Experimental Evaluation of Mutation Testing Approaches to Python Programs

Anna Derezinska and Konrad Halas
Institute of Computer Science
Warsaw University of Technology
Warsaw, Poland
A.Derezinska@ii.pw.edu.pl

***Abstract*ð Mutation testing of Python programs raises a problem of incompetent mutants. Incompetent mutants cause execution errors due to inconsistency of types that cannot be resolved before run-time. We present a practical approach in which incompetent mutants can be generated, but the solution is transparent for a user and incompetent mutants are detected by a mutation system during test execution. Experiments with 20 traditional and object-oriented operators confirmed that the overhead can be accepted. The paper presents an experimental evaluation of the first- and higher-order mutation. Four algorithms to the 2nd and 3rd order mutant generation were applied. The impact of code coverage consideration on the process efficiency is discussed. The experiments were supported by the MutPy system for mutation testing of Python programs.**

***Keywords*ð mutation testing; Python; dynamically typed language; higher order mutation**

I. INTRODUCTION

Dynamic typing of a programming language means that a programmer is not forced to define types of variables. Testing of dynamically typed programs is posing a challenge, as simple type inconsistencies cannot be detected by static analysis of a program and are only manifested at run-time.

Python is an interpreted programming language with dynamic typing system. Its reference implementation is a bytecode interpreter written as an open source software in C. The Python project was started by Guido van Rossum in 1991 and now is coordinated by the Python Software Foundation [1]. Python supports multiple programming paradigms, including the procedural and object-oriented approaches. Types are specified by the Python interpreter based on values assigned to a variable, and they cannot be determined before a program execution.

The mutation testing assumes modifying a program under test by injecting small changes to the code that are syntactically and semantically correct. This condition cannot be verified in Python programs, and in other dynamically typed ones, before the *mutant* is run. Applying a mutation that is incorrect due to types used in a modified expression, an *incompetent* mutant will be created [2]. In general, incompetent mutants include also *still-born* mutants that can be detected at the compilation stage and other static analysis. However, many mutants that are typically still-born for a statically typed language like Java could not be detected for the Python language. In this paper, we discuss these incompetent mutants that are syntactically correct, but violate dynamic binding properties of the type system at runtime.

A *competent* mutant can be running against tests and manifests a different behavior than the original program, while this difference is not caused by inconsistencies in types. A test is said to *kill* a competent mutant if it detects the behavior difference. If none of tests has killed a mutant, either the test suite is insufficient or the mutant is *equivalent* one.

One of solutions towards mutation of dynamically typed programs is the mutation of a running code [2]. In this paper we focus on another approach based on the experiences with Python programs. Firstly, we selected mutation operators that cover the broad aspects of the programming language, but avoided operators that generate many incompetent mutants [3]. All mutants are run against tests, but incompetent mutants are detected in the run-time and excluded from the considerations. Experiments performed on Python programs allow to assess the overhead that encounter in this approach. This pragmatic approach was implemented in the Mutpy tool supporting the mutation process of Python programs [4].

There are several methods to deal with the main obstacle of the mutation testing, i.e. the high execution cost [5],[6]. One of them is application of the higher order mutation testing [7]. A mutant is created by providing one change to one code position, so-called *first order mutation* (FOM), or many changes to the same program. The latter case is called *higher order mutation* (HOM).

The current version of MutPy was extended with higher order mutation facilities. We have examined how the mutation testing of Python programs can benefit from the HOM approach. The costs of dealing with incompetent mutants were also studied in the context of the higher order mutation testing. Moreover, the impact of a mutation process option on its cost was evaluated. The option referred to the combination of a code coverage-based method with mutant generation and test execution. In all experiments, the actual costs given in terms of the mutation time were measured.

The structure of the rest of the paper is as follows. Section II outlines briefly related work. Section III presents several issues of the mutation testing process applied to Python

programs. An experimental setup and the results are discussed in Section IV. Section V concludes the paper.

II. Related Work

A. Mutation of Dynamically Typed Programs

The problem of dynamically typed programs was addressed by Bottaci in [2]. He described the mutation analysis introduced during a mutant run-time, when information about type of program elements is available. Postponing the time of a mutation injection prevents creation of incompetent mutants. The ideas were discussed on examples of JavaScript, a representative of dynamically typed languages, although they were neither implemented nor experimentally verified.

B. Higher Order Mutation

The higher order mutation technique was investigated in the context of different purposes.

One of goals is creation of mutants that are better in error simulation than simple 1st order mutants (FOMs). A special category of HOMs that are hard to be killed can force creation of better tests [8][9][10].

Other studies concentrated on cost reduction in terms of lowering the mutant number [7] [11] [12]. The application of HOMs discussed in this paper follows those research. Another cost reduction context treated with the higher order mutation referred to the equivalent mutant problem [13][14].

Brief surveys on application of the higher order mutation can be found in [12] [14].

