The York Extensible Testing Infrastructure (YETI)

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

***Abstract*—Random testing is a simple, effective technique for finding faults. It can be automated and then does not need any input from the tester and only reports results once the testing session is over.**

This article presents the York Extensible Testing Infrastructure (YETI), a random testing tool whose reference implementation is in Java. YETI provides a strong decoupling between the strategies and the actual code, making its engine language agnostic. The tool runs at a high level of performances with 10^6 calls per minute on Java code. It benefits from a graphical user interface and allows testers to interact with the testing process in real-time. This is useful to abort or modify the parameters of the testing session when needed. We illustrate the efficiency of YETI with a study testing all classes in java.lang and some classes in the well known open source project iText.

***Keywords*-automated; testing; random;**

I. Introduction

Automated random testing is a methodology often neglected by programmers and software testers because it seems overly simple. It has however advantages over other techniques: it is completely unbiased, it makes a large number of calls over a short period of time, and it can be completely automated.

Two issues have, however, prevented automated random testing from being more widely accepted:

- All tools are language specific and whether the results would also apply to other languages is always in question.
- It is difficult to know a priori when to stop testing.

The York Extensible Testing Infrastructure (YETI) provides a framework for executing random testing sessions. YETI supports multiple programming languages through a *language-agnostic meta model*. Testing strategies apply to all supported languages thanks to a strong decoupling between the strategies used and the programming language binding.

In YETI, 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 Java, if programmers use `assert` statements then violations are interpreted as failures. If programmers do not use assert statements, a testing session with YETI is a robustness test that considers undeclared runtime exceptions as failures.

Unlike competitors, YETI also supports a *graphical user interface* (GUI) that allows test engineers to monitor the testing session and modify some of its characteristics while testing. In particular, it allows the software tester to evaluate the likelihood of discovering further faults in the program with the present parameters by observing the evolution of the number of bugs found in real-time. YETI is available as a free, open-source project.[1]

To validate our approach we made one million calls at random on each class in `java.lang` and it discovered 45 faults. We also tested a package of iText and it discovered 120 faults. Eventually, we present scenarios where the GUI allows a finer control over the testing session.

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

II. Model

This section presents the meta-model of YETI. This meta-model describes elements of a programming language from a random testing perspective.

YETI uses four main notions (see Figure 1): *routines*, *types*, *variables*, and *modules*. A type is a collection of variables[2] and routines that return values (of that type). A routine is a computation unit that uses variables of a given type and returns values of a given type. A variable is a couple made of a label and a value. A module is a collection of routines.

Note that routines contain typing indications. These are the information needed by the infrastructure to make calls and to create or store instances.

Subtyping ($\trianglelefteq$) is a relationship between two types and implies that the two types conform one to another as shown in the middle part of Figure 1. The rule $(PROP)$ is non-standard because it does not try to establish the type but rather uses it to establish the existence of routines and attributes. This is because our model is descriptive of the actual programs rather than used to define a type judgement. We assume that YETI already knows the type of the variables and either creates them or uses an instance returned by a method call. As an example, such a variable could be

[1] http://www.yetitest.org

[2] We use variables instead of values to represent values previously generated by routines. This implies that they can contain duplicates.

YETI Meta-Model:

$$
\begin{array}{lll r}
v, v_0, ..., v_k & & & values \\
n, n_0, ..., n_k, r & & & names \\
f, f_0, ..., f_k, r & & & failures \\
R & ::= & (v_1 : T_1 \times ... \times v_k : T_k) \rightarrow v_0 : T_0 & routines \\
V & ::= & (n, v) & variables \\
M & ::= & (R_1, ..., R_k) & modules \\
T & ::= & ((R_1, ..., R_k), (V_1, ..., V_l)) & types \\
data & ::= & R|V|M|T & data \\
L & ::= & ()|(data_1, ..., data_n)|L_1 \oplus L_2|L \oplus data & lists
\end{array}
$$

Subtyping rules in YETI:

$$\frac{T_1 \trianglelefteq T_2 \quad T_2 \trianglelefteq T_3}{T_1 \trianglelefteq T_3} \quad (TRANS)$$

$$T \trianglelefteq T \quad (REFL)$$

$$\frac{T_1 \trianglelefteq T_2 \quad T_1 = ((R_{11}, ..., R_{1k}), (V_{11}, ..., V_{1l})) \quad T_2 = ((R_{21}, ..., R_{2m}), (V_{21}, ..., V_{2n}))}{\forall i \exists j | R_{1i} = R_{2j} \quad \forall i \exists j | V_{1i} = V_{2j}} \quad (PROP)$$

