Using Type Analysis for Dealing with Incompetent Mutants in Mutation Testing of Python Programs

Anna Derezińska[a] and Anna Skupinska

Warsaw University of Technology, Institute of Computer Science, Nowowiejska 15/19, Warsaw, Poland
A.Derezinska@ii.pw.edu.pl

Abstract: Mutation testing of dynamically typed languages, such as Python, raises problems in mutant introduction and evaluation of mutant execution results, which may provide to the application of incompetent mutants. Type analysis technique has been proposed to support mutation testing in Python. Based on the static information available in a program and on the type impact analysis, prospects of type errors are detected. The method has been developed in a type analyser, which has been combined with a mutation tool for Python programs. In mutation testing of programs in which many incompetent mutants would be created, the approach could lower the number of such mutants. The final contribution depends on the mutation operators and programming structures used in a mutated program. Preliminary experiments do not confirm the efficiency improvement in terms of time execution.

1 INTRODUCTION

One of the characteristic features of the Python programming language is dynamic typing. It might contribute to the easiness of programming and code clarity. On the other hand, many of the developer errors are not detected at the implementation stage. Therefore, numerous exceptions could become apparent only during the execution of a program.

Mutation testing is a method that supports the evaluation and improvement of a program and its test suite quality (Papadakis et al., 2019). However, dynamic typing influences the creation of mutants and the assessment of mutation testing results. One of the significant problems is the encountering of so-called *incompetent* mutants that are mainly associated with *type errors* detected during a mutant execution (Bottaci, 2010), (Derezinska and Hałas, 2014a).

Dynamic nature of Python impacts on the allowed mutations or combinations in genetic algorithms. Incompetent mutants degrade efficiency of mutation testing and could even provide ambiguous results. High cost of mutation testing is an obstacle in the method application (Pizzoleto et al., 2019). Therefore, a mutation tool should limit the creation of such kind of mutants. A straightforward approach could be the avoidance of mutation operators that may lead to the creation of many incompetent mutants and bounding a number and/or places of mutants being created by other mutation operators (Derezinska and Hałas, 2014b). However, this strategy could entail unwanted restrictions in the mutation testing capabilities.

Another approach could be a type analysis based on information about types accessible from the program statement evaluation and from annotations, which are increasing commonly used in the current

---

[a] https://orcid.org/0000-0001-8792-203X

Python programs (Python, 2021). Any kind of static type analysis could not resolve all situations, but potentially could point at selected cases of incompetent mutant occurrence.

The primary goal of this paper is to investigate a static type analysis to optimize the mutation testing of Python programs. The main contribution is a dedicated approach to the type analysis, its development combined with a mutation testing tool for Python programs, and a preliminary evaluation of this proof of concept.

The paper will be structured as follows. The next Section describes the issues of mutation testing in Python programs, especially concerning incompetent mutants. In Section 3, an approach to type analysis is explained. Development of the type analysis support in the mutation tool for Python is addressed in Section 4. We finish the paper with notes about related work in Section 5, and with the conclusions.

2   BACKGROUND

We first introduce the basic notion of mutation testing and discuss its specific features in the context of Python programs.

2.1   The Basics of Mutation Testing

Mutation testing has been primarily used for the evaluation of test set quality. An original program is modified to create many program variations, so-called *mutants*. A mutant is generated applying a *mutation operator* in a certain place(s) of the original code. Mutation operators reflect possible code changes caused by programmer errors.

Mutants are executed against a test set under concern. Differences detected in a mutant behavior denote that the injected error(s) has been discovered by the tests. The more mutants are *killed* in this way, the more the test set is able to reveal errors. The mutation testing can also support the generation of new test cases that supplement the test set. Mutants that could not be killed by any test, i.e., *equivalent* mutants, impede the process, as these mutants not always could be automatically recognized.

2.2   Competent and Incompetent Mutants

Application of mutation testing in dynamically typed programming languages leads to specific kinds of mutants (Bottaci, 2010). Competent mutants have code changes that could have been introduced by a developer of an original program. Mutation operators are aimed at generating mutants that could have been produced by a competent developer. A mutant will be called *competent* if a mutated code has no injected errors that could undesirably interrupt its execution and its modifications could be detected by test cases.

