Generating Tests to Analyse Dynamically-Typed Programs

Stephan Lukasczyk
University of Passau
Passau, Germany
stephan.lukasczyk@uni-passau.de

***Abstract*—The increasing popularity of dynamically-typed programming languages, such as JavaScript or Python, requires specific support methods for developers to avoid pitfalls arising from the dynamic nature of these languages. Static analyses are frequently used but the dynamic type systems limit their applicability. Dynamic analyses, in contrast, depend on the execution of the code under analysis, and thus depend on the quality of existing tests. This quality of the test suite can be improved by the use of automated test generation but automated test generation for dynamically-typed programming languages itself is hard due to the lack of type information in the programs. The limitations of each of these approaches will be overcome by iteratively combining test generation with static and dynamic analysis techniques for dynamically-typed programs.**

***Index Terms*—Dynamic Analysis; Static Analysis; Python; Test Generation; Type Inference**

I. Problem Statement

Dynamically-typed programming languages, such as JavaScript or Python, have gained more and more popularity in practice over the last years. JavaScript is the de-facto standard programming language for the web; Python has a similar role in the fields of data science, big data, and machine learning. Well-known rankings, such as the *TIOBE Index*[1] or the *IEEE Spectrum Top Programming Languages* ranking[2] reflect this fact. At the time of writing, Python is ranked third, after Java and C, in the TIOBE Index, and is ranked first in the IEEE Spectrum ranking.

Reasons for the increased popularity include the dynamic typing of these languages and their dynamic nature, which allows, for example, to dynamically modify objects during program runtime. Furthermore, they offer a high level of abstraction to developers, which enables rapid development [1] and prototyping. Those particular features, however, also cause potential problems: the lack of a visible, static type system and the dynamic features can lead to subtle errors in programs that are difficult to find. Studies show that a dynamic type system and the dynamic features of a language can be a source of reduced development productivity [2], code usability [3], and code quality [4]–[6]. Besides that, type errors are primarily encountered in dynamic languages due to their lack of a (visible static) type system [7]. Further work reports the frequent usage of dynamic features and other aspects of these programming styles in practice [8], [9].

[1]https://www.tiobe.com/tiobe-index/, accessed 2019–09–16.
[2]https://spectrum.ieee.org/computing/software/the-top-programming-languages-2019, accessed 2019–09–16.

```python
def work(obj, b, pred):
    # ...
    if b.holds(pred):
        delattr(obj, "value")
    # ...
    out(obj)

def out(x):
    # ...
    if global_var:
        x.value = global_value
    # ...
    print(x.value)
```

Listing 1. Example Python code snippet using dynamic features

Consider the Python example program in Listing 1. Method `work`, which takes three arguments of unspecified types, deletes the attribute `value` from its parameter object `obj` depending on how a method `holds` of object `b` evaluates using `pred` as an input value. Afterwards, `out` is called with parameter `obj`, whose type is also not specified. Depending on some global variable's value, the attribute `value` is re-added to the parameter object. If it is not added, but was removed in `work` the attribute access in the last line will fail with an exception. The example shows how dynamic typing and dynamic features of the language can be used in a way that makes software analysis difficult; nevertheless, they are widely used in practice [9].

Static analysis is a commonly used approach for many projects to prevent the introduction of bugs at an early stage of development [10]–[13]. Especially for dynamically-typed languages, static analyses might miss potential issues, or report a large number of false-positive results, depending on the approximation of the static analysis and the granularity level of the tool configuration [14]. In the example, a static analysis tool, such as a linter, is able to issue a warning because of the manipulation of the object `obj`, which can cause program failures when the attribute `value` is accessed afterwards. It cannot reason, however, whether it is a real problem, that is, whether `value` is added again in method `out` since this depends on the run-time value of `global_var`.

A dynamic analysis is able to spot whether there is a case where the attribute `value` is not re-added in `out`. In this case, the warning of the static analysis is confirmed to be a real

problem in the example program. However, a major drawback of a dynamic analysis is that it needs to execute the analysed code, for example, by the test suite of the analysed project. Unfortunately, this makes the analysis depend on the test suite; a test suite that covers only small parts of the code prevents the dynamic analysis to reason about the non-executed parts.

