Master Thesis
Software Engineering
Thesis no: MSE-2008-08
June 2008

Search-based software testing and complex test data generation in a dynamic programming language

Stefan Mairhofer

School of Engineering
Blekinge Institute of Technology
Box 520
SE – 372 25 Ronneby
Sweden

This thesis is submitted to the School of Engineering at Blekinge Institute of Technology in partial fulfillment of the requirements for the degree of Master of Science in Software Engineering. The thesis is equivalent to 20 weeks of full time studies.

Contact Information:
Author(s): Stefan Mairhofer
E-mail: s.mairhofer@gmail.com

University advisor(s): Dr. Robert Feldt
Department: School of Engineering

School of Engineering
Blekinge Institute of Technology
Box 520
SE – 372 25 Ronneby
Sweden

Internet : www.bth.se/tek
Phone : +46 457 38 50 00
Fax : + 46 457 271 25

ABSTRACT

Manually creating test cases is time consuming and error prone. Search-based software testing (SBST) can help automate this process and thus to reduce time and effort and increase quality by automatically generating relevant test cases. Previous research have mainly focused on static programming languages with simple test data inputs such as numbers. In this work we present an approach for search-based software testing for dynamic programming languages that can generate test scenarios and both simple and more complex test data. This approach is implemented as a tool in and for the dynamic programming language Ruby. It uses an evolutionary algorithm to search for tests that gives structural code coverage. We have evaluated the system in an experiment on a number of code examples that differ in complexity and the type of input data they require. We compare our system with the results obtained by a random test case generator. The experiment shows, that the presented approach can compete with random testing and, for many situations, quicker finds tests and data that gives a higher structural code coverage.

Keywords: Search-Based Software Testing, automatic test data generation, dynamic programming language, object-oriented

CONTENTS

Paper

Appendix A

Appendix B

Appendix C

LIST OF TABLES

LIST OF FIGURES

ACKNOWLEGMENTS

I would like to thank my supervisor Dr. Robert Feldt for his support, guidance and constructive ideas throughout this work. This thesis would not have been possible without his valuable time. I am thankful for the numerous meetings and discussions that we had, which were a great inspiration and helped me to focus on the subject and most important issues.

HMT - FORMAT

The thesis is structured according to the 'Hybrid Master Thesis' (HMT) format, which was proposed in the summer 2007 by members of BTH, and is still in the experimental phase. The idea of the HMT format, is to have a hybrid form between an IEEE/ACM paper and a traditional master thesis. One of the reasons behind the HMT format is to increase the number of theses that can be published as papers. A further reason is to help students focus their writing and express themselves clearly.

According to the HMT format, the document should be divided into two major parts. The former part (Part A) follows the IEEE/ACM structure and focuses on the most relevant areas of the thesis project. It should comprise a maximum of 15 pages. The latter part (Part B) consists of a number of Appendices that cover in detail different aspects, such as methodology, validation and experiment. Part B should typically have a length of 15-40 pages, whereas it can grow upwards if necessary.

Search-based software testing and complex test data generation in a dynamic programming language

Stefan Mairhofer
Dept. of Systems and Software Engineering
Blekinge Institute of Technology
SE-372 25 Ronneby, Sweden
s.mairhofer@gmail.com

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 have mainly focused on static programming languages with simple test data inputs such as numbers. In this work we present an approach for search-based software testing for dynamic programming languages that can generate test scenarios and both simple and more complex test data. This approach is implemented as a tool in and for the dynamic programming language Ruby. It uses an evolutionary algorithm to search for tests that gives structural code coverage. We have evaluated the system in an experiment on a number of code examples that differ in complexity and the type of input data they require. We compare our system with the results obtained by a random test case generator. The experiment shows, that the presented approach can compete with random testing and, for many situations, quicker finds tests and data that gives a higher structural code coverage.

1. Introduction

The development of software products is a competitive activity and the time available to bring a product to market is often limited. Furthermore, the complexity and size of software systems have increased in recent years. In order not to fail on the market, it is important to also achieve a high quality. Together, these trends put big demands on development organisations; they need to develop more and larger systems quicker. This is often especially challenging for activities focusing on increasing quality, e.g. software testing.

Search-based software testing (SBST) can reduce time and effort by automatically generating relevant and adequate test cases. SBST has been successfully applied in previous studies, especially in structural testing [5, 7, 10, 15]. Although previous studies have shown the applicability of SBST to generate test cases for structural testing, they were often limited to simple types of input data, such as numerical values. Numbers are a very common input data 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 an appropriate initialisation of data. In such situations, the generation of test data to pursue a specific goal, becomes much more complex than it is for simple numerical values.

Another aspect is the generation of test cases for dynamic languages, which have grown in popularity in recent years. A dynamic language is a high-level programming language that allows a program to change its behaviour dynamically at runtime. Often they are not strict on the type of objects that are sent as arguments or produced in method invocations. Thus, it is possible to change parts of a program while the system is still running and to adapt software systems more easily [11]. Furthermore, dynamic languages are often seen as very flexible and productive. In previous research, SBST tools were developed for mostly statically typed languages, such as C/C++ [9] and Java [13]. To our knowledge, SBST has not yet been applied to a dynamic programming language and it is not known how SBST can be adapted for this.

In this paper we introduce RuTeG (**Ru**by **Te**st 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 mainly on two aspects:

- How can SBST be applied on a dynamic programming language for test case generation?
- How can different test input data, such as objects and complex data structures, be generated?

This paper is structured as follows. Section 2 gives background information on Ruby, genetic algorithms and structural coverage. Section 3 presents related work. Section 4 introduces the ruby test case generator RuTeG, and its components. Section 5 describes the experiment, while Section 6 contains the results. Sections 7–8 end the paper with discussion and conclusion, respectively.

2. Background

2.1. Ruby

This study is focused on the test case generation for dynamic languages. One such dynamic language is Ruby [12], which has continued to grow in popularity in recent years. Ruby is a fully object-oriented language, which means that everything is an object, including primitive types such as bytes, integers, booleans and chars. Another interesting feature of Ruby is 'duck typing'. Objects are described by what they can or can not do, instead of being associated to a specific type. Therefore the interpreter does not care what type the data is, but only if it can be used in a given context. Finally, a further characteristic is the reflective ability of Ruby, which means that much information about the code itself is accessible during 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 [18].

2.2. Genetic algorithms

The genetic algorithm (GA) is an example of an evolutionary algorithm and is a heuristic search technique that is inspired by Darwin's evolutionary theory [5]. The basic idea of the algorithm is to start with a randomly initialized population of individuals. Each individual is a potential candidate solution of a given problem. A fitness function is used to evaluate the adequacy and quality of each individual. After this, a selection process, which is biased towards the fitness associated to each individual, extracts a subset from the current population. This means that fitter solutions are more likely to be selected. These selected individuals are combined to form a new generation of population. The combination is usually done through a crossover operation, which takes two individuals and exchanges their information at a random selected position. Often a mutation process is applied, to prevent that individuals become too similar and thus that the population freezes. The mutation operation modifies randomly some information of a selected individual. After the generation of a new population, each individual is evaluated again, and the process is repeated, until a specific termination criterion is fulfilled. Figure 1 shows the flowchart of a simple genetic algorithm.