The first source of incompetent mutants could be mutants that have syntax errors (*still-born* mutants). Errors of such kind can be detected during program compilation, which is time-consuming. However, incompetent mutants with syntax errors could be avoided by a careful creation of mutants within a tool. Thus, we can omit this problem in our discussion.

Another cause of incompetent mutants are execution errors generated mainly by incompatible types. Python is a programming language with a dynamic typing system. Therefore, a variable could be of any type. Usage of an incompatible type could result in the following situations:

- calling an attribute that the type has not got,
- calling a variable that the type is not possible to be called,
- using an operator not supported by the type.

An idea to detect situations that might lead to the creation of those kinds of incompetent mutants is a program analysis aimed at type occurrence.

A mutation operator can create an incompetent mutant if it cannot verify that such a mutant could not be created. Therefore, mutation operators that create too many incompetent mutants have been excluded from mutation testing. To check that a created mutant is a competent one, it should be compiled and tested, which requires some time and resources.

One of the main causes that a mutant becomes an incompetent one is a *TypeError*. It can occur when a variable has to perform an action not supported by this type. For example, a concatenation of strings would be accomplished with an addition operator. Original code:

```
if n >= 1
    out=="I have" + str(n)+"books"
```

Mutated code:

```
if n >= 1
    out=="I have" - str(n)+"books"
```

Errors of this kind could be introduced by mutation operators that manipulate variables or their actions. Because of dynamic typing, mutation operators of this kind cannot detect whether the introduced mutations could be supported by types that will be taking part in the actions.

2.3   Dealing with Incompetent Mutants

While tracing of types, the incompetence of some mutations could be predicted, such as AOD (*arithmetic operation deletion*), AOR (*arithmetic operation replacement*). It also makes possible to apply the VOR mutation operator (*variable to constant replacement*) without creating a great number of incompetent mutants.

There are some mutation operators that would be not influenced by the tracing of types. For example, LOR (*logical operator replacement*) and ROR (*relational operator replacement*) do not create incompetent mutants that could be detected by the considered tracing of types. These operators avoid introducing such changes after which the same types would not be supported and, therefore, avoid creating incompetent mutants. For example, an equivalence relation "==" would be not substituted by ">=" although in both cases a relational operator is used.

In Mutpy, the mutation testing tool for Python programs (Derezinska and Halas, 2014a), mutants are created while traversing an abstract syntax tree (AST). For each tree node, acceptable mutation operators are applied to generate mutants. In this set of mutants, some could be recognized as certainly incompetent and rejected from further analysis.

Checking a mutant incompetence relies on verification whether a mutated operation supports the types of the operation elements. A test is performed using variables of the same type as elements of the mutated operation. A mutated operation is realized in a Python *try* expression. If a 'TypeError' is caught, the operation is not supported by the types under concern. A mutation is allowed for this set of types, if no exception is caught. An operation could have many combinations of possible simple types and all such combinations are tested by MutPy. If all combinations of types have risen an error, the mutant will be certainly incompetent and will not be created.

MutPy will be enhanced by type analysis. The Type Analyzer associates variables with possible types which they could possess. A variable could be identified as having many possible types, or as a variable which type cannot be recognized.

Even the incomplete knowledge of variable types gives an opportunity to use other mutation operators, which could not be applied due to a large number of incompetent mutants generated by these operators. For example, information about the types of variables that could be substituted by constants could be used in the application of a mutation operator that swaps a variable with a constant of the same type. Therefore, we want to address the following research questions:

(1) Is it worthwhile to apply type analysis to cope with incompetent mutants in mutation testing of Python programs?
It is associated with the following detailed issues:
(2) Does the application of type analysis lower the number of incompetent mutants in mutation testing of Python programs?
(3) Does the application of type analysis save the time of mutation testing of Python programs?

3   TYPE ANALYSIS

An approach to type analysis of Python programs will be presented. It is based on tracing of variable types and processing of dedicated *Type_tree* structures.

3.1   Evaluating Types in Python

Certain information included in a Python program could help in revealing the types of variables. It could allow us to detect mutants that could be incompetent due to type error and avoid creating them.

In comparison to built-in types, tracing of the types created by a user is more complicated, as their attributes could be changed during a program execution. It is even possible to change attributes of single objects, resulting in a unique special type. Therefore, in this work we only focus on tracing built-in types, such as *int, float, bool, string, list,* and *set,* because their attributes do not change during a program execution.