YETI primitives and instruction:

$$
\begin{array}{rcl}
primitive & ::= & selectVariableFromType(T) \\
 & | & addVariableToType(T, V) \\
 & | & selectRoutineFromModule(M) \\
 & | & executeRoutineWithArguments(R, l) \\
 & | & oracle(primitive) \\
 & | & report(f) \\
instruction & ::= & primitive \\
 & | & n\mathbf{:=}primitive|n\mathbf{:=}data|n\mathbf{:=}L \\
 & | & instruction \ \mathbf{;} \ instruction \\
 & | & \mathbf{forall} \ n \ \mathbf{in} \ list \ \mathbf{do} \ instruction \ \mathbf{end} \\
 & | & \mathbf{ifIsFailure} \ instruction \ \mathbf{then} \ instruction \ \mathbf{else} \ instruction \\
 & | & \mathbf{whileNotFinished}\{instruction\}
\end{array}
$$

$$
\begin{array}{rl}
selectVariableFromType : & T \mapsto V \\
addVariableToType : & T \times V \mapsto T \\
selectRoutineFromModule : & M \mapsto R \\
executeRoutineWithArguments : & R \times l \mapsto V|f \\
oracle : & V|f \mapsto V|f \\
report : & f \mapsto \emptyset
\end{array}
$$

Figure 1. YETI meta-model formal definitions.

classified as an instance of a super type of its defining type and not as an instance of its defining type. If this model defined a programming language it would be incomplete. In our case we only aim at soundness – method calls should not fail because of typing issues, but only because of bugs. This allows YETI to reason about incomplete systems and to use only the types it knows and the available interfaces.[3]

[3]This has the practical implication that we do not have to consider the transitive closure of types in YETI, thus allowing to exclude classes that impact the performances in a negative way.

A. Modelling Random Testing

To model random testing we add primitives to the simple model outlined above. The lower part of Figure 1 presents the primitives needed by random testing. Unless specified otherwise, each instruction has an intuitive definition.

When YETI tests a module it iteratively and randomly selects one of the module's routines and executes with random values coming from type specific value pools (see Figure 2). The result values of each routine are either reported as faults or otherwise added to the corresponding value pool. Thus the longer the testing runs the value pools become richer and may drive the routines into previously uncovered regions.

```
$whileNotFinished\{$
  $r$:=$selectRoutineFromModule(M_{underTest})$;
  $l$:=();
  forall $t$ in $(T_1...T_n)$
      $|i \neq 0 \; and \; r = (v_1 : T_1 \times ... \times v_k : T_k) \rightarrow v_0 : T_0$
  do
   $l$:=$l \oplus selectVariableFromType(t)$
  end;
  $res$:=$oracle(executeRoutineWithArguments(r, l))$;
  ifIsFailure $res$ then
   $report(res)$ else $addVariableToType(T_0, res)$
$\}$
```

Figure 2. Main random testing algorithm.

While instructions have a precise semantics we assume that primitives are actually user-defined. User-specific strategies and bindings are thus easy to define. For example, in the case of a pure random strategy on Java, the primitives would have the following semantics:

- $selectVariableFromType(T)$ selects one of the variables of T at random and returns it.
- $addVariableToType(T, V)$ adds V to T and returns the result.
- $selectRoutineFromModule(M)$ returns one of the routines in M at random.
- $executeRoutineWithArguments(R, l)$ makes the actual execution of the code and returns a created variable (if any).
- $oracle(V|f)$ decides on the validity of a failure or a value according to the semantics of the target programming language.
- $report(f)$ aggregates failures and only actually report unique failures.

Other semantics are however possible. For example Random+ [1] either selects values in the existing pool of objects, generates new ones on the fly, or takes "interesting" values. This would then mean a different semantics for $selectVariableFromType$.

As previously shown, several specializations of the model are possible. The next chapter presents both the implementation of the core of YETI and the implementation of the Java binding.

