Challenges to support automated random testing for dynamically typed languages

Stéphane Ducasse
RMoD – INRIA Lille Nord Europe, France
http://stephane.ducasse.free.fr/

Manuel Oriol
University of York, UK
manuel@cs.york.ac.uk

Alexandre Bergel
Pleiad Lab, Department of Computer Science (DCC), University of Chile, Chile
http://bergel.eu

Abstract

Automated random testing is a proven way to identify bugs and precondition violations, and this even in well tested libraries. In the context of statically typed languages, current automated random testing tools heavily take advantage of static method declaration (argument types, thrown exceptions) to constrain input domains while testing and to identify errors. For such reason, automated random testing has not been investigated in the context of dynamically typed languages. In this paper we present the key challenges that have to be addressed to support automated testing in dynamic languages.

1. Introduction

Random testing is a form of automatic testing that randomly generates data input and test cases. Random testing shines in its effectiveness to identify software faults against manual testing [CPO+11]. Random tests are appealing because they are relatively easy and cheap to obtain.

Autotest [CPL+08, COMP08] and Yeti [OT10][1] have proven that automated random testing is an effective way to identify bugs and generate test cases which reproduce them. Autotest typically finds more bugs than any kind of manual testing in very small amounts of time [COMP08]. While all types of testing find different kinds of bugs, there is an overlap between bugs found by random testing and bugs found by other techniques.

Dynamically typed languages, including Pharo, [BDN+09] should be able to take advantages of the benefits of automated random testing. Unfortunately, current automated random testing tools heavily rely on of static type annotation: method signatures, including argument types, return types and thrown exceptions, constrain input domains used by a random test. In addition, static types qualify the software faults found by random tests (*e.g.,* whether a fault is effectively a bug or a false positive). In this paper we present the key challenges and paths that further researchers may follow to support automated testing in dynamic languages.

First we present the principles of random automated testing in Java (Section 2) by explaining the strategies of YETI, one of the best automated random testing tools. We subsequently identify the challenges that dynamic languages pose and we sketch some possible tracks of solutions (Section 3). The focus of this article is not to propose a solution but to stress the challenges that have to be addressed to support automated random testing for dynamically typed languages.

2. Random Automated Testing in Java: The Facts

2.1 Random Automated Testing Principles

To explain how random testing tools for statically typed languages work, we present how the York Extensible Testing Infrastructure (YETI) – a language agnostic automated random testing tool – works.

YETI is an application coded in Java. It tests 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. YETI is a lightweight platform with around 10,000 lines of code for the core, the strategies and the Java binding and is available under BSD license.

Figure 1 shows the general process of an automated random testing tool. An instance database is created with some seeds (for example, 0, 1, -1 for numbers). Such database is used during the instance generation period. The instance generation step requires instances for both the receiver and arguments of a message. It uses class type profiles (types method declaration) which are stored in another database.

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

[Preprint of International Workshop on Smalltalk Technologies 2011.]

Then tests are created by calling methods, the tests may check the returns values and the thrown exceptions. During tests execution, when a new instance is created it may be added to the instance database. To determine whether a fault is actually found, the declared exceptions and precondition (argument types are used) are key to determine whether a fault is a bug in the analyzed software or whether it is a false positive.