Mutant creation could be supported by the following functions:
- obtain the type of newly created variables
- obtain the type of function parameters
- obtain the type returned by a function
- identify situations, when a variable could have many types
- assign possible types that could have a variable in different stages of a program execution.

Recognizing of a type that could be assigned to a variable would be impossible in several circumstances, such as: (i) function parameter without annotation, (ii) the type of a variable returned in a function call which type of the returned value is not known, (iii) the type of a variable taken from a container of a content having an unknown type.

This second situation is illustrated by the following example in which the type of a returned variable is not known. It could be an integer or a string of characters.

```
def unknown_type(answer:bool):
    a = 0
    if  answer:
        a = 'letter'
    return a
```

3.2   Determining Type of Variables

There are three sources of information that could be used for the identification of a variable type:

1. Annotations of functions
2. Annotations of variables
3. Assignment of a value to a variable

Annotations could inform about the types of created variables, function parameters, or values returned by functions. According to the Python paradigm, annotations are optional and have no impact on the variable type during program execution. However, it could be assumed that if annotations are present, they give information about programmer intentions and the corresponding variables, arguments, or return values are of the given types. Therefore, an annotation, if it is given, would be treated as a primary initial type of a variable.

A subsequent code extract shows an example of annotations for an argument, a return value of a function, and of a variable annotation:

```
def annotation_example(a:int):int
    b: int = 3
    return a:b
```

However, types of variables that have neither assigned values, arguments, or returned values, nor have annotations; or their annotations do not specify any type, cannot be identified using annotations.

The trird possibilty to determine a variable type is an assignment operation. A type of the left hand side variable becomes the type of the assigned object. Assignment operations encounter also in *for* loops, in which values are assigned to an iterated variable.

Specification of a variable type due to its assignment is sometimes impossible. An assigned object could have an unknown type, e.g., be a result of a function with an argument without annotation. In these cases, the variable type also remains unknown after assignment.

Annotations have been used as information for programmers and do not influence variable types, assuming there is no additional code to implement typing control. Therefore, in case of type conflicts, information originated from an assignment operator overpass this from annotations.

3.3   Tracing Types of Dynamic Variables

In a language with dynamic types, such as Python, the type of a variable could be further changed by an assignment operation. Consequently, the types identified by annotations or by the first assignments could be replaced. Identification of a variable type used at a particular mutation place, would require the observation of the variable changes in the execution paths of a program.

A variable could have many possible types. For example, a type could be changed in an *if* statement, and it could not be determined whether the condition is satisfied or not. Therefore, many possibilities are taken into account for a variable of this kind.

```
def many_type_example(condition):
    a =0
    if condition;
        a = ""
```

Tracing of type changes could be performed directly in Python code or at the AST of the code. The AST comprises all necessary program information, including annotations, and supports the semantic analysis, therefore type analysis could be based on AST and additional data structures.

During the type analysis, a program AST is scanned and for each variable a special structure, called *Type_tree*, is built. A Type_tree consists of two kinds of nodes: internal and external. An external node corresponds to a variety of potential types of a variable and comprises many internal nodes. In an external node, one of its internal nodes is marked as a current position of the variable. An internal node includes a single possible type of a variable and a reference to its external node. Apart of "real" types, an internal node can include a *void* type that denotes no change in the variable type. A type in an internal node can be overwritten if the variable is assigned to a value of another type. An external node with their internal nodes is deleted when its parent node has been overwritten.

A Type_tree is created and modified during the type analysis. However, Type_tree is not a static tree that gives information about a variable type in each moment of a program execution.

Type analysis is tightly coupled with the language grammar reflected in AST. According to the Python grammar (Python, 2020), a complex expression consists of one or more closures. Each closure has its header and its suite. Closure headers are at the same ident level and begin with the unique keywords. The suite contains a group of expressions controlled by the closure.

Scanning of types in complex expressions has been described by a set of automata-like specifications. The current state is determined by two factors: a kind of current position in AST and a phase regarding *try* expression processing. Taking into account all possible states, a set of rules has been identified. Each rule specifies modifications performed on the Type_tree, i.e., creation of external or internal nodes, category of created internal nodes, searching and selection of nodes, deleting of nodes, as appropriate.

