Search-based Software Testing and Test Data Generation for a Dynamic Programming Language

Stefan Mairhofer, Robert Feldt, Richard Torkar
Dept. of Computing
Blekinge Institute of Technology
Karlskrona, Sweden
robert.feldt@bth.se

ABSTRACT

Manually creating test cases is time consuming and error prone. Search-based software testing can help automate this process and thus reduce time and effort and increase quality by automatically generating relevant test cases. Previous research has mainly focused on static programming languages and simple test data inputs such as numbers. This is not practical for dynamic programming languages that are increasingly used by software developers. Here we present an approach for search-based software testing for dynamically typed programming languages that can generate test scenarios and both simple and more complex test data. The approach is implemented as a tool, RuTeG, in and for the dynamic programming language Ruby. It combines an evolutionary search for test cases that give structural code coverage with a learning component to restrict the space of possible types of inputs. The latter is called for in dynamic languages since we cannot always know statically which types of objects are valid inputs. Experiments on 14 cases taken from real-world Ruby projects show that RuTeG achieves full or higher statement coverage on more cases and does so faster than randomly generated test cases.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and debugging—*Testing tools (e.g. data generators, coverage testing)*

General Terms: Experimentation.

1. INTRODUCTION

While complexity software systems have increased in recent years the markets have also matured and users require higher quality which puts additional stress on activities, such as testing, to increase quality [3]. Search-based software testing (SBST) can reduce time and effort by automatically generating relevant test cases. It has been successfully applied, especially in structural testing [7, 11]. However, a majority of studies are limited to simple input data types, such as numerical values. Numbers are very common as input data in real-world software but there are many other input data types that are frequently used, especially in object-oriented programs. Often parameters are objects themselves that maintain an internal state, or are complex and compound data structures that require appropriate initialization. In such situations, the generation of test data to pursue a specific goal, becomes more complex than it is for simple numerical values.

Another aspect is the generation of test cases for dynamic languages such as Ruby, Python and Javascript, which have grown in popularity in recent years. Dynamic languages share a number of runtime characteristics that are not, or only partially, available in static languages: dynamic typing, interpretation, and runtime modification [12]. Often they are not strict on the type of objects that are sent as arguments or produced in method invocations.

In this paper we introduce RuTeG (the Ruby Test case Generator), a tool written in Ruby that can create test cases for Ruby source code. The goal is the automatic generation of test cases to achieve full statement coverage. In this paper we focus on two aspects: (i) How can search-based techniques be applied on a dynamic programming language? and (ii) How can different test input data, such as objects and complex data structures, be generated? We focus on the dynamic typing aspect of dynamic programming languages, which makes it harder to constrain the search space since there are many more possible test data and parameters that could be valid.

2. BACKGROUND

There is no single definition of what constitutes a dynamic language, rather the term is used quite liberally within computer science. The common theme is that these languages allow and execute behaviors at runtime that other languages either do not allow at all or do only statically, at compile time [12]. Even if these behaviors can be emulated in nearly any language of enough complexity, dynamic languages inherently supports them and makes them easy to use.

Three main characteristics are commonly attributed to dynamic languages: dynamic typing, runtime modification and interpretation [12]. The latter refers to the fact that dynamic languages are often interpreted instead of being compiled. However, this has more to do with implementation techniques suited to support the dynamic aspects rather than to being an inherent property of the languages themselves. In practice, the effect for the programmer and user is that she does not have to think about whether or when the source code is actually compiled since it happens seamlessly and automatically [12].

Runtime modification is a central characteristic of dynamic languages. It often involves some type of reflection where objects can be queried on their type and for the code implementing a method, etc. Types, values and code can also be dynamically changed or added at runtime. The actual behaviors that are allowed vary from language to language. Also, the use of these features are not common in actual dynamic programming languages and often only a small percentage of code make use of such dynamic features, while the rest is designed in a more traditional manner. For these reasons we exclude the testing of code that makes use of runtime modification. However, dynamic typing is a central aspect of most programming languages considered to be dynamic, and this is our main focus for this study.

One of the dynamic languages that are growing in use and popularity is Ruby. Ruby is a fully object-oriented language, which means that everything is an object, including primitive types such as bytes, integers, booleans and chars. Ruby is dynamically typed, also expressed as Ruby having 'duck typing'. Objects are described by what they can or can not do, i.e. by the methods it responds to at runtime, instead of being associated to a specific type. When Ruby code is executed, the execution environment need not care what type an object has, only if it implements the methods that are called on it. Even if our focus is on testing code, with this dynamic typing, it is helpful for our implementation that Ruby has extensive support for reflection. Much information about objects can be gained by querying objects at runtime. These features make Ruby an attractive programming language, however it may also complicate the search for adequate test cases, because of its dynamic nature [19].