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 bound to YETI. More details can be found on the specifics of our .NET binding in a separate publication [2].

YETI 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 through 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 and typical call of YETI is:

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

The options used on this command-line have the following meaning: `-Java` indicates that the tested program is in Java, `-yetiPath=.` indicates 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 we are testing `java.lang.String`.

While testing, traces of faults found are output to 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:

```
...
@Test public 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 keep no logs except exception traces, or `-rawlogs` to output the logs to the terminal.

B. Graphical User Interface (GUI)

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

Figure 3 shows YETI's graphical user interface when using the random+ strategy. At the top left 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, 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 subpanel specific to the current strategy (in the example, it considers how to use "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 on demand when 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 testing a routine.

In the central panel, four graphs describe the evolution of the system over time: 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 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% (see Section IV for a further evaluation of the performances of the GUI).

C. Core Architecture

The core architecture of YETI is 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 1 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 extended.

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 bindings in YETI. The Java binding uses class loaders to find definitions of classes it tests. It uses reflection to make calls – for languages not supporting reflection (e.g. C) we need to generate a reflexive layer and extract information through the source code. 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.

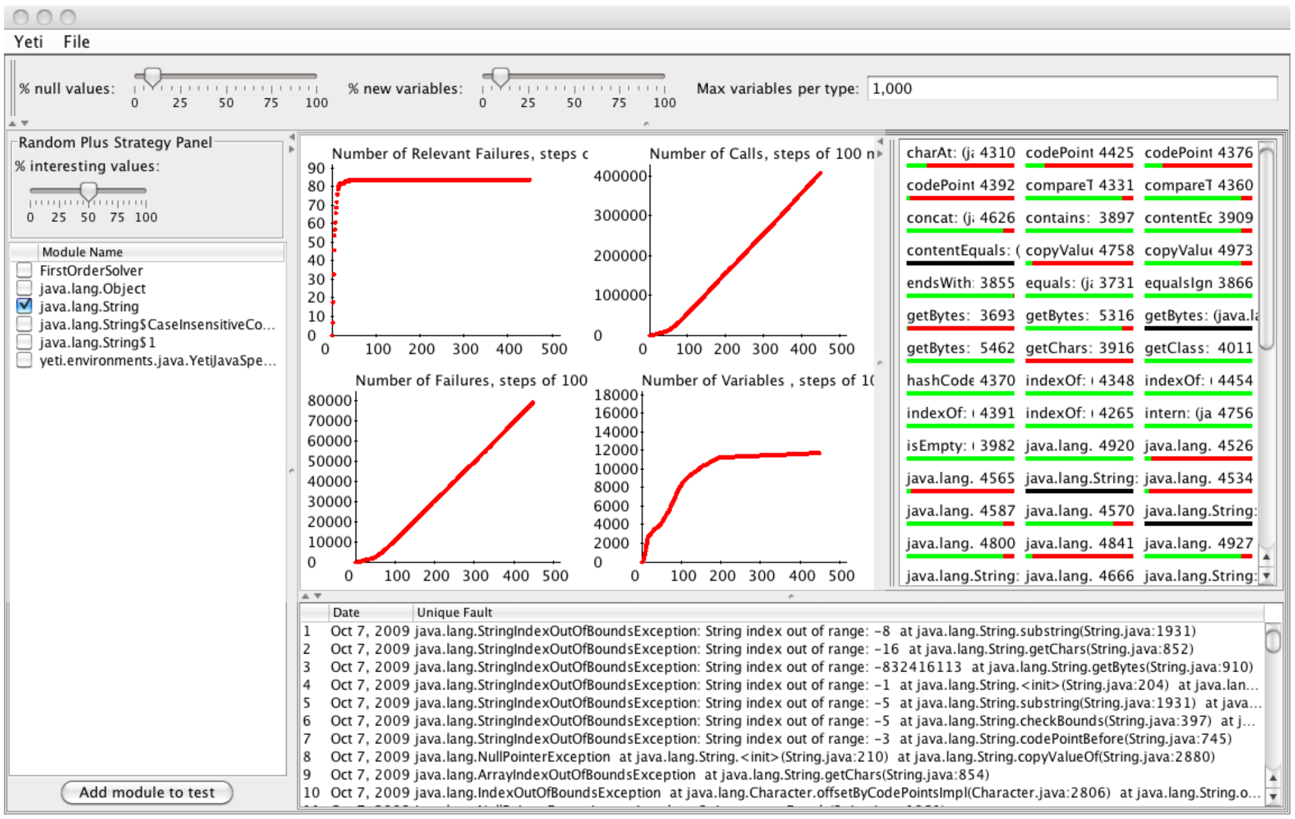
Custom class loader: The Java binding defines a custom class loader mainly to perform two tasks: prefetching classes found following the `-yetiPath` option and create types and modules from classes loaded in YETI.

As soon as YETI starts it prefetches the classes: it loads all classes in a given path. Each of these classes then defines both a module and a type. By default, no other class in the transitive closure 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 – through the graphical user interface or the command-line.

Subsequently to loading, 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.

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

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`.


Yeti
File
Random Plus Strategy Panel
Number of Relevant Failures, steps c
Number of Calls, steps of 100 n
Number of Failures, steps of 100
Number of Variables , steps of 1(
Add module to test

% null values: 0 25 50 75 100

% new variables: 0 25 50 75 100

Max variables per type: 1,000

% interesting values: 0 25 50 75 100

	Module Name
☐	FirstOrderSolver
☐	java.lang.Object
✓	java.lang.String
☐	java.lang.String$CaseInsensitiveCo...
☐	java.lang.String$1
☐	yeti.environments.java.YetiJavaSpe...

charAt: (ja 4310	codePoint 4425	codePoint 4376
codePoint 4392	compareT 4331	compareT 4360
concat: (ja 4626	contains: 3897	contentEc 3909
contentEquals: (	copyValue 4758	copyValue 4973
endsWith: 3855	equals: (ja 3731	equalsIgn 3866
getBytes: 3693	getBytes: 5316	getBytes: (java.la
getBytes: 5462	getChars: 3916	getClass: 4011
hashCode 4370	indexOf: (4348	indexOf: (4454
indexOf: (4391	indexOf: (4265	intern: (ja 4756
isEmpty: (3982	java.lang. 4920	java.lang. 4526
java.lang. 4565	java.lang.String:	java.lang. 4534
java.lang. 4587	java.lang. 4570	java.lang.String:
java.lang. 4800	java.lang. 4841	java.lang. 4927
java.lang.String:	java.lang. 4666	java.lang.String:

```
    Date         Unique Fault
1   Oct 7, 2009 java.lang.StringIndexOutOfBoundsException: String index out of range: -8  at java.lang.String.substring(String.java:1931)
2   Oct 7, 2009 java.lang.StringIndexOutOfBoundsException: String index out of range: -16  at java.lang.String.getChars(String.java:852)
3   Oct 7, 2009 java.lang.StringIndexOutOfBoundsException: String index out of range: -832416113  at java.lang.String.getBytes(String.java:910)
4   Oct 7, 2009 java.lang.StringIndexOutOfBoundsException: String index out of range: -1  at java.lang.String.<init>(String.java:204)  at java.lan...
5   Oct 7, 2009 java.lang.StringIndexOutOfBoundsException: String index out of range: -5  at java.lang.String.substring(String.java:1931)  at java...
6   Oct 7, 2009 java.lang.StringIndexOutOfBoundsException: String index out of range: -5  at java.lang.String.checkBounds(String.java:397)  at j...
7   Oct 7, 2009 java.lang.StringIndexOutOfBoundsException: String index out of range: -3  at java.lang.String.codePointBefore(String.java:745)
8   Oct 7, 2009 java.lang.NullPointerException  at java.lang.String.<init>(String.java:210)  at java.lang.String.copyValueOf(String.java:2880)
9   Oct 7, 2009 java.lang.ArrayIndexOutOfBoundsException  at java.lang.String.getChars(String.java:854)
10  Oct 7, 2009 java.lang.IndexOutOfBoundsException  at java.lang.Character.offsetByCodePointsImpl(Character.java:2806)  at java.lang.String.o...
```

Figure 3. YETI graphical user interface.

E. Strategies

By default YETI contains four main strategies:

Pure random: is a 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.

Random+: adds 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: is a random+ 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: is a random+ strategy where all three probabilities periodically decrease and increase over the testing session.

While the first two strategies were described previously in literature [1], the last two are new as nobody thought that modifying the probabilities over time might make sense. Informal tests show that in the default settings, apart from pure random that does not find easily some bugs, all these strategies yield similar performances at finding faults.

Figure 4 shows the overall process followed when YETI needs an instance to make a call. *Pint* is the probability to

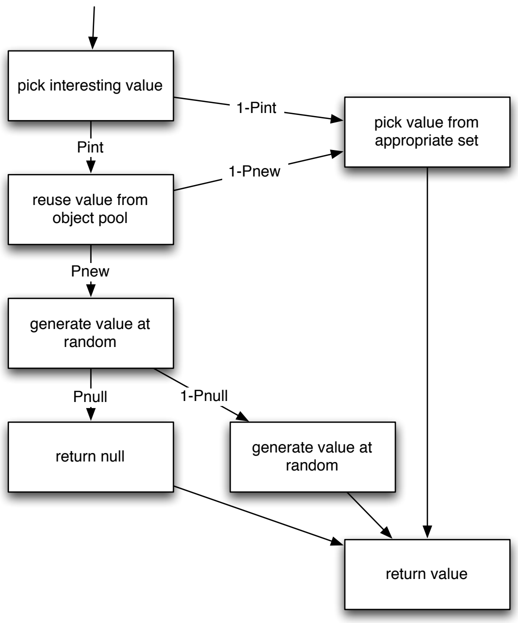


Figure 4. Generation of values.

use an interesting value, *Pnew* the probability to generate a new value and *Pnull* the probability to use a null value. These can be manipulated at runtime through the sliders shown in Figure 3.

IV. Evaluation

To evaluate the performances of our approach, we first evaluate 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[4] a well known library – 1,969,220 downloads on SourceForge[5], 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 made two small classes – `Perf` and `Perf1` shown in Figure 5 – that we tested 30 times each with one million tests. Results are presented in Figures 6 and 7.