Deficiencies in the test suite can be overcome using automated test generation. Test generation is, however, difficult for Python since nothing is known about the parameter types of the involved methods. A static analysis is able to determine that object `b` needs to provide a method `pred` and object `obj` needs to have an attribute `value`, but it cannot decide which is the correct type for the object `b`. If a test generator would know the correct type, it could generate a test in which it sets `global_var` to `False`, creates an object with an attribute `value` as value for `work`'s parameter `obj`, and creates an object with a constant method `holds` returning `True` as value for `b`; the value of `pred` can be arbitrary. This test case would expose the issue and would raise an exception in the last line caused by the access to the non-existing attribute `value`.

The insight underlying this research is the following: While each of the three techniques, static analysis, dynamic analysis, and test generation, has fundamental limitations in the context of dynamic languages, they can be overcome by combining all three techniques into one framework.

II. State of the Art

A. Automated Test Generation

Automated test generation is a well-established field of ongoing research. Many approaches to automatically generate tests exist for statically-typed languages, such as RANDOOP [15] or EVOSUITE [16] for Java. An annual competition demonstrates the success of such tools [17].

For dynamically-typed languages automated test generation is still a field with many open questions. Only few such tools exist, for example, AUGER[3] for Python, which generates unit tests from recorded execution traces. This approach also has the drawback that one needs to execute the code before unit tests are generated, which is a non-trivial problem.

Other approaches target JavaScript, but are focussing on specific properties of the JavaScript language or its usage in web development. JSEFT [18] incorporates the interactions with the document-object model (DOM) into test generation. ARTEMIS [19] or SYMJS [20] specifically target the feedback from event handlers. Neither approach is suitable to automatically generate tests at an API level for a dynamically-typed programming language.

B. Type Inference for Dynamic Languages

The lack of static type information motivated research from various fields and for a variety of programming languages. Several approaches propose subsets of a dynamic language, usually by removing or restricting the usage of dynamic features, such that the resulting language can then utilise a static type system. An example for this approach is RPYTHON [21], for which a static type inference system exists [22]; RPYTHON is, for example, used in the Python interpreter PYPY.

TYPESCRIPT [23] by Microsoft follows a different approach. It is a statically typed superset of JavaScript and uses a so called transpiler to generate JavaScript code, which will then be used for web packages or applications. A large problem with these approaches is scalability: they require large, often manual, changes to existing code bases to, for example, remove the usage of dynamic features, or add type annotations to methods and variables.

To avoid this manual effort, several type inference strategies have been proposed that allow to infer the variable and method types from existing code automatically. Examples are probabilistic approaches using natural language [24], the use of a machine-learning classifier on documentation comments [25], or deep learning approaches that were trained on type-annotated source code and allow to infer type information [26], [27]. The existing approaches are based on assumptions on the program, such as existing comments, that limit their applicability on programs that do not fulfill these assumptions.

C. Hybrid Static and Dynamic Analysis Techniques

Several studies, for example [10]–[13], show the success of static analysis techniques, especially for statically-typed programming languages; further work also shows their effectiveness for dynamically-typed languages [14], [28]. In the field of dynamically-typed languages the analysis techniques are often also dynamic to overcome limitations inherited from the analysed programming language. Studies on bad coding practices in JavaScript [1] or code smells in Python [29] utilise such a dynamic analysis.

JNUKE [30] combines static and dynamic analyses for Java for run-time verification. PALUS [31] is an approach that combines static and dynamic analyses for test generation for Java. It dynamically infers models from executions and expands them using a dependence analysis; these models are then used to guide the test generation. Variations of analyses are also used, for example, for mutability analysis [32]. Nevertheless, existing approaches target statically-typed programming languages and do not incorporate automated test generation for dynamic analysis inputs, which is proposed by this research.

III. Proposed Approach

A. General Overview

In this paper, I propose a novel combination of analysis techniques as shown in Figure 1. Static and dynamic analyses will be used to detect possible issues related to dynamic typing in the code with a static analysis and rule out false-positive results using a dynamic analysis. To increase the quality of the analyses, automated test generation will be incorporated in the process, which utilises results from the analyses to reason about the expected method inputs and their types. The generated tests are used as further input for the dynamic analysis in order to also execute parts of the code under analysis that is not covered by the provided test suites of the project and to narrow down

[3]https://github.com/laffra/auger, accessed 2019–09–16.