3. RELATED WORK

Search-based techniques have been extensively applied to generate test data and scaled up to industrial use [11, 16]. Harman and McMinn [6] conducted a theoretical and empirical study, comparing random testing, hill climbing, and genetic algorithms on a number of test projects. The outcome of the study showed that evolutionary algorithms are suitable in many situations when it comes to the generation of input test data for structural testing, whereas in some cases simpler search techniques perform surprisingly well, and are able to surpass evolutionary algorithms. They also proposed a memetic algorithm that combines different local and global searchers for improved performance.

The efficiency of the search depends not only on the used algorithm. Also the quality of the fitness function contributes to the success rate. It expresses the 'goodness' of test cases in a numerical value, and is used to guide the search. Watkins and Hufnagel [21] compared in their work different fitness functions that were used in previous studies. They divide the fitness functions in two major categories: Approximation level (or control-oriented approaches) and distance level (or branch-oriented approaches). The first, approximation level, is an indicator about how close the actual path taken, deviates from the target sub-goal. The second, distance level, examines the branch node and gives information about how close the test case was, in order to fulfill the branch condition. Some fitness functions are a combination of approximation and distance level. Examples for that are the fitness functions proposed by e.g. Wegener et al. [22].

3.1 Generating Complex Input Data

Zhao and Li [23] have developed an automatic test data generator in C/C++ for dynamic data structures. They divide pointer operations into four possible categories: Assignment, creation, deletion and comparison statements. The comparison between pointer values is further categorised into equal and unequal conditions. An accompanying table is maintained to keep track of the current values and constraints of pointers. Thus, along the search path, pointers are modified to satisfy predicate conditions, as long as they do not violate any constraints kept within the accompanying table. This approach was tested on a small number of test programs, which showed its applicability. However, this approach is limited to simple dynamic data structures, such as binary trees. In [2], Arcuri and Yao examined the possibility to generate even more complex data types—again not for a dynamic language—using search-based approaches. The experiment they conducted gave inconclusive results, even though the contribution, in their case, was mainly on examining different search algorithms' possibility to solve a testing problem.

Alshraideh and Bottaci [1], on the other hand, focused on the test data generation to cover branches with string predicates. They address in their study string equality, string ordering and regular expression matching. They applied a fitness function that depends on the string predicate. Thus, for string equality they use the binary Hamming distance, character distance, edit distance, and string ordinal distance, while for string ordering, the ordinal value method and single character pair ordering is applied. The search for adequate test data is done using a GA. To improve the efficiency of the search, the input domain is restricted to characters within an ordinal range from 0 to 127. Further the solution candidates are biased towards string literals that appear within the program under test. The experiment done in their study shows that the most effective result for string equality was obtained using the edit distance fitness function, while no significant difference was found in the fitness function for string ordering.

Marinov [10] developed Korat for creating test data that are structurally complex. Korat is implemented and generates test data for the statically typed Java language. Korat solves imperative predicates, added by the developer, by creating increasingly more linked, and thus complex, test data objects. The solver systematically searches for structures that are inputs to the predicates. The search is bounded by a limit on the size of the generated data.

Lakhotia et al. [8] combined a hill climb search with symbolic execution and a constraint solver to handle dynamic data structures involving pointers. Results were promising compared to a concolic unit testing engine.

3.2 Test Case Generation for Object-Oriented Programs

Tonella [15] presented one of the first approaches that applied search-based software testing to object-oriented programs for structural testing. GA was used in the study to generate an adequate sequence of object creation and method invocation, in order to maximize a given coverage criterion. The main focus was on the generation of a method call sequence, while input parameters were randomly generated.

Wappler and Wegener [18] introduced a new type of fitness function, when it comes to object-oriented programs, namely the method call distance. It penalizes test cases which terminates prematurely in a sequence of method calls, because of a possible runtime exception. Such a test case cannot reach the method under test and so is not a potential solution.

In another contribution, Wappler and Wegener [19] used strongly typed genetic programming to guarantee the feasibility of generated test cases. Feasibility in this context refers to the method call sequence, which guarantees that a method is only invoked after the creation of its object. Using GA and the method call dependency graph, a set of test cases was generated to achieve full branch coverage. The fitness function consisted of a composition of distance level, approximation level and method call distance. Later Wappler and Schieferdecker [17] described a method to generate test cases for maximizing branch coverage for non-public methods. Their idea was to start with a static code analysis to identify call points, which are method invocations to non-public methods. This information is then used to generate test cases that are rewarded by the fitness function if they are able to reach a call point, and penalized in case that they miss its target.

