Testing .NET Code with YETI

Manuel Oriol, Sotirios Tassis
Department of Computer Science
University of York,
York, United Kingdom
Email: manuel@cs.york.ac.uk, sotirios_tassis@yahoo.gr

***Abstract*—Testing code is one of the central techniques for quality assessment of code. Generating test cases manually, however, is costly and inherently biased by the human point of view. While this might not be an issue for end-user code, it is problematic for developing libraries.**

The York Extendible Testing Infrastructure (YETI) is an automated random testing infrastructure supporting multiple programming languages (Java, JML, and .NET). It tests code at random and decouples the engine from the strategies and the language used. This article presents the .NET binding.

***Keywords*-Automatic test software, Software engineering.**

I. Introduction

Automated random testing is a methodology often neglected by programmers and software testers because it is deemed as overly simple. It has, however, advantages over other techniques because it is completely unbiased and allows the execution of a high number of calls over a short period of time.

The York Extendible Testing Infrastructure (YETI) provides a framework for executing random testing sessions in a language agnostic way. At its core lies a testing engine that performs calls one after the other on methods picked at random with arguments generated at random.

Oracles are however language-dependent. In the presence of specifications YETI checks inconsistencies between the code and the specifications. In the case of languages such as .NET, YETI uses code-contracts[1] as oracles. In case a precondition of the method under test is violated, then it does not interpret it as a failure. In case no precondition is violated, any exception can be interpreted as a bug if it is not declared in the documentation. If programmers do not use contracts, a testing session of .NET programs with YETI is then a robustness test that reports all runtime exceptions. Because .NET does not declare runtime exceptions, all triggered must then be compared with exceptions declared in the documentation to check whether they were expected or they are faults.

Unlike competitors, YETI also supports a graphical user interface that allows test engineers to monitor the testing session and modify some of its characteristics while testing.

[1] http://research.microsoft.com/en-us/projects/contracts/

Section II describes its current implementation for .NET. Section III presents related work. We eventually conclude in Section IV.

II. Implementation

The core of YETI is an application coded in Java, allowing to test programs at random in a fully automated manner. It is designed to support various programming languages – for example, functional, procedural and object-oriented languages can easily be supported. It contains three parts: the core infrastructure, the strategy, and the language-specific bindings. YETI is a lightweight platform with around 5000 lines of code for the core, graphical interfaces and the strategies. The .NET binding in itself consists of around 1000 lines of code in Java and 1700 lines of code in C#.

A. Using YETI

YETI can be launched on the command-line. A typical call to YETI that tests .NET assemblies is:

```
java yeti.Yeti -dotnet -time=10mn -randomPlus
-testModules=Assembly1.dll:Assembly2.dll
```

The options used on this command-line have the following meaning: `-dotnet` indicates that the tested program is in .NET bytecode, `-time=10mn` indicates that the testing session will last 10 minutes, `-randomPlus` indicates that the strategy random+ will be used, and `-testModules=Assembly1.dll:Assembly2.dll` indicates that both `Assembly1.dll` and `Assembly2.dll` will be tested.

While testing, traces of faults found are output in the terminal. At the end of the testing sessions, YETI outputs generated test cases reproducing the faults found during the testing session as well.

Note that it is also possible to avoid the overhead of keeping the traces in the system (and calculating the minimal test cases) by specifying `-nologs` to throw away all logs except exception traces, or `-rawlogs` to output the logs to the terminal.

B. Core Architecture

YETI uses the core notions of types, routine, modules and variables by defining respectively Java classes `YetiType`, `YetiRoutine`, `YetiModule`, and `YetiVariable`.

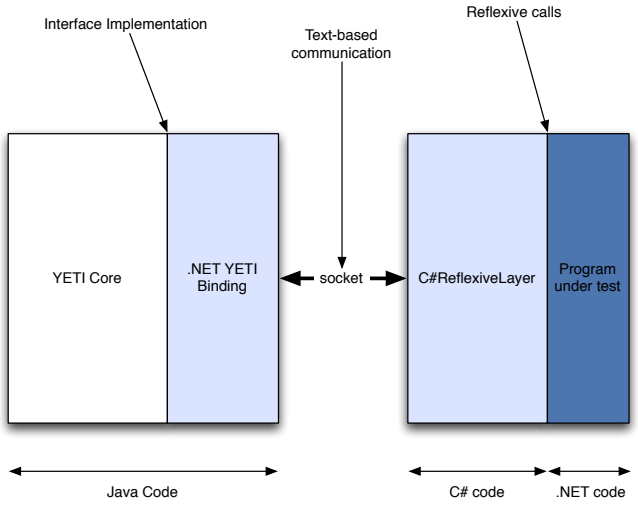

Figure 1. Yeti .NET binding architecture.