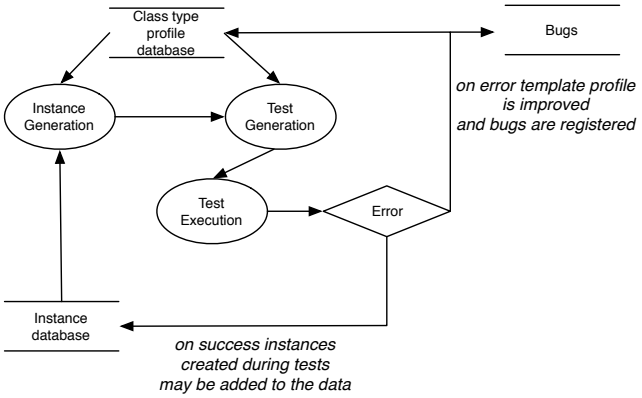

Figure 1. Automated random testing process

YETI contains three parts: the core infrastructure, the strategies, and the language-specific bindings. The core infrastructure contains the code to represent routines (methods or functions), a graph of types (class profile database), and a pool of typed objects (instance database). Routines use arguments of certain types and return an object of a certain type (if any) for which they are considered constructors.

```
// Input: Program/Strategy
// Output: found bugs
foundBugs = new Vector<Bug>();
M0 = strategy.getModuleToTest();
while (not endReached){
  R0=strategy.getRandomRoutineFromModule(M0);
  Vector<Variables> arguments =
    new Vector<Variable>();
  for(T in R0.getArgumentTypes()){
    arguments.addLast(strategy.getInstanceOfType(T));
  }
  try {
    new Variable(languageBinding.call(R0,arguments));
  } catch (Exception e) {
    if (languageBinding.representsFailure(e)){
     foundBugs.add(e);
    }
  }
}
```

Figure 2. Algorithm of automated random testing. Bolded source lines show where a dynamic language cannot provide the necessary typing information.

Similarly to other automated random testing tools, YETI follows the algorithm in Figure 2. In this algorithm, getModuleToTest, getRandomRoutineFromModule, and getInstanceOfType are defined within the strategies. How to make a call (call) and how to interpret the results (representsFailure) are both defined in the language binding.

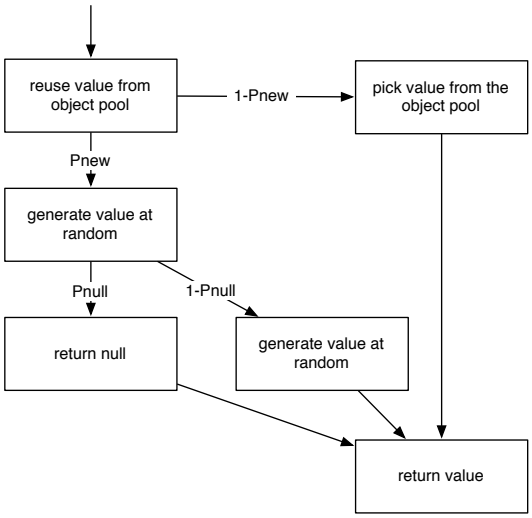

Figure 3. Generation of values.

By default YETI uses a strategy that generates calls and selects values at random. Two main probabilities can be adjusted: the percentage of null values p_{null}, and the percentage of newly created objects to use when testing p_{new}. Figure 3 shows the overall process followed when YETI needs an instance for a test and calls getInstanceOfType. By default, YETI uses 10% as a default value for both p_{new} and p_{null}.

In the Java binding, YETI uses class loaders to find definitions of classes to tests, reflection to make calls, and a separate thread to run them. Any undeclared RuntimeException or Error is interpreted as a failure and failures are grouped into unique failures by comparing their call stack trace beyond the first line. Receiver and arguments are discarded: to avoid the fact that if we would reuse sets of arguments/receiver we could trigger the same bug.

To understand how strong typing impacts the testing process, we identify 4 main areas where the typing information is used extensively. For each of these areas, we indicate whether a dynamically typed language has an easy way of supporting it:

Type description and pool of objects. Types are necessary to construct message arguments. In YETI, a type is mainly made of a list of supertypes, a list of subtypes, a list of "constructors" (by constructors we mean method returning an instance of the class), and a list of instances. Constructors are all routines that return a value of the type, and the list of instances is a pool of objects of that type.

Many dynamically typed languages offer support for listing existing instances for a given class, which can be used as example for feeding the testing tool. Supertypes and subtypes are easily and efficiently accessible via reflec-

tion. It is however more difficult to know the return type of a routine because 1) it is not declared and 2) it may vary over multiple executions.

Instance generation. In YETI, generating an instance is done through a "constructor" for such an instance. If the routine to call needs arguments, these are typed and it is easy to reuse instances of such types already present in the system through their type or create new ones through their own constructors.

Test generation and execution. To generate and execute a test, YETI either uses or generates instances of the needed types and makes a call.

Without restriction to given types for the arguments, calling a method with random instances is very unlikely to produce a meaningful test.

Feedback and bug identification. In YETI, because the type is supposedly valid due to the typing information, collecting runtime exception is meaningful as only a superficial check on the APIs is needed to make sure that unique failures are in fact bugs.