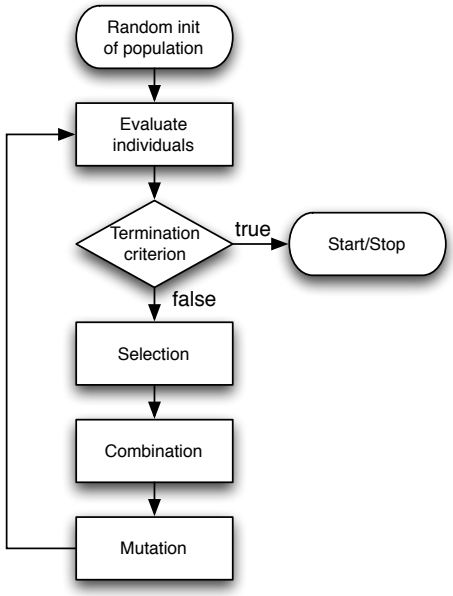


Figure 1. Flowchart of a genetic algorithm

2.3. Structural coverage

Structural coverage is an indirect measure of test quality. It can be used to identify areas of code that is not covered by test case scenarios. If parts of the source code remain uncovered, means that it was not tested. Therefore the current set of test cases should be extended. On the other hand, full coverage does not guarantee the correctness of the code. There are many different coverage criterions, whereas some are more powerful than others. Probably the best known are statement coverage, decision or branch coverage, modified condition/decision coverage, multiple condition coverage, and path coverage.

3. Related work

This section gives an overview of previous work in search-based software testing, especially for the automatic generation of test cases for structural testing.

3.1. Search-based test data generation

Jones et al. [5] have developed an automatic test data generator in Ada83 using GA to search input data to achieve full branch coverage. The input variables, that form the individuals of the population, are encoded into a concatenated bit string, on which search operations are applied, to create new generations. In order to fulfill the search goal and thus to reach all branches, the system moves in a breadth-first order within a control flow tree, from one branch to the next. To determine the suitability of an individual for each sub-goal, the authors use two different approaches based on the branch distance level, namely the Hamming distance and a simple numerical reciprocal function.

Pargas et al. [10] used an alternative implementation of GA to achieve full statement and branch coverage. The implemented tool TGen was written in the C programming language. Unlike in the work of Jones et al. [5], their evaluation is based on the control dependency graph. Further the individuals of the population are represented in its natural solution space, without applying any binary encoding. Thus each individual is a set of values according to the number of input variables.

GA is only one of many heuristic search techniques that were used to search for adequate test data. Previous studies have used a number of different search algorithms, such as gradient descent [6], simulated annealing [14], tabu search [3], immune genetic algorithm [2], particle swarm optimisation [21], to name but a few.

Some studies were focused on the comparison of different heuristic search techniques. Michael et al. [8, 9] have developed a tool in C/C++ called GADGET, to generate test data for condition-decision coverage. The evaluation of test cases is based on a simple branch distance measure. The search methods supported by GADGET are, random testing, gradient descent, simulated annealing, genetic algorithms, and differential genetic algorithms. The tool was tested on a number of projects, to determine strengths and weaknesses of different search techniques. The results showed that in most cases heuristic search techniques perform very well, while random testing fails to reach specific targets.

Harman and McMinn [4] conducted an empirical study, comparing random testing, hill climbing, and genetic algorithms on a number of test projects, with different size in terms of line of codes. The goals of the study were to determine when evolutionary algorithms are suitable and the performance, compared to other search techniques. The fitness function used in their study for the test case evaluation is a combination of approximation and distance measure. 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 surprising well, and are able to surpass evolutionary algorithms.

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 [19] compared in their work different fitness functions that were used in previous studies. They divide the fitness functions in two major categories, namely 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 fitness function used by Pargas et al. [10] falls into this category. 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. The fitness function used by Jones et al. [5], falls into this category. Some fitness functions are also a combination of approximation and distance level. Examples for that are the fitness functions proposed by Wegener et al. [20] and Tracey et al. [15].

3.2. Test data generation for complex input data

To the best of our knowledge, there are only few studies which are related to the generation of non-numerical input data. Zhao and Li [22] have developed an automatic test data generator in C/C++ for dynamic data structures. They divide pointer operations into four possible categories, namely assignment, creation, deletion and comparison statements. The comparison between pointer values is further categorized into equal and unequal conditions. An assistant 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 they do not violate any constraints kept within the assistant table. This approach was tested on a small number of test programs, which showed its applicability. However, this approach is limited on simple dynamic data structures, such as binary trees.

Alshraideh and Bottaci [1] 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.

3.3. Test case generation for object-oriented programs

Tonella [13] 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 maximise a given coverage criterion. The main focus lied on the generation of a method call sequence, while input parameters were randomly generated. The most important role was the mutation phase of test cases, which was categorised into different parts, namely the mutation of input value, constructor change, insertion/removal of method invocation, and one-point crossover. The presented solution was tested on a set of code examples, and showed that GA's are suitable to generate test sequences for object-oriented programs.

Wappler and Wegener [17] introduced, in their contribution, a new type of fitness function (when it comes to object-oriented programs), namely the method call distance. It penalises test cases which terminates prematurely in a sequence of method calls, because of a possible runtime exception. Such test cases are not able to reach the method under test, and therefore not to consider as a potential test case solution.

In another contribution, Wappler and Wegener [18] 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. The idea of the solution is based on a method call dependency graph, which is a bipartite, directed graph with methods on one side and classes on the other side. A link from a method to a class means that the method can only be called, after an instance of the class is created, while a link from a class to a method means that an instance of a class is created or delivered by the method. Using GA and the method call dependency graph, a set of test cases were generated to achieve full branch coverage. The fitness function consisted of a composition of distance level, approximation level and method call distance.

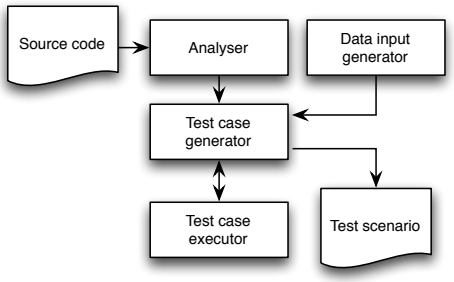


Figure 2. Basic structure of RuTeG

Finally, Wappler and Schieferdecker [16] described a method to generate test cases for maximising branch coverage for non-public methods. Their idea is 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 penalised in case that they miss its target. Their solution is built on the test case generator presented by Wappler and Wegener above.

4. The Ruby test case Generator

In this section we introduce the tool RuTeG, an automatic test case generator for Ruby. Figure 2 shows the basic structure and the main components of the tool.

The system starts by receiving the input source code and a selected class under test (CUT). Having these prerequisites, the analyser loads dynamically the CUT and searches for useful information that is needed for the test case generator. The test case generator produces a set of test individuals, by forming a sequence of object creations and method invocations. The input data for the parameters are generated using the data generator. After a complete initialization of individual test cases, they are executed on the program under test. From this, the test case generator receives feedback that

influences the successive steps of the search. Once the search goal is achieved, the tool delivers the set of test case scenarios.

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 focuses on performing its task at runtime, and does only a minimum of static code analysis. This means that the 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 that are declared within the target class. Every method becomes a method under test (MUT), and thus it 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 single argument. Furthermore it keeps track of the coverage achieved during the search process in addition to the adequate or disqualified data generators.

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.

Having problem specific data generators, improves also the search for adequate test values. Let us suppose that we have a method that takes a `String` as input and checks then whether it is a valid ISBN code or not. If we apply a simple `String` generator, which randomly produces any sequence of character, then it will be very difficult, if not impossible, to find a valid ISBN string. On the other hand, if we define a `String` generator, that produces only values according to a predefined pattern, then the success rate to find a valid input value will increase.

A further advantage of having multiple data generators for specific problems, is that they can be easily combined to produce more complex data. For example, a generator that produces an array, could use a positive `Fixnum` generator to define the size, and then fill it with random data of a specific type with other existing generators. This array generator in turn could be used again to produce nested arrays.

4.3. Test case executor

Generated test cases are executed, to obtain information such as the 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, namely 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 that 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, namely to find appropriate input values and to form a reasonable sequence of method invocations. Both are something that can not be done right away. Indeed, a Genetic Algorithm (GA) is used to search for possible test cases. Thus, a population of test individuals is maintained, which evolves during the search process in hope of finding 'good' test cases.

An individual itself can not be executed, but contains all necessary information to produce a complete test case. This information can be divided into three different categories, namely the constructor, the method call sequence, and the invocation of the method under test. The representation of an individual can be seen in Figure 3.