The research on the higher order mutation and the corresponding experiments were performed on C and Java languages. The approach was also applied to programs and queries [15], as well as at the system level based on Java programs [12]. To the best of our knowledge, it has not been studied for dynamically typed languages, including Python.

C. Code Coverage in Mutation Testing Process

Information about code coverage can be taken into account during the mutant generation process. This procedure was successfully applied for C# programs since the second version of CREAM [16][17].

Running of tests is another part of mutation process that can be influenced by the code coverage results. Tests that do not cover a mutated statement of a mutant are assumed not to be able to kill the mutant. In this way the number of pairs mutant-test was limited in some tools [18][19]. A base of Pitest, a tool recently developed for mutating Java programs [20], is promotion of combining the test coverage and mutation approaches. In the regression mutation testing ReMT [21] mutant-coverage checking is used for selecting the subset of tests that actually execute the mutant statement.

D. Mutation Tools for Python Programs

The first mutation tool (2002) dealing with Python programs was probably Pester based on the Java mutation tool Jester [22]. It introduces mutations at the source code level treated as simple text replacements without any validation. The tool has been not upgraded for many years.

In 2011 MutPy version 0.2 - a mutation testing tool based on a manipulation of an abstract syntax tree (AST) was published [4]. It was further refactored to MutPy version 0.3. It has an improved mutant generation algorithm and an extended set of mutation operators. The tool also supports cost reduction techniques, the higher order mutation and a code coverage option. The experiments discussed in this paper were conducted with MutPy v. 0.3. The most current version is 0.4.

PyMuTester [23] is a simple mutation tool, which provides only two mutation operators: *IfCondition Negation* and *Loop Skipping*. It introduces mutations into the Python abstract syntax tree. PyMuTester classifies mutants into three groups: killed, unreachable and alive, where the last two are stored as a mutation process result.

Mutant [24] is a proof of concept project, which modifies programs at the bytecode level and it supports only doctests, i.e. tests inside a function description. It provides three mutation operators: *ComparisonMutation*, *ModifyConstantMutation*, and *JumpMutation*.

Recently a new tool has been launched, which generates mutants based on a similar concept to MutPy at the AST level. This is Elcap [25] that provides currently the following mutation operators: *StringMutator*, *NumberMutator*, *ArithmeticMutator*, *ComparisonMutator*, *LogicalMutator*, *FlowMutator*, *YieldMutator* and *BooleanMutator*. Elcap seems to be the most comprehensive tool, except MutPy, but it supports no object-oriented or Python-specialized operators.

III. Mutation Testing of Python Programs

In this Section we discuss the main problems of the mutation testing process proposed for Python programs. We present how these issues were solved in the current version of the MutPy tool [4].

A. Dealing with Incompetent Mutants

In dynamically typed programs, variables have no types specified by a programmer before run-time. The types of operands in an expression, and hence the applicability of expression operators, are resolved during a program execution. For example, a valid expression is an addition of variables $a=b+c$. If the AOR mutation operator (Arithmetic Operator Replacement) is applied to this expression, the mutated code will be the following $a=b-c$.

If variables *b* and *c* are of the integer type (*int*), the original and the mutated code can be properly executed, although their results might be different. If these variables are of string type (*str*), the addition is a valid concatenation of strings, while subtraction of strings does not exist in Python. In the latter case Python will raise the *TypeError* exception, and the mutant will be *incompetent* one.

If a mutant is run under supervision on a mutation testing system the *TypeEror* exception can be handled by the system. The whole situation is transparent for a user, and the mutant recognized as incompetent does not influence the mutation

score calculated by the system. Yet, some overhead for generating and running this mutant increases the whole mutation time.

It is theoretically possible that the *TypeEror* exception was caused not by a mutation, and the tests did not capture it. This situation is, though, very unlikely. In our experiences with various mutation operators [3] and different Python programs all incompetent mutants were correctly classified.

B. Supported Mutation Operators

Different mutation operators were tested in the context of Python programs in preliminary experiments. Next, mutation operators implemented in four mutation tools were systematically reviewed, namely Mothra for Fortran [18], Proteum for C [26], MuJava for Java [27], and CREAM for C# [17]. However, many of those operators are not applicable due to a different structure of Python programs. New operators related to selected Python structures, such as *decorator*, *index slice*, and *loop reversing* [1] were also proposed. Some operators generate mostly incompetent mutants, hence were classified as trail ones and excluded from the further experiments [3]. The following 20 operators are included in the final set that is currently supported:

1) Structural mutation operators

AOD - Arithmetic Operator Deletion

AOR - Arithmetic Operator Replacement

ASR - Assignment Operator Replacement

BCR - Break Continue Replacement

COD - Conditional Operator Deletion

COI - Conditional Operator Insertion

CRP - Constant Replacement

LCR - Logical Connector Replacement

LOD - Logical Operator Deletion

LOR - Logical Operator Replacement

ROR - Relational Operator Replacement

2) Object-Oriented mutation operators

EHD - Exception Handler Deletion

EXS - Exception Swallowing

IHD - Hiding Variable Deletion

IOD - Overriding Method Deletion

IOP - Overridden Method Calling Position Change

SCD - Super Calling Deletion

SCI - Super Calling Insertion

3) Python-related mutation operators

DDL - Decorator Deletion

SIR - Slice Index Remove

Operator DDL deletes any Python decorator, e.g. *@classmethod*, from a function or a method. Operator SIR removes one argument from a collection slice., e.g. *x[:5:2]* is mutated to *x[::2]*.