The number of set-up nodes in Type_tree depends on the composite statements encountered in the program. External nodes created for *if* expressions have at most two internal nodes: one of the *if* body and the second of *else*, because constructions of kind *if elif* are treated in the same way as *if* inside of *else*. External nodes that handle *for*, *while* or *try* structures could have many internal nodes, as there are many places in which the expression could be got out of. Though, the number of internal nodes is limited by the number of *break* statements in a loop, or *except* closures of an exception handling expression.

For example, the variable x is annotated with *int* type, but the assignment inside the conditional statement could change the type to *float*.

```
def TypeTree_example(x:int):
    if x != 0:
        x = 1/x
    return x
```

During the AST processing, a Type_tree of the variable x has been developed. Figure 1 illustrates the development phases of the Type_tree. Dark nodes stand for internal nodes with a current position. In the bottom row, the external nodes are shown. Internal nodes have their types (*int* and *float*) or are empty (void). In step I, an empty node is created for the identified variable. The variable has its annotation *int* (II). Processing of the statement *if* results in the creation of a new node with the void type (III). After analysis of the assignment inside *if*, the type is changed to *float* (IV). The next internal node becomes void, as there are no clauses *else* or *elif* (V). The final structure indicates that at stage VI the variable could have one of two different types, *int* or *float*.
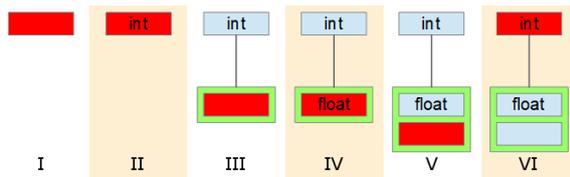

Figure 1: Evaluation of *Tree_type* of the variable x.

4 MUTATION TESTING WITH TYPE ANALYSIS

The provided type analysis in Python has been motivated by the advancement of mutation testing of Python programs. The developed solution has been combined with MutPy (its core of 2017 Nov 21) – a tool for mutation testing of Python programs (Derezinska and Halas, 2014a) (MutPy, 2021).

4.1 Extending MutPy with a Type Analyzer

MutPy creates and runs a set of mutants that are built from an original Python program using a set of mutation operators and taking into account the coverage results (Derezinska and Halas, 2014b). A few strategies for applying higher order mutation can also be selected.

Enhancement of MutPy with Type Analyzer allows to avoid the creation and running of these incompetent mutants that have been identified. The general flow of the mutation testing process with the type analysis is presented in (Figure 2). Actions and conditions supplemented by the type analysis are denoted by the '*' character.
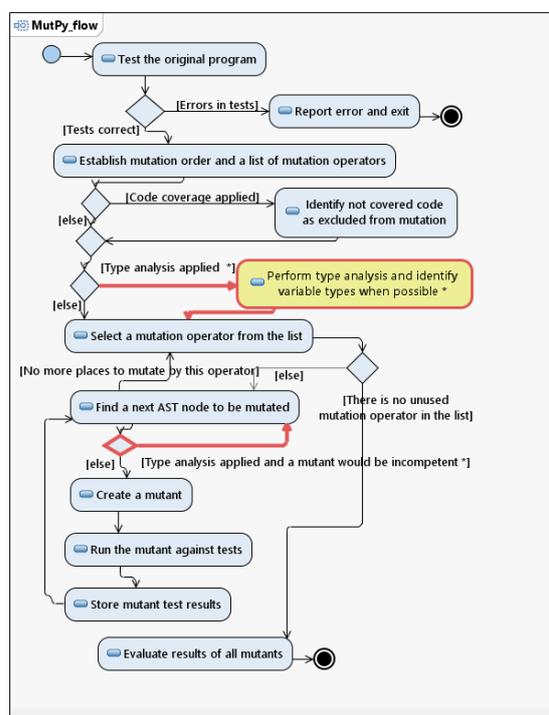

Figure 2: Mutation testing realization with type analysis.

During the initial phase, an original program is tested using a selected test suite. Next, a configuration of a mutant generator is established. It depends on the mutation order (first, second, …), a selection strategy in case of higher order mutation, a set of mutation operators to be performed, options about application of code coverage and type analysis. An abstract syntax tree of the original program is created. The further program analysis and manipulation is performed on the AST.