Using genetic algorithms for state-based testing is also relevant for testing object-oriented programs [9]. However, additional static or dynamic analysis would be needed to extract the states and transitions that the search could then be used to target; it is not clear that this can be done directly from an object-oriented program *per se*.

Worth taking into account in this context is also Pacheco's et al. feedback-directed random testing [14] and the usage of dynamic detection of likely invariants [4], which has been shown to efficiently generate effective test cases for object-oriented programs [13]. The difference to our approach is mostly the goal—where we focus on dynamic programming languages and they on Java and C#. With respect to dynamic languages, in this case PHP, Wasserman et al. [20] also analyzed runtime values, confirming that the approach is useful and can compete with static analysis. In this paper we also focus on a dynamic programming language (Ruby); however, we implement a search-based approach to generate *complex* input data types.

4. THE RUBY TEST CASE GENERATOR

In this section we introduce the tool RuTeG and its four main components.

4.1 Analyser

The analyser extracts information that is used later in the process to generate and adapt test cases. Since Ruby is a reflective and dynamic language, the analyser mainly performs its task at runtime, and does only a basic static code analysis. This means that the class under test (CUT) is loaded dynamically into the system and investigated. The analyser delivers a `CUTInfo` object, that contains information about the constructor and its arguments. Further it maintains a list of methods defined on the CUT. Every such method under test (MUT) is associated with a `MUTInfo` object. This in turn contains information about the MUT, such as its argument list, and the methods invoked for each of its arguments. Furthermore it keeps track of the coverage achieved during the search process in addition to the adequate or disqualified data generators.

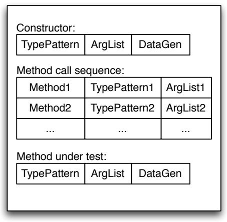

Figure 1: Representation of an individual.

4.2 Data Generator

Data generators produce input values that are passed as arguments to method invocations. Finding appropriate data is a very important although difficult task. There is a large set of possible input types and also the domain of input values for a specific type can be quite large. Since it is difficult for a single data generator, to cover all the different possibilities, a major design decision was to have different generators for specific problems. This gives the user the flexibility to define new generators that can produce context relevant data. Hence, the set of data generators must be modifiable, by adding or removing user defined generators. This however requires a common shared interface, such that the application can independently run, without changing its behaviour for each generator.

4.3 Test Case Executor

Generated test cases are executed, to obtain information, such as code coverage. The test case executor keeps track of the coverage achieved by previous test scenarios. Therefore it is possible to determine, whether the current test case contributes to the code coverage and, thus, if it will be part of the final set of test scenarios.

Test cases are divided into three major parts: the constructor; the sequence of method calls to modify the state of an object; and the invocation of the current method under test. In case the execution of a test scenario leads to an exception, it is possible to determine the responsible part. This is done to prevent a false evaluation in the search process.

4.4 Test Case Generator

The test case generator is the core of RuTeG and is responsible for producing test scenarios. There are two major tasks: *i*) to find appropriate input values, *ii*) to form a reasonable sequence of method invocations.

Both tasks can not be done right away and we employ a GA to search for a combination of valid input values and method invocations. The algorithm starts with a random initialisation of the population, which is then evolved to find fitter individuals. Each individual in the population is an encoded representation of a test case (see Figure 1 for an example). An individual itself can not be executed, but contains all necessary information to produce a complete test case: the constructor to create an object from the class under test, the method call sequence to modify the state of an object, and the invocation of the method under test.

The constructor consists of an argument type pattern, the argument list, and the data generators that were used for the creation of input values. The method call sequence is similar to the constructor section, with the exception that each method call contains additional information about the name of the method to be called. The invocation of the method under test contains information about the argument type pattern, the argument list, and which data generator was applied for the creation of the input value. Information about the argument type patterns and the applied data generators are not relevant for the creation of the phenotype, but it is used for statistical purposes and for simplifying the combination and mutation phases of the GA.

The search algorithm starts with a randomly initialised population of individuals. Each individual is transformed into an executable test case and evaluated. The evaluation is based on the fitness function, f, according to $f = (\epsilon \cdot \alpha) + \left(\frac{\rho_{\text{exec}}}{\rho_{\text{tot}}} \cdot (1 - \alpha)\right)$, where ϵ is the code coverage achieved by the test case, α is a weighting parameter between 0 and 1, ρ_{exec} the number of executed control structures, and ρ_{tot} the total number of existing control structures. Thus, the fitness value is a value between 0 and 1, where a value close to 1 indicates better individuals.