```
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);
        }
}
```

Figure 5. Classes used for measuring performances.

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).

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 6 shows box and whisker plots for the two raw tests.

Figure 7 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.

[4]http://www.lowagie.com/iText/ downloaded Sep. 1, 2009
[5]http://sourceforge.net/projects/itext/

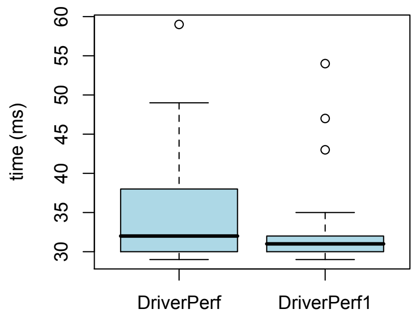


Figure 6. Evaluation of the code of Perf and Perf1 with an external driver.

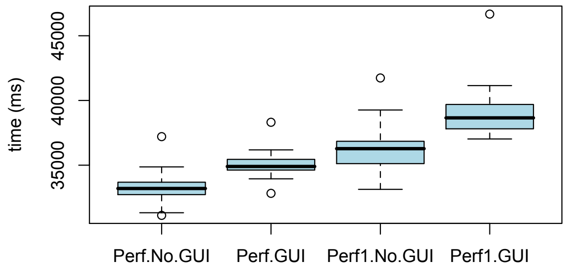


Figure 7. Evaluation of the code of Perf and Perf1 tested by YETI, with and without GUI. Averages: 33200(Perf No GUI), 35018(Perf GUI), 36103(Perf1 No GUI), 39021(Perf1 GUI)

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 8 for a 50 minute session testing `java.lang.String`.

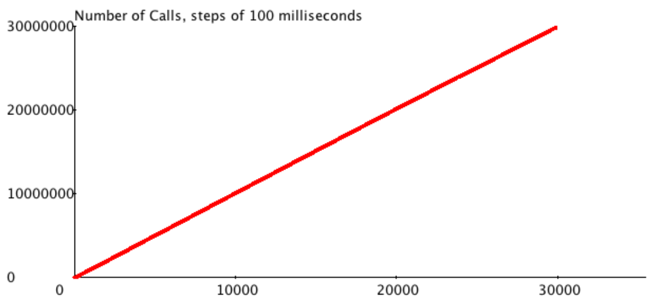


Figure 8. Number of calls over time while testing `java.lang.String`

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		1
StringBuilder	36	2	1				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

B. *Testing* `java.lang`

We tested all classes in `java.lang` with YETI. For each class we requested $1,000,000$ tests. Except for a couple of classes that were too memory intensive – such as `StringBuffer` – that we tested with $100,000$ tests only. 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 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 exceptions 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`. While these faults are real, it is to be noted that we only uncover runtime faults (runtime exceptions and errors) and it has no knowledge of methods that should be called in specific orders. Faults resulting from the misuse of such methods are still reported as faults by the TOOL. To obtain more data to assess how useful YETI would be for regular developers, we present similar results for a regular open source project in the next section.

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

In order to have a more representative project for end-users, we tested the package `com.lowagie.text` from iText. Table II shows the aggregated results of the tests. 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 unique 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 as the code often serves as documentation.

D. *Reporting in Real-Time*

Other random testing tools do not include any way of interacting with the infrastructure in real-time. YETI allows it and it benefits test engineers because they can adapt the testing process to fit their needs. In this subsection, we show two scenarios where having real-time reporting in YETI allows test engineers to adapt and circumvent issues.

