an invalid object instance. Therefore we must distinguish between two further cases: (1) The POG was generated based on a constructor $\dot{ctor}$. (2) The POG was generated based on a method $\dot{m}$. In the first case, the invalid object instance can be created by a mocked constructor method $\ddot{ctor}$. The mocked constructor uses an extended parameter list $\vec{\ddot{ctor}}$, which allows us to directly set all class fields $\mathcal{F}$ based on $\vec{t}$.

$$\forall_j \pi_j \vdash \pi_i \mid \neg\Psi(\pi_i) : t[\dot{ctor} \mapsto \ddot{ctor}] \rightarrow \tilde{S}_j, \vec{t} \rightarrow \vec{\ddot{ctor}} \rightarrow \mathcal{F}] \mid [\![\gamma_i^{inv}]\!]_{\mathcal{F}} \rightarrow \bot \tag{6}$$

```
1 public bool AddCutMocked(double cutLength, double usedLength,
  double length) {
2   if ((Length - UsedLength - (Cuts.Count * MinSpace)) <
    cutLength)
3   { return false; }
4   UsedLength += cutLength;
5   Cuts.Add(cutLength);
6   UsedLength = usedLength;  // Inject invalid field values
7   Length = length;    // based on parameter list
8   return true;
9 }
```

Listing 1.4: Mocked method to inject simulated errors

In the second case, the invalid instance must be simulated by a mocked copy $\ddot{m}$ of $\dot{m}$. The mocked method must set all referenced class fields $\mathcal{F}$ of c_i based on its own extended parameter list $\ddot{\vec{m}}$.

$$\forall_j \, \pi_j \vdash \pi_i \mid \neg\Psi(\pi_i) : t[\dot{ctor} \mapsto \ddot{ctor}] \rightarrow \tilde{S}_j, \vec{t} \rightarrow \ddot{\vec{m}} \rightarrow \mathcal{F} \mid [\![\gamma_i^{inv}]\!]_{\mathcal{F}} \rightarrow \bot \qquad (7)$$

In our running example, the failed POG was generated to cover an invariant during the execution of `AddBar`. Therefore, we need to inject the simulated error through a mocked method, which is listed in 1.4. We have added the additional parameters *useLength* and *length* to the original parameter list. These values are used in line 6 and 7 to set the invalid object state. Now, we only need to replace calls to `AddBar` by calls to `AddBarMocked` when testing `CreateBars`. We can simulate the invalid object state of `Bar` by calling `AddBarMocked`, e.g. with `usedLength=7000` and `length=5000`. The results of the robustness test will show that the created cutting layouts are invalid. Now we know that we need to add extra sanity checks to validate the correctness of `cuttingLayouts` before return them. This prevents undiscovered errors in AddBar from spreading into code sections where the origin of invalid cutting layouts may be difficult to find.

4 Reference-Implementation

Applying robustness tests and injecting simulated errors requires more effort than regular testing. Especially creating all required mocks is very labour intense when doing it manually. Therefore, we have implemented our methodology into a new mocking framework to face three major challenges: First, we need to ensure that every tested method is visible and accessible within the test suite. Second, we need to initialise object instances with deliberate states hosting the tested methods. Third, we must create the possibility to inject simulated errors to apply robustness testing. We meet those challenges by creating three different layers of mocked code: The first layer mocks the original source code to provide access to all class fields and methods. The second layer contains the mocked test methods to validate unverified POGs and vulnerable code sections. All steps in both layers can be applied automatically and do not require manual work. The

Algorithm: `InitList`(c)
Globals: $RecursionDepth$, $MaxRecursion$, $CollectionSize$
begin
  if $RecursionDepth[c] > MaxRecursion$ **then return** $\{c\}$
  $IP \leftarrow \emptyset$
  $RecursionDepth[c]++$
  foreach $f \in \mathcal{F}_c$ **do**
    if $IsSimpleType(f)$ **then** $IP = IP \cup Type(f)$
    else if $IsCollection(f)$ **then**
      for $i = 0 \rightarrow CollectionSize$ **do**
        if $HasKeyType(f)$ **then** $IP = IP \cup InitList(KeyType(f))$
        $IP = IP \cup InitList(ValueType(f))$
      end
    else $IP = IP \cup InitList(Type(f))$
  end
  $RecursionDepth[c]--$
  return IP;
end