In our experiment we have used $\alpha = 0.5$, but for future work it might be beneficial to investigate (dynamic) adaptation of this parameter to overcome plateaus in the fitness landscape or extending the coverage tool so that full branch coverage can be used instead. Currently the information about covered control structures is extracted from the output of the coverage tool.

The combination of two individuals for the constructor and the method under test, concerns the argument list. In this case, the information of the argument list is exchanged at a randomly selected position. If the individuals to be combined, have argument lists of different length, the shorter one is padded with null arguments. After the combination all arguments after a null argument are discarded.

If individuals have two completely different type patterns, then the result may be a new combination of types. While the parents were applicable, in the sense of not leading to an exception when executed, one or both of the children may have inapplicable type patterns. We distinguish between applicable and inapplicable type patterns when evaluating the individuals and repeat the cross-over operation until applicable type patterns are found. We limit the number of retries and go back to select a new set of individuals for selection if the retries limit is reached.

The cross-over of the method call sequences of two individuals is done by selecting, randomly, two positions for each individual, at which the sequences after the chosen positions are substituted with each other. This cut-and-splice type of cross-over operator allows for growing or shrinking the length of the method call sequences.

The mutation operator is applied with a predefined probability to randomly selected individuals. For the constructor and the method under test, there are two possibilities of mutation: *i*) Generate a new type pattern. *ii*) Produce a new input value for one of the existing arguments. The generation of a new type pattern is applied to cover type combinations, that otherwise would not be tested. A separate table is maintained to keep track on already executed type combinations. In the case where all possible type patterns have been tested at least once, the mutation concerns only the argument value. For the mutation of the method call sequence, a position is randomly selected, at which a method is either added or removed from the current sequence. In case of the addition, a method from the class under test is randomly chosen and added to the sequence.

5. EXPERIMENT

In this experiment we wanted to test the applicability of RuTeG and examine which code portions are difficult to cover. We also wanted to evaluate the strength of the system in comparison to random test generation. Below we describe the random test generator we compared to, the parameters used for RuTeG and the test cases we used.

5.1 Random Test Generator

There are several possible random test generators that RuTeG could be compared to. However, in practice we argue that if any automated test generation is used, it is likely to be completely random rather than incorporating some learning aspect. We have thus opted to generate test sequences and input data fully randomly. The generation uses the constructor for the class to be tested and then randomly creates a sequence of method invocations of random length. The random length is bounded to be a maximum of two times the number of methods in the class under test. Method calls, data generators and their specific input data are then randomly generated to create a sequence of the given length.

Every test case is generated independent from previous test cases; there is no search or evaluation being applied for the random testing component.

5.2 RuTeG Search and Parameter Settings

The genetic search used in RuTeG uses a tournament selection method for the selection of individuals, a one-point crossover for the combination of argument lists and a cut and splice method for the combination of the method sequences. The cross-over rate is 1.0 since two parents are always allowed to be mated.

The mutation operator is applied with a probability of 0.2 and either affect the argument list by generating a new input pattern to cover new combinations that has not been used so far, or by generating new input values. The mutation operator can also affect the method sequence by adding or removing methods at a random position.

No specific constraints was used for the data generators. Only the base type generators for integers, floats, arrays, etc. was used as well as a test case specific ones such as an ISBN string generator for the ISBN test case.

Table 1 shows the parameter settings used in the experiment.

5.3 Test Methods Used in Experiment

A number of different test candidates were selected, that varied in their code complexity and structure as well as the complexity of input data they require. They ranged from classical code snippets, to more complex methods taken from the Ruby Standard Library and open source projects. Columns 1-4 in Table 2 lists the test candidates; below we give a brief summary of a select few. Projects can be found at `http://rubyforge.org/` and `http://raa.ruby-lang.org/`.

ISBN Checker. The ISBN checker is a small tool that takes a string as input, and checks whether it is a valid

Table 1: Parameter settings for the experiment.

Param.	Setting	Comment
Population size	50	
Selection method	Tournament	Tournament size = 4
Mutation rate	0.2	Mutate the input pattern or mutate an input value
Cross-over rate	1.0	One-point xover for argument lists, cut & splice for method call lists
Initial sequence length	Random	Random with a max length of two times the number of methods in the class under test, cut & splice cross-over can change length dynamically
Weight param. α	0.5	Half fitness on coverage, half on covering control structures

ISBN10 or ISBN13 code. An ISBN code is a sequence of numerical characters of length 10 or 13, whereas the last character can be any number between 0 and 9 or 'X'. **RBTree.** RBTree is a sorted associative collection using Red-Black Tree as the internal data structure. **RubyGraph.** RubyGraph is an implementation for directed and undirected graph data structures and algorithms. The tool includes a number of different graph algorithms, such as Breadth First Search (BFS), Depth First Search (DFS) and the Floyd-Warshall algorithm. **RubyChess.** RubyChess is a standalone chess engine that comes with a graphical Ruby/Tk user interface, to play chess against the computer.