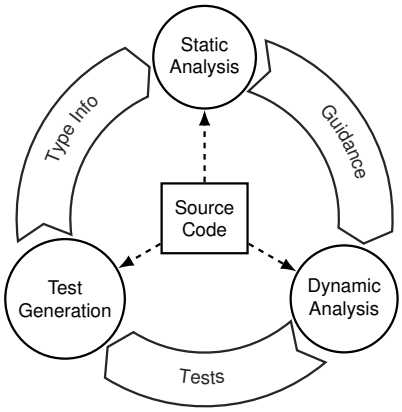

Figure 1. The proposed combination of test generation with static and dynamic analysis techniques

the set of possible types. The figure shows how the involved components complete each other by providing their results as additional input to the remaining components.

A possible application scenario could be the following: The proposed approach receives the source code of a project to analyse as input. It runs a static analysis on this code which detects potential issues in the code, such as dynamic manipulation of objects that are known to be prone to programming errors [9]. The program analysis proceeds to check whether an issue is a real problem or a false-positive due to an imprecise static analysis, a check which is done by a dynamic analysis. If the project under test provides a test that executes the relevant parts of the source code this test can be used to run the analysis. In case there is no test executing the relevant parts the test generation starts in order to provide a test case that can be used for the analysis execution. The analysis chain can afterwards proceed with the next suspicious program location.

The output of the approach is a result, whether or not erroneous parts in the code have been found. Furthermore, when test generation is involved, the proposed approach can emit the generated test cases. They can be used by the developer, for example, as exposing tests to debug and fix the found issues, or to improve the quality of the existing test suite.

B. Generating Unit Tests

Many approaches exist for statically-typed languages like Java. Approaches for dynamic languages, such as JavaScript, target specific features and usage scenarios of those languages. A general purpose unit test generation approach for a dynamic language is missing, which I propose with this paper.

Central to the approach is the knowledge about the expected input types of methods in order to generate tests for them. To gather this knowledge type inference strategies shall be used. This work aims to incorporate several existing type inference approaches to allow a fast and precise reasoning about the expected input types. They will furthermore be improved to be more suitable for the iterative test generation.

The test generation is done at an API level, that is, it creates sequences of method calls to generate objects and adjust their state, in order to use them as matching inputs for a particular method. To achieve this, the test generation utilises a search-based approach similar to EvoSuite [16]. If the expected types for the method under test are known, for example, from annotations in the code, this information will be used for the object creation. Otherwise, test generation will start with random object types, utilise type-inference techniques, and will collect information about occurring errors from wrong object types, in order to find the matching type that can then be used for the generation. The purpose of this procedure is to cover parts of the code that are not covered by the existing test suite of the project under analysis.

C. Combining Static and Dynamic Analyses

A large number of issues in software systems written in a dynamic language are due to the usage of the dynamic features the language provides [29]. The lack of static type information is prone to wrong types or access to non-existing object attributes. The detection of the usage of such dynamic features, as shown in the introducing example in Listing 1, can often be done by a simple static analysis, such as a code linter. In the context of this work, I will introduce new analysis techniques specifically targeting the dynamic typing and dynamic nature of the programming language. This approach, however, is either prone to a large number of false positives, or it misses various issues, depending on the sensitivity of the analysis. A dynamic analysis, however, can safely detect and report the true issues only to the developer.

Combining dynamic and static analysis with test generation in order to find errors that are related to the dynamic nature of a programming language is a prerequisite to realise this goal. Candidates for static analysis can be, for example, linting tools; their mean in this research is to detect possible error locations. The dynamic analyses shall then be used to check whether the found issues of the static analysis is a false-positive result or a real error that should be fixed by a developer.

IV. Evaluation Plan

In order to evaluate the proposed approaches we plan to quantitatively compare the resulting tools with existing approaches. The work will target the Python programming language, since it is the most commonly used dynamic programming language at the time of writing. We plan to use popular publicly available Python software projects as subject systems for evaluations; it will primarily focus on known real bugs, specifically caused by or related to the dynamic typing or the dynamic feature usage in these projects, which we will collect and provide as a separate database of bugs to enable further research.

The evaluation of the test generation will focus on common-used metrics, such as the coverage, achieved by the generated tests, the time consumption during test generation, or the number of detected faults. The evaluation of the combined static and dynamic analysis with test generation will report quantities and statistics on the frequency of suspicious program statements and the number of correctly detected issues as well

as false-positive rates. This shall give an insight in the quality of the proposed approaches.