As mentioned previously, testing an untyped routine leads to calls that have a high risk of failing. It is also not useful to know that a routine fails when using arguments of types which were not foreseen to be usable there.

As we can see in the previous description, typing information is used at every single step of the process.

2.2 YETI Facts

By using such typing information YETI is able to run a million tests per minute – this represents a barely noticeable overhead over reflexive invocation. So far, YETI found thousands of bugs and was able to test many programs such as the Java core libraries, GWT [2], and all programs in the Qualitas Corpus [TAD+10]. YETI also has a graphical user interface which allows the test engineers to monitor how a testing session performed so far.

3. Challenges for Dynamic Languages

We now explore the challenges and possible solutions posed by the absence of static types to support automated random testing.

3.1 Instance generation and execution

Obtaining instances to execute the tests is key in automated random testing. Instances are necessary to feed random tests. Constructing messages (which received objects as argument) intended to be sent to object receivers are used as input of a test.

In Smalltalk, the fact that no constructor is natively supported by the language makes the generation of well formed instance challenging. One way to circumvent this lack of constructor is to use existing instances (and thus well formed) as an input for random testing. An approach is to use heuristics based on method categories and some patterns for methods. The fact that in Pharo and Squeak the method new automatically invokes the method initialize is a good step to obtain more systematically well initialized instances.

Getting instances is easily done through exploring the instances currently in the system. In Smalltalk, it is possible to get access to all the instances of a class using messages allInstances. Reflection is simple and efficient to use. Accessing existing instances can provide specific information that generated instances could not present.

However, using existing instances as input in random tests presents two issues:

1. *Partially initialization* – Since Smalltalk does not offer any guarantee on whether an object is properly initialized or not, the standard Pharo distribution contains uninitialized or partially initialized instances. Those objects have their invariant broken. As an example we have found an instance of the class Point which contains the nil value as x and y and totally unused by the system. A properly initialized point contains numerical values instead.
2. *Fragile objects* – Randomly modifying existing objects may lead to situation where the system is put in a dangerous state. Such objects may either be discarded for not being used as a random test input or be copied. Copying objects may be an option, however it may lead to an overhead in case of a deep copy, if ever possible at all.

In addition, since we do not know the supposed type of the arguments, it is not clear what instances to pass.

Using type inference. To know the argument type, one possibility is to use type inference. Different approaches are possible: lightweight with low precision [PMW09] or heavy computation (and with the problem of the availability of the approach) [SS04]. We believe that a lightweight approach is a first step to be used. As an example, RoelTyper only considers single method body to identify argument type of the method, therefore the precision of the type information is low. Another heavy approach could be to annotate methods with type information but this clearly does not scale.

Using libraries like MethodWrappers or other techniques to monitor method execution [BFJR98, Duc99] could be a possible way to collect type information. However, this is not the panacea because (1) we face a bootstrap problem - we need to use some pre-existing information (existing tests could be used to run but again the results would be driven by test quality and in absence of large coverage we may get incomplete type information). In statically type languages, the process does not require existing tests to run. (2) not all the methods should be executed randomly.

[2] http://code.google.com/webtoolkit/

Executing destructive methods. Once we get access to instances to which messages can be sent, we should consider which methods to invoke. There again care is important because it is not trivial to identify potentially destructive methods. Smalltalk offers powerful features such as pointer swapping, class changing, quitting the VM, unloading classes.... Not invoking these methods is important. The Finder tool manually specifies a list of problematic methods. Another approach can be to ask programmers to tag the methods with meta information. As a general point, we believe that knowing methods which may endanger the system when run randomly is a valuable information that can make the system more robust in presence of tools such as the Finder in Pharo or Squeak [BDN+09].

3.2 How to identify errors?

While the other problems can be fixed with the (tedious) addition of meta-data, identifying that a bug is found is a difficult question. Indeed raised exceptions are not part of method declaration in Smalltalk as this is the case in Java. Therefore we cannot simply identify a bug by looking if the exception raised was part of the declared raised exception. In addition, since there are no contracts, they cannot be used to filter wrongly passed arguments. In addition, since a wide range of objects can be passed as arguments (and could be invalid) we cannot use the fact that the arguments are valid when analyzing an exception. The combination of the absence of argument type, contracts and declared exception is clearly what makes the real error identification a challenge.