The outcome of this experiment is compared with the results obtained by random testing. The random test case generator uses the `Analyser`, to receive information about the class under test and its methods, and produces test cases, by randomly selecting any type combination, data generator, and method sequence.

Each test candidate was tested 30 times, to obtain a good estimated result and to make sure that the data was consistent. A test run terminated when full code coverage was achieved, or when a predefined time was exceeded. This predefined time varied between different test candidates, since they differ in their complexity and required input data. However, the time constraint is the same regardless of the used test case generator. Experiments were run on a PC with an Intel Core Duo 2.00GHz CPU, 2GB of RAM and Windows XP SP2. The Ruby version used was 1.8.6.

6. RESULTS

Table 2 shows the average code coverage achieved by RuTeG and the random test case generator for each test candidate. RuTeG could achieve full code coverage in 11 of 14 cases, where the lowest average code coverage was 88%. On the other hand, the random test case generator could find test scenarios that cover all the code only in 4 of 14 cases. The lowest average code coverage achieved by random testing (RT) was 68%.

We then tested the null hypothesis that there is no observable difference between the coverage achieved by RuTeG and by random testing. After testing the data for normality, using the Shapiro-Wilk test at the 5% level, we applied the t-test on the data sets for each test case used. In cases where a difference was perceived, the probability that the results were obtained by chance is less than 5% (*) or even less than 1% (**). In Table 2 one can see that H_0 can be rejected on ten occasions.

Furthermore, Table 2 shows the time to maximum coverage for both test case generators. Hence, RuTeG does not only achieve higher code coverage, but also finds solutions quicker than the random test case generator. The difference in time is not always significant, but in such situations one can consider also the difference in code coverage. We consider the time needed as more significant than the number of iterations, generations or evaluations, since the absolute time makes for a more fair comparison between different techniques.

Figures 2, 3 and 4 show the results of some test candidates in a line chart and is typical of the other tests in the experiment. The solid line represents RuTeG, whereas the dashed

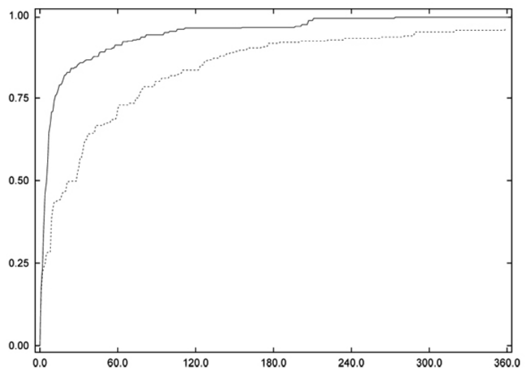

Figure 2: Average code coverage for ** (`power!`) method.

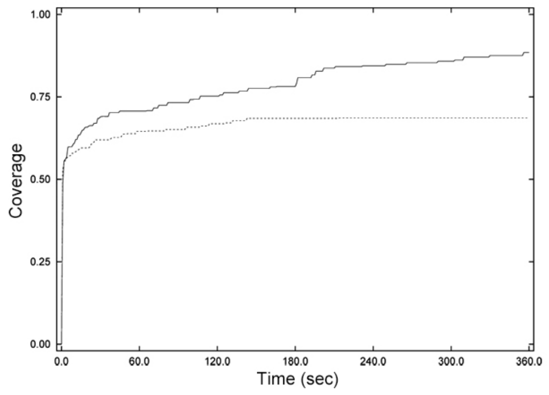

Figure 3: Average code coverage for `move` method.

Table 2: Average code coverage achieved by RuTeG and random testing (RT), with t-test where * indicates $p < 0.05$ and ** indicates $p < 0.01$; and the time to maximum coverage expressed in seconds.

