# Testing .NET Code with YETI

Manuel Oriol, Sotirios Tassis

***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 and validates our prototype with testing both toy and real-world examples.**

***Index Terms*—IEEEtran, journal, LaTeX, paper, template.**

## 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. The main characteristics of YETI is that it supports multiple programming languages through a language-agnostic meta model. Various testing strategies apply to all supported languages thanks to a strong decoupling between the strategies used and the programming language binding.

Oracles are language-dependent. In the presence of specifications YETI checks inconsistencies between the code and the specifications. In the case of languages such as .NET, code-contracts[1] can be used to direct the testing of the code. In case a precondition of the method under test is violated, then we do 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

Unlike competitors, YETI also support a graphical user interface that allows test engineers to monitor the testing session and modify some of its characteristics while testing. To validate our approach we classes using code-contracts and tried to find seeded bugs in those classes. We also applied our technique to the .NET system libraries.

Section II presents informally YETI's meta-model and its main algorithms. Section III describes its current implementation for .NET. Section IV evaluates the YETI binding for .NET. Section V presents related work. Section VI describes future work. We eventually conclude in Section VII.

M. Oriol is with University of York.
S. Tassis was a student at University of York.

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

## II. Model

This section describes the meta-model used by YETI to represent programs and data. The actual model was previously describe in an article describing the Java binding of YETI [1], please refer to it for a complete definition of the model.

YETI uses four main notions:

**Routines:** A routine is a computation unit that uses variables of a given type and returns values of a given type:

$$R ::= (T_1 \times ... \times T_k) \rightarrow T_0 \quad (1)$$

**Types:** A type is a collection of variables and a collection of routines that return values(of that type):

$$T ::= ((R_1, ..., R_k), (V_1, ..., V_l)) \quad (2)$$

**Variables:** A variable is a couple between a label and a value:

$$V ::= (n, v) \quad (3)$$

**Modules:** A module is a collection of routines:

$$M ::= (R_1, ..., R_k) \quad (4)$$

This model is used by YETI as the backbone of any binding. Note that types can be organized through a subtyping relationship which imposes that all variables of a subtype are also present in the super type and that all types constructors of the subtype are also constructors of the supertype. YETI ensures that the these properties are verified.

While these definitions model a program from a high level that is not concerned with actual values and computation, it is enough for devising strategies. As an example, Figure 1 shows the algorithm – in C#-like pseudo code – of the pure random strategy.

What Figure 1 does not show is the generation of new variables other than through making calls. In this example, we assume that this happens in `T.getArgumentAtRandom()`. It could however be added to the algorithm without any issue.

Other strategies are however possible. For example Random+ [2] either picks selects values in the existing pool of object, generate new ones on the fly, or take interesting values. This implies that the model should be enriched to take into account a set of interesting values in each type. In practice, this is exactly what happens, and up to now, we always managed to introduce backward compatible changes.

```
M0 = .../** module to test **/
while (not endReached){
  R0=M0.getRandomRoutine();
  Vector<Variables> arguments =
           new Vector<Variable>();
  for(T in R0.getArguments()){
   arguments.addLast(
         T.getArgumentAtRandom());
  }
  try {
    new Variable(R0.call(arguments));
  } catch (Exception e) {
    if (not
         (e is PreconditionViolation)){
     foundBugs.add(e);
    }
  }
}
```

Fig. 1. Algorithm for the random strategy.

## III. 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 is a tool that can be launched on the command-line. A typical call of 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. For example:

```
Exception 15
Value cannot be null.
Parameter name: to
   at System.Net.Mail.MailMessage..ctor(String
              from, String to)
   at System.Net.Mail.MailMessage..ctor(String
              from, String to, String subject,
              String body)
```

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. Graphical User Interface (GUI)

By specifying the `-gui` option, YETI shows a graphical user interface that allows test engineers to interact directly with the system while the testing session proceeds.

Figure 2 shows YETI's graphical user interface when using the random strategy. At the top of the interface, two sliders correspond to the percentage of null values and the percentage of new variables to use when testing. In short, each time a test is made, each parameter of the routine to test can either be void, be newly generated or a new variable. These sliders indicate which probability to use. In the top part there is also a text field to limit the number of instances per type in the system (which is necessary for long-running sessions).

