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 [] 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 library.

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. We eventually conclude in Section VI.

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

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

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, strategies and the Java binding. As explained in Section II, the core infrastructure provides extendibility trough specialization. To create specific strategies or language bindings, one subclasses abstract Java classes.

A. Using YETI

YETI is a tool that can be launched on the command-line. A typical call of YETI is:

```
java yeti.Yeti -Java -yetiPath=. -time=10mn -randomPlus
-testModules=java.lang.String:java.lang.StringBuilder
```

The options used on this command-line have the following meaning: `-Java` indicates that the tested program is in Java, `-yetiPath=.` indicate that classes in the current directory (and its subdirectories) will be preloaded, `-time=10mn` indicates that the testing session will last 10 minutes, `-randomPlus` indicates that the strategy random+ will be used, and `-testModules=java.lang.String` indicates that both `String` and `StringBuilder` will both be tested.

While testing, traces of faults found are output in the terminal. For example:

```
Exception 5
java.lang.NullPointerException
        at java.lang.String.replace(String.java:2207)
```

At the end of the testing sessions, YETI outputs generated test cases reproducing the faults found during the testing session as well:

```
...
public static void test_5() throws Exception {
double v0=0.0d; // time:1254919729044
java.lang.String v1=java.lang.String.valueOf(v0);
                                    // time:1254919729044
java.lang.String v25=new java.lang.String();
                                    // time:1254919729106
java.lang.String v26=v25.replace(null,v1);
                                    // time:1254919729106
/**BUG FOUND: RUNTIME EXCEPTION**/ // time:1254919729114
/**YETI EXCEPTION - START
java.lang.NullPointerException
        at java.lang.String.replace(String.java:2207)
YETI EXCEPTION - END**/
/** original locs: 59 minimal locs: 4**/
}
...
/** Non-Unique bugs: 223, Unique Bugs: 104,
Logs size (locs): 2172**/
/** Testing Session finished, number of tests:1178,
time: 1007ms, number of failures: 223**/
/** Processing time: 3119ms **/
/** Testing finished **/
```

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 paramater 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 panel specific to the current strategy (in the example, the considers using "interesting" values when possible) and 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 for being able to test 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 and presents results in the form of a colored gauge where