Algorithm 1: Algorithm to generate initialisation parameter lists

third layer contains the actual test cases.
Layer 1 contains a mocked version $\bar{\mathbf{Prog}}$ of **Prog** to make the complete code base testable. We need to consider that we do not want to test complete methods but explicit control flows of unverified POGs. Those control flows might be extracted from private or abstract methods. In $\bar{\mathbf{Prog}}$ we set the visibility of each class field and tested method to `public`. We have chosen this way, because it is language independent. other solutions for accessing private symbols require language specific runtime flags or reflection APIs. We remove the `abstract` attribute from each tested class and method. Instead, we add an empty default implementation and a corresponding default return value to each pure abstract method so our program can be compiled. To be able to initialise every object type, we add a default constructor and a static initialisation method to each user-defined type. To create those initialisation methods, we use the recursive Algorithm 1 to inspect aggregated object types $c \in \mathcal{C}$. The algorithm extracts a list with parameters representing the aggregated primitive values. To that end, we also create items to fill used collection types, such as Lists, Arrays or Dictionaries. The number of items are added is set with the constant $CollectionSize$. Recursively analysing the key and value types, we merge the resulting parameter lists. To prevent endless recursion steps, we track the recursion depth with the map $RecursionDepth(c \rightarrow \mathbb{N})$ until the maximum recursion depth $MaxRecursion$ has been reached.
Layer 2 contains the mocked methods and constructors to inject invalid return types and object states. Invalid return values and object states are injected by setting corresponding class fields or by creating corresponding return values, instead of computing them. We use Algorithm 1 to extract the list of required

primitive types in order to manipulate or initialise the aggregated object. These extracted primitive types are added to the parameter list of the mocked method. Thereby, we can use the parameter list to explicitly control return values and object states. An example is given in Listing 1.7, in line 6 and 7.
Layer 3 contains the actual robustness tests calling the mocked test methods in layer 2. The parameter lists of test cases in this layer combine the extended parameter lists of called mocked methods. Errors can be injected by assigning corresponding parameter values. This might require additional manual work, if the list of parameters is to long and cannot be automatically covered by a symbolic execution tool like Pex.

5 Case Studies

The real world case study 'Settings Manager' (SM) is extracted from an industrial machine control software. The case study 'Cutting Stock' (CS) is the program hosting our running example. The program creates a list of cutting layouts based on the lengths and quantities of material and pieces. The case study 'Lending Library' (LL) is a small code example to manage the rental and return of items. All three studies are implemented in C# and use Code Contracts as specification language. Table 1 summarises the main properties.
We compare our methodology with results of current verification and testing techniques. We apply Microsoft Code Contracts as formal verification framework and IntelliTest as automatic test case generation framework. To get detailed POG information, we have implemented our own POG generator based on [3] and [13]. The results are summarised in Table 2. To benchmark the achieved benefits, we analyse the results of the generated robustness tests, and analyse whether they triggered an error within the vulnerable code. The resulting table lists those errors as "Robustness Errors". This table also lists the automatically achieved test coverage by regular test cases on unverified proof obligations.

Table 1: Overview case studies

	Settings Manager (SM)	**Cutting Stock (CS)**	**Lending Library (LL)**
LOC	1277	634	432
Preconditions	46	32	12
Postconditions	22	19	13
Invariants	21	13	15
Proof Obligations	187	115	52

5.1 Case Study: Settings Manager

We have created 187 POGs to cover all 89 single specifications. The verification framework left 4 POGs unverified, covering different preconditions. The

```
1 public object GetValue(string targetName) { [...]
2  Contract.Requires(targetValues.ContainsKey(targetName));
3  Contract.Requires(targetScopes.ContainsKey(targetName));
4  Contract.Ensures(Contract.Result<System.Object>() != null);
5  if (!targetValues.ContainsKey(targetName))
6  { throw new UnkownTargetException("[...]"); }
7  // Exception when targetName is no key [...] }
8  SettingScope targetSope = targetScopes[targetName];
```

Listing 1.5: Vulnerable code section which can causes a software crash

automatically generated test suite achieved 92% branch coverage on those 4 unverified control flows. Analysing those 4 unverified POGs, we could identify 18 vulnerable proof obligations. Two of them were not sufficiently secured against undiscovered errors.
Listing 1.5 shows one of those unsecured code sections. The precondition in line 3 could not be verified for each caller. To handle this incomplete verification result, related approaches create test cases to validate the unverified caller. In addition, we use robustness tests to analyse the consequences of a failing precondition. The robustness tests created by our approach call this method with a value for *targetName* which explicitly invalidates the precondition in line 2, while respecting all other assumptions. Thereby, we discovered the potential `KeyNotFound`-Exception in line 8, which would cause a software crash. This distinguishes a robustness test from a regular test. Regular test coverage could be achieved by testing this method while respecting both preconditions, but such tests would not trigger the error in line 8. This method was programmed based on the assumption that both containers (`targetScopes` and `targedNames`) share the same keys. Therefore the programmer checked the key only for one container. After discovering this risk, the developer could add additional exception handling similar to the one in lines 5-6.