If code coverage is affected, code nodes not used in the tests are identified in AST in order not to be employed in a mutant creation. If the option of type analysis is allowed, it is performed at this stage and appropriate Type_trees are generated.

Then, for each mutation operator from the list, mutation testing is performed. A mutation operator scans the AST and looks for the nodes that could be mutated by the operator. If the type analysis was selected, realization possibility of a mutation in a located node is checked. If, according to the type analysis, the mutant is an incompetent one, this code position would be discarded and the mutant not created.

The created mutant is run with the tests and its testing results stored appropriately. After traversal of all AST nodes and processing of all mutation operators, the final mutation results are evaluated.

4.2 Experiments on Type Analyzer

The Type Analyzer has been implemented as a proof of concept, and the conducted experiments focused on the verification of its main capabilities in the processing of incompetent mutants. We wanted also to observe the impact of selected program construction and annotation usage on the type analysis and mutation testing results. The following six subjects have been used in experiments[2]:

1) A very simple program that includes a conditional statement (*if*) and arithmetic operations. No annotations are used.
2) The same program as (1) but with the application of annotations in function parameters.
3) A very simple program with two similar methods. One of the methods uses annotations, while the second does not.
4) A very simple program with many arithmetic operations on parameters of a method.

5) A program that calculates a game statistics. It manipulates on numbers and strings. Creates a relative high number of mutants.
6) A program that processes C++ code and creates its inheritance tree.

The numbers of mutants obtained in the experiments are given in Table 1. For each program, the results are provided in two rows: the upper row (denoted by "-") when no type analysis was used, and the bottom row ("+") including outcomes with type analysis. The number of mutants are given in four columns: (i) all generated mutants, (ii) mutants killed by a test set associated with the program, (iii) mutants not killed by the tests but not counted as incompetent mutants, and (iv) created mutants that were identified as incompetent during the program execution.

There is also another mutant category that could be recognized by MutPy during test execution, i.e., mutants abandoned by a time limit. For all programs discussed, there were no such timeout mutants.

Table 1: Number of mutants in mutation testing.

		Number of mutants			
		all	killed	not killed	incompetent
1	-	6	3	0	3
1	+	6	3	0	3
2	-	6	3	0	3
2	+	6	3	0	3
3	-	2	0	0	2
3	+	1	0	0	1
4	-	12	0	3	9
4	+	3	0	3	0
5	-	481	259	169	53
5	+	433	259	169	5
6	-	257	6..8	100..102	149..151
6	+	257	6..8	100..102	149..151

To compare the approach efficiency, execution times were also measured. Each program was executed 200 times. The average time values are presented in Table 2.

Table 2: Execution time (including mutant generation and testing time) in [s].

	Average execution time [s]	
	Without Type Analysis	With Type Analysis
1	0.0509	0.0527
2	0.0556	0.0596
3	0.0443	0.0479
4	0.1703	0.1675
5	9.4028	9.4247
6	11.0237	13.5065

---

[2]https://galera.ii.pw.edu.pl/~adr/MutPywithTypeAnalyzer/

Comparison of programs 1) and 2) points at longer processing time of an annotated program 2). This is probably due to the bigger number of AST nodes, as annotations are included in AST. This impact of annotations is higher when the type analysis is applied, because annotations are used in the type determination. In these programs, there were no mutants with TypeErrors, so the type analysis could not lower the number of incompetent mutants and caused only a slightly longer execution time because of the overhead provided. Detected incompetent mutants originated from a "by zero division" which was present in one of tests.

Programs 3) and 4) also differ in annotations. In 3), there were no annotations, and only two incompetent mutants, one of which was statically detected. The small number of mutants could not lower the execution time, which was a bit longer due to the overhead. In the annotated program 4), all incompetent mutants (9) were recognized and the execution time was a bit shorter than for 3). The number of incompetent mutants is 3 times higher than the number of killed mutants, so in this case mutation testing with the type analysis was beneficial.

The remaining programs, 5) and 6) were existing programs developed independently from these experiments. In 5), most of the incompetent mutants, 48 out of 53, were detected by the static type analysis. Even though, the approach does not spare the execution time. In this program, the more serious problem was a high number of nonkilled mutants (169), mostly equivalent ones. The type analysis was not dealing with this kind of mutants and all of them reminded in the second version of mutation testing.