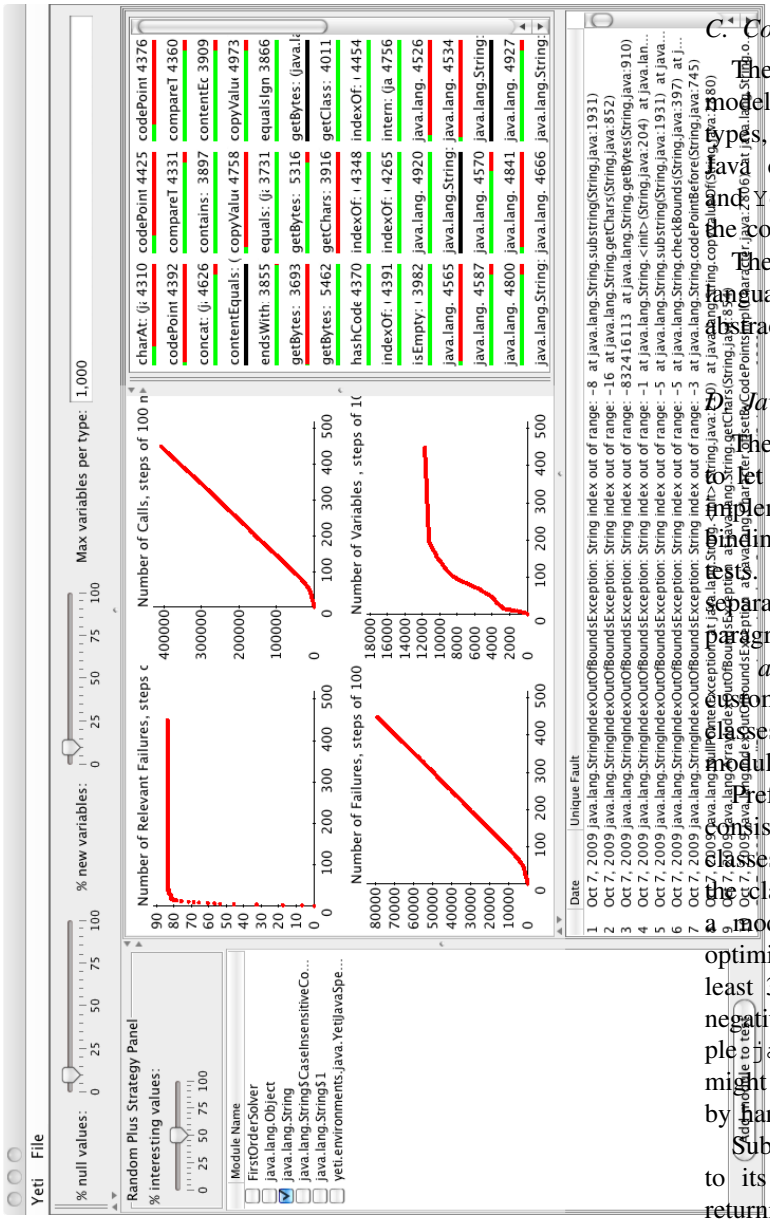

Fig. 2. YETI graphical user interface.

black indicates that the routine was not tested, green is the proportion of routines tested successfully, yellow represents routine calls that cannot be interpreted – for example YETI had to stop a thread –, and red indicates failures.

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. In ten minutes testing sessions of `java.lang.String` on a dual core MacBook Pro, the slowdown incurred by the GUI was 4.4%.

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 so that the constraints on Figure **??** are naturally respected.

The primitives and instructions are implemented either in language-specific classes or in strategy-specific classes through abstract classes that must be implemented.

D. Java Binding

The Java binding redefines classes from the base framework to let YETI test Java programs. It is currently the reference implementation for making new bindings for YETI. The Java binding uses class loaders to find definitions of classes it tests. It uses reflection to make calls. Tests are run in a separate thread so that infinite loops can be stopped. The next paragraphs explain each of these points in more detail.

a) Custom class loader.: The Java binding defines a custom class loader mainly to perform two tasks: prefetching classes in the `-yetiPath` option and create types and modules from classes loaded in YETI.

Prefetching classes is effected as soon as YETI starts, it consists in loading all classes in a given path. Each of these classes then defines both a module and a type. None of the classes in the transitive closure of classes defines either a module or a type. This is mainly due to performance optimizations as this would automatically result in loading at least 30 classes in the system, some of which have a very negative impact on performances of the system (for example `java.lang.StringBuffer`). Instead, test engineers might decide to load such classes – or other helper classes – by hand in the graphical user interface.

Subsequently, each method of each loaded class is added to its module, while any method (including constructors) returning an object of a given type is added as a "constructor" for that type.

b) Reflection.: The Java binding subclasses `YetiRoutine` with two classes: `YetiJavaConstructor` and `YetiJavaMethod`. Each of these classes uses reflection to make the calls.

c) Threads.: The Java binding uses two threads: a worker thread to perform tests and a second to control – and possibly stop –the first one. Associated to the thread group of the worker thread, we use the Java security model and do not grant any permission. That way, the code cannot create files open sockets or in general do any potentially dangerous operation. The only operation not ruled out is exiting the program. Note also that we forbid the system to test `wait`, `notify`, and `notifyAll`.

E. Strategies

By default YETI contains four main strategies:

Pure random strategy that generates calls and selects values at random. Two main probabilities can be used: the

percentage of null values, and the percentage of newly created objects to use when testing.

Randomadds the utilisation of "interesting" values. For the Java binding such values for integers contain `MAX_INT`, `MIN_INT`, all values between -10 and 10 etc. It also defines the probability to use such interesting values.

Random+ Decreasing strategy where all three probabilities start at 100% and then linearly decrease until they are at 0 at the end of the session.

Random+ Periodic strategy where all three probabilities periodically decrease and increase over the testing session.

While the first two strategies were described previously in literature [2], the last two are new as nobody thought of modifying the probabilities over time.

IV. Evaluation

To evaluate the performances of our approach, we first evaluate test its raw speed with a test class that we made for the occasion. We also ran two series of tests: we first tested all classes in `java.lang` 10^6 times each. We also tested a package from iText[1] a well known library – 1,969,220 downloads on SourceForge[2], 84% positive advices, activity of 99.85% – for manipulating PDF documents in Java.

All tests were run using a MacBook Pro 2.53 GHz Intel Core 2 Duo, 4GB of 1067 MHz DDR3 of RAM, under MacOSX with the Java(TM) SE Runtime Environment (build 1.6.0_15-b03-219) – with default value of 64MB of RAM reserved.

A. Performances

To test raw performances of YETI, we have made two small classes – `Perf` and `Perf1` shown in Figure 3 – that we tested 30 times each with one million tests. Results are presented in Figures 4 and 5.

```
public class Perf{
        int i=0;
        public void test(){
                i++;
        }
}
public class Perf1{
        public void test(int i){
                int j = i;
                i++;
                assert(i>j);
        }
}
```

Fig. 3. Classes used for performances measurements.