The left panel contains a list of modules loaded in the system. The modules being tested are ticked, while others can be used as helpers to create variables to use on-demand making a routine calls. A button is available to add modules at runtime for programming languages that support it. This last part is useful in cases where a module is missing to enable the testing of a routine.

In the central panel, four graphs describe the evolution of the system: the top-left one shows the evolution of the number of unique failures found – all failures without redundancy –, the bottom-left indicates the raw number of failures over time, the bottom-right indicates the current number of instances in the system, the top-right panel indicates the total number of calls effected by YETI.

The panel on the right shows all methods tested in the system.

The bottom panel reports unique failures as they are found: each line is a unique failure.

In order for the graphical interface not to slow down the testing process, we use two threads. The first one samples data for building graphs every .1 seconds, the second one updates the graphical user interfaces and waits .1 second between two updates. A special care has also been taken for not showing all samples in the graphs when not needed: we only show one point per pixel on the x-axis.

### C. Core Architecture

The core architecture of YETI is very consistent with the model described in Section II. YETI uses the core notions of types, routine, modules and variables by defining respectively Java classes `YetiType`, `YetiRoutine`, `YetiModule`, and `YetiVariable`. YETI also maintains the types constraints as discussed in Section 2.

Extending YETI to support a new programming language is done through specializing a set of abstract classes in the

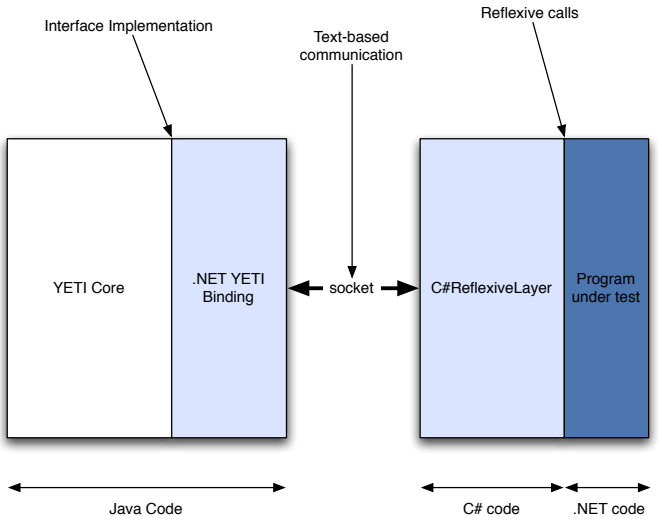


Fig. 3. Yeti .NET binding architecture.

YETI core. In the case of the .NET binding, it was necessary to make the Java 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 3 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 is a very simple text-based interpreter for extracting information from the .NET code under test, making calls of 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 in the binding's Java part at the end of the testing session.

## IV. Evaluation

This section presents the evaluation of the .NET binding of YETI. In this evaluation, we first tested examples that we coded with code contracts and in which we seeded bugs, we then show the results for testing `System.dll` for a short amount of time.

### A. Evaluating through seeded bugs

We tested five assemblies instrumented with Code-Contracts – pre-conditions, post-conditions, assertions and object invariants. This strategy focused on measuring performance, its fault finding ability and any limitations that the binding has. We repeated all experiments 30 times to ensure that the outcomes holds statistical validity. The experiment used a computer equipped with an Intel Core 2 Duo T5750 at 2.00 GHz with 3.00GB memory running under Windows Vista Home Premium Service Pack 1.

Table I presents an overview the number of faults seeded and found by YETI. On average, with testing sessions of 100 seconds, YETI found 77% of the seeded bugs.

Table **??** also presents statistics about testing seeded code: the number of calls to first fault, the time to first fault, the total number of failures (redundant) and the number of method calls in 10s testing sessions. Note that in each case the testing session reached a plateau and it was not expected to find further faults. The main interesting result is that on similar classes, the Java binding of YETI performed around 800'000 calls per minute [1] (1'120'000 method calls for 100 seconds). Therefore the overhead of using sockets incurs around 100% penalty over making native calls.