Program 6) has no incompetent mutants caused by TypeErrors, so their number remains unchanged. The mutant numbers are expressed as ranges min and max over the set of all program executions. Incompetent mutants were mainly due to the calling of attributes or container spots that do not exist. This could be utilized in further development of the MutPy tool extension to avoid incompetent mutants based also on different reasons. The whole execution time with the type analysis was longer because of the overhead and no saving in the incompetent mutant number.

The presented experiments give a preliminary assessment of the proof of concept but cannot be counted as a representative set of professional Python programs. The possibility to generalize the outcomes in terms of threats to validity is bounded.

5 RELATED WORK

There are different approaches to handle types in the context of Python programs.

One of the approaches is realized by type controllers, such as mypy – an optional static type checker for Python (mypy, 2021). A type controller verifies the variables which types have been specified by a developer, obtaining in result a mixture of static and dynamic types in the same program. Mypy detects types based on annotation, logical reasoning, and a special kind of comments. The current mypy supports programs that use Python 3 function annotation syntax (conforming to Python Enhancement Proposal 484).

Although variable types are statically processed in both cases: type controllers and Type Analyzer, but these kinds of tools are aimed at different tasks. In the Type Analyzer the code is not directly analyzed and not modified and it is sufficient to scan AST produced by the Python scanner. MyPy introduces some changes into the program and has to interpret comments, which are excluded from the ordinary scanning.

The most important difference refers to the determination of the type of a variable. Type Analyzer works on dynamic types, i.e., the variable types are not stable but have to be traced. Type Analyzer has to deal with the situations, when many types are possible. Type controllers deal with static types that are supposed not to change. Variables without specified types remain dynamic, and their correctness is due to the Python compiler and not to the type controller.

Type Analyzer reports only about its internal errors and can assume that the syntax errors introduced by a programmer would be detected by the Python scanner. Whereas type controller verifies the static type rules and informs about any rule violation, also syntax errors might be reported if a code is changed.

In (Monat, Quadjaout and Mine, 2020), a static analysis was proposed to use type information to detect all exceptions that can be raised and not caught. The type analysis is flow-sensitive and takes into account the fact that variable types evolve during program execution and, conversely, run-time type information is used to alter the control flow of the program, either through introspection or method and operator overloading. Some limitations to the Python language are assumed.

Combination of static and dynamic typing capabilities, i.e., gradual typing, is presented in a language dialect – Reticulated Python (Vitousek, Siek and Baker, 2014). It provides type checking based on the annotations interpreted as if they were static

obligations. It is performed on AST during a module load time.

Mutation testing and annotations have been treated by (Gopinath and Walkingshaw, 2017). Though, it was focused on the evaluation of the annotation quality by mutation testing (MutPy) and static analysis supported by mypy.

One of the most comprehensive tools for mutation testing of Python programs has been MutPy (Derezinska and Hałas, 2014b). Other previous tools had a very limited number of mutation operators and do not cover any object-oriented or Python-specific features of the language. MutPy was used in different experiments and refactored to improve its quality (Derezinska and Hałas, 2015). The current basic version is available at (MutPy, 2021).

Recently, other tools for mutation testing of Python programs have been developed: Cosmic Ray (Bingham, 2017) and mutmut (Thoma, 2020). However, to the best of our knowledge, there is no detailed information about dealing with type errors and processing incompetent mutants in those tools.

6 CONCLUSIONS

In programs of dynamically typed languages, type correctness cannot be verified by compilers before program execution. Therefore, mutants that activate type errors can be created in the mutation testing. This kind of mutant could be killed by any test and is not supportive in the evaluation of a test suite quality. In this work, we have addressed the problem of incompetent mutants in Python programs.

A type analysis has been proposed that assists with the identification of a subset of potential incompetent mutants before creation and testing of a mutant. The approach has been implemented and integrated with MutPy – the mutation testing tool of Python. This proof of concept has been evaluated in preliminary experiments. As expected, the number of incompetent mutants could be lowered. This effect depends strongly on the selected mutation operators. Those mutation operators that are prone to generate incompetent mutants would benefit from the solution.