While `Perf` does not exhibit failures, `Perf1` exhibits failures when `i` is equal to `MAX_INT` if assertions are enabled (which we chose to do in this test).

[1]http://www.lowagie.com/iText/ downloaded Sep. 1, 2009

[2]http://sourceforge.net/projects/itext/

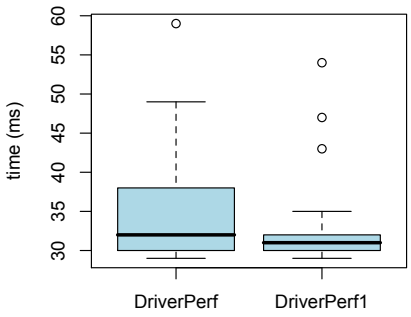

Fig. 4. Evaluation of the code of Perf and Perf1 with an external driver.

To make sure that we only tested the overhead of the infrastructure, we also made one million calls on each of the methods from another program. Figure 4 shows box and whisker plots for the two raw tests.

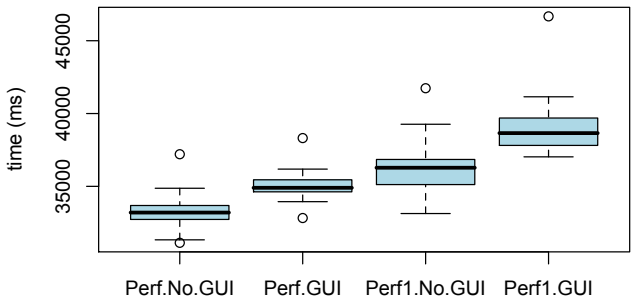

Fig. 5. Evaluation of the code of Perf and Perf1 tested by YETI, with and without GUI. Averages: 33200(Perf No GUI), 35018(Perf GUI), 36103(Perf1 No GUI), 39021(Perf1 GUI)

Figure 5 shows box and whisker plots for 30 YETI testing sessions of one million tests made on `Perf` and `Perf1`, both with and without the GUI. We can see that there is a slowdown of more than a factor 1000 over the direct invocation. The GUI also incurs respectively a 5.5% and 8.1% overhead.

B. Testing `java.lang`

We tested all classes in `java.lang` with YETI. For each class we requested 100000 tests. Table I shows our results. The first column of numbers indicates the total number of throwables that the tests triggered. We then classified each throwable as either a fault or not. This is due to programmers not necessarily declaring runtime exceptions but rather indicating in the documentation that these exceptions are normal. For throwables that where not ruled out, we then classify them by

TABLE I
RESULTS OF TESTING JAVA.LANG

	Total throwables	Faults	NullPointer	NoClassDefFoundError	IndexOutOfBounds	AssertionError	IllegalArgument
Boolean	1	0	0				
Byte	2	2	2				
Character	33	1	1				
Character.Subset	0	0					
Character.UnicodeBlock	2	0					
Class	13	3	3				
ClassLoader	10	10	8	2			
Compiler	0	0					
Double	4	2	2				
Enum	2	0					
Float	4	2	2				
InheritableThreadLocal	0						
Integer	2	2	2				
Long	2	2	2				
Math	0						
Number	0						
Object	0						
Package	1	1	1				
Process	0						
ProcessBuilder	5	2	2				
Runtime	2	0					
RuntimePermission	4	0					
SecurityManager	39	0					
Short	2	2	2				
StackTraceElement	2	0					
StrictMath	0						
String	87	5			2	3	
StringBuffer	57	3			2		
StringBuilder	36	2	1				
System	5	0					
Thread	18	5	5				
ThreadGroup	4	0					
ThreadLocal	0						
Throwable	2	0					
Void	0						
EnumConstantNotPresentException	1	1	1				
All Other Exceptions	0	0	0				
All Errors	0	0	0				
Total	340	45	34	2	4	3	2
Percentages			75.6	4.4	8.9	6.7	4.4

column. Note that the documentation of certain classes states up front that some kind of exceptions are to be expected – for example `NullPointerException` in `String`. Other classes declare all runtime exceptions. This indicates that several teams actually collaborated to make this API.

The small number of faults other than `NullPointerExceptions`, shows the overall quality of `java.lang` in terms of API and implementation. Most exception seem indeed to be omitted in the documentation rather than real bugs in the system.