C. Generation of Mutants - AST tree, node coverage

Mutation operators define program changes that are interpreted at the program source level. However, specified changes can be introduced at different levels of a program abstraction, e.g.: in a source code of C, a syntax-based tree, an intermediate language level, as bytecode for Java or CIL for .NET C#.

In the system under concern, a Python program is transformed into its standard form of the Abstract Syntax Tree (AST) [1]. The injection of a mutation is based on a manipulation of the tree nodes. The mutated tree is passed to the Python interpreter in the desired intermediate form. The current version of MutPy speeds up generation of mutants by working on the same AST for many mutation operators and mutation places.

One of factors that can influence a mutant generation process is consideration of the code covered by tests. The system runs the original program and collects data about AST nodes covered by tests. The solution was inspired by the AST instrumenting [28]. Consequently, we can generate either all mutants, or optionally only those mutants that refer to the AST nods covered by the tests. The latter case is especially beneficial for programs under development that have been not completed yet.

The details of mutant generation and evaluation algorithms are beyond the scope of this paper.

D. Killing Mutants

Execution of a mutant can take a very long time or be unlimited due to an injected code change. A delayed execution of a mutant that finishes in the reasonable time limits can be classified as alive if the result is equal to the original program, or killed elsewhere. A mutant with the much longer execution time in comparison to the original program, or a mutant without ñstopò will be treated as a killed one by the timeout condition.

We would like to have controlled the whole mutation process in an automated way. Therefore, mutants can be killed by timeout. While executing mutants, the MutPy system run two processes, one is a supervisor and the second one is devoted to a mutant with its tests. The supervisor process can stop a mutant if the time limit is exceeded. The threshold time was used to avoid situations, where a heavily overloaded computer could interrupt a mutant that was about to finish shortly its work.

The system measures execution time of an original program. Then, a maximum of this execution time and a given threshold time (1 sec in the current experiments) is calculated. The timeout limit is equal to this maximum multiplied by a given timeout coefficient (set to 5 in the current experiments).

As a result, two main categories of killing mutants are recognized by the system: *killing by tests* and *killing by timeout*.

E. Mutation Results

The result of a mutation process is expressed as so-called *mutation score* and calculated as follows:

$$MS(P,T) = (K + U) / (M - IM - E) \quad (1)$$

where P ï a program under test, T ï a test suite, K ï the number of mutants killed by tests, U ï the number of mutants killed by timeout, M ï the number of all generated mutants, IM - the number of incompetent mutants, E ï the number of equivalent mutants.

However, in the current process no equivalent mutants are automatically detected. Therefore, if no other data about equivalent mutants are present the approximate result is calculated under assumption of E being set to null.

F. Running tests - testing to the first kill, covered mutants

Each generated mutant is run under the system supervision against a given set of tests. In fact, the intermediate form of a mutant is interpreted, as Python is an interpreted language. There are different scenarios of running tests in the mutation systems, as for example two supported in Proteum [26] and Javalanche [19]: *partial mutation testing* - where a mutant is run until it is killed and *full mutation testing* **ï** running all tests against a mutant. The MutPy approach is based on the partial mutation testing. The testing of a mutant is finished when a mutant is killed by a test, or killed by timeout, or it is classified as incompetent, or all tests had been applied. It means that not all tests are run if it is not necessary, and hence the overall execution time is lowered, which was required in the current version of the tool. On the other hand, we do not count how many tests killed a mutant, as for example in the CREAM system [17].

An initial set of tests considered for one mutant can also be limited performing *selective testing*. In the MutPy system, there are so far two possibilities: either all tests are taken, or only tests that cover the mutant under concern are selected. The latter approach is used together with the option for generation only mutants covered by tests (Sec. III.C).

G. Higher Order Mutation

In the higher order mutation (HOM) more than one change is provided in a mutated program. Using this approach, the number of mutants generated and run with tests is decreased [7] [11] [12]. The lowered number of mutants results in shorter time of mutation testing.

The following four HOM algorithms for mutant generation discussed in [12] or [7] were considered for Python programs. Each procedure assumes that an ordered list of the lower level mutants exist. The general ideas about creating 2nd order mutants from the FOMs are shown below. The creation of 3rd order mutants was accomplished in an analogous way.