Project	Method	SLOC	CC	Cov. RuTeG	Cov. RT		Time RuTeG	Time RT
Triangle	triangle_type	26	8	100%	81%	**	59	99
ISBN Checker	valid_isbn10?	18	7	100%	100%		29	84
	valid_isbn13?	13	6	100%	100%		34	80
AddressBook	add_address	10	3	100%	100%		56	97
RBTree	rb_insert	49	7	100%	88%	**	68	92
Bootstrap	bootstrapping	38	9	100%	86%	*	54	88
RubyStat	gamma	116	6	98%	92%	**	209	213
RubyGraph	bfs	39	12	100%	93%	*	79	86
	dfs	34	10	100%	96%	*	70	72
	warshall_floyd_shortest_paths	26	11	100%	100%		155	196
Ruby 1.8	rank	56	13	100%	92%	*	111	202
	** (power!)	59	16	100%	96%	**	274	356
RubyChess	canBlockACheck	23	10	94%	74%	**	285	333
	move	111	26	88%	68%	**	356	143
TOTAL (Average):		44.1	10.3	98.6%	90.4%		131.4	152.9

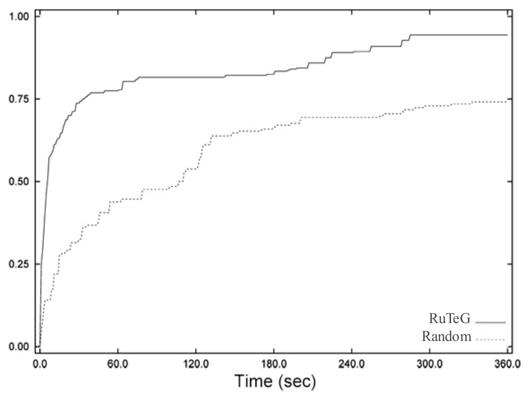

Figure 4: Average code coverage for `canBlockACheck`.

line represents the random test case generator. The average code coverage is displayed on the y axis and the time, expressed in seconds, on the x axis. From the line charts it is possible to see that in some cases, there is already at the beginning a major difference between the two results. This can be seen especially in Figures 2 and 4. There is also a difference observable at the end, where the average code coverage achieved by RuTeg is higher than the average code coverage of random testing. In Figure 2 the two results differ at the beginning, whereas the random test case generator is able to catch up with RuTeG as the time progresses. However, there is still a significant difference in the final result, in which RuTeG could cover more code than the random test case generator. From Figure 3 it can be seen, that both data generators could cover much code within a short time. However, the random test case generator hardly found new test scenarios that could contribute to the coverage, whereas the line of RuTeG shows a slow increase, which results in a higher code coverage at the end compared to random testing.

7. DISCUSSION

In this study, we presented a possible solution to automatically generate test cases for a dynamic programming language. Furthermore, the generation of input values is not limited on numbers and string values, but can produce different complex types of input data. We implemented a tool (RuTeG) in Ruby, that applies GA to produce test scenarios, with the goal to achieve full code coverage. RuTeG was tested in an experiment and compared with the results of random test case generation. The results show that the presented approach offers a possibility to automatically generate test cases for a dynamic programming language. In most of the cases, RuTeG could cover more code and find quicker solutions compared to the random test case generator. A clear improvement was not always observable, but for all test candidates, RuTeG was at least as efficient as random testing. There was no situation in which the random test case generator outperformed RuTeG.

A weakness of RuTeG could be observed in the generation of method sequences, especially when there is a strong dependency between the methods, such that a specific order is required. As long as there are only few methods that play an important role to satisfy a certain condition, it is possible to find adequate test scenarios. However, the more complex the method sequence becomes, the more difficult it is to find possible test cases. This could, for example, be observed while applying RuTeG on the Ruby/TK library.

A further limitation of RuTeG, is the set of data generators. When the standard or available data generators are not sufficient to find input values for a certain argument, then the specification of an additional data generator is required. RuTeG can not currently evolve data generators automatically such that better input values are produced. However, as the system is extended a reuseable set of generators can be built up much as described in [5].

RuTeG is able to test methods with required arguments, arguments that have a default value, and arguments of variable length. An unsolved problem is still the generation of code blocks as parameters. RuTeG currently generates an empty code block if required. This, however, does not guarantee the generation of successful test scenarios.

If we compare the implementation of RuTeG and the random test case generator the latter is, naturally, simpler. The core of RuTeG is the Test Case Generator, which implies a GA. The size of the RuTeG-specific code is 780 lines of code. Not included are the Analyser, Test Case Executor, and the different data generators, which are the same in both cases.

The applicability and efficiency of the tool was tested in an experiment on 14 openly available software artefacts. Even though the total number of test cases, and lines of code they cover in total is limited, they are representative of real-world Ruby code.

One characteristic of Ruby, which can also be found in other dynamic programming languages, is its reflective ability. This makes it easier to collect relevant information about classes and methods at runtime. In such a situation it does not matter, where parts of a class are defined, as long they are available when the object is created. RuTeG makes use of this ability to search for methods that may change the internal state of an object as well as identifying their arguments. However, for dynamic programming languages where run-time creation and addition of code is more common it is not clear that our approach could be directly adapted.