5.2 Case Study: Cutting Stock

This case study is comprised of 64 specifications, which are covered by 115 POGs. The verification framework left 7 POGs unverified (4 invariants, 2 preconditions, 1 postcondition). The automatically generated test suite achieved 94% branch coverage on those 7 unverified control flows. We could identify 23 vulnerable proof obligations.
One of them was already discussed above and used as the running example. Another unverified POG covers the postcondition of `GetFreeClamp` in Listing 1.6. This method is called by a different method `AssignClamps`, which of course depends on the validity of that postcondition. `AssignClamps` calculates the required Clamp positions for the cutting machine, which is used to produce the generated cutting-layouts. Wrong clamp positions may cause damage to the machine, e.g. when the saw hits a wrongly positioned clamp. Therefore, we want to test the behaviour of `AssignClamps` when this postcondition fails, in order to

```
1 private Clamp GetFreeClampMocked(double minPos_1, double
  maxPos_2, bool free_3, double minPos_4, double maxPos_5) {
2   Clamp clamp = new Clamp();
3   clamp.Init(free_3, minPos_4, maxPos_5);
4   return clamp; }
```

Listing 1.7: Mocked method to inject invalid return values

guarantee safe error handling. The corresponding robustness test must inject an invalid return value for `GetFreeClamp` into `AssignClamps`. Our framework generates the mocked copy `GetFreeClampMocked` in Listing 1.7. In `AssignClamps`, all method calls to the original methods are replaced in order to call the mocked copy. As described in Section 4, the mocked method uses an extended parameter list to initialise the returned object: `free_3`, `minPos_4`, `maxPos_5`. These parameters map to the basic field values of class `Clamp`. Thereby it is possible to return a `Clamp` instance which is not null and which does not meet the defined postcondition. The analysis of this robustness test shows that the simulated error caused an invalid return value for `AssignClamps`. An invalid return value would cause invalid cutting layouts, leading to faulty production in the real world. What makes this bug particular dangerous is the absence of easily detectable errors such as exceptions or a crash. The problem would not have been detected until someone tried to produce the erroneously calculated cutting layouts. Analysing the robustness test, the developer can add an additional sanity check within `AssignClamps` and make sure that the results meet all requirements.

```
1 private Clamp GetFreeClamp(double minPos, double maxPos) {
2   Contract.Ensures(Contract.Result<Clamp>() == null || (
    Contract.Result<Clamp>().free && Contract.Result<Clamp>().
    minPos <= minPos));
3   foreach (Clamp clamp in clamps) {
4     if(clamp == null)  continue;
5     if(clamp.minPos <= minPos && clamp.maxPos >= maxPos)
6     { return clamp; }
7   }
8 return null; }
```

Listing 1.6: Code section with an unverified postcondition

5.3 Case Study: Lending Library

The smallest case study, 'Lending Library' was specified using 30 conditions, which were covered by 54 generated POGs. The verification step left 6 proof obligations unverified (3 invariants, 2 postconditions, 1 precondition). Based on those 6 unverified POGs, we could identify 12 vulnerable code sections. The corresponding robustness tests discovered 3 critical code sections where additional sanity checks were required. Now someone could wonder about the vulnerable sections where no robustness error was found. The answer is very simple. The other 9 vulnerable code sections already contained error handling code, so no additional code needed to be added. An example is given in Listing 1.8. This

```
1 public bool ReturnItem(RentalItem item) {
2   if(!item.Rented) { return false; }
3   [...] }
```

Listing 1.8: Existing checks also handle simulated errors

method requires that the given `RentalItem` is actually rented and not returned. This state is encoded as an boolean class field. The related precondition could not be verified. However, this flag is already checked in line 2 and the robustness test could not trigger any new error.