- *Between-Operators* (BTO) ï a 2nd order mutant is a combination of two not-used FOMs. The first FOM is taken from the top of the list. The second one is the first FOM that has been generated with a different mutation operator that the first selected mutant.
- *Each-Choice* (ECH) ï two successive, unused FOMs from the list are combined into a 2nd order mutant.
- *FirstToLast* (FTL) ï a 2nd order mutant is a combination of a not-used FOM from the top of the list with a not-used FOM from the end of the list.
- *Random* (RND) ï a 2nd order mutant is generated by a combination of two randomly selected FOMs that were not used before.

Applying the higher order mutation, several issues have to be decided.

Firstly, the same statement could be modified by several mutations. For example, in an expression **ï** x, the unary operator can be modified by the AOR mutation operator or the AOD one. We assume that such a situation should be avoided. This is resolved during modification of the abstract syntax tree of a Python program. The same node of the tree is not modified twice in one mutant, if necessary another mutation place is considered. Therefore, it is ensured that in a 2nd order mutant two different nodes of the syntax tree are changed, in a 3rd order mutant three nodes are changed, etc.

Another issue is an odd number of the first order mutants considered in generation of 2nd order mutants with algorithms FTL, ECH and RND. In this boundary case, a mutant with the lower order is generated. The same first order mutant is not used once again in another higher order mutant, if it is not assumed by the algorithm. The similar solution is applied to analogous situations with higher orders accordingly. In should be noted that in typical case of many mutants generated for a program this effect has no observable influence on the overall results.

IV. Experiment Setup and Result Analysis

In this Section we describe the performed experiments and discuss their results.

A. Experiment Subjects

The following four Python projects were used in the experiments:

- *astmonkey* ï a library for manipulation of abstract syntax tree of Python,
- *bitsring* [29] ï a library for creating and analyzing of binary data,
- *dictset* [30] ï a project implementing data structures supporting operations on a collection of sets,
- *Mutpy* v 0.2 ï a mutation tool of Python programs (a previous working version).

All projects were developed for a professional use and distributed with their unit test suits. The basic complexity measures of the projects and their tests are summarized in TABLE I.

TABLE I. SUBJECT OVERVIEW

Project	Classes		LOC		Test methods
	Program	*Tests*	*Program*	*Tests*	
astmonkey	3	4	689	302	66
bitstring	12	67	4217	4677	456
dictset	1	38	711	1598	173
MutPy	36	32	883	800	86

TABLE II. MUTATION TIME IN [S]

Project	Mac Book	IBM x3755				
	All	*All*	*Mod*	*Comp*	*Tests*	*Others*
astmonkey	46.6	98.7	25.0	19.0	54.5	0.3
bitstring	2355.0	2700.0	90.6	455.9	2151.6	1.9
dictset	26.3	60.4	15.1	8.8	36.4	0.2
MutPy 0.2	47.2	69.0	17.6	3.0	48.1	0.3

B. Experimental Environment

In the experiments, MutPy version 0.3 [4] ï a mutation testing tool for the Pyhon 3.x source code [1] was used. MutPy applies mutation on the abstract syntax tree (AST) level. It provides 20 mutation operators, including 15 standard and 5 object-oriented. MutPy was used to generate mutants, run mutants with tests and evaluate the results. MutPy supports higher order mutation and code coverage analysis to improve the mutation process efficiency.

In order to effectively run different projects on remote powerful processors, an additional *MutPyhelper* tool was developed. It supported fetching MutPy and the subjects from remote repositories and running experiments with different parameters on various cores of a server in parallel.

The experiments were performed mainly on an IBM x3755 server comprising four 16-core processors AMD Opteron 6276 2.33 GHz (in sum 64 cores) with 256 GB RAM. The reference experiments were also carried out on MacBook Pro having 4-core IntelCore i5 2.5 GHz and 4 GB RAM.

C. Mutation Time

The details of mutation time are presented in TABLE II. The whole mutation time (column *All*) denotes the time required by a project and one set of the input configuration (including one mutation order, one coverage option, but application of all mutation operators). Data given in TABLE II. refer to the first order mutation and no coverage analysis. The mutation time consist of the following time components: an injection of changes into an AST tree (column *Mod*), a transformation of the modified tree into the executable intermediate code format (column *Comp*), running mutants with tests (column *Tests*), and the remaining time (*Others*). The sum of the modification and compilation time is equal to the generation time of all mutants of a given order.

Analyzing the time distribution we can observe that the biggest part of the mutation time is devoted to running tests. Therefore, the reduction of the test execution numbers could give substantial benefits.

The times given for the multicore IBM server relate to execution of one project with one configuration set on one core. The overall mutation times of comparable experiments performed on the MacBook computer were about 12%-56% faster, as shown in the first column.

However, the experiments were carried out for many different configurations. Therefore, the usage on a multicore processor benefited from the execution of many experiments in parallel under the control of the MutPyhelper tool.

Further comparison of mutation times is given in the tables comprising results for experiments with the 1st, 2nd and 3rd order mutation, discussed in the next subsections.

D. Impact of Coverage on First-Order Mutation

The results of the 1st order mutation of Python programs are shown in TABLE III. The number of all generated mutants is given in the column *All*. These mutants were executed with tests and classified as killed by tests, killed by timeout, incompetent, or not killed. Their numbers are given in the subsequent columns.