The search algorithm starts with a randomly initialized population of individuals. Each individual is transformed into an executable test case and evaluated. The evaluation is based on the following fitness function,

$$f_{\text{fitness}} = (\text{cov} \cdot p) + \left(\frac{\text{executed\_cs}}{\text{total\_cs}} \cdot (1-p) \right)$$

where p is a value between 0 and 1, *cov* is the code coverage achieved by the test case, *executed_cs* the

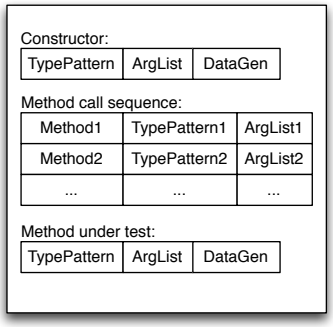


Figure 3. Representation of an individual

number of executed control structures, and *total_cs* 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 then a value close to 0.

Individuals are selected from the population, whereas fitter individuals are more likely to be selected than others. They are then combined and eventually mutated to form a new generation of a population. These operations are applied differently on each part of an individual.

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. On the other hand, the combination of the method call sequence is done by selecting randomly two positions for each individual, at which the sequence is replaced.

Similar is the case for the mutation, which is applied with a predefined probability to randomly selected individuals. For the constructor and the method under test, there are two possibilities of mutation, namely to generate a new type pattern, or to 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 case that all possible type patterns have been tested at lest once, then 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 form the current sequence.

Since Ruby features duck typing, it is not possible to determine from the method signature, which types of parameters can be applied. Furthermore, through the combination and mutation of individuals, it may happen that new combinations of types are produced. Such a possible situation shows the following example.

Table 1. Test candidates

Test Projects[1]	**Methods**
Triangle[2]	triangle_type
ISBN Checker	valid_isbn10? valid_isbn13?
AddressBook	add_address
RBTree	rb_insert
Bootstrap	bootstrapping
RubyStat	gamma
RubyGraph	bfs dfs warshall_floyd_shortest_paths
Ruby 1.8[3]	rank ** (power!)
RubyChess	canBlockACheck move

[1] Projects can, if not otherwise stated, be found at `http://rubyforge.org/` and `http://raa.ruby-lang.org/`
[2] Custom defined test candidate
[3] Ruby Standard Library

Let us suppose we have a method that takes as input two parameters and applies the + operator. In this case, possible input type patterns are (`Fixnum`, `Fixnum`) or (`String`, `String`), but any other combination such as (`Fixnum`, `String`) or (`String`, `Fixnum`) results in an exception.

To reduce the number of raised exceptions, the system learns to distinguish between applicable and inapplicable type patterns. During the search process, operations may lead to new type combinations. Such a combination is tested for applicability, and in case that the pattern is inapplicable, the operation may be repeated to find a better solution.

The system learns also to distinguish the quality of data generators. Thus, if a new input value of a specific type is required, then the selection is biased towards better generators. The evaluation of data generators is based on the fitness value assigned to test cases. Data generators that produced input values which led to better solution candidates, will result in a higher evaluation.

5. Experiment

In this experiment we want to test the applicability of RuTeG and examine which code portions are difficult to cover. Therefore, a number of different test candidates are selected, that vary in their code complexity and structure as well as the complexity of input data

they require. They range from classical code snippets, to more complex methods taken from the Ruby Standard Library and open source projects. The test candidates are listed in Table 1.

The outcome of this experiment is compared with the results obtained by random testing. The random test case generator uses the `Analyser`, to get 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 is tested 30 times, to obtain a good estimated result and to make sure that the data is consistent. A test run terminates when full code coverage is achieved, or when a predefined time is exceeded. This time varies 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.

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 was 68%.

We wanted to be sure that these results were statistically significant and not obtained by chance. Thus, we considered the following null hypothesis:

H_0: there is no statistical difference in the results between the two generators

H_1: there is a statistical difference in the results between the two generators

The Student's t-test determines whether the means of the two results are statistically different from each other. Furthermore, the normality of the data was tested by use of the Shapiro-Wilk method. 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% (**). Therefore we can reject H_0 and consider the results obtained by the two generators as statistically different.

Furthermore, Table 2 shows the time to maximum coverage for both test case generators. It can be seen that 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 we have to consider also the difference in code coverage.

Figure 4 shows the results of some test candidates in a line chart and is typical of the other tests in the experiment. The solid line represents the empirical data obtained with RuTeG, whereas the dashed line represents the data of 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 4(a) and 4(d). 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 4(b) 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.

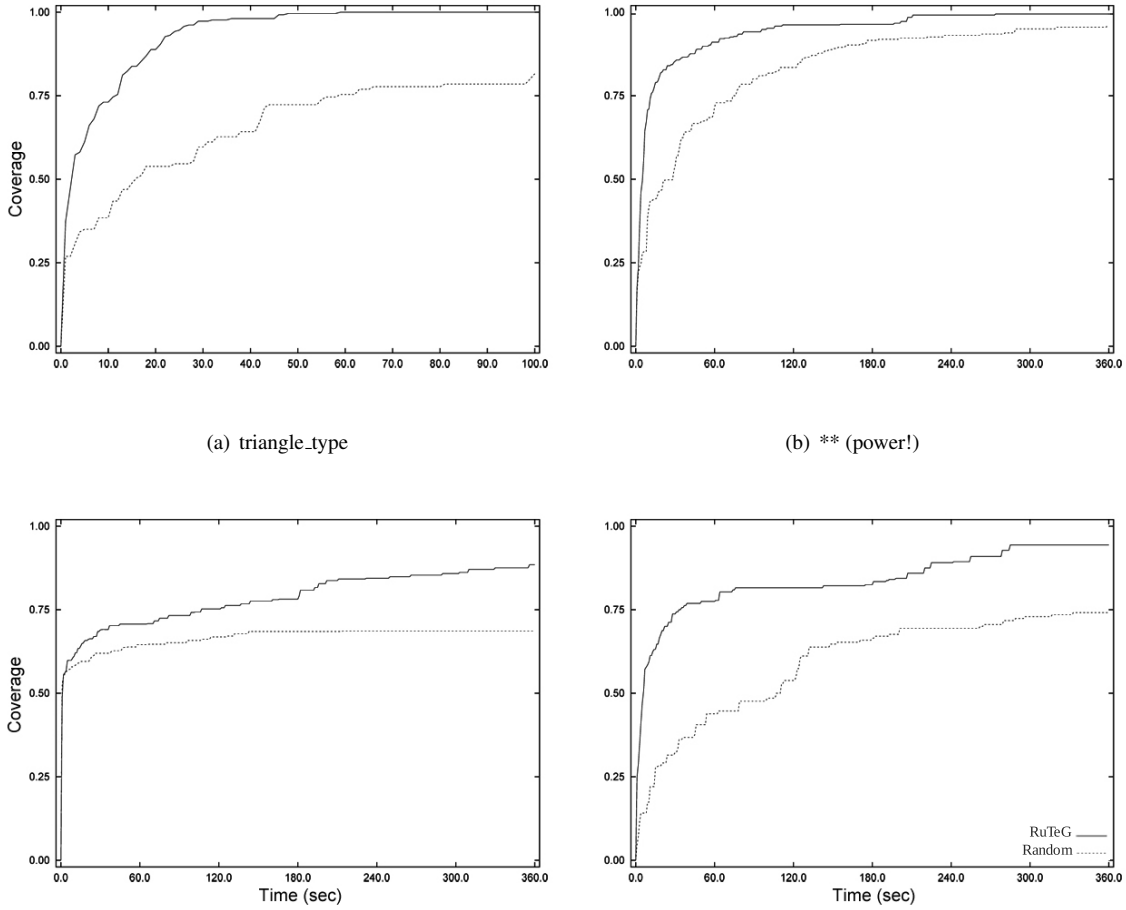
From Figure 4(c) it can be seen, that both data generators could cover much code within a short time. However, the random test case generator hardly found new test scenarios that could contribute to the coverage, whereas the line of RuTeG shows a slow increase, which results in a higher code coverage at the end compared to random testing.