Extending YETI to support a new programming language is done through specializing a set of abstract classes in the YETI core. In the case of the .NET binding, it was necessary to make the language-agnostic part communicate with the .NET code through a custom-made communication layer. We chose to use a socket-based solution for its modularity and potential for distributed testing sessions.

Figure 1 shows the main structure of the YETI .NET binding. The architecture consists of three main different parts:

- the main YETI core in white,
- the .NET binding in light blue,
- the program under test in dark blue.

The .NET binding itself is coded both in Java and C#. The C# part is the C#ReflexiveLayer. It consists of a very simple text-based interpreter for extracting information from the .NET code under test, making calls to routines in the program under test, maintaining a variable pool and returning results. The Java part is a simple stub between standard YETI structure and the interpreter.

The communication between the two parts of the .NET binding is text-based. When first loaded the C# reflexive layer queries all information in loaded assemblies and sends it to the YETI part. This constitutes the initialization of the .NET binding. Then the Java part of the binding translates method calls ordered by the main YETI infrastructure in a text-based, sends them through the socket and reads the result of the computation. The C# reflexive layer performs the calls and returns the results, be it an exception trace, a newly created variable or only the indication of no failure.

The failures are stored in the core Java part and the logs are processed at the end of the testing session.

C. Validation

We validated our approach by testing five different assemblies containing 20 seeded bugs in total. YETI found 77% of the seeded faults over a testing sessions of 100 seconds for each assembly.

Performances were satisfactory as YETI performed around 300,000 calls per minute.

III. Related Work

YETI is the latest in a serie of recent random testing tools like JCrasher [1], Jartege [2], Eclat [3], or AutoTest [4]. YETI is however different from these tools in at least three important ways: (1) it is made to easily support multiple programming languages, (2) it is intended to allow test engineers to modify the parameters of the testing during the testing session itself, and (3) it is at least 3 orders of magnitude faster than competing tools.

The .NET binding of YETI is however the first random testing tool to interpret correctly code-contracts violations and to leverage on that to filter bugs.

IV. Conclusions

This article presented the .NET binding of YETI. The binding allows YETI to test .NET programs in an automated and random way. It is the first random system to interpret failures of code contracts in a correct way automatically.

To validate our approach we used YETI on code in which we seeded bugs and showed that we were able to find 77% of the seeded bugs. This allowed us to identify some types of bugs that YETI cannot find in .NET.

Future work will focus on enriching YETI with an automated parser for documentation in order to help with the automated identifications of bugs.

References

[1] C. Csallner and Y. Smaragdakis, "Jcrasher: an automatic robustness tester for java," *Software: Practice and Experience*, vol. 34, no. 11, pp. 1025–1050, 2004.

[2] C. Oriat, "Jartege: a tool for random generation of unit tests for java classes," Centre National de la Recherche Scientifique, Institut National Polytechnique de Grenoble, Universite Joseph Fourier Grenoble I, Tech. Rep. RR-1069-I, June 2004. [Online]. Available: http://arxiv.org/abs/cs.PL/0412012

[3] C. Pacheco and M. D. Ernst, "Eclat: Automatic generation and classification of test inputs," in *ECOOP 2005 — Object-Oriented Programming, 19th European Conference*, Glasgow, Scotland, July 25–29, 2005. [Online]. Available: http://pag.csail.mit.edu/pubs/classify-tests-ecoop2005-abstract.html

[4] I. Ciupa, A. Pretschner, A. Leitner, M. Oriol, and B. Meyer, "On the predictability of random tests for object-oriented software," in *International Conference On Software Testing, Verification And Validation (ICST 2008)*, July 2008.