Challenges in Automated Test Data Generation for Dynamically Typed Programming Languages

Simon Bihel[1,2]

[1] Université de Rennes 1, France
[2] École Normale Supérieure de Rennes, France
simon.bihel@ens-rennes.fr

Supervisor: Dr. Shin Yoo
COINSE Lab, KAIST, South-Korea

Monday 15th May, 2017 – Friday 4th August, 2017

Abstract. Despite popular usage, most of the dynamically-typed languages still lack automated test data generation tools. Most of the existing tools and approaches depend critically on static and explicit types, which makes it hard to port any of them over to dynamically typed languages. Some projects have tried to tackle the challenges of dynamically-typed languages or complex data structure but without much success and without generating momentum in the research community. After trying to develop a better tool than existing ones we were able to refine what problems are in the way of automated test data generation. This paper exposes them and tries to give some clues on how to tackle them.

Keywords: software engineering, testing, coverage, experimental, dynamic typing

1 Introduction

Testing is an essential part of software development. Especially due to the lack of formal specifications, tests are the pillars to ensure well-behavior of programs, be it lack of crash or coherent results. Various metrics exist to evaluate how thoroughly tested a software is. One of these is the *code coverage* that describes the degree to which the source code of a program is executed when a particular test suite runs.

Automating the generation of tests is essential to free the programmer from writing them as is it a time-consuming and error-prone task. Evolutionary methods like search-based software testing have been of interest for several decades [29]. In particular, the automated test data generation problem [19] which consists of generating inputs to explore different kinds of properties for a given program.

Past works on test generation automation have been focused on statically-typed languages. Having information on the type of test inputs allows tools to focus on the values of inputs and the depth of complex structures. Having no knowledge on the inputs adds a new step in the process of testing as we have to find fitting types for parameters. It also induces limitations to the kind of bugs a tool can detect as well as a heavier cost due to tries with wrongly typed inputs.

While developing an automated test generator for Python (presented in Section 4) we came to the conclusion that we would not have come further than existing works (explained in Section 2). Thus we decide to write down (Section 3) the challenges that the community will have to tackle in order to have a fully automated test generator for dynamically-typed languages and more generally dynamic languages.

2 Background

While a lot of work has been put in automated testing for statically typed languages, dynamically typed languages have received very little attention. Only a few projects exist and each has a different approach, different levels of automation, and uses different features that usually ship with dynamic languages.

Multiple reasons can explain this void. Culturally, languages like Python are not under the same level of industrial scrutiny for structural coverage, compared to C or Java. But with rise of significant projects (ranging from popular web applications to scientific tools) written with dynamic languages the light has to be moved toward this problem. Despite this interest the automation of tests generation is still a daunting task.

Before presenting others automated testing tools in 2.4 we will first give an overview of search-based software engineering in 2.1, 2.2 and explain some techniques used by practitioners for dynamic languages testing in 2.3.

2.1 Software testing

Testing consists of a System Under Test (SUT) (e.g. a whole program, a function, an object's method, etc.) to which we will apply perturbation by sending inputs and then analyze the output.

There are a broad variety of test and metric classes. Tests focus on one particular aspect of software to be finer grained, faster to run, to be more comprehensible in the bugs they find, have more precise goals to achieve for developer, etc.

In automated testing, where a tool generates tests without the help of the programmer, we can identify some test classes. We will focus on dynamic testing, as opposed to static testing (e.g. static analysis). We have two kinds of testing, black-box and white-box. We will focus on white-box but we will present both.

Black-box testing consists of analysing only the inputs and outputs of the SUT. It is used to test functionalities, integration in other systems or even purely random testing is useful to discover failure domains [1]. A failure domain can be a range of values for which the SUT does not have a satisfying response (e.g. crashes).

Because we are focusing on the problem of automated testing without much information of what the SUT should output we will not be focusing on this kind of testing.

White-box testing , on the other hand, works with more information on the SUT (e.g. access to its source code or runtime monitoring) we can have more insightful feedbacks. We can isolate different units (i.e. parts) of the software, analyze the data and control flow, or test only modified parts (by what is called regression). With access to the innards of the SUT we can make sure that every parts of it are tested. We use a metric called *code coverage* to evaluate the scope of tests.

Multiple coverage criteria exist: Has every function been called? Has each statement been executed? Has each boolean sub-expression been evaluated both to true and false? Has each branch of each control structure been executed? That last criterion is called branch coverage and it is the one we are focusing one. It is part of standards [7,31] for quality assurance and safety.