For each project, two kinds of results are compared. The first row of any subject relates to the experiments in which no preliminary data about coverage by tests was taken into account. The second row corresponds to the experiments in which the mutants are generated only for such AST nodes that were covered by tests. In the column *Coverage*, the measures of the line coverage are given.

The number of mutants generated in the second case is lower than the number of all mutants. However, the number of mutants killed by tests and by timeout remained the same for all projects. The last column presents the mutation score. In the experiments with coverage, the mutation results are higher because the number of all considered mutants was lower. The mutation score is more precise if we are consciously interested in the parts of the code covered by tests. Otherwise, the additional tests should be generated for the remaining parts of the code and evaluated in the mutation process.

In both kinds of the experiments, the number of detected incompetent mutants was similar and took about 3%-16% of all generated mutants in dependence of the mutated program. Based on these moderate values, we can also approximate a mutation time overhead necessary to cope with incompetent mutants. Moreover, these mutants were automatically handled and they did not influence the mutation score calculation.

The number of test runs carried out during the experiments are also given in a column of TABLE III. When the code coverage was considered in the test selection, this number was significantly decreased. In order to kill the same mutants, it was enough to make from 6% to 30% of all test runs.

TABLE III. EXPERIMENTAL RESULTS FOR 1ND ORDER MUTATION AND PROGRAM COVERAGE

Poject	Coverage	Mutant number						Number of tests executed	Mutation time [s]	Mutation score
		All	*Killed by tests*	*Killed by timeout*	*Incompetent*		*Not killed*			
astmonkey	-	521	338	1	51	9.8%	131	22230	98.7	72.1%
	93%	489	338	1	48	9.8%	102	5711	81.3	76.9%
bitstring	-	3774	2759	30	162	4.3%	823	733260	2700.0	77.2%
	97%	3653	2759	30	162	4.6%	761	160997	1646.7	78.6%
dictset	-	311	183	0	50	16.1%	78	28500	60.4	70.1%
	97%	298	183	0	50	16.8%	65	8655	53.2	73.8%
MutPy 0.2	-	270	104	2	6	2.2%	158	16843	69.0	40.2%
	89%	140	104	2	6	4.3%	28	1059	30.6	79.1%

Therefore, the whole mutation time, given in the last but one column, was reduced. The mutation time decrease was about 44%-88%. This effective cost reduction was substantial. It originated from the shorter time of generating the lower number of mutants and, above all, from the shorter time of running the lower number of tests.

These observations are similar to the experiments with code coverage data performed with CREAM on C# programs [17]. Therefore, mutation tools should be combined with some code coverage facilities in order to generate mutants for the covered code only, at least as an option. This is not significant for programs fully covered by tests. However, it is especially important for programs that are under development and have been partially not covered by tests yet. In this way program extracts that are not completed are automatically excluded from the mutation process.

Another facility, implemented in MutPy is selective testing. In this case, only those tests are running that are covering the mutant under concern. This approach can reduce the mutation cost even for programs fully covered by tests. Other criteria of test selection or prioritization [21] could also be profitable, but had not been verified for Python programs yet.

E. Second and Third Order Mutation

The subject programs were also mutated using four algorithms for generation of 2nd and 3rd order mutants. The results of mutant generation and execution are presented in Tables IV and V. In the similar way as for the FOMs, the numbers of generated, killed by tests and by timeout, and incompetent mutants are shown in the subsequent columns.

1) Number of Generated Mutants

The decline in the number of generated mutants was consistent to the theoretical assessment based on the algorithms applied for the higher order mutation. The number of 2nd order mutants generated with ECH, FTL or RND algorithm was about a half lower than the number of the corresponding first order mutants. For the 3rd order mutation this number was about 33%.

In case of the Between-Operators (BTO) algorithm this number is not so predictable as it depends of the distribution of FOMs for different operators. The number of 2nd order mutants generated using BTO varied from 50% to 80%. The analogous number of 3rd order mutants was in the range 37%-74%. In both cases the percentages are related to the number of all FOMs.

2) Number of Test Runs

As expected, the decrease of the number of generated mutants also caused the lowering of the number of test executions. In the 2nd order mutation the decline of test runs was about 23%-63%, and in 3rd order about 10-44%. These values are related to the number of test executions with FOMs.

The change of the overall mutation time was influenced by three factors: the lower number of mutants, the decreased number of test executions, but on the other hand by increased time of mutant generation. Mutation time for the 2nd order mutation was equal, on average, from 40% to 83% of the first-order mutation time. The analogous time for the 3rd order mutation lasts from 25% to 90%.

3) Mutation Score

The mutation score calculated for the higher order mutations gave higher results than for the first order ones. In general, the lower number of mutants survived because in a mutant more than one change was injected. However, it should be noted that the results for different orders are not directly comparable, as the mutation score is calculated for the different number of all mutants. In case of 2nd order, the mutation score was increased for about 8%-21%, while for the 3rd order mutation for about 13%-40%.