The type analysis adds overhead to the mutation process. Most of the work is performed once before mutant generation. Hence, the evaluation of a program with many mutation operators and many incompetent mutants could benefit from the type analysis, in comparison to the situation when only a few mutation operators selected to avoid incompetent mutants are applied. However, the preliminary results showed that the time overhead of type analysis could surpass the time lowering caused by a slight drop in the number of incompetent mutants.

Type analysis was intended to overcome obstacles in the application of a variety of mutation operators that would have been excluded or limited due to the creation of incompetent mutants. Time measurement does not confirm this supposition. However, detailed efficiency evaluation needs experiments on a more comprehensive set of programs. Combination of the approach with other mutation testing tools (Bingham, 2017), (Thoma, 2020) is an open question, as it is not known how they handle incompetent mutants.

REFERENCES

Bottaci, L., 2010. Type Sensitive Application of Mutation Operators for Dynamically Typed Programs. In: *Proceedings of 3rd International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE Comp. Soc. pp 126-131. doi: 10.1109/ICSTW.2010.56.

Bingham, A. 2017. Mutation Testing in Python, [Online] [Accessed 18 Jan 2021] Available from: balabit.github.io/coderegation/notes/2017-05-10-Austin-Bingham-Mutation-Testing-in-Python.

Derezinska, A., and Hałas, K., 2014a. Experimental Evaluation of Mutation Testing Approaches to Python Programs. In: *Proceedings of IEEE International Conference on Software Testing, Verification, and Validation Workshops*. IEEE Comp. Soc. pp. 156-164. doi: 10.1109/ICSTW.2014.24.

Derezinska, A., and Hałas, K., 2014b. Analysis of Mutation Operators for the Python Language. In: Zamojski, W., Mazurkiewicz, J., Sugier, J., Walkowiak, T., Kacprzyk, J.: (eds) *DepCos-RELCOMEX 2014*. AISC, vol. 286. Springer Int. Pub. Switzerland. pp. 155-164. doi: 10.1007/978-3-319-07013-1_15.

Derezińska, A. and Hałas, K., 2015. Improving Mutation Testing Process of Python Programs, In: Silhavy, R., Senkerik, R., Oplatkova, Z.K., Silhavy, P., Prokopova, Z. (eds.) *Software Engineering in Intelligent Systems*, AISC, vol. 349, Springer. pp. 233-242. doi: 10.1007/978-3-319-18473-9_23.

Gopinath, R. and Walkingshaw, E., 2017. How good are your types? Using mutation analysis to evaluate the effectiveness of type annotations. In: *Proceedings of 10th IEEE International Conference on Software Testing, Verification and Validation Workshops*, pp. 122-127. IEEE Comp. Society.

Monat, R., Ouadjaout, A. and Mine, A. 2020. Static Type Analysis by Abstract Interpretation of Python Programs. *In: 34th European Conference on Object-Oriented Programming, ECOOP*, No17, pp. 17:1-17:29. doi: 10.4230/LIPIcs.ECOOP.2020.17.

MutPy mutation testing tool for Python. [Online] [Accessed 17 Jan 2021] Available from: https://github.com/mutpy/mutpy.

mypy - optional static type checker for Python. [Online] [Accessed 11 Jan 2021] Available from: http://mypy-lang.org/.

Papadakis, M., Kintis, M., Zhang, Jie, Jia, Y., Le Traon, Y., and Harman, M., 2019. Chapter Six - Mutation testing advances: an analysis and survey. *Advances in Computers.* 112, pp. 275-378. Elsevier. doi:10.1016/bs.adcom.2018.03.015.

Pizzoleto, A. V., Ferrari, F. C., Offutt, J., Fernandes, L., and Ribeiro, M., 2019. A systematic literature review of techniques and metrics to reduce the cost of mutation testing. *Journal of Systems and Software.* 157, Elsevier. doi:10.1016/j.jss.2019.07.100.

Python documentation. [Online] [Accessed 13 Dec 2020] Available from: https://docs.python.org/3/.

Thoma, M. 2020. Mutation Testing with Python. [Online] [Accessed 18 Jan 2021] Available from: https://medium.com/analytics-vidhya/unit-testing-in-python-mutation-testing-7a70143180d8.

Vitousek, M. M., Kent, A. M., Siek, J. G., and Baker, J., 2014. Design and evaluation of gradual typing for Python. In: Black, A. P., and Tratt, L., (ed.), *DLS*, pp. 45–56, ACM.