```
// A class where random does not find
// the bug unless very lucky
public class MyClass{
   public int div(int i){
      return i/(4556767-i);
   }
}
// Helper class to generate the interesting
// value 4556767 in priority
public class HelpMyClass {
   public static int value(){
     return 4556767;
   }
}
```

Figure 9. A class to test.

Using results from other techniques: Figure 9 shows the code of `MyClass`, a class that only contains a method that calculates $i/(4556767-i)$ (i being its argument). Obviously, passing 4556767 as an argument will lead to a failure. When using a black-box random testing approach, for the tool to test that method with 4556767 requires it to be lucky or to wait for a potentially long time for the testing to perform. A more realistic approach is that users might launch the testing session and then, because they do not find a fault at the beginning of the session, decide to find values from the code or the documentation. In our case a software tester might then create the class `HelpMyClass` (see Figure 9) that contains a method returning that value and load it at runtime.

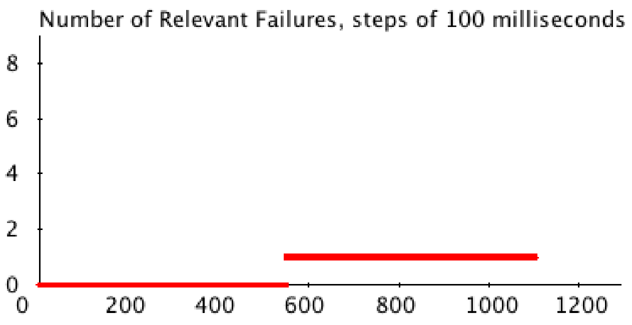


Figure 10. Faults evolution when introducing HelpMyClass at 56s, `div` had been called 3×10^5 times out of 2×10^6 total calls.

Figure 10 shows that the fault is then found immediately after the test engineer loads `HelpMyClass` into the system (56s after the beginning of testing).

Testing untested methods and adapting the strategy at runtime: One of the visible issues in Figure 3 is that the method `contentEquals` from `String` is not tested by YETI. The reason is that it uses instances of the class `StringBuffer` and even if the JVM knows it, YETI does not. The reason is that, for performances reasons, YETI does not load the whole transitive closure of types used in the tested program. The natural reaction in such a case is for the test engineer to add the module (load the class using the button in the lower-left part of the GUI).

The issue is that `StringBuffer` is a buffer that can be initialized with a pre-defined capacity (an `int`). If numerous instances are initialized using large values, then the performances drop due to the large amount of memory needed by the JVM. A recovery strategy is to limit the number of instances per type.

Figure 11 presents the graphs obtained in such a scenario. It is clearly visible in the figure that introducing `StringBuffer` has a positive impact on the number of faults. It also impacts however the performances. By reducing the number of instances per type we then improve the situation, which also allows YETI to find more faults.

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

Table II
RESULTS OF TESTING COM.LOWAGIE.TEXT