7. Discussion

RuTeG was tested in an experiment in which the outcome was compared with the results of a random test case generator. The results show that the presented approach offers a possibility for a dynamic programming languages to automatically generate test cases and both simple and more complex test data. In most of the cases, RuTeG could cover more code and find solutions faster compared to the random test case generator, as can be seen in Figure 4. An 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.

The difference in the results between the two test case generators can probably be explained by RuTeG's capability to explore different possibilities of input data and then focus on a set of promising solutions. This may be the reason, why it was possible to find already at the beginning, a number of test cases to cover most of the code.

One reason for the difference in the final achieved



(a) triangle_type

(b) ** (power!)

(c) move

(d) canBlockACheck

Figure 4. Line charts of the experimental results

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

Methods	Cov. RuTeG	Cov. RT	t-test	Time RuTeG	Time RT
triangle_type	100%	81%	**	59	99
valid_isbn10?	100%	100%		29	84
valid_isbn13?	100%	100%		34	80
add_address	100%	100%		56	97
rb_insert	100%	88%	**	68	92
bootstrapping	100%	86%	*	54	88
gamma	98%	92%	**	209	213
bfs	100%	93%	*	79	86
dfs	100%	96%	*	70	72
warshall_floyd_shortest_paths	100%	100%		155	196
rank	100%	92%	*	111	202
** (power!)	100%	96%	**	274	356
canBlockACheck	94%	74%	**	285	333
move	88%	68%	**	356	143

coverage may be due to the evolution of individuals. New individuals are generated based on collected information of previous individuals. In Figure 4(c) there is a continuous slow increase of the coverage, while the random test case generator could hardly find new test scenarios that contributed to the coverage. In Figure 4(d) one can see that there is a phase in which the coverage achieved by RuTeG did not increase. Once an individual was able to cover new code, several new test scenarios were found that contributed to the coverage. Thus, these two results can probably be explained because of the evolution of individuals.

A possible 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 cases. However, the more complex the method sequence becomes, the more difficult it is to find possible test cases. This could be observed while applying RuTeG on the RubyTK library.

In this study we wanted to identify which characteristics are typical for a *dynamic programming language* and how they affect the automatic generation of test cases. We selected Ruby for the implementation of our tool. One characteristic of Ruby 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. Thus, methods can be defined in different modules and included within a class. RuTeG makes use of this reflective ability to search for available methods that may change the internal state of an object. RuTeG identifies also the kind of arguments, whether its specification is required or if they have a default value associated. Arguments can also have a variable length or require a code block. This information is collected and available at runtime.

Another characteristic, as described in Section 2, which many dynamic programming languages have in common, is duck typing. Objects are described by what they can or cannot do, instead of being associated with a specific type. This makes it difficult to identify the input data for method invocations, also because an argument can be used in different ways. Often methods behave differently, depending on the argument's current type. RuTeG presents a possible approach to classify such applicable type combinations and to disqualify inappropriate types.

Throughout the experiment it was possible to identify different kinds of *complexity*. One concerns the input type of data. Basic types are easier to generate than objects that consists of multiple values. In the latter case, a single value or a combination of values can be decisive to satisfy a given condition. Changing one value may modify the entire structure or meaning of an object. This was observable for the RubyGraph test candidate. If we consider for example a cyclic graph, then the removal of a single edge may result in a completely different graph and thus have an affect to the executed code.

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. Here, 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, such that a specific order is required. An example, in which the method sequence plays an important role, was the RubyChess test candidate.

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 in finding 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 applicability and efficiency of the tool was tested in the experiment on 14 *test candidates*. Some test candidates are code snippets that were often used in many testing papers. Other test candidates are taken from the Ruby Standard Library and open source projects, because we wanted to test the tool on more realistic and complex code examples.

The Ruby Standard Library consists of a number of classes, which were used to search for complex test candidates. However, methods with the highest cyclomatic complexity are parsers. A parser may have many control structures to respond differently for each keyword, but can not really be considered as a challenging test candidate. Other methods with a relatively high cyclomatic complexity have basic types as arguments. Hence, it was difficult to find test candidates that met our expectations. Therefore we extended our search to open source projects to find test candidates with different complexity and input data.

Apart from the test candidates mentioned in the experiment, we applied RuTeG on other classes and methods from the Ruby Standard Library. For some methods it was never possible to achieve full code coverage, even after repeating the tests several times. After analysing the reason why it failed to cover specific portions of the code, we could locate some errors. This was due to wrong computations and the usage of undefined variables, which results in exceptions and, in the end, uncovered code. This or similar cases show when and how the system can help to improve the quality of the code.

It is important to ensure the correctness of the empirical results and to avoid a misleading conclusion.*Statistical conclusion validity* is related to the reliability of the observed results. 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. On the other hand, RuTeG learns during the search process to distinguish between better and weaker data generators and applicable type combinations, and can therefore focus on more promising solutions.

Furthermore, it should also be mentioned that we applied only GA as a heuristic search algorithm to generate possible test cases. We do not know how other search techniques perform, such as hill climbing, simulated annealing or tabu search, to name but a few. Even if they cannot achieve a higher coverage, it may be possible that they find different solutions quicker.

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 generalizability.

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. Furthermore, RuTeG applies the ParseTree to collect some of the relevant information at runtime, which is a Ruby tool that presents the code in an abstract syntax tree using S-expressions. The collected information can probably be obtained also in other dynamic programming languages, but in a different way. 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. Therefore we cannot be sure whether the sample is representative of the Ruby code, but we can use the results as an indicator.

8. Conclusion

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 could find test scenarios that cover all the code only in 4 of 14 cases. The statistical significance of the difference in the results was tested by a Student's t-test. A difference was also observable in the time required to find possible test cases, where RuTeG could find solutions quicker than the random test case generator.

There are different kinds of complexity that have a major effect on the generation of tests cases. These are self-defined types, complex and compound data structures, input data with a small solution space, and method sequences to change the internal state of an object. The complexity of method sequences is a sensitive factor in the automatic generation of test cases. The more complex the method sequence becomes, the more difficult it is to find possible solutions. Very complex and dependable method sequences may not occur frequently, but in such situations, both test case generators will most likely fail to achieve high code coverage.

The goal of RuTeG is to find test scenarios in order to cover as much code as possible. However, code coverage is not a very strong coverage criterion. A possible future step would be aiming for branch or condition coverage.

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. Acknowledgments

I would like to thank my supervisor Dr. Robert Feldt for his support, guidance and constructive ideas throughout this work. This thesis would not have been possible without his valuable time. I am thankful for the numerous meetings and discussions that we had, which were a great inspiration and helped me to focus on the subject and most important issues.

References

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

[2] A. Bouchachia. An immune genetic algorithm for software test data generation. In *HIS '07: Proceedings of the 7th International Conference on Hybrid Intelligent Systems (HIS 2007)*, pages 84–89, Washington, DC, USA, 2007. IEEE Computer Society.

[3] E. Díaz, J. Tuya, and R. Blanco. Automated software testing using a metaheuristic technique based on tabu search. In *ASE*, pages 310–313, 2003.

[4] M. Harman and P. McMinn. A theoretical & empirical analysis of evolutionary testing and hill climbing for structural test data generation. In *ISSTA '07: Proceedings of the 2007 international symposium on Software testing and analysis*, pages 73–83, New York, NY, USA, 2007. ACM.

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

[6] B. Korel. Automated software test data generation. *IEEE Transaction on Software Engineering*, 16(8):870–879, 1990.

[7] P. McMinn. Search-based software test data generation: a survey. *Software Testing, Verification and Reliability*, 14(2):105–156, 2004.