V. Contributions

With my work I plan to investigate approaches to combine static and dynamic analyses for dynamically-typed programs. Central to this is the generation of tests to overcome limitations of the dynamic analysis due to non-executable code in the project under analysis. Test generation allows to create further inputs that execute the previously non-executable parts of the code under analysis.

I therefore propose an automated test generation framework for dynamically-typed programs. The aim of this test generation framework is to be general purpose, that is, it is not targeting specific characteristic aspects of a language, such as interactions with a web page's DOM, but it generates tests at the API level.

Furthermore, I will contribute the collection of bugs related to dynamic typing and the dynamic nature of the language from real-world projects to enable further research.

Lastly, I propose novel analysis techniques that specifically target the dynamic typing and the dynamic nature of the programming language.

VI. Conclusion

The purpose of this research is to develop a novel combination of static and dynamic analyses with test generation for dynamically-typed programs. The proposed approach overcomes the inherent limitations of each of the involved components and aids developers to prevent the introduction of subtle programming errors caused by the dynamic nature of the programming language.

The resulting tools and evaluation results will be made publicly available, in order to support open science, to enable other researchers to reproduce the results, and to make them comparable with other approaches.

Acknowledgements

The author is advised by Prof. Dr. Gordon Fraser. The author and his advisor are members of the Chair of Software Engineering II at the University of Passau, Germany. Thanks to my supervisor and my colleagues for their valuable inputs.

References

[1] L. Gong, M. Pradel, M. Sridharan, and K. Sen, "DLint: Dynamically Checking Bad Coding Practices in JavaScript," in *Proc. ISSTA*. ACM, 2015, pp. 94–105.

[2] S. Kleinschmager, S. Hanenberg, R. Robbes, É. Tanter, and A. Stefik, "Do Static Type Systems Improve the Maintainability of Software Systems? An Empirical Study," in *Proc. ICPC*. IEEE Comp. Soc., 2012, pp. 153–162.

[3] C. Mayer, S. Hanenberg, R. Robbes, É. Tanter, and A. Stefik, "An empirical study of the influence of static type systems on the usability of undocumented software," in *Proc. OOPSLA*. ACM, 2012, pp. 683–702.

[4] L. A. Meyerovich and A. S. Rabkin, "Empirical Analysis of Programming Language Adoption," in *Proc. OOPSLA*. ACM, 2013, pp. 1–18.

[5] B. Ray, D. Posnett, V. Filkov, and P. T. Devanbu, "A Large Scale Study of Programming Languages and Code Quality in Github," in *Proc. FSE*. ACM, 2014, pp. 155–165.

[6] Z. Gao, C. Bird, and E. T. Barr, "To Type or Not to Type: Quantifying Detectable Bugs in JavaScript," in *Proc. ICSE*. IEEE/ACM, 2017, pp. 758–769.

[7] Z. Xu, P. Liu, X. Zhang, and B. Xu, "Python Predictive Analysis for Bug Detection," in *Proc. FSE*. ACM, 2016.

[8] A. Holkner and J. Harland, "Evaluating the dynamic behaviour of Python applications," in *Proc. ACSC*, ser. CRPIT, vol. 91. Australian Computer Society, 2009.

[9] Z. Chen, W. Ma, W. Lin, L. Chen, Y. Li, and B. Xu, "A Study on the Changes of Dynamic Feature Code when Fixing Bugs: Towards the Benefits and Costs of Python Dynamic Features," *SCIENCE CHINA Information Sciences*, vol. 61, no. 1, pp. 012 107:1–012 107:18, 2018.

[10] N. Ayewah, D. Hovemeyer, J. D. Morgenthaler, J. Penix, and W. Pugh, "Using Static Analysis to Find Bugs," *IEEE Software*, vol. 25, no. 5, pp. 22–29, 2008.

[11] A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Gros, A. Kamsky, S. McPeak, and D. R. Engler, "A Few Billion Lines of Code Later: Using Static Analysis to Find Bugs in the Real World," *Commun. ACM*, vol. 53, no. 2, pp. 66–75, 2010.

[12] B. Johnson, Y. Song, E. R. Murphy-Hill, and R. W. Bowdige, "Why Don't Software Developers Use Static Analysis Tools to Find Bugs?" in *Proc. ICSE*. IEEE Comp. Soc., 2013, pp. 672–681.