In our case of dynamically-typed languages, working on code coverage seems to be a good idea to make sure we are exploring every parts and functionalities of a function as they might be dependent on the type of parameters. This particular problem of exploring every branch of a function is called test data generation. One way to tackle this problem of finding certain values to pass tests is to use search-based techniques.

2.2 Search-based software engineering

Search-based software engineering [15] is the application of metaheuristic algorithms to problem related to software engineering. Such algorithms comprise

genetic algorithms, hill climbing, etc. The goal is iterate and find an optimum by executing for real the system under test or optimization.

When we are trying to explore different branches, we are trying to satisfy some boolean conditions. We will thus try to find values that can reach and pass these control structure. With feedback to know how close we are to go through a condition, we can iterate by tweaking parameters and re-executing the SUT. This way of solving appeared in 1990 [19] and is called *search-based test data generation*. It has been extensively applied to branch coverage [25,20,26,18].

One example is the Alternating Variable Method [28] that consists in playing with one parameter until no positive change is seen, and then focus on another parameter.

2.3 Testing techniques

Following is a list of testing methods that the reader can refer to to understand the scope of related tools.

Instrumentation is the insertion of additional code in the SUT during compilation or linking. It is often needed for white-box testing. This code can provide various information like counters to record which statement are executed or compute distances as needed in branch coverage search.

Mock objects simulate to behavior of real objects. They are used to have more control on the execution, e.g. to remove some side effects to test one particular property of a program or simply to provide instrumentation. It is a versatile tool vastly used [23,33,13,34]. In the most extreme use cases they can be used to test only function calls for example, and methods only return either null or another mock object.

Object testing can be done by creating an instance, modifying its state and then measuring some metrics. To modify the state of the instance a method call sequence is defined. After that, a final method call is done and metrics like code coverage or method calls can be measured.

2.4 Research works

Ducasse et al. [11] have already exposed challenges and milestones in random testing of dynamically-typed languages. They are talking about random testing but with a white-box approach. They also bundle object-oriented with dynamic types problems. The challenges they highlight are instance generation and execution, error identification, and result understanding. Our work exposes more challenges as well as more practical details which are explained in the next section. They also propose milestones that we will discuss in Section 5. They propose static analysis and dynamic type collection for type inference, regression from existing instances, hand-written contracts for automated instances generation, and sandboxing to protect the environment. In the end their work is different to ours because we are primarily focusing on basic dynamic types and test coverage.

RuTeG [24] (Ruby Test case Generator) is a Ruby tool that targets the dynamic and object-oriented features of Ruby (not just dynamic types). They do object testing and do code (line) coverage.

The center of the tool is the test case generator. For the inputs it uses generators, basic types (numbers, strings, objects, arrays) are supported natively but for the rest it is to the user to write problem-specific generators. That means that this tool is not fully automated (but the authors hint at the re-usability of a set of generators [12]). The test case generator uses a Genetic Algorithm for each step of the test unit (constructor, method call sequence, method under test). In general, an individual will be composed of type patterns and according data generators.

Before starting the search, the source code is analyzed statically to reduce the search space. Method names and argument lists information will be collected as well as methods called on arguments. It will identify methods impacting the state of the instance and also identify control structures (this is possible because they ignore run-time code addition). It also disqualify inappropriate types. The authors claim that the tool performs better than fully random generation.

SpLATS [4] (SpLATS Lazy Automated Test System) is another tool for Ruby that was written during a student project. It has a lazy approach and uses mock objects for that purpose. It will collect information at runtime on which methods should be implemented and will output most of the time other mock objects or else primitives.

Only methods of classes constructed by standard methods can be tested. It is assumed that there are no side effects and no interactions with the environment. No inheritance. No blocks or variable sized arguments.

It constructs all possible sets of arguments with only nil or mocks. When implementing a method, it will return either nil or another mock object.

SPLAT [22] (Simple Python Lazy Automated Tester) is a tool for Python with a lazy approach and is another student project. It generates unit tests for instruction coverage. It works with bytecode and uses what is called meta-parameters in Python to make mock objects. It uses a unit tests library to have automatically generated assertions on the return value or exceptions raised. It also gather structure information from the bytecode.

The tool does not handle internal methods (those surrounded by '__') which means it cannot capture primitive operations (e.g. addition). It also excludes string predicates but points to a study [2] addressing string predicates with a search-based approach. It does not seem to generate recursive parameters like lists of lists.