Table 2: Comparsion between both approaches

	SM	**CS**	**LL**
Unverified Proof Obligations	4	7	6
Autom. achieved code coverage	92%	94%	98%
Identified Vulnerable Code Sections	18	23	12
Discovered Robustness Errors	2	5	3

6 Conclusion and Future Work

Our case studies have shown that automatic test frameworks already achieve high coverage rates on unverified code sections. This poses the risk that such test suites might never be checked manually by the corresponding developer to identify insufficient test cases, as shown in Listing 1.2. That makes the inspection of code sections that rely on the correctness of the tested code particularly important. Even tested methods, entirely covered, may still contain errors. Therefore, only testing unverified code sections is insufficient when combining formal verification and dynamic testing techniques. Undiscovered errors in tested code my spread into other code sections, even those sections that have been previously verified. Such errors may be hard to debug, as they might be camouflaged after having been propagated through different methods. This was demonstrated in our running example extracted from the Cutting Stock case study. These risks are not handled by current approaches.

To reduce this residual risk, we have presented a new methodology to use robustness testing to handle incomplete verification results. We extract the guarantees of unverified proof obligations and use them to create and inject simulated errors. Those errors test the behaviour of vulnerable code sections in situations when those guarantees fail. The presented reference-architecture demonstrates how robustness tests can be generated and how simulated errors can be injected. By injecting simulated errors, the developer can analyse the consequences of failed guarantees. They can add further exception handling and sanity checks to prevent the propagation of previously undiscovered errors into other methods. The software can then handle errors in a controlled way rather than defaulting to unpredictable behaviour. It could be argued that developers can always add more

exception handling and state checking. But this would be very labour intense when applied for every return value and argument. Furthermore, these sanity checks must be tested as well, requiring many robustness tests to cover the corresponding program paths. The presented methodology helps the developer to localise precisely those code sections, where additional error handling is required.

Finally, one major issue regarding formal verification needs to be addressed in future work. There is still no proper way to tell whether the defined specifications are sufficient and cover all necessary requirements. When the specification is insufficient, the number of generated POGs may be too small to properly analyse dependencies between them in order to identify vulnerable sections. Future work must find more sophisticated coverage rates for specifications.

References

1. Müller, P.: Modular specification and verification of object-oriented programs. Springer-Verlag (2002)
2. Hoare, C.A.R.: An axiomatic basis for computer programming. Communications of the ACM **12**(10) (1969) 576–580
3. Beckert, B., Hähnle, R., Schmitt, P.H.: Verification of object-oriented software: The KeY approach. Springer-Verlag (2007)
4. Amor, H.B., de Carvalho, J.V.: Cutting stock problems. In: Column generation. Springer (2005) 131–161
5. Christakis, M., Müller, P., Wüstholz, V.: Collaborative verification and testing with explicit assumptions. In: International Symposium on Formal Methods, Springer (2012) 132–146
6. Christakis, M., Müller, P., Wüstholz, V.: Guiding dynamic symbolic execution toward unverified program executions. In: Proceedings of the 38th International Conference on Software Engineering, ACM (2016) 144–155
7. Christakis, M.: Narrowing the gap between verification and systematic testing. PhD thesis, National Technical University of Athens, Greece (2015)
8. Czech, M., Jakobs, M.C., Wehrheim, H.: Just test what you cannot verify! In: International Conference on Fundamental Approaches to Software Engineering, Springer (2015) 100–114
9. Kanig, J., Chapman, R., Comar, C., Guitton, J., Moy, Y., Rees, E.: Explicit assumptions-a prenup for marrying static and dynamic program verification. In: International Conference on Tests and Proofs, Springer (2014) 142–157
10. Fähndrich, M. In: Static Verification for Code Contracts. Springer Berlin Heidelberg, Berlin, Heidelberg (2010) 2–5
11. Xie, T., Tillmann, N., Lakshman, P.: Advances in unit testing: theory and practice. In: Proceedings of the 38th International Conference on Software Engineering Companion, ACM (2016) 904–905
12. de Halleux, J., Tillmann, N. In: Moles: Tool-Assisted Environment Isolation with Closures. Springer Berlin Heidelberg, Berlin, Heidelberg (2010) 253–270
13. Huster, S., Heckeler, P., Eichelberger, H., Ruf, J., Burg, S., Kropf, T., Rosenstiel, W.: More flexible object invariants with less specification overhead. In: International Conference on Software Engineering and Formal Methods, Springer (2014) 302–316