	Total throwables	Number of Faults	NullPointer	IndexOutOfBounds	NumberFormatException	IllegalArgumentException	No class def found	ClassCast	IllegalState	NegativeArraySize	Runtime	ArrayStore	StackOverflow	Project-defined
Total	138	120	38	20	2	5	6	4	1	6	1	2	2	33
Percentage		86.2	31.9	16.8	1.7	4.2	5.0	3.4	0.8	5.0	0.8	1.7	1.7	27.7

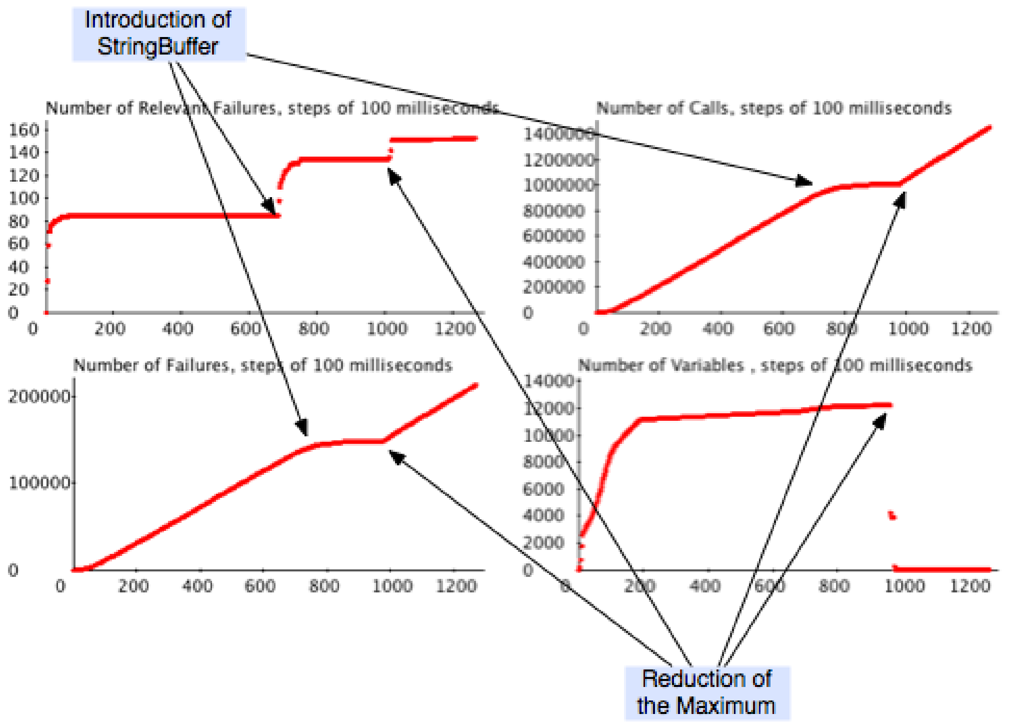


Figure 11. Testing `String`, adding `StringBuffer`, and then reducing the number of instances per type (in this case 2).

cheap to run and proved to find bugs in Java libraries and programs [3], [4], as well as 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], [10], [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], [13], [14], [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 as a component that would run by itself without further interactions. In our experiments 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 is that YETI exhibits the best performances reported so far. For example, JCrasher [4] takes around 19.9s for creating and executing 14382 test cases (around $45 * 10^3$ tests per minute) and AutoTest executes an average of 10^3 calls per minute. In the case of JCrasher, this is because each test case is written first to disk and then executed through JUnit, in the case of AutoTest, the discrepancy is yet to be explained. While it clearly depends on the reference implementation being in Java and testing Java programs – other bindings might exhibit slower performances –, it is worth noting that this has a direct impact on the capability of the tool to test thoroughly large amounts of code in short periods of time.

VI. Conclusions

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

On Java code, YETI runs at a very high speed – around 10^6 calls per minute on fast code – and provides a graphical user interface that allows test engineers to diagnose the testing process while testing. Test engineers are then 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 120 in the open source library. We also measure the performances of YETI as well as presented scenarios where test engineers would benefit from such features.

Future work will focus on bringing more languages into YETI and improve further its interface to include a method-focused mode to control the input of one method and call it in priority. A further goal will be to leverage the support for the multiple languages and allow automated random conformance testing within YETI.

References

[1] 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.

[2] M. Oriol and S. Tassis, "Testing .net code with yeti," in *15th IEEE International Conference on Engineering of Complex Computer Systems, ICECCS 2010, Oxford, United Kingdom, 22-26 March, 2010*. IEEE Computer Society, 2010.

[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.