The highest increase in the mutation score was for the MutPy 0.2 project and the 3rd order mutation with the BTO algorithm. It can be observed that the biggest differences in the mutation score were obtained for a project with the lowest initial result. The analogous results were reported in [7] for experiments in Java.

TABLE IV. EXPERIMENTAL RESULTS FOR 2ND ORDER MUTATION

Project	Algorithm	Mutant number						Number of tests executed	Mutation time [s]	Mutation score
		All	*Killed by tests*	*Killed by timeout*	*Incompetent*		*Not killed*			
astmonkey	BTO	418	277	0	122	29.2%	19	14052	82.1	93.6%
	ECH	262	184	1	33	12.6%	44	9953	69.9	80.8%
	FTL	262	211	1	42	16.0%	8	8599	68.9	96.4%
	RND	262	210	1	37	14.1%	14	8599	65.3	93.8%
bitstring	BTO	2288	2014	33	163	7.1%	78	203843	1559.4	96.3%
	ECH	1887	1527	23	117	6.2%	220	273542	1481.1	87.6%
	FTL	1888	1681	18	111	5.9%	78	174883	1091.3	95.6%
	FTL, Cov.	1863	1670	18	105	5,6%	70	63100	941.2	96.0%
	RND	1887	1664	16	120	6.4%	87	181779	1086.5	95.1%
dictset	BTO	167	112	0	43	25.8%	12	9726	39.9	90.3%
	ECH	156	102	0	39	25.0%	15	10362	37.7	87.2%
	FTL	156	106	0	41	26.3%	9	8404	36.4	92.2%
	RND	156	108	0	40	25.6%	8	8668	41.6	93.1%
MutPy 0.2	BTO	137	78	2	5	3.7%	52	7114	52.1	60.6%
	ECH	137	68	2	5	3.7%	62	7763	45.9	53.0%
	FTL	137	76	1	5	3.7%	55	7246	39.1	58.3%
	RND	137	76	1	6	4.4%	54	7309	39.0	58.8%

It should be noted that modification of the mutation process makes the results not comparable in general. For example, in the higher order mutation process the number of all mutants was lowered, therefore, the mutation score was a different measure for each order (first, second or third). This effect is different from the evaluation of other cost reduction techniques performed for C# programs, where the normalized metrics were used for the direct comparison of different approaches (e.g.: selective mutation, mutant clustering and sampling)[17][31].

In general, the number of not killed mutants was decreased. The observed also the lower number of potential equivalent mutants. However, the system did not supported any automatic analysis of equivalent mutants and we cannot forward the verifiable statistical data.

4) Incompetent Mutants

The number of incompetent mutants was in the most cases equal to or lower than the number for the 1st order mutation. This was caused by the lower number of all mutants. However, the lowering of the mutant number caused that the percentage of incompetent mutants was higher.

An exception is the BTO algorithm, for which in few cases the number of incompetent mutants was higher than the number of FOMs. This effect is consistent with the BTO accomplishing. Let us consider two mutation operators, one generating 10 valid FOMs and the second generating 2 incompetent ones. Hence, summing up there would be 2 incompetent FOMs and all (in this case 10) 2nd order mutants would be incompetent because in this situation the competent FOMs are merged with the incompetent one. An analogous situation occurred with, for example, the *astmonkey* project. The most of its mutants were generated by CRP and AOR operators, where the CRP 1st order mutants were valid and many AOR were incompetent. Therefore, we can observe a considerable increase of the number of incompetent mutants for *astmonkey* HOMs with the BTO algorithm.

F. Code Coverage Combined with 2nd and 3rd Mutation

Experiments on the higher order mutation were also performed with options taking into account code coverage during mutant generation and test execution, similarly as for FOMs. The results of the *bitstring* project and FTL algorithm are shown in the rows *FTL, Cov.* of TABLE IV. and TABLE V.

This mutation was the most effective one, generating the lower number of mutants and using the lowest number of test runs. However, the reduction of the mutant number was insignificant as the test coverage of *bitstring* was quite high (97%) and the number of mutants was already reduced by the HOM technique. The numbers of incompetent mutants were similar in both cases. The reduction of test runs was essential due to selection of the tests covering the mutants. Therefore, the overall mutation time was decreased.