[13] M. Beller, R. Bholanath, S. McIntosh, and A. Zaidman, "Analyzing the State of Static Analysis: A Large-Scale Evaluation in Open Source Systems," in *Proc. SANER*. IEEE Comp. Soc., 2016, pp. 470–481.

[14] K. F. Tómasdóttir, M. Aniche, and A. van Deursen, "Why and How JavaScript Developers Use Linters," in *Proc. ASE*. IEEE Comp. Soc., 2017, pp. 578–589.

[15] C. Pacheco and M. D. Ernst, "Randoop: Feedback-Directed Random Testing for Java," in *Proc. OOSPLA*. ACM, 2007, pp. 815–816.

[16] G. Fraser and A. Arcuri, "EvoSuite: Automatic Test Suite Generation for Object-Oriented Software," in *Proc. ESEC/FSE*. ACM, 2011, pp. 416–419.

[17] U. R. Molina, F. M. Kifetew, and A. Panichella, "Java Unit Testing Tool Competition: Sixth Round," in *Proc. SBST@ICSE*. ACM, 2018, pp. 22–29.

[18] S. Mirshokraie, A. Mesbah, and K. Pattabiraman, "JSEFT: Automated JavaScript Unit Test Generation," in *Proc. ICST*. IEEE Comp. Soc., 2015, pp. 1–10.

[19] S. Artzi, J. Dolby, S. H. Jesen, A. Møller, and F. Tip, "A Framework for Automated Testing of JavaScript Web Applications," in *Proc. ICSE*. ACM, 2011, pp. 571–580.

[20] G. Li, E. Andreasen, and I. Ghosh, "SymJS: Automatic Symbolic Testing of JavaScript Web Applications," in *Proc. FSE*. ACM, 2014, pp. 449–459.

[21] D. Ancona, M. Ancona, A. Cuni, and N. D. Matsakis, "RPython: A Step Towards Reconciling Dynamically and Statically Typed OO Languages," in *Proc. DLS*. ACM, 2007, pp. 53–64.

[22] E. Maia, N. Moreira, and R. Reis, "A Static Type Inference for Python," in *Proc. DYLA*, 2011.

[23] G. M. Bierman, M. Abadi, and M. Torgersen, "Understanding TypeScript," in *Proc. ECOOP*, ser. LNCS, vol. 8586. Springer, 2014, pp. 257–281.

[24] Z. Xu, X. Zhang, L. Chen, K. Pei, and B. Xu, "Python Probabilistic Type Inference with Natural Language Support," in *Proc. FSE*. ACM, 2016, pp. 607–618.

[25] Y. Luo, W. Ma, Y. Li, Z. Chen, and L. Chen, "Recognizing Potential Runtime Types from Python Docstrings," in *Proc. SATE*, ser. LNCS, vol. 11293. Springer, 2018.

[26] V. J. Hellendoorn, C. Bird, E. T. Barr, and M. Allamanis, "Deep Learning Type Inference," in *Proc. ESEC/FSE*. ACM, 2018, pp. 152–162.

[27] R. S. Malik, J. Patra, and M. Pradel, "NL2Type: Inferring JavaScript Function Types from Natural Language Information," in *Proc. ICSE*. IEEE/ACM, 2019, pp. 304–3015.

[28] D. Liu and A. Petersen, "Static Analyses in Python Programming Courses," in *Proc. SIGCSE*. ACM, 2019, pp. 666–671.

[29] Z. Chen, L. Chen, W. Ma, X. Zhou, U. Zhou, and B. Xu, "Understanding Metric-based Detectable Smells in Python Software: A Comparative Study," *Information & Software Technology*, vol. 94, pp. 14–29, 2018.

[30] C. Artho and A. Biere, "Combined Static and Dynamic Analysis," *Electr. Notes. Theor. Comput. Sci.*, vol. 131, pp. 3–14, 2005.

[31] S. Zhang, D. Saff, Y. Bu, and M. D. Ernst, "Combined Static and Dynamic Automated Test Generation," in *Proc. ISSTA*. ACM, 2011, pp. 353–363.

[32] S. Artzi, A. Kiezun, D. Glasser, and M. D. Ernst, "Combined Static and Dynamic Mutability Analysis," in *Proc. ASE*. ACM, 2007, pp. 103–113.