Figure 4 presents the seeded bugs for the first three assemblies. The last two are omitted because of their complexity. All the code is available online.[2]

Overall, seeding bugs helped with discovering the difficulties that the random strategy has to find certain types of faults:

- Post-conditions that break only with specific values:
  ```
  Contract.Ensures(
  Contract.Result<int>() != 0)
  ```
- Post-conditions that break only on values that exist in a small ranges:
  ```
  Contract.Ensures(r<222 && r>200)
  ```
- Post-conditions that break if specific conditions hold simultaneously:
  ```
  Contract.Ensures(!(o == null &&
  s.Equals("End Of Evaluation") &&
  l == 12000))
  ```
- Faults that require a specific sequence of method calls in order to raise an exception.

In addition, we deduce that the performance of the binding tool is satisfactory and the use of sockets is reasonable. When YETI performed testing, neither the Java application (i.e. yeti.environments.csharp) nor the CsharpReflexiveLayer ever crashed or raised an exception at any of the testing sessions on the YetiTestAssemblies.

*1) Testing real-world assemblies:* We tested the limitations of the binding, its performance and if it could recognize non Code-Contracts exceptions of real usable assemblies. The benchmarks that the second strategy used are System.dll and mscorlib.dll, which are the basic libraries of C# and C++ programming languages, respectively. The experiment used a computer equipped with an Intel Core 2 CPU 6600 2 at 2.40 GHz with 3.24GB memory running under Windows XP

[2]http://www-users.cs.york.ac.uk/ñanuel/yetidocs/dotnet/YetiTestAssemblies.zip

TABLE I
NUMBER OF SEEDED FAULTS FOUND BY YETI. TESTING SESSIONS OF 100 SECONDS.

| Class | # methods | # attributes | # Seeded Faults | Max # Faults YETI found |
|---|---|---|---|---|
| YetiTestAssembly1.exe | 6 | 2 | 3 | 2 (67%) |
| YetiTestAssembly2.exe | 7 | 2 | 3 | 2 (67%) |
| YetiTestAssembly3.exe | 7 | 3 | 3 | 3 (100%) |
| YetiTestAssembly4.exe | 14 | 5 | 7 | 5 (71%) |
| YetiTestAssembly5.exe | 20 | 5 | 10 | 8 (80%) |
| Total | 54 | 17 | 26 | 20 (77%) |

TABLE II
STATISTICS ABOUT TESTS WITH SEEDED FAULTS. TESTING SESSIONS OF 100 SECONDS.

| Class | # Method calls to first fault | Time to first fault (ms) | # Failures | # method calls |
|---|---|---|---|---|
| YetiTestAssembly1.exe | 18.2 | 41.9 | 28734 | 508114 |
| YetiTestAssembly2.exe | 138.9 | 93.3 | 5673 | 563401 |
| YetiTestAssembly3.exe | 83.6 | 73.4 | 8549 | 534354 |
| YetiTestAssembly4.exe | 72.2 | 93.3 | 5104 | 368856 |
| YetiTestAssembly5.exe | 41.7 | 62.5 | 8030 | 447874 |

Professional Service Pack 3. Each test was run 10 times and consisted of a 10-second testing session.

One of the issues with testing such libraries is that they contain calls that make the system quit. As a consequence CsharpReflexiveLayer crashed many times due to random system calls – such as Process.Kill() –forced it to quit. Another reason was that calls could affected the design and implementation of ChsarpReflexiveLayer. For example the testing procedure made calls to the System.IO.StreamWriter class and these calls always affected the part of CsharpReflexiveLayer code that writes the created values in the report file. Testing these assemblies was however the best way to discover any limitations of the tool.

The graphs in Table III depict plausible faults and not certain faults as each fault should be double checked with what the documentation specifies for each class. Such testing sessions are simple robustness testing and in a production environment, all valid exception traces should be kept and reused later on to indicate that they do not correspond to a real fault.

When testing System.dll no plateau was reached due to the length of the testing session. THis was however necessary to obtain testing sessions that did not crash.