TABLE V. EXPERIMENTAL RESULTS FOR 3RD ORDER MUTATION

Poject	Algorithm	Mutant number						Number of tests executed	Mutation time [s]	Mutation score
		All	*Killed by tests*	*Killed by timeout*	*Incompetent*		*Not killed*			
astmonkey	BTO	388	260	2	123	31.7%	3	9952	89.2	98.9%
	ECH	176	134	0	20	11.4%	22	6082	46.3	85.9%
	FTL	177	134	0	27	15.3%	16	5531	49.3	89.3%
	RND	176	137	0	37	21.0%	2	4860	51.6	98.6%
bitstring	BTO	1817	1616	20	175	9.6%	6	87886	981.2	99.6%
	ECH	1259	1051	21	88	7.0%	99	149531	923.2	91.6%
	FTL	1258	1103	13	94	7.5%	48	107521	790.7	95.9%
	FTL, Cov	1242	1084	13	92	7.4%	53	41321	-	95.4%
	RND	1258	1129	14	99	7.9%	16	73376	683.3	98.6%
dictset	BTO	117	82	0	32	27.4%	3	5053	31.1	96.5%
	ECH	104	73	0	27	26.0%	4	5370	32.5	94.8%
	FTL	104	68	0	31	29.8%	5	5050	30.5	93.2%
	RND	104	72	0	28	26.9%	4	5000	28.1	94.7%
MutPy 0.2	BTO	112	82	2	7	6.3%	21	5488	41.4	80.0%
	ECH	93	51	2	3	3.2%	37	4920	37.6	58.9%
	FTL	93	55	2	5	5.4%	31	4736	38.8	64.8%
	RND	91	59	2	4	4.4%	26	4249	37.7	70.1%

G. Threats to Validity

The external validity of experiments was lowered by using four practically used projects of a different size and various application domains. However, an evaluation of mutants originated from more programs would increase the representativeness of results.

An assessment of thousands of mutants cannot be done manually. Therefore, the threats of validity are concerned with approximations of results. The first approximation corresponds to the rule of classifying incompetent mutants. A ñTypeErrorò exception could be caused not only by an introduced mutation. However, in none of cases referring to the presented programs, as well as in experiments with other programs using the previous versions of MutPy, an exception caused by a different reason has been found. On the other hand, the number of incompetent mutants generated for the selected mutation operators and the 1st order mutation was usually about several %, except the *dictset* project with about 16%. The percentage of incompetent mutants was greater for the higher-order mutation mainly due to the reduced number of overall mutants, but in any case all incompetent mutants were automatically excluded from the mutation score calculation.

The second, more severe, approximation is related to equivalent mutants. In the discussed experiments, the equivalent mutants were not automatically classified.

V. CONCLUSIONS

To the best of our knowledge, the experiments presented here are the first comprehensive evaluation of the mutation testing applied to Python programs, especially dealing with the higher order mutation. In general, the experiments on HOMs confirmed the similar experiences and benefits as with other languages. The practical mutation approaches should also take into account code coverage both during mutant generation and selection of mutant-test pairs to be executed.

Dynamically typing of data is a distinguishing feature of Python programs in relation to other general purpose languages. This causes a problem of incompetent mutants that are invalidated due to a type inconsistency detected at run-time. We have showed one practical solution, in which a moderate number of incompetent mutants is generated and they are coped with the mutation system. The overhead for incompetent FOMs in order of 2%-16% can be accepted, especially when the whole mutation process is performed automatically without a user supervision and the accuracy of results is not declined.

Python programs are principally not very fast, as the language is interpreted. This may be seen as an obstacle for applying the mutation technique. Therefore, the latest MutPy version was extended with methods boosting the mutation process, such as the higher order mutation and coverage analysis. On the other hand, the enhanced reflexing and

modifying mechanism makes Python suitable for introduction of mutation.

The MutPy is so far the most mature tool for the mutation testing in Python. Apart from the reported research experiments, it has been using by students of an advanced dependability course, in which mutation testing methods on different programming languages are studied.

The current version of MutPy made a big progress in terms of implementation of mutation operators and the decline of the mutant generation and execution times. There are still several ideas that could contribute to a mutation cost reduction and to a user convenience. The future extension can take into account regression testing, test priorities, or parallel execution of mutants. In the praxis of developing bigger programs, the tests are often run not on a local computer of a developer, but remotely. Therefore, a mutation tool could be associated with a continuous integration server, as for example Jenkins [32].

Considering research experiments on Python programs, an open issue remains an extensive evaluation of equivalent mutants and further study of additional mutation operators. We could also examine benefits of a hybrid approach that combines two solutions, namely one - a mutation at the parser tree level with the detection and elimination of incompetent mutants at run-time, and second ï a mutation introduced at run time. The latter case would be devoted to those operators that could create many incompetent mutants, but are valuable from the testing point of view, also with mutation on simple and object values [2].

REFERENCES

[1] Python Programming Language, http://python.org

[2] L. Bottaci, ñType sensitive application of mutation operators for dynamically typed programsò, in Proc. of 3rd International Conference on Software Testing, Verification and Validation Workshops (ICSTW), pp. 126-131. IEEE Comp. Soc., 2010.

[3] A. DereziĆska, K. Hałas, ñAnalysis of mutation operators for the Python language,ò in Advances in Intelligent and Soft Computing, Springer Inter. Pub. AG, Switzerland, 2014, in press.

[4] MutPy, https://bitbucket.org/khalas/mutpy

[5] M. P. Usaola, P. R. Mateo, ñMutation testing cost reduction techniques: a survey,ò IEEE Software 27(3), pp. 80ï 86, 2010.