These works were published at least 5 years ago but without follow-up. While writing our tool, we realised there was hardly anything new but maybe a potential different subset of cases handled. We thus decided to write in details all problems that a perfect tool should solve and we aim at building precise foundations to build on to in future works. As each of these projects have different

problematics depending on the language they are tackling, we aim to identify some generalization for dynamically-typed languages.

3 Challenges

This section presents challenges of testing dynamically-typed languages based on previous research [11] and our experience. This list can be added to an already long list of challenges for non dynamic languages testing [27].

3.1 Pure dynamic types problems

Type errors One big limitation for an automated code coverage tester is the inability to identify crashes. As we are trying to find coherent types for parameters, if a type error is raised, how can we know it is the parameters that are the problem and not the code? It distances us from the fault-based testing approach which is especially useful for object-oriented programming systems [17].

Relations between parameters Parameters search can get incredibly complex if we think even beyond complex data structures, with for example functions or indexes. Also an object can be constructed by a function. Because of that, if we knew which parameters were involved in an operation that raised a type error, we could not be sure that changing just these parameters would improve things.

Working but unfit types A parameter of a certain type might not generate type errors but make the search much more difficult. If we have for example a floating-point number where an integer is enought then simply approaching a round value might be difficult.

Difficulty to reduce domain of possible types A parameter can have different types for different branches. Because of that, it is not possible to reduce the possible types of a parameter for all branches but it is possible for one branch.

3.2 The bigger picture

Because dynamically-typed languages usually integrate other paradigms, like object-oriented, we need to consider the impact of dynamic types. Also, known challenges in static languages become even more problematic.

Execution Controlling the environment of the SUT is essential for performance and security purposes. Two problems arise. The first one is for the language environment, such as primitives, as it can be modified dynamically by programs. The second problem is about identifying side effects to block them and protect the external environment.

Understanding results In a case outside of simple code coverage, you might want to analyze the result and judge whether it is right. If a function ends up working for different parameters types than the ones it was designed for we are facing the automated oracle problem [5].

Infinite search spaces Many statically-typed languages have bounded types by the number of bits. On the other hand many dynamically-typed languages have infinite range for integer for example, or seemingly with an infinite value being implemented by default. Having information on whether we are approaching a condition becomes essential then.

Dynamic data structures Generating data structures of variable size has been the subject of many works. [21] uses both a search-based approach and a set of constraint solving rules. Tools for object-oriented software testing like [35,10] use available instances to generate new instances. Arcuri and Yao [3] focused only on containers (e.g. lists, trees, etc.) which are data structures designed to store any arbitrary type of data, using search-based techniques. Addressing this problem is also important for variable sized arguments as they can be seen as lists or dictionaries/hash tables.

Dynamic insertion of code All tools that we have seen exclude from their scope functions that execute code contained in a string (e.g. '`eval`' in Python). One argument given is that it is not a commonly used feature. Nonetheless in the case of branch coverage, as we rely on instrumentation to actual detect branch forks, one problem arise. We cannot have unique identifiers for instrumentation function calls as the code might be inserted or executed multiple time.

Objects state Manipulating the state of an object instance is difficult but necessary for testing. We have seen that it is generally done by having a list of method calls. With languages that do not have constructors like Smalltalk is it even impossible to do pure random testing.

4 Contribution

Our tool started as a fork of CAVM [18] which is a search based test data generation tool for C. It uses the Alternating Variable Method and can generate complex data structures.

Our tool handles primitive types and lists (that can contain different types and lists). It basically encloses the AVM search catching type and index errors.

4.1 Philosophy

Ideally we wanted to have a tool the most universal possible, i.e. without Python specific features. We focused on primitive type (e.g. integers, strings) with lists

to have a complex data structure. Lists can have elements of different types, even other lists.

When an index error occurs, because we do not have a distance function to tell use how close we are to the index asked, we are mocking list parameters. We decorate the access method and bundle with the index error exception the index asked. We did this to avoid growing wrong lists in the dark because of a lack of feedback. It also mean that when a index error occurs we are going to grow the list instead of reducing another parameter if it is used as the index.

4.2 Technical difficulties

In Python, exceptions (i.e. errors) carry very little information. If the exception arises from an operation involving two objects, what will be available is the code line number, the two types involved and the values of the objects in frame (i.e. function call).

In addition to that, we cannot really use mocked objects to decorate all methods and catch errors. For example with an operation, it will check on one object if the according method works, and then check the other object.