Testing the mscorlib.dll assembly did not result in many crashes of the CsharpReflexiveLayer. The tool shows similar performances. The main difference is the fact that the Relevant Failures number is relatively small in comparison to the equivalent when testing System.dll and it also reaches a plateau very quickly. The reason for this is that the traces of the exceptions raised by mscorlib.dll had a significant number of reandom characters and strings that confused the communication layer.

Testing System.dll and mscorlib.dll outlined the need for YETI to process automatically documentation in order to extract information about expected exceptions because such information is not available through C# interfaces. YETI cannot thus decide definitively if an exception of a method call should be characterised as a bug or not.

It is worth mentioning that the Java application of the binding never crashed or raised any exception at any testing execution.

## V. RELATED WORK

With programs becoming increasingly complex, the process of testing them has to become smarter and more efficient. Automated random testing is a technique that is cheap to run and proved to find bugs in Java libraries and programs [3], [4], in Haskell programs [5].

YETI is the latest in a serie of recent random testing tools like JCrasher [4], Eclat [3], Jtest [6], Jartege [7], RUTE-J [8], and in particular AutoTest [9]–[11]. 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 first characteristics comes from the meta-model that YETI uses that is not bound to a specific paradigm. It also implies that some domain-specific optimizations [12]–[15] will not be possible without additional support because YETI does not know values or language-specific constructs.

The second point is the truly innovative point. All other approaches considered the testing tool to be a component that would run by itself without further interactions. In our experience the runtime monitoring has become the main means of knowing how the testing session evolves and possibly to improve it by modifying some parameters.

To illustrate the last point, Randoop [16] produced 4 millions test of .NET programs in 150 hours. While the intent is a bit different because we do not use feedback to generate test cases, it takes around 12mn to perform the same amount of method calls on .NET programs in YETI – in an admittedly uncontrolled way. Further evaluations should be conducted to compare effectiveness of both approaches.

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.

TABLE III
STATISTICS ABOUT TESTS OF SYSTEM.DLL AND MSCCORLIB.DLL. TESTING SESSIONS OF 10 SECONDS.

| Class | # Method calls to first fault | Time to first fault (ms) | # Failures | # method calls |
|---|---|---|---|---|
| System.dll | 14 | 107.8 | 487.9 | 2269 |
| mscorlib.dll | 14.4 | 146.8 | 553 | 2231.7 |

## VI. LIMITATIONS AND FUTURE WORK

The approach currently has a few limitations:

- The socket communication can mess up with special characters used in exception traces. As we previously mentioned special characters might interfere with the regular process of the socket communication because it is text-based. It is likely that another format might be better for such communications. This will be investigated in the future.
- There is no way of stopping infinite loops in the .NET binding. This is a serious limitation for the .NET binding at the moment as it is not possible to test some classes due to that limitation. A clean implementation is however much more difficult to realize, but we are planning to do it in the near future.
- There is no restart of the system after the .NET part crashed. Currently, if the .NET part crashes, we do not restart the program. This is a limitation we need to overcome in order to test realistically the system libraries.
- It is time-consuming to interpret failures as there is no parsing of the textual documentation. Such a parsing is going to be necessary if we want to provide a significant support for .NET users. It would also benefit other languages users (even Java) and make the interpretation of the testing much more reliable. It is however a challenging problem as documentation is typically informally written.

The efficiency of the socket-based communication also made clear that using socket-based communication between the program under test and the testing infrastructure was a viable alternative (100% overhead only). This means that even in the Java biding it could be interesting to use such an approach for testing classes that may quit the program. While this is seldom the case for libraries, this might be useful for end-user classes.

## VII. 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.

On .NET code, YETI runs with unparalleled speed –around $10^6$ calls in 3 minutes on fast code – to date and provides a graphical user interface that allows test engineers to diagnose what happens while testing and thus be able to change parameters used in the testing session while it proceeds.

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. We also tested core .NET libraries and showed some limitations of YETI.

Future work will fix a couple of issues found through experiments. In particular, we plan to enrich YETI with an automated parser for documentation in order to help with the automated identifications of bugs.

## REFERENCES


[1] M. Oriol, "The york extendible testing infrastructure (yeti)," in *Submitted to Fundamental Approaches to Software Engineering (FASE'10)*, March 2010.