[6] Y. Jia, M. Harman, ñAn analysis and survey of the development of mutation testingò, IEEE Transactions on Software Engineering 37(5), pp. 649-678, 2011.

[7] M. Polo, M. Piattini, I. Garcēa-Rodrēguez, ñDecreasing the cost of mutation testing with second-order mutants,ò Softw. Test. Verif. Reliab., 19(2), pp. 111ï 131, 2009.

[8] Y. Jia, M. Harman, ñConstructing subtle faults using higher order mutation testingò, 8th IEEE International Working Conference on Source Code Analysis and Manipulationò, pp. 249ï 258, 2008.

[9] Y. Jia, M. Harman, ñMILU: A Customizable, runtime-optimized Higher Order Mutation Testing Tool for the full C language,ò in Proc. of the Testing: Academic & Industrial Conference - Practice and Research Techniques, IEEE Comp. Soc., 2008, pp. 94ï 98.

[10] Y. Jia, M. Harman, "Higher Order Mutation Testing," Information and Software Technology, vol. 51, pp. 1379-1393, 2009.

[11] M. Papadakis, N. Malevris, "An Empirical Evaluation of the First and Second Order Mutation Testing Strategies," in 3rd Inter. Conf. on Software Testing, Verification, and Validation Workshops (ICSTW), IEEE Comp. Soc., 2010, pp. 90-99.

[12] P. R. Mateo, M. P. Usaola, J. L. F. Aleman, ĂValidating 2nd-order mutation at system level,ò IEEE Trans. on Software Engineering, 39(4), pp.570ï 587, 2013.

[13] M. Kintis, M. Papadakis, and N. Malevris, ñIsolating first order equivalent mutants via second order mutation,ò 5th Inter. Conf. on Software Testing, Verification and Validation, IEEE, 2012, pp. 701-710.

[14] L. Madeyski, W. Orzeszyna, R. Torkar, M. Józala, ĂOvercoming the equivalent mutant problem: a systematic literature review and a comparative experiment of second order mutation,ò IEEE Trans. on Softw. Eng., in press.

[15] G. Kaminski, U. Praphamontripong, P. Ammann, J. Offutt, ñA logic mutation approach to selective mutation for programs and queries,ò Information and Software Technology, pp. 1137ï 1152, 2011.

[16] A. DereziĆska, A. Szustek, ñObject-Oriented testing capabilities and performance evaluation of the C# mutation system,ò in: T. Szmuc, M. Szpyrka, J. Zendulka (eds.) CEE-SET 2009, LNCS vol. 7054, 2012, pp. 229-242.

[17] A. DereziĆska, M. Rudnik, ñQuality evaluation of object-oriented and standard mutation operators applied to C# programsò, in C.A. Furia and S. Nanz (eds.): TOOLS Europe 2012, LNCS, vol 7304, Springer Berlin Heildelberg, 2012, pp. 42ï 57.

[18] K. N. King, A. J. Offutt, "A Fortran language system for mutation-based software testing,ò Software - Practice and Experience, 21(7), pp. 685-718, July 1991.

[19] D. Schuler, A. Zeller, ĂJavalanche: efficient mutation testing for Java,ñ in Proc. of the 7th Joint Meeting of the Europ. Soft. Eng. Conf. and the ACM SIGSOFT Symp. on the Found. of Soft. Eng., 2009, pp. 297-298.

[20] PIT, http://pitest.org/

[21] L. Zhang, D. Marionov, L. Zhang, S. Khurshid, ñRegression Mutation Testing,ò in Proc. of Int. Symp. on Software Testing, ISSTAô2012, pp.331-341, 2012.

[22] Jester, http://jester.sourceforge.org

[23] PyMutester,http://miketeo.net/wp/index.php/projects/python-mutant-testing-pymutester

[24] Mutant, http://github.com/mikejs/mutant

[25] Elcap, http;//github.com/sk-/elcap

[26] M. E. Delamaro, J. C. Maldonado, ñProteum-a tool for the assessment of test adequacy for C programs,ò in Proc. of the Conference on Performability in Computing Systems (PCS 96), 1996, pp. 79ï 95.

[27] Y-S. Ma, A. J. Offutt, Y-R. Kwon, ñMuJava: an automated class mutation system,ò Soft. Testing, Verification & Reliability, vol 15, no 2, pp. 97-133, June 2005.

[28] A. Dalke, ñInstrumenting the AST,ò http://www.dalkescientific.com /writings/diary/archive/2010/02/22/instrumenting_the_ast.html

[29] bitstring, https://code.google.com/p/python-bitstring/

[30] dictsect, https://code.google.com/p/dictsext /

[31] A. DereziĆska, ñA quality estimation of mutation clustering in C# programs,ò in W. Zamojski et al. (Eds.) New Results in Dependability & Comput. Syst., AISC 224, Springer Switzerland, 2013, pp.119-129.

[32] Jenkins - integration server, http://jenkins-ci.org/