Not all classes can be inherited which is another limitation for mocking as inheritance allows to keep the type. Even though it is discouraged to write conditions on types (e.g. '`isinstance`' function in Python) it is still something that is used.

5 Future Works

As the purpose of this paper was to establish goals in the aim of automated testing for dynamically-typed languages, we will now plan on what to do next.

5.1 Pursuing the original goal

Studying the time spent on trying invalid types would give some hints on whether it is viable to extend the search for very complex structures such as objects. If we were to use decorator with a catch for exceptions for all objects methods, it would be interesting to compute the overhead. This overhead would depend on whether the SUT uses exceptions a lot as the overhead of the catch will be noticeable only when an exception occurs.

We could compare our approach to [30] which uses the multiple-dispatch feature of Julia (methods have different implementation to handle different types for parameters).

Overall we should focus at first on a simple tool that works for every situation. The actual search for values is secondary, but still needed to be sure we explore every needed type for arguments.

5.2 Beyond random testing

This section aims at giving clues to go beyond random testing and use previous work done on dynamic languages.

Data-flow and fine grained runtime monitoring. Data-flow would allow us to know which parameters have an impact on other variables or even on other parameters. This would be useful if we knew which variables were involved in an operation resulting in an error. Which in turn would require wrap every elementary operation. We would have to use a wrapper because in the case of a language like Python we cannot override basic types.

A lot of work has been done on regression testing [14,16,32,36]. Of course this is quite the opposite of the goal of automated test generation. Having a starting set of inputs that we know are valid greatly reduces our main problem of finding appropriate types. But our work on automated testing can be used as an extension and enhancement of existing, hand-written tests. For big projects, there should be preexisting tests for functionalities testing. Knowledge can also be extracted by watching the whole program with the graph of function calls.

Some languages like Python have type annotations. We could ask the programmer to precise with annotations when parameters should be custom types and assume other parameters are primitives.

On-the-fly type analysis [8] and verification [9] for dynamically-typed object-oriented languages are possible but do not make sense without valid inputs.

Genetic Algorithms are possible as [6] shows how to produce mutants for dynamically-typed languages by generating mutants using types information available at run-time.

6 Conclusion

In this paper we presented challenges in dynamically-typed languages software testing. We saw that challenges arise from both the paradigm and the implementation. Detailing and isolating down problems will allow more coherent future works.

Acknowledgment

Thanks to Shin Yoo and the whole COINSE team for being so friendly and welcoming. Thanks also to KAIST for hosting me.

This internship was supported by the Région Bretagne with the *Jeune à l'International* grant as well as the ENS Rennes with the *Aide Transport pour la Mobilité à l'International* grant.

References

1. Ahmad, M.A.: New Strategies for Automated Random Testing. Ph.D. thesis, University of York (2014)

2. Alshraideh, M., Bottaci, L.: Search-based software test data generation for string data using program-specific search operators. Software Testing, Verification and Reliability 16(3), 175–203 (2006)
3. Arcuri, A., Yao, X.: Search based software testing of object-oriented containers. Information Sciences 178(15), 3075–3095 (2008)
4. Bardsley, E., Glassberg-Powell, C., Keeley, C., Slade, J., Wood, T.: Splats: Lazy automated test system (2011)
5. Barr, E.T., Harman, M., McMinn, P., Shahbaz, M., Yoo, S.: The oracle problem in software testing: A survey. IEEE transactions on software engineering 41(5), 507–525 (2015)
6. Bottaci, L.: Type sensitive application of mutation operators for dynamically typed programs. In: Software Testing, Verification, and Validation Workshops (ICSTW), 2010 Third International Conference on. pp. 126–131. IEEE (2010)
7. Vocabulary of terms in software testing. Standard, British Standards Institute (1998)
8. Chambers, C., Ungar, D.: Iterative type analysis and extended message splitting: Optimizing dynamically-typed object-oriented programs. LISP and Symbolic Computation 4(3), 283–310 (1991)
9. Chugh, R., Rondon, P.M., Jhala, R.: Nested refinements: a logic for duck typing. ACM SIGPLAN Notices 47(1), 231–244 (2012)
10. Ciupa, I., Leitner, A., Oriol, M., Meyer, B.: Artoo: adaptive random testing for object-oriented software. In: Proceedings of the 30th international conference on Software engineering. pp. 71–80. ACM (2008)
11. Ducasse, S., Oriol, M., Bergel, A.: Challenges to support automated random testing for dynamically typed languages. In: Proceedings of the International Workshop on Smalltalk Technologies. p. 9. ACM (2011)
12. Feldt, R.: Biomimetic software engineering techniques for dependability. Chalmers University of Technology (2002)
13. Freeman, S., Mackinnon, T., Pryce, N., Walnes, J.: Mock roles, objects. In: Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications. pp. 236–246. ACM (2004)
14. Gligoric, M., Badame, S., Johnson, R.: Smutant: a tool for type-sensitive mutation testing in a dynamic language. In: Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering. pp. 424–427. ACM (2011)
15. Harman, M., Jones, B.F.: Search-based software engineering. Information and software Technology 43(14), 833–839 (2001)
16. Haupt, M., Perscheid, M., Hirschfeld, R.: Type harvesting: a practical approach to obtaining typing information in dynamic programming languages. In: Proceedings of the 2011 ACM Symposium on Applied Computing. pp. 1282–1289. ACM (2011)
17. Hayes, J.H.: Testing of object-oriented programming systems (oops): A fault-based approach. In: Object-oriented methodologies and systems, pp. 205–220. Springer (1994)
18. Kim, J., You, B., Kwon, M., McMinn, P., Yoo, S.: Evaluating CAVM: a new search based test data generation tool for C. In: Proceedings of the International Symposium on Search Based Software Engineering (2017)
19. Korel, B.: Automated software test data generation. IEEE Transactions on software engineering 16(8), 870–879 (1990)
20. Lakhotia, K., Harman, M., McMinn, P.: A multi-objective approach to search-based test data generation. In: Proceedings of the 9th annual conference on Genetic and evolutionary computation. pp. 1098–1105. ACM (2007)