[8] C. Michael and G. McGraw. Automated software test data generation for complex programs. In *ASE '98: Proceedings of the 13th IEEE international conference on Automated software engineering*, page 136, Washington, DC, USA, 1998. IEEE Computer Society.

[9] C. C. Michael, G. McGraw, and M. A. Schatz. Generating software test data by evolution. *IEEE Transactions on Software Engineering*, 27(12):1085–1110, 2001.

[10] R. P. Pargas, M. J. Harrold, and R. Peck. Test-data generation using genetic algorithms. *Software Testing, Verification & Reliability*, 9(4):263–282, 1999.

[11] S. Stinckwich and S. Ducasse. Introduction to the smalltalk special issue. *Computer Languages, Systems & Structures*, 32(2-3):85–86, 2006.

[12] D. Thomas, C. Fowler, and A. Hunt. *Programming Ruby. The Pragmatic Programmer's Guide*. Pragmatic Programmers, 2004.

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

[14] N. Tracey, J. Clark, K. Mander, and J. McDermid. An automated framework for structural test-data generation. In *ASE '98: Proceedings of the 13th IEEE international conference on Automated software engineering*, page 285, Washington, DC, USA, 1998. IEEE Computer Society.

[15] N. Tracey, J. Clark, J. McDermid, and K. Mander. A search-based automated test-data generation framework for safety-critical systems. In *Systems engineering for business process change: new directions*, pages 174–213, New York, NY, USA, 2002. Springer-Verlag New York, Inc.

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

[17] 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.

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

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

[20] J. Wegener, A. Baresel, and H. Sthamer. Evolutionary test environment for automatic structural testing. *Information and Software Technology Special Issue on Software Engineering using Metaheuristic Innovative Algorithms*, 43(14):841–854, 2001.

[21] A. Windisch, S. Wappler, and J. Wegener. Applying particle swarm optimization to software testing. In *GECCO '07: Proceedings of the 9th annual conference on Genetic and evolutionary computation*, pages 1121–1128, New York, NY, USA, 2007. ACM.

[22] 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.

Appendix A

Appendix A gives a detailed description of the main components of RuTeG, which were briefly introduced in the paper. These components are separated in Analyser (Section A.1), Data Generator (Section A.2), Test Case Executor (Section A.3) and Test Case Generator (Section A.4).

A.1 Analyser

The Analyser collects information from the input source code, which is relevant for the later search process of the system. In this section we describe how the Analyser performs its task, what information it can collect, and what is generated and delivered to the test case generator.

A.1.1 Ruby's AST and S-expressions

Most information can be obtained by using the ParseTree[1], which is a Ruby tool that presents the code of a class or a specific method in an abstract syntax tree using s-expressions. S-expression, also known as sexp, stands for symbolic expression, and is probably best known in the context of LISP[2]. S-expression is a notation for presenting tree structures in a linear text enclosed in parenthesis, containing either atom elements or further s-expressions [17]. In the context of the ParseTree tool, the first element of an s-expression, which is a symbol, defines the meaning of the entire expression. The following example presents the s-expression of a code snippet. For the sake of clarity, figure 5 shows the s-expression as a tree.

```
class TestClass

  def method1(a,b)
    sum = a + b
    puts sum
  end

  def method2(*c)
    l = c.length
    puts l
  end

end
```

s(:class, :TestClass, s(:const, :Object), s(:defn, :method1, s(:scope, s(:block, s(:args, :a, :b), s(:lasgn, :sum, s(:call, s(:lvar, :a), :+, s(:array, s(:lvar, :b)))), s(:fcall, :puts, s(:array, s(:lvar, :sum)))))), s(:defn, :method2, s(:scope, s(:block, s(:args, :"*c"), s(:lasgn, :l, s(:call, s(:lvar, :c), :length)), s(:fcall, :puts, s(:array, s(:lvar, :l)))))))

[1] https://rubyforge.org/projects/parsetree/ - Aman Gupta, Eric Hodel, Luis Lavena, Ryan Davis - Version 2.1.1

[2] http://www-formal.stanford.edu/jmc/recursive.pdf - John McCarthy about LISP

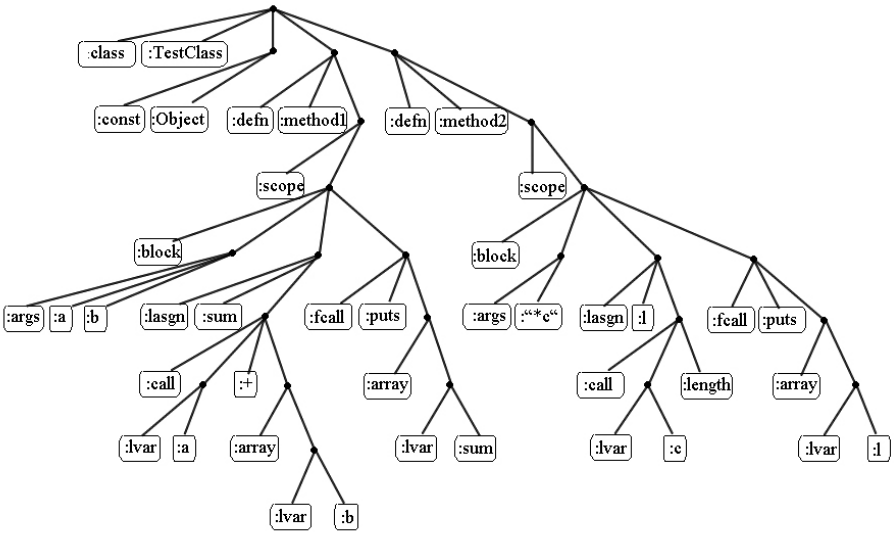


Figure 5: S-expression presented as a tree

ParseTree, in its current version, defines more than 100 different node elements, whereas only a few of them are important to the Analyser. These relevant nodes are described in the following subsections.

A.1.1.1 Methods

The definition of methods can be distinguished into instance methods and class methods. Instance methods are presented with a leading `:defn` symbol, followed with the name of the method and an s-expression. Class methods differ slightly from instance methods. They start with a leading `:defs` symbol, followed with an s-expression containing `:self`, the method name and finally another s-expression.

instance method	s(:defn, :method_name, s(...))
class method	s(:defs, s(:self), :method_name, s(...))

Table 3: S-expression for method definitions

A.1.1.2 Arguments

So far we have seen how methods are defined. Now we want to find the argument list assigned to each method. Arguments are presented in the same way for instance and class methods. The leading node element for arguments is `:args`. If a method is defined without arguments, then the s-expression is present but empty.

```
def method_name
```

For the method definition above, the s-expression for the argument list becomes

```
s(:args)
```

In Ruby there are four different possibilities to define arguments for a method. The first, and probably the most common, is a predefined number of arguments. This means that they are obligatory and must be specified with the method invocation, in order to avoid an ArgumentError exception.

```
def method_name(a)
```

The corresponding s-expression for the argument list looks as follow

```
s(:args, :a)
```

Aside from obligatory arguments, Ruby offers also the possibility to define default values for parameters.

```
def method_name(a, b=nil, c="default")
```

For this example it is possible to call the method with either one, two or three arguments. In case that an argument is not specified for the method invocation, it will be assigned to the default value. The s-expression for the argument list changes to

```
s(:args, :a, :b, :c, s(:block, s(:lasgn, :b, s(:nil)),
s(:lasgn, :c, s(:str, "default"))))
```

Here we are able to determine the argument variables, but from the initial part of the s-expression it is not possible to tell, which of them are obligatory and which contain a default value. Therefore it is necessary to examine also the attached s-expression. From this we can see, that both b and c have a value assigned and thus they are optional, which means they have a default value. The third possibility that Ruby offers, is to pass an argument list of variable length. This is done by placing an asterisk in front of the parameter name.

```
def method_name(a, *b)
```

