Mock-object generation with behavior

Nikolai Tillmann Wolfram Schulte
Microsoft Research, One Microsoft Way, Redmond WA, USA
{nikolait,schulte}@microsoft.com

Abstract

Unit testing is a popular way to guide software development and testing. Each unit test should target a single feature, but in practice it is difficult to test features in isolation. Mock objects are a well-known technique to substitute parts of a program which are irrelevant for a particular unit test. Today mock objects are usually written manually supported by tools that generate method stubs or distill behavior from existing programs. We have developed a prototype tool based on symbolic execution of .NET code that generates mock objects including their behavior by analyzing all uses of the mock object in a given unit test. It is not required that an actual implementation of the mocked behavior exists. We are working towards an integration of our tool into Visual Studio Team System.

1. Introduction

Unit testing is a popular way to guide software development and testing. It is central to the Extreme Programming [2] and Test-Driven Development [1] paradigms. Each unit test should target a single feature so that a failing unit test identifies the broken feature as concisely as possible.

However, in practice it is often difficult to test features in isolation. Mock objects [4] are a commonly used technique to substitute parts of the program that are irrelevant for a tested feature. Mock objects answer queries with fixed values similar to those that the substituted program computes.

Today developers usually define mock objects by hand. Several frameworks [9] exist which provide method stubs. The behavior of the stubs must still be programmed by the developer. A capture-replay approach is used in [5] to distill actual behavior of an existing program into mock objects.

We have implemented a prototype tool which generates mock objects and their behavior; it does not need an implementation of the mocked behavior. By default, the behavior of a mock object is defined by the sequence of result values returned by method calls with the mock object as the receiver. The developer can extend the considered behaviors. Our tool employs symbolic execution [3] to analyze how a given .NET unit test and program under test use a mock object. The tool implements constraint solving techniques to determine if different result values will cause the execution of different execution paths. Finally, the tool generates code that creates a concrete mock object and defines its behavior for each discovered execution path. Concrete mock objects enable traditional debugging and fast regression tests.

We are working towards an integration of mock object generation into Visual Studio Team System. Our tool is currently integrated in Visual Studio as an Add-In.

In Section 2 we give an overview of symbolic execution, in Section 3 we illustrate the generation of mock objects, and Section 4 provides concluding remarks.

2. Symbolic Execution

Symbolic execution is a way to explore all execution paths of a program. Symbolic execution is similar to concrete execution, but it supplies symbols as inputs to the program instead of concrete values. Consequently, the values the program computes are expressions over the input symbols. When symbolic execution reaches a conditional branch, it considers all branch targets. For example, when reaching an if-statement, it splits the current execution path into two. Each execution path can be characterized by a path condition: When reaching an if-statement, the condition is conjoined to the path condition for the then-path and the negated condition to the path condition of the else-path.

Many conditional branches are implicit. For example, accessing an object member fails if the receiver is `null`. Likewise, a unit test fails if an assertion does not hold.

Our tool implements *concolic execution* [6], a combination of concrete and symbolic execution. To this end, our tool instruments the program's code. When the instrumented program is executed with concrete inputs, it not only performs the concrete computation along some execution path, but it also derives the path condition that characterizes the taken execution path. We obtain an initial path condition by executing the program with random input values. Our tool implements constraint solving techniques to

21st IEEE International Conference on Automated Software Engineering (ASE'06)
0-7695-2579-2/06 $20.00 © 2006 IEEE

find values that do not satisfy a given path condition. When supplied with these new values, the program will execute along a different path. In this way, our tool explores all execution paths within user-supplied bounds.

3. Example Tool Usage

Consider the method `AppendFormat` of the `StringBuilder` class in the .NET base class library. Given a string with formatting instructions, and a list of values to be formatted, it computes a formatted string.

```
public StringBuilder AppendFormat(
  IFormatProvider provider,
  string format, object[] args);
```

The first parameter of type `IFormatProvider` "provides a mechanism for retrieving an object to control formatting" according to the MSDN documentation:

```
public interface IFormatProvider {
  object GetFormat(Type fmtType); }
```

A non-trivial test calling `AppendFormat` needs an object that implements `IFormatProvider`. First, our tool generates a mock type that implements the interface:

```
class MFormatProvider : IFormatProvider {
  public object GetFormat(Type fmtType) {
    IMethodCallOracle call =
      Oracle.NextMethodCall(this,
        MethodBase.GetCurrentMethod());
    return call.ChooseResult<object>(); }}
```

The mock method `GetFormat` obtains from a global `Oracle` a handle `call` representing the current method call. The oracle is the glue that enables symbolic execution as well as concrete execution of unit tests with arbitrary mock objects. It provides the values which define the behavior of the mocked methods. Here the generated code only chooses a result value. The developer can extend the code, e.g. adding the choice to throw an exception, perform a callback, or change some accessible state.

For symbolic execution, `ChooseResult` returns a fresh symbol representing an arbitrary result value. For concrete execution, the oracle can be instructed to either return random concrete values, or a sequence of fixed values.

As described before, our tool will analyze all uses of the result. The following use occurs in `AppendFormat`. It calls `provider.GetFormat` after checking `provider!=null`:

```
cf = (ICustomFormatter)provider.
  GetFormat(typeof(ICustomFormatter));
```

Depending on the result of `GetFormat`, the cast to `ICustomFormatter` might fail. Our tool finds the failure, and it generates the following code snippet. It creates a concrete mock object and instructs the oracle that during the concrete execution of a unit test the first call to `m.GetFormat` should return a new `object`. This will cause the cast to fail, since `object` does not implement `IFormatProvider`.

```
MFormatProvider m = new MFormatProvider();
IMethodCallOracle call0 =
  Oracle.DefineMethodCall(m, 0,
    m.GetType().GetMethod("GetFormat"));
call0.SetResult<object>(new object());
```

4. Conclusion

Symbolic execution is a program analysis technique that works well on small programs. Mock objects are the preferred technique to unit test features in isolation. Using symbolic execution to generate mock objects is a promising approach to unit test big projects. [7] evaluates symbolic execution on parts of the .NET base class library, and [8] contains a broad overview of how unit testing can benefit from symbolic execution, and it discusses how to prune undesirable behaviors of generated mock objects.

References

[1] K. Beck. *Test Driven Development: By Example*. Addison-Wesley, 2003.

[2] K. Beck and C. Andres. *Extreme Programming Explained: Embrace Change (2nd Edition)*. Addison-Wesley Professional, 2004.

[3] J. C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, 1976.

[4] T. Mackinnon, S. Freeman, and P. Craig. Endotesting: Unit testing with mock objects, May 2000. XP2000.

[5] D. Saff, S. Artzi, J. H. Perkins, and M. D. Ernst. Automatic test factoring for java. In *Proc. 20th ASE*, pages 114–123, New York, NY, USA, 2005. ACM Press.

[6] K. Sen, D. Marinov, and G. Agha. CUTE: A concolic unit testing engine for C. In *Proc. ESEC/FSE'05*, pages 263–272. ACM Press, 2005.

[7] N. Tillmann and W. Schulte. Parameterized unit tests. In *Proc. 13th ESEC/FSE*, pages 253–262, New York, NY, USA, 2005. ACM Press.

[8] N. Tillmann and W. Schulte. Unit tests reloaded: Parameterized unit testing with symbolic execution. *IEEE Software*, 23(4):38–47, 2006.

[9] Wiki. Mock objects. www.mockobjects.com.