Another characteristic, that many dynamic programming languages have in common, is 'duck typing'. This makes it difficult to identify the input data for method invocations, also because an argument can be used in different ways. RuTeG presents a possible approach to classify such applicable type combinations and to disqualify inappropriate types. We think this type of mechanism is essential to limit the size of the search space that needs to be considered.

But also the usage of basic types may become quite complex, especially when there is only a small solution space, in which a certain condition can be satisfied. This may concern single arguments, but also a combination of arguments, which is the case for the triangle test candidate. In that situation, each argument depends on other values, and only if all three arguments have the same positive numerical value, then it is possible to form an isosceles triangle.

Another complexity factor is the sequence of method invocations. This may concern an object passed as argument, but also the object under test. Often it is not the input value that determines whether a specific code portion is executed, but the internal state of an object. In order to satisfy a certain condition, it may be necessary to call a specific method multiple times. But also the sequence of method calls may increase in complexity, especially when there is a dependency between each method, e.g. a specific order is required.

RuTeG addresses the different kinds of complexity with the definition and selection of specific data generators and the evolution of test candidates. This can help to find additional test cases that contribute to a higher code coverage, and is probably the reason for the better results in the experiment compared to random testing.

The fitness function we use is a very simple one and several extensions can be considered. For example, it has been common in literature to use some indication of the distance from boundary values in conditions that would make missed branches to be taken. We use no such measures in our approach and thus there is only a minimal of help in guiding the search. On the other hand, this strengthens our results since we still get better results than a random search; it is likely that we can improve coverage further by extending the fitness measure.

7.1 Validity Analysis

In our experiment we tested each test candidate 30 times, to obtain a good estimated result and to make sure that the data was consistent. The results were then presented as the average of all test runs. In addition we applied the Student's t-test, to analyse the statistical significance.

A possible threat to *internal validity* may be the comparison of the results with the random test case generator. There are different possibilities to implement such a generator. The implementation of the used random test case generator selects randomly one of the available data types and existing data generators, to produce a possible input value. This may not be the natural solution for static programming languages, where the input type is known and values randomly generated by a selected data generator. However, for dynamic programming languages, the situation differs, since we can not know which types are valid. Therefore, the random test case generator must randomly choose between all available data generators, if we want the same level of automation. This in turn may have some disadvantages for the random test case generator. The larger the set of available data generators, the less efficient is the random test case generator.

Construct validity addresses the issue whether a test measures what it claims to measure. A way to ensure construct validity, is to use multiple and different measures that are relevant for the purpose. We wanted to test the performance of the implemented tool on a number of test candidates. Therefore we measured the time that was needed to achieve a certain level of code coverage. In addition we wanted to test the quality of the tool, which was done by measuring the coverage achieved by the generated test cases.

External validity is related to generalisability. RuTeG makes use of Ruby's reflective ability. This is a characteristic that many dynamic programming languages have in common, whereas the information that they provide may differ from Ruby. Thus, RuTeG is partially a Ruby specific implementation. However, the core of RuTeG, namely the test case and data generator, is independent from Ruby specific code and thus applicable in any other dynamic programming language. Another possible threat to external validity could be the selection of test candidates, which was not chosen randomly from the population, since we wanted to have candidates to cover different criteria.

8. CONCLUSIONS

In this study we implemented RuTeG, a tool to automatically generate test cases for the dynamic programming language Ruby. RuTeG can be used for different kinds of input values. The system was tested on 14 test candidates, which differ in their code complexity and structure as well as the complexity of input data they require. The result of the experiment showed the applicability of the tool and that it was possible to find test cases to cover specific portions of code.

RuTeG could achieve full code coverage in 11 of 14 cases, while the random test case generator only succeeded on 4 occasions. In 10 of 14 cases RuTeG found test cases that showed a significantly higher code coverage than random generation. A difference was also observable in the time required to find possible test cases, where RuTeG could find solutions faster than the random test case generator.

The goal of RuTeG is to find test scenarios in order to cover as much code as possible. However, statement coverage is not a very strong coverage criterion. A possible future step would be aiming for branch or condition coverage; currently there are no tools to evaluate these coverage measures for Ruby code. Even so it is encouraging that RuTeG works well even with simplistic coverage criteria and choices.

The current version of RuTeG searches for applicable type combinations, and then for each type selects an adequate data generator. This intermediate step is not necessary. A more efficient solution would be to use directly the set of available data generators. In this case the system would search for a combination of applicable generators instead of data types, which may improve the performance but also the quality of generated test cases.

9. REFERENCES

[1] M. Alshraideh and L. Bottaci. Search-based software test data generation for string data using program-specific search operators. *Software Testing, Verification & Reliability*, 16(3):175–203, 2006.