[2] I. Ciupa, M. Oriol, B. Meyer, and A. Pretschner, "Finding faults: Manual testing vs. random+ testing vs. user reports," in *IEEE International Symposium on Software Reliability Engineering (ISSRE)*, Nov 2008.

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

[5] K. Claessen and J. Hughes, "Quickcheck: A lightweight tool for random testing of haskell programs," in *ACM SIGPLAN Notices*. ACM Press, 2000, pp. 268–279.

[6] "Jtest. parasoft corporation. http://www.parasoft.com/." [Online]. Available: http://www.parasoft.com/jsp/products/home.jsp?product=Jtest

[7] 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

[8] J. H. Andrews, S. Haldar, Y. Lei, and F. C. H. Li, "Tool support for randomized unit testing," in *RT '06: Proceedings of the 1st international workshop on Random testing*. New York, NY, USA: ACM Press, 2006, pp. 36–45.

[9] I. Ciupa, A. Leitner, M. Oriol, and B. Meyer, "Experimental assessment of random testing for object-oriented software," in *Proceedings of ISSTA'07: International Symposium on Software Testing and Analysis 2007*, 2007.

[10] ——, "ARTOO: Adaptive Random Testing for Object-Oriented Software," in *International Conference on Software Engineering (ICSE 2008)*, april 2008.

[11] 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.

[12] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball, "Feedback-directed random test generation," in *ICSE '07: Proceedings of the 29th International Conference on Software Engineering*, 2007. [Online]. Available: http://people.csail.mit.edu/cpacheco/

[13] P. Godefroid, N. Klarlund, and K. Sen, "Dart: directed automated random testing," in *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*. New York, NY, USA: ACM Press, 2005, pp. 213–223.

[14] T. Chen, R. Merkel, P. Wong, and G. Eddy, "Adaptive random testing through dynamic partitioning," in *Proceedings of the Fourth International Conference on Quality Software*, vol. 00. Los Alamitos, CA, USA: IEEE Computer Society, 2004, pp. 79 – 86. [Online]. Available: http://ieeexplore.ieee.org/search/wrapper.jsp?arnumber=1357947

[15] T. Y. Chen and R. Merkel, "Quasi-random testing," in *ASE '05: Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*. New York, NY, USA: ACM Press, 2005, pp. 309–312.

```
namespace YetiTestAssembly1{...
public int methodReturnInt1(int r, int t)
{
    Contract.Ensures(Contract.Result<int>() != 0);
    r = t;
    return r;
}

public void setDoubleAttr1(double d)
{
    Contract.Ensures(d > 10.5);
    doubleAttr1 = d;
}
public float methodForFloat1(Object1 ob, char c,
                                          long l)
{
    Contract.Requires(ob != null && c != 'm');
    Contract.Ensures(l > 100);
    float f = (Single)l;
    return f;
}}

namespace YetiTestAssembly2{...
public int methodReturnInt2(int r, int t)
{
    Contract.Ensures(Contract.Result<int>() > 5);
    r = t;
    return r;
}

public void setOb2Attr(Object2 d)
{
    Contract.Ensures(d !=null);
    ob2 = d;
}
public float methodForFloat2(double ob, int l)
{
    Contract.Requires(ob != 0.0);
    Contract.Ensures(Contract.Result<float>() != 0);
    float f = (Single)(l*ob);
    return f;
}}

namespace YetiTestAssembly3{...
public static int methodReturnInt3(int r, int t)
{
    Contract.Requires(t != 4);
    Contract.Ensures(Contract.Result<int>() > 1);
    r = t;
    return r;
}

public void setOb3Attr(Object3 d, char c, String s)
{
    Contract.Ensures(d !=null && c!='m');
    ob3 = d;
}
public float methodForFloat3(double ob, int l)
{
    Contract.Requires(ob != 0.0);
    Contract.Ensures(Contract.Result<float>() != 100);
    float f = (Single)(l*ob);
    return f;
}}
```

Fig. 4. Seeded bugs examples.

[16] C. Pacheco, S. K. Lahiri, and T. Ball, "Finding errors in .net with feedback-directed random testing," in *ISSTA '08: Proceedings of the 2008 international symposium on Software testing and analysis*. New York, NY, USA: ACM, 2008, pp. 87–96.