In this case the first argument is assigned to the first parameter variable as usual, whereas the remaining arguments are collected and assigned to a new array. The s-expression for this example looks as follow

```
s(:args, :a, :"*b")
```

From this s-expression we can clearly distinguish, which one of these two parameters is the argument list of variable length. The forth and final possibility that Ruby offers, is an argument containing a code block. This is handled separately by the ParseTree and therefore not part of the `:args` expression. Code blocks have its own symbol node, namely `:block_arg`. Hence the definition of the following argument as a code block

```
def method_name(a, &code)
```

becomes

```
s(:block_arg, :code)
```

no argumetns	s(:args)
required arguments	s(:args, :parameter_name)
default arguments (=)	s(:args, :parameter_name, s(...))
argument list (*)	s(:args,:"*parameter_name")
code block	s(:block_arg, :parameter_name)

Table 4: S-expression for arguments

A.1.1.3 Method invocations

So far we know how to extract method definitions and their argument list. Further we can say what kind of argument is required, namely if its specification is obligatory, if it has a default value and thus optional, if it is a list of variable length, or if it is assigned to a code block. Since Ruby doesn't use any type specification for arguments, we have to look on the method invocations in order to be able to disqualify inadequate data types. Method invocations are represented with a leading `:call` node. There exist two different forms of `:call` s-expressions. The first one is defined as

```
s(:call, s(...), :method_call)
```

which is produced by method invocations without argument list. Therefore

```
a.length
```

becomes

```
s(:call, s(:lvar, :a), :length)
```

Method invocations can be nested as the following example shows.

```
a.length.to_s
```

The corresponding s-expression for this method invocation becomes

```
s(:call, s(:call, s(:lvar, :a), :length), :to_s)
```

However, in this situation we are interested only in the innermost s-expression, since this is the method call associated to the variable in question. The second and all succeeding method calls, refer to the return value of the preceding. The second form of the `:call` s-expression is defined as

```
s(:call, s(...), :method_call, s(...))
```

which is produced by method invocations with an argument list. Therefore

```
a + b
```

becomes

```
s(:call, s(:lvar, :a), :+, s(:array, s(:lvar, :b)))
```

Also in this situation it is possible that `:call` s-expressions are nested. In any case we are always interested in the innermost method call, that is directly related to the variable.

method call without argumetns	s(:call, s(...), :method_call)
method call with arguments	s(:call, s(...), :method_call, s(...))

Table 5: S-expression for method invocations

A.1.2 SexpProcessor

From the previous section we have seen a possibility to extract information form a given s-expression. However, such an s-expression can become quite complex, especially for longer codes. Fortunately there exists a SexpProcessor for Ruby, that can be used to write an own customized SexpProcessor, whereas there are two major rules that must be followed. First, everything that comes in must be processed. This means that every element must be observed by shifting it form the list. Also nested s-expressions must be examined by passing them as arguments to the process method. A further basic rule is, that the information that comes in should be the same that comes out. This means that the outcome of the SexpProcessor is again an s-expression that matches with the input.

By following these rules we can write our own SexpProcessor that gathers all the information that is relevant for the later search process. This can be done by creating a new class that inherits from the class SexpProcessor. Within this class we can define process methods that are called for s-expressions with a specific initial symbol. In order to do so, we must define a method with the name process_*symbol_name*, where *symbol_name* corresponds to one of the node elements. Therefore, for the Analyser we have to define following methods

process_defn(expression)
process_defs(expression)
process_args(expression)
process_block_args(expression)
process_call(expression)

For all other node elements within the s-expression, the default process method is executed. By defining customized methods, we can collect relevant information that is later delivered to the Analyser.

A.1.3 Result returned by the Analyser

The Analyser creates a CUTInfo object which contains information about the class under test. These are the constructor and its argument list. For each argument it is know, whether its specification is required or optional. Also the methods invoked for each argument are stored, to reduce later the set of possible input types. The CUTInfo object maintains in addition a list of methods that are declared within the class under test. These are associated with a MUTInfo object that contains information about the argument list of the method. Furthermore, it keeps track of the coverage reached during the search process, and about adequate or disqualified data generators. Figure 6 shows an example of an object delivered by the Analyser.

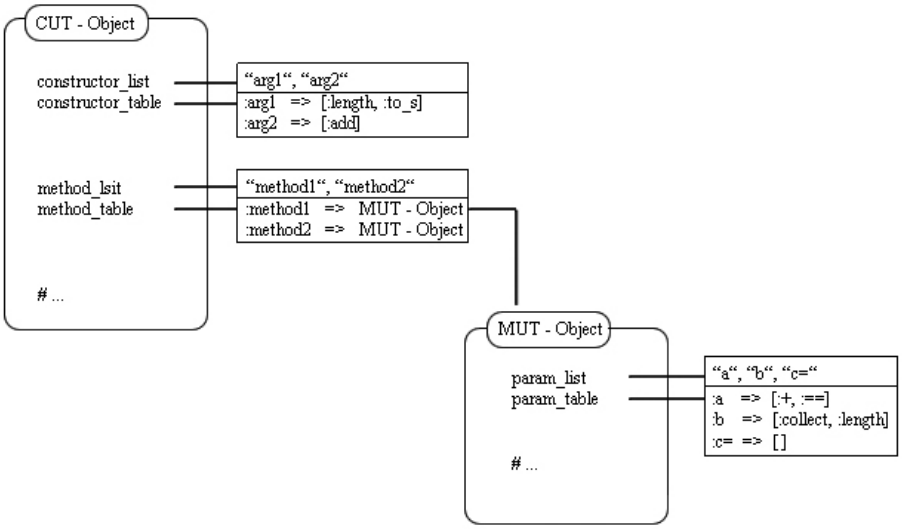


Figure 6: Example of the generated outcome delivered by the Analyser

A.2 Data Generator

Data Generators produce input values for method invocations. The set of possible input types is large, ranging from simple data types such a numbers, to more complex data such as objects and arrays. Also the domain of input values for a specific type can be quite large. Finding appropriate values, is a very important although difficult task. In this section we present a possible approach for the generation of input data.

A.2.1 Characteristics of a Data Generator

A Data Generator has to cover a large domain of types and input values. It is probably too much for a single generator to cover all these different possibilities. Therefore, a major design decision was to have different data generators for specific problems. The responsibility lies with each single generator to produce qualitative input values. Since multiple generators can be applied to the system, they have to share a common interface, such that the application can behave independently, regardless of which generator is currently used.

It is very unlikely to have a specific data generator that produces input values for all possible problems. Therefore, the set of data generators must be modifiable, by removing or adding user defined generators. How this can be achieved and what a user must know to create its own specific generator, is described in the following subsections.

A.2.2 Defining problem-specific Generators

When writing a new data generator, there a few things that a developer must consider. Every generator must include the module BaseGen which implies methods that are required by each data generator.

```
class NewDataGen
  include BaseGen
  # data generator specific code
end
```

Each generator must declare a method with the name *generate*. The return value of this method is a randomly generated input value. For simple generators, this may be a single value, such as it is the case for numbers or strings. But this is not enough for all possible types. A generator, that produces an object of a specific class, probably requires a prior initialisation. Therefore, the return value is a pair of an initialisation code string and the value. This design was chosen because of its simplicity in the later process. However, it is not an optimal solution, and complicates the usability. In case that no initialisation is required, then the value *nil* should be returned instead of an initialisation string. This is clarified through the following examples. The first example returns a random number between *min* and *max*, and therefore doesn't require any prior initialisation.

```
def generate
  random_number = rand((max - min).abs + 1) + min
  return nil, random_number
end
```

In the second example, an object is created and its attribute changed, before returning the value.

```
def generate
  variable_name = # a unique identifiable variable name
  random_value  = # a random generated value
  init  = "#{variable_name} = UserDefClass.new \n"
  init += "#{variable_name}.set_a_value = #{random_value} \n"
  return init, variable_name
end
```

The reason why the generator doesn't create an instance of a class and returns its object, instead of returning the code as a string, is that the code is needed in the later process to generate and deliver test case scenarios. If the generator returns an instance of a class, then it would be difficult to reproduce the same object for the final test case. More details about the generation of test case scenarios, is discussed in the section A.4 about the Test Case Generator.