As a first way to distinguish between various situations we can consider the object receiver contained in the top frame of the method stack and to identify different errors.

Receiver on top of stack and message not understood. The idea here is how do we make sure that one sends messages that are understood by the receiver. Such situation should not happen because the tools should enumerate the methods from the receiver class. However, even such simple assumption is not always true: a message can be cancelled in superclasses using "should not implement" exception and superclass methods are banned from subclasses.

Receiver not on the top of the stack and error. This is the regular case when we get an error is because you send a message and this method sends another one and along the road there is an error. The question then is what to do? Such a situation can arise due to different causes:

Badly initialized objects. For example, executing methods on an object that is not well initialized or whose invariant is broken can happen. The automatic detection of such case is difficult. Such object can either be created by a bogus initialization or the result of inadequate method execution. It is thus important to know that instances in the instance database are in a valid state.

Breaking precondition. It may happen that messages not respecting non explicit invariant lead to errors. For example sending the message reciprocal to 0@0 leads to a problem because x reciprocal does not work on number zero.

```
(0@0) reciprocal
   ->  x reciprocal @ y reciprocal.
      ->  x reciprocal
```

Probably the receiver being different of 0 should be a precondition on Number»reciprocal and similarly, x != 0 and y != 0 should be the precondition of Point»reciprocal.

Another example is #() first (accessing the first element of an empty array). Here clearly a precondition would help to capture that this behavior is not an error but just a normal behavior. In addition considering specific error raised by the class can be a good starting help.

```
#() first
   -> errorSubscriptBounds:
```

The definition of the at: method (which leads to the previous error) shows that some errors could be used to build up a list of raised errors. Such errors could then be used to define preconditions.

```
Object>>at: index
   "Primitive. Assumes receiver is indexable.
Answer the value of an indexable element in the receiver.
Fail if the argument index is not an Integer or is out of bounds.
Read the class comment for a discussion about that the fact
that the index can be a float."

   <primitive: 60>
   index isInteger ifTrue:
      [self class isVariable
         ifTrue: [self errorSubscriptBounds: index]
         ifFalse: [self errorNotIndexable]].
   index isNumber
      ifTrue: [^self at: index asInteger]
      ifFalse: [self errorNonIntegerIndex]
```

Since there is no contract declared, software bugs cannot be distinguished from contract violation. This clearly shows that preconditions could be useful for automated random testing in addition to the other properties they bring to software quality.

3.3 Understanding results

While performing some preliminary experiments with automated random testing in Smalltalk, we identified also the following challenge: How can we identify the impact of a bogus instances on the resulting generated errors?

For example we got an instance of Point, the point nil@nil as an available instance and it generated a lot of false positives by generating errors when executing methods that were

totally correct when we picked any other instances available in the system. We identified this instance in a ad-hoc way and we believe that tools to support the understanding of the results are needed. Notice that this problem does not happen in YETI because YETI uses types and contracts as specifications for valid input. In Smalltalk, this is obviously not possible so the nil@nil example is a good one.

If somebody had specified formally Point, then nil@nil would have been illegal. In the case of Pharo, this instance lives in the system, so it was taken as possible input, and polluted your output. We analyzed why such instance was there and it was unused. Since Pharo 1.3 this instance was removed, still this is an interesting case for understanding the impact of an instance of the resulting method that are raising errors.

To illustrate this challenge, here are the data we got when performing our experiments. The existing instance nil@nil broke a lot of messages: in our experiments we obtained 3695 problematic messages. While checking the problematic messages we found that we obtained 371 buggy receivers (points with at least one zero) and a number of buggy methods r, reciprocal, guarded, negated, degrees, max, theta, asFloatPoint, abs, isZero, truncated, rightRotated, eightNeighbors, ceiling, leftRotated, asIntegerPoint, min, normal, transposed, normalized, sign, floor, fourNeighbors, deepCopy, fourDirections, angle, rounded.

When filtering the buggy selectors by removing the bogus instance (nil@nil) from the instance database, we reduced the number of problematic methods to two: normalized and reciprocal. Such methods raised errors because of points with one zero. We identified the problems with nil@nil because we looked at all the instances that generated errors and noticed it. In this case this was simple: we got points with nil, or at least one zero. We tried to see how a tool could have help us and report a problem but we did not found a simple approach based on a correlation or statistics. In our dataset, sorting the number of problematic methods according to the receiver was not a great help.