PLACE PHOTO HERE

**Manuel Oriol** Biography text here.

**Sotirios Tassis** Biography text here.

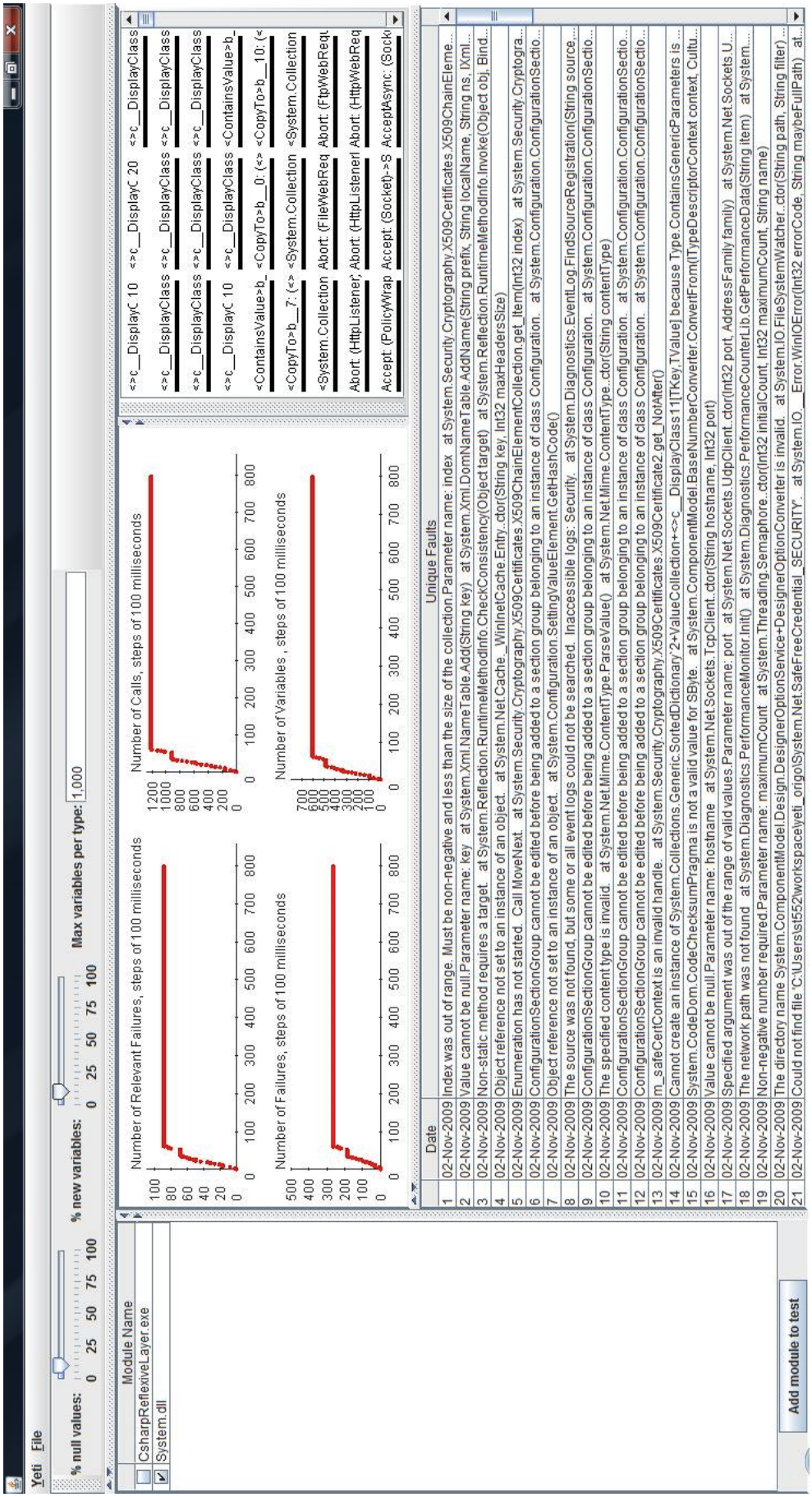


Fig. 2. YETI graphical user interface.