The last specification required for each data generator is the type of data that it produces. This is done, so that the system is able to find all data generators of a certain type. The return type is specified by setting the instance variable @gen_type, as the following example shows.

```
class NewDataGen
  @gen_type = :UserDefClass
  # data generator specific code
end
```

Finally, the new data generator must be registered before it can be used. This is done by adding its symbolic name to the set of existing data generators, as the following example shows.

```
DataGenClass.add(:NewDataGen)
```

On the other hand, it is also possible to remove registered generators. This is done as following.

```
DataGenClass.remove(:NewDataGen)
```

Knowing theses few restrictions, a developer can write data generators that best fit to the context of the program under test. A further advantage of having multiple data generators for specific problems, is that they can be easily combined to produce more complex data. For example, a generator that produces an array, could use a positive fixnum generator to define the size, and then fill it with random data of a specific type with other existing generators. The array generator in turn can be used again to produce nested arrays.

A.2.3 Standard Data Generators

RuTeG contains already a number of data generators for standard types, which can be used to test methods with common input data. However, these data generators select random values from a large domain, and may not be efficient enough for specific problems. This subsection presents the set of standard data generators, and the values that are possible to be produced. Table 6 lists all predefined data generators that are already implemented in RuTeG.

There are two data generators that produce numerical input values. The first generator creates a random Fixnum, which is an integer value, between the default range of [-32768, 32767]. This range can be modified through defined methods, and therefore used in other data generators to produce for example only positive numbers or numbers within a small range. The second numerical data generator creates a random Float value. Also here is the possibility given, to change the range of the domain. The default value of the precision is set to three, which means that there are at most three digits after the decimal point.

The String generator produces a sequence of alphanumerical characters. The default value for the maximal length of the sequence is set to 20, however this value is modifiable. The set of possible characters consists of all lower and upper case alphabetic letters and numerical values, whereas it is possible to change the domain of characters.

Other standard data generators that produce basic input values, are the NilClassGen and ObjectGen. The former creates a nil value, which is also an object of class NilClass in Ruby. The latter generates a simple object of type Object. For both generators there no set from which a value could be randomly selected, and thus they very simple.

Other common input types are arrays. Therefore one of the standard data generators produces an array of random length, with random values. Both the

length and possible data are configurable. The maximal default length is set to 10. The array is filled with random values produced by other data generators. The default set of data generators implies all the above mentioned standard data generators. Thus, the array is filled with values of the same type produced with a random selected data generator. There is a small chance, that the array is filled with values of different type, where for each value a new data generator is randomly selected, however this may produce arrays that are unrealistic and never used in any context.

Data Generator	**Characteristics**
FixnumGen	values: [-32768, 32767]
FloatGen	values: [-32768, 32767] precision: 3
StringGen	values: [a-z, A-Z, 0-9] max length: 20
NilClassGen	nil
ObjectGen	Object.new
ArrayGen	values: [FixnumGen, FloatGen, StringGen, NilClassGen, ObjectGen] max length: 10

Table 6: Standard Data Generators

A.3 Test Case Executor

To obtain information about generated test cases, they are executed by the Test Case Executor. Rcov[3], a code coverage tool for Ruby, is used to measure the statement coverage achieved by the current test scenario. Therefore, the test case must be executed within a hook block, as the following example shows

```
rcov.run_hooked do
  # test case scenario
end
```

After running the test case within the hook block, the Rcov tool provides information about which line of code was executed from the current class under test. This information is obtained by calling the following line.

```
lines, marked_info, count_info = rcov.data(file_name)
```

Here, *lines* is an array of strings that represents the lines of the source code under test, *marked_info* is an array that holds true or false values to indicate

[3]http://eigenclass.org/hiki/rcov - Mauricio Julio Fernandez Pradier - Version 0.8.1.2

whether the corresponding line of code was executed or not, and *count_info* is an array of numbers that represents how many times a line was reported as executed. The following example shows how the information is presented by the Rcov tool.

count_info[]	*marked_info[]*	*lines[]*
1	true	`def print_10_times(msg)`
0	false	`  # this is a comment line`
2	true	`  1.upto(10) do |counter|`
10	true	`    puts "#{counter}: #{msg}"`
0	false	`  end`
1	true	`  puts "\'#{msg}\' was printed 10 times"`
0	false	`end`

Rcov returns the information for the whole source file, and therefore we have to extract the lines that match to a specific method, since we are interested in the coverage of the current MUT. Furthermore, lines that are comments are not executed, and therefore always marked as false. Fortunately Rcov provides an additional method *is_code?*, that can be used to determine whether a given line contains executable code or not. With this information we are able to calculate the coverage for the MUT.

The Test Case Executor keeps track of the coverage achieved by previous test cases, and is updated when uncovered lines are executed. From this we are able to distinguish between test cases that contribute to the coverage and test cases that execute already covered lines of code.

Test cases are divided into three major parts, namely 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 that 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.

A.4 Test Case Generator

The test case generator is the core of RuTeG and is responsible for producing possible test cases. A test case is a sequence of method calls that create an object, change its state, and finally invoke the method under test. There are two major tasks, namely to find an appropriate list of arguments and to form an adequate sequence of method invocations. Both are something that can not be done right away. Indeed, a Genetic Algorithm (GA) is used to search for possible test case scenarios. This section describes how the test case generator performs this task.

A.4.1 Searching for test cases

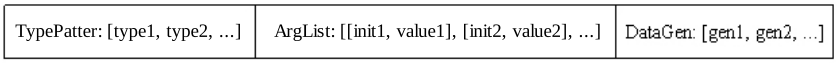
A GA is applied to search for possible test case scenarios. The idea of GA is to have a population (Section A.4.1.1) of candidate solutions that evolve during the search process. The algorithm starts with a random initialisation of the population. Each individual of the population is executed and evaluated (Section A.4.1.2) according to a predefined fitness function. A selection

operation (Section A.4.1.3) chooses the fittest individuals that are combined (Section A.4.1.4) to form a new generation of the population. Individuals can mutate (Section A.4.1.5) with a predefined probability, to prevent that they become too similar and thus that the population freezes. Afterwards, each individual is evaluated again and the process is repeated, until a termination criterion is fulfilled. This is either a maximum number of repetitions or the achievement of full code coverage. In the following subsections we describe how the population is set up and discuss in detail each phase of the algorithm.

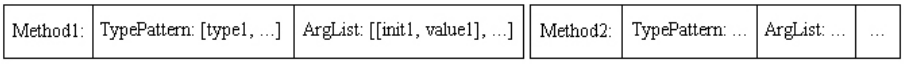
A.4.1.1 The population

The population is a set of individuals. An individual is an encoded representation of candidate solutions, also called genotype. This means that an individual itself can not be executed, but it contains all necessary information to produce a complete test case, which is called phenotype.

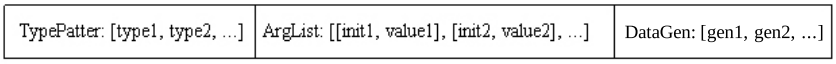
The encoded information of an individual is based on a simplified model presented by Feldt et. al. [10]. It can be divided into three categories; the constructor to create an object from the class under test, the method call sequence to modify the state of an object, and the invocation of the method under test. The constructor consists of the argument type pattern (Section A.4.2), 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, with the exception that it contains additional information about the method name.



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 the later combination and mutation phase. Figure 7 summarises the encoded representation of an individual.