21. Lakhotia, K., Harman, M., McMinn, P.: Handling dynamic data structures in search based testing. In: Proceedings of the 10th annual conference on Genetic and evolutionary computation. pp. 1759–1766. ACM (2008)
22. Lee, W.Y.: Splat: Simple python lazy automated tester (2012)
23. Mackinnon, T., Freeman, S., Craig, P.: Endo-testing: unit testing with mock objects. Extreme programming examined pp. 287–301 (2000)
24. Mairhofer, S., Feldt, R., Torkar, R.: Search-based software testing and test data generation for a dynamic programming language. In: Proceedings of the 13th annual conference on Genetic and evolutionary computation. pp. 1859–1866. ACM (2011)
25. McMinn, P.: Search-based software test data generation: a survey. Software testing, Verification and reliability 14(2), 105–156 (2004)
26. McMinn, P.: Iguana: Input generation using automated novel algorithms. a plug and play research tool. Department of Computer Science, University of Sheffield, Tech. Rep. CS-07-14 (2007)
27. McMinn, P.: Search-based software testing: Past, present and future. In: Software testing, verification and validation workshops (icstw), 2011 ieee fourth international conference on. pp. 153–163. IEEE (2011)
28. McMinn, P., Kapfhammer, G.M.: Avmf: An open-source framework and implementation of the alternating variable method. In: International Symposium on Search Based Software Engineering. pp. 259–266. Springer (2016)
29. Miller, W., Spooner, D.L.: Automatic generation of floating-point test data. IEEE Transactions on Software Engineering (3), 223–226 (1976)
30. Poulding, S., Feldt, R.: Automated random testing in multiple dispatch languages. In: Software Testing, Verification and Validation (ICST), 2017 IEEE International Conference on. pp. 333–344. IEEE (2017)
31. Software considerations in airborne systems and equipment certification. Standard, Radio Technical Commission for Aeronautics (1992)
32. Steinert, B., Haupt, M., Krahn, R., Hirschfeld, R.: Continuous selective testing. In: International Conference on Agile Software Development. pp. 132–146. Springer (2010)
33. Taneja, K., Zhang, Y., Xie, T.: Moda: Automated test generation for database applications via mock objects. In: Proceedings of the IEEE/ACM international conference on Automated software engineering. pp. 289–292. ACM (2010)
34. Tillmann, N., Schulte, W.: Mock-object generation with behavior. In: Automated Software Engineering, 2006. ASE'06. 21st IEEE/ACM International Conference on. pp. 365–368. IEEE (2006)
35. Tonella, P.: Evolutionary testing of classes. In: ACM SIGSOFT Software Engineering Notes. vol. 29, pp. 119–128. ACM (2004)
36. Yoo, S., Harman, M.: Regression testing minimization, selection and prioritization: a survey. Software Testing, Verification and Reliability 22(2), 67–120 (2012)