Being able to represent results and the influence of receiver/arguments on the generated problems is thus of importance to reduce noise.

4. Paths towards random testing

To enable automated random testing for dynamic languages, we propose to follow these milestones:

- *Dynamic Type Inference using Random Testing.* One of the first actions to be done is to use type inference to get some type information. The minimum is to use a simple static analysis as proposed by RoelTyper [PMW09]. Combining an approach like the one of RoelTyper with the type collected by MethodWrapper is an interesting track to follow. However, using MethodWrapper or any execution based appraoch requires both tests availability and
- *Random testing for dynamically typed languages using existing instances.* Once type inference is available, it becomes possible to use instances in an image and perform either random testing or exhaustive testing of programs in an automated way. The performance associated to the testing is then dependent on the quality of the existing instances. It seems thus likely that such a technique should be applied while the program to test is stalled after being run for a while.

  In parallel, identifying classes defining an initialize method (or inheriting one) should be considered to identify classes where creating instances using the method new may provide well initialized instances.
- *Automated random testing for dynamically typed languages.* The next step is to be create meaningful instances at random. As of now, creating random instances is easy. Making sure these are meaningful is quite difficult without additional support. It might be useful in this case to add support for contracts (pre-, postconditions, and class invariants) and to let programmers specify such contracts to decide whether instances are valid for testing. This might, for example, allow the filtering for testing of instances such as nil@nil.
- *Sandboxing for testing.* As a more long-term goal it might also be interesting to consider how to restrain testing so that it does not corrupt the tested images and external resources.

Acknowledgements

This work was supported by Ministry of Higher Education and Research, Nord-Pas de Calais Regional Council and FEDER through the 'Contrat de Projets Etat Region (CPER) 2007-2013' and the Royal Academy of Engineering.

References

[BDN+09] Andrew P. Black, Stéphane Ducasse, Oscar Nierstrasz, Damien Pollet, Damien Cassou, and Marcus Denker. *Pharo by Example*. Square Bracket Associates, 2009.

[BFJR98] John Brant, Brian Foote, Ralph Johnson, and Don Roberts. Wrappers to the rescue. In *Proceedings European Conference on Object Oriented Programming (ECOOP'98)*, volume 1445 of *LNCS*, pages 396–417. Springer-Verlag, 1998.

[COMP08] Ilinca Ciupa, Manuel Oriol, Bertrand Meyer, and Alexander Pretschner. Finding faults: Manual testing vs. random+ testing vs. user reports. In *IEEE International Symposium on Software Reliability Engineering (ISSRE)*, Nov 2008.

[CPL+08] Ilinca Ciupa, Alexander Pretschner, Andreas Leitner, Manuel Oriol, and Bertrand Meyer. On the predictability of random tests for object-oriented software. In *International Conference On Software Testing, Verification And Validation (ICST 2008)*, July 2008.

[CPO+11] I. Ciupa, A. Pretschner, M. Oriol, A. Leitner, and B. Meyer. On the number and nature of faults found by random testing. *Software Testing, Verification and Reliability*, 21(1):3–28, 2011.

[Duc99] Stéphane Ducasse. Evaluating message passing control techniques in Smalltalk. *Journal of Object-Oriented Programming (JOOP)*, 12(6):39–44, June 1999.

[OT10] Manuel Oriol and Sotirios 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.

[PMW09] Frédéric Pluquet, Antoine Marot, and Roel Wuyts. Fast type reconstruction for dynamically typed programming languages. In *DLS '09: Proceedings of the 5th symposium on Dynamic languages*, pages 69–78, New York, NY, USA, 2009. ACM.

[SS04] S. Alexander Spoon and Olin Shivers. Demand-driven type inference with subgoal pruning: Trading precision for scalability. In *Proceedings of ECOOP'04*, pages 51–74, 2004.

[TAD+10] Ewan Tempero, Craig Anslow, Jens Dietrich, Ted Han, Jing Li, Markus Lumpe, Hayden Melton, and James Noble. Qualitas corpus: A curated collection of java code for empirical studies. In *2010 Asia Pacific Software Engineering Conference (APSEC2010)*, December 2010.