[2] A. Arcuri and X. Yao. Search based software testing of object-oriented containers. *Information Sciences*, 178(15):3075–3095, 2008.

[3] A. Bertolino. Software testing research: Achievements, challenges, dreams. In *2007 Future of Software Engineering*, FOSE '07, pages 85–103, Washington, DC, USA, 2007. IEEE Computer Society.

[4] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 69:35–45, December 2007.

[5] R. Feldt. *Biomimetic Software Engineering Techniques for Dependability.* PhD thesis, "Dept. of Computer Engineering, Chalmers University of Technology", 2002.

[6] M. Harman and P. McMinn. A theoretical and empirical study of search-based testing: Local, global, and hybrid search. *IEEE Transactions on Software Engineering*, pages 226–247, 2009.

[7] B. Jones, H.-H. Sthamer, and D. Eyres. Automatic structural testing using genetic algorithms. *The Software Engineering Journal*, 11:299–306, 1996.

[8] K. Lakhotia, M. Harman, and P. McMinn. Handling dynamic data structures in search based testing. In *Proceedings of the 10th annual conference on Genetic and evolutionary computation*, GECCO '08, pages 1759–1766, New York, NY, USA, 2008. ACM.

[9] R. Lefticaru and F. Ipate. Automatic state-based test generation using genetic algorithms. In *Proceedings of the Ninth International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*, pages 188–195, Washington, DC, USA, 2007. IEEE Computer Society.

[10] D. Marinov. *Automatic testing of software with structurally complex inputs.* PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, USA, 2005. AAI0807757.

[11] P. McMinn. Search-based software test data generation: A survey. *Software Testing, Verification & Reliability*, 14:105–156, June 2004.

[12] T. Mikkonen and A. Taivalsaari. Using JavaScript as a real programming language. *Sun Microsystems Laboratories Technical Report TR-2007*, 168, 2007.

[13] C. Pacheco, S. K. Lahiri, and T. Ball. Finding errors in .NET with feedback-directed random testing. In *Proceedings of the 2008 international symposium on Software testing and analysis*, ISSTA '08, pages 87–96, New York, NY, USA, 2008. ACM.

[14] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball. Feedback-directed random test generation. In *Proceedings of the 29th international conference on Software Engineering*, ICSE '07, pages 75–84, Washington, DC, USA, 2007. IEEE Computer Society.

[15] P. Tonella. Evolutionary testing of classes. *SIGSOFT Software Engineering Notes*, 29(4):119–128, 2004.

[16] T. Vos, A. Baars, F. Lindlar, P. Kruse, A. Windisch, and J. Wegener. Industrial Scaled Automated Structural Testing with the Evolutionary Testing Tool. In *2010 Third International Conference on Software Testing, Verification and Validation*, pages 175–184. IEEE, 2010.

[17] S. Wappler and I. Schieferdecker. Improving evolutionary class testing in the presence of non-public methods. In *Proceedings of the 22nd IEEE/ACM international conference on Automated software engineering*, ASE '07, pages 381–384, New York, NY, USA, 2007. ACM.

[18] S. Wappler and J. Wegener. Evolutionary unit testing of object-oriented software using a hybrid evolutionary algorithm. In *CEC'06: Proceedings of the 2006 IEEE Congress on Evolutionary Computation*, pages 851–858. IEEE, 2006.

[19] S. Wappler and J. Wegener. Evolutionary unit testing of object-oriented software using strongly-typed genetic programming. In *Proceedings of the 8th annual conference on Genetic and evolutionary computation*, GECCO '06, pages 1925–1932, New York, NY, USA, 2006. ACM.

[20] G. Wassermann, D. Yu, A. Chander, D. Dhurjati, H. Inamura, and Z. Su. Dynamic test input generation for web applications. In *ISSTA '08: Proceedings of the 2008 International Symposium on Software Testing and Analysis*, pages 249–260, New York, NY, USA, 2008. ACM.

[21] A. Watkins and E. M. Hufnagel. Evolutionary test data generation: A comparison of fitness functions. *Software Practice and Experience*, 36(1):95–116, 2006.

[22] J. Wegener, A. Baresel, and H. Sthamer. Evolutionary test environment for automatic structural testing. *Information and Software Technology*, 43(14):841–854, 2001.

[23] R. Zhao and Q. Li. Automatic test generation for dynamic data structures. In *SERA '07: Proceedings of the 5th ACIS International Conference on Software Engineering Research, Management & Applications*, pages 545–549, Washington, DC, USA, 2007. IEEE Computer Society.