It is however compelling to see that YETI can actually uncover 45 faults in a library as used and tested as `java.lang`. To obtain more data to assess how useful YETI would be for regular developers, we present similar results for a project slightly less tested in the next section.

C. Testing `com.lowagie.text` from iText

TABLE II
RESULTS OF TESTING COM.LOWAGIE.TEXT

	Total throwables	Number of Faults	NullPointer	IndexOutOfBounds	NumberFormatException	IllegalArgumentException	No class def found	ClassCast	IllegalState
Anchor									
Annotation	1	1	1						
Cell	20	5	2		2	1			
Chapter									
ChapterAutoNumber									
Chunk	3	3	1			1	1		
Document	18	18	4						
DocWriter									
ElementTags									
Font	3	3	1				2		
FontFactory	4	4	3						
FontFactoryImp	1	1							
GreekList									
Header									
HeaderFooter									
Image	4	4	3	1					
ImgCCITT									
ImgJBIG2	1	1	1						
ImgRaw									
ImgTemplate									
ImgWMF	2	2	2						
Jpeg	2	2	2						
Jpeg2000	2	2	2						
List	1	1	1						
ListItem	3	3	1				1	1	
MarkedObject	1	1	1						
MarkedSection	4	3	2	1					
Meta	2	2	2						
PageSize	1	1							
Paragraph	2	2						2	
Phrase	4	4	2	1				1	
Rectangle	3	3	2						
RectangleReadOnly	17	17							
RomanList	1	1					1		
Row									
Section	6	3		2					1
SimpleCell	1	1							
SimpleTable	1	1				1			
SpecialSymbol									
Table	26	26	3	13		2	1		
Utilities	4	4	2	2					
ZapfDingbatsList									
ZapfDingbatsNumberList									
All Exceptions									
Total	138	119	38	20	2	5	6	4	1
Percentage		86.2	31.9	16.8	1.7	4.2	5.0	3.4	0.8

In order to have a more representative project for end-users, we tested the package `com.lowagie.text` from iText. Because this package is quite time intensive to test, we decided to test all classes in the package at the same time for only a 100000 tests. The testing session lasted for 15 minutes and output 138 failures. Comparatively to the previous section, the code almost did not declare any runtime exception. Only five classes – `Cell`, `MarkedSection`, `Phrase`, `RectangleReadOnly`, and `Section` – did it

in an informal way.

It is worth noting that two classes – `RectangleReadOnly` and `Document` – exhibit a high number of project-defined errors. We investigated and found out that these two classes have poor documentation rather than poor code. This is to be expected in a free open source project.

Eventually, even over long running sessions (when logs are not stored within YETI) the number of routine calls effected, grows linearly with time as shown in Figure 6 for a 50 minute session testing `java.lang.String`.

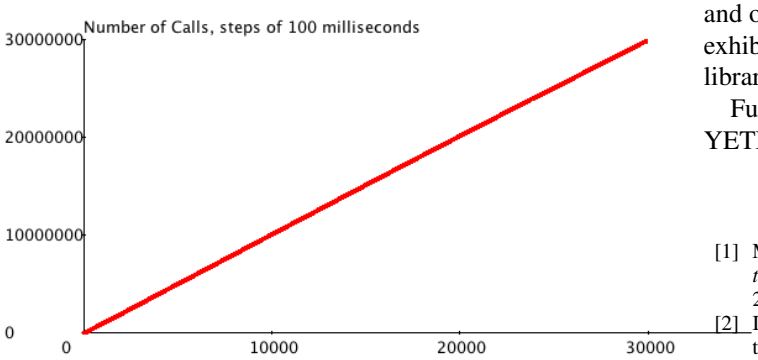

Fig. 6. Number of calls over time while testing `java.lang.String`

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.

The last characteristics clearly depends on the reference implementation being in Java and testing Java programs – other bindings might exhibit slower performances. It is however worth noting that this has a direct impact on the capability for the tool to test thoroughly big amount of code in short period of time.

VI. Conclusions

This article presented YETI, a new tool to run automated random testing sessions on multiple programming languages. The current reference implementation works on Java but other bindings exist for JML and for .NET.

On Java code, YETI runs with unparalleled speed –around 10^6 calls per minute 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 and types used in the testing session while it proceeds.

To validate our approach we used YETI on `java.lang` and on a package of a popular open source library. This lead to exhibit 45 faults in `java.lang` and 119 in the open source library.

Future work will focus on bringing more languages into YETI and improve further its interface as well as its strategies.

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.

Manuel Oriol Biography text here.

Sotirios Tassis Biography text here.