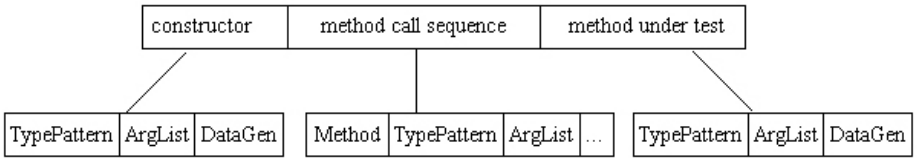


Figure 7: Encoded representation of an indivdual

A.4.1.2 Evaluation of individuals

The selection of an individual depends on the defined fitness function. It has a significant impact on the success rate of the search process and therefore it is important to choose a well constructed fitness function. The aim of the tool is to find test cases, that cover as much code as possible, hence the more code and control structures are executed, the better is an individual. The fitness function used in RuTeG is defined as

$$f_{fitness} = (cov \cdot p) + \left(\frac{executed\_cs}{total\_cs} \cdot (1 - p) \right)$$

where p is a value between 0 and 1, *cov* is the code coverage achieved by the test case, *executed_cs* the number of executed control structures, and *total_cs* 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 then a value close to 0.

A.4.1.3 Selection phase

During the selection phase, individuals are chosen from the current population. They are then combined to form new individuals for the next generation. Two common selection methods are the roulette wheel selection and the tournament selection.

The roulette wheel selection associates a probability to each individual, that is proportional to its fitness value. An individual is then randomly selected according to its probability. This means, that individuals with a higher fitness value are more likely to be selected than individuals with a low fitness value.

Another selection method is the tournament selection. It chooses with the same probability a number of individuals that will be part of the tournament. The winner, namely the fittest individual, is then selected for the combination. The number of participants in a tournament is not restricted. The higher the number of participants, the lower is the chance for weak individuals to be selected. The tournament selection is the default method for RuTeG and used in the later experiment.

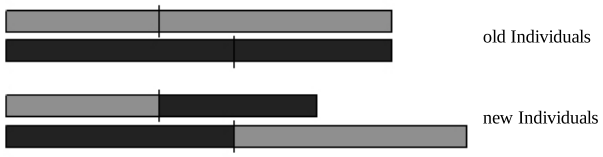
A.4.1.4 Combination phase

During the combination phase, two selected individuals exchange their information. This results in having two new individuals which will be part of the new

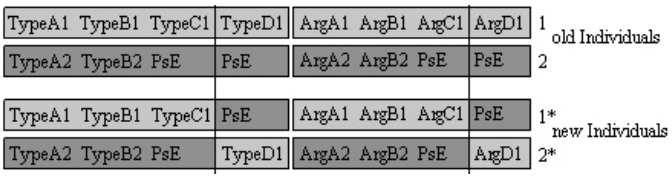
generation. There are different possibilities to combine individuals. Two common methods are the one-point crossover combination and the cut and splice combination.

For the one-point crossover combination, the information of the individuals is exchanged at a random selected position. This is clarified through the following example.

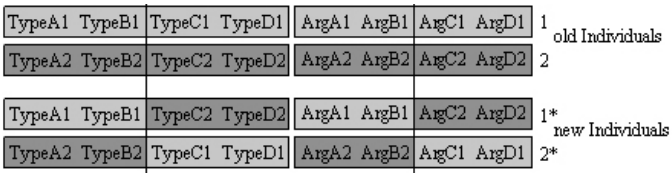
For the cut and splice combination, the position where the information is exchanged, is independent between the two individuals. This results in two individuals of different length, as the following example shows.



In this context, the combination of two individuals can be applied on different parts. For the constructor and the method under test, the combination between individuals concerns the argument list. For this, the one-point crossover combination is applied. This is shown in the following example.



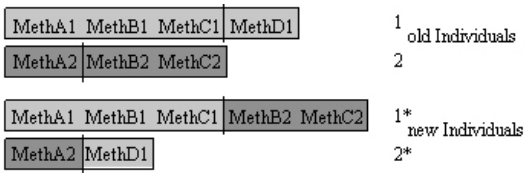
If the length of the argument list doesn't match between the two individuals, then the missing elements are substituted with temporary pseudo elements (PsE). This may be the case for default parameters or parameters with variable length. The following example shows such a situation.



While the first of the two new individuals (1*) is still a valid method call, the second individual (2*) is not feasible, because of its pseudo element between the arguments. In this case, all the arguments after the pseudo element are cut off, and thus discarded.

If individuals have two completely different type patterns, then the result may be a new combination of types. While the former patterns were applicable, which means that they didn't lead to an exception, the resulting patterns may be inapplicable. As we will see soon in the following subsections, we are able to distinguish between applicable and inapplicable patterns. In case that the new individuals result in an inapplicable pattern, then the combination is repeated, until applicable type patterns are found, or a maximum number of tries exceeded.

For the combination of the method call sequence, the cut and splice method is applied. This means that the sequence of two individual is cut at two different randomly selected positions, and exchanged. This is shown in the following example.



A.4.1.5 Mutation phase

During the mutation phase, individuals of the new population are randomly selected, and mutated with a predefined probability. The mutation of an individual can affect different parts, such as the constructor, the method sequence, and the method under test.

For the constructor and the method under test, there are two possibilities of mutation, namely to generate a new type pattern, or to 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 wouldn't be tested. A separate table is maintained to keep track about already applied type combinations. If no type pattern can be found that hasn't already been tested, then the mutation concerns only the argument value. For the generation of a new input value, a data generator of the corresponding type is selected. The selection is biased towards the evaluation of the data generators (Section A.4.3). This means that the probability that a specific generator is selected to produce an input value, depends on its evaluation. We will see soon more about the selection and evaluation of data generators, in the following subsections.

For the mutation of the method call sequence, a position is randomly selected, at which a method is either added or removed form the current sequence. In case of the addition, a method from the class under test is randomly chosen and added to the sequence.

A.4.2 Input type pattern

Ruby features duck-typing, and therefore it is not possible to determine which parameter type can be applied, by just having the method signature. On the other hand, trying each possible type and getting overwhelmed with exceptions may not be very efficient. Fortunately, the CUTInfo and MUTInfo objects, returned by the Analyser, contain information about which methods are called for each argument. From this it is possible to construct a list of types associated with a value that expresses the likelihood of being a good candidate. This value is calculated as following

$$f_{type} = \begin{cases} 1 - \left(\frac{|(M_{arg} - M_{type})|}{|M_{arg}|} \right) & \text{if } |M_{arg}| > 0 \\ 1 & \text{otherwise} \end{cases}$$

where M_{arg} is the set of methods invoked for the parameter in question, and M_{type} the set of methods that a type or class can respond to. This means that if a class has defined all required methods, then the value is equal 1. The less the number of common methods, the lower is the value assigned to the type, where a value equal to 0 expresses that a class has none of the required method calls defined. This list is used as a ranking of possible candidates from which a type is selected with a probability that is proportional to the assigned value. This however doesn't guarantee that no further exceptions are raised. A possible scenario shows the following example.

```
def add(a,b)
  a+b
end
```

In this case, possible input type patterns are (Fixnum, Fixnum) or (String, String), but any other combination such as (Fixnum, String) or (String, Fixnum) results in an exception. Therefore, the system learns during the search process about inappropriate type patterns, to reduce the number of raised exceptions. This is done by maintaining different categories for type combinations, namely applicable, suspicious, and critical patterns. If a certain type pattern is applied to a method call, and its execution doesn't result in an exception, then this pattern is added to the set of applicable patterns. On the other hand, if an exception of type NoMethodError or TypeError is raised, then the type pattern is added to the set of suspicious patterns. Suspicious patterns have a low chance to be applied again, in order to prove their applicability. If they continue to fail, then they are moved to the set of critical patterns. Critical type patterns are avoided by the system.

A.4.3 Selection of data generators

Finding applicable type patterns for method invocations is only the initial step in search of adequate test input data. The second step is the selection of a data generator, since there are multiple generators that can produce the same type of data. At the beginning, each generator of a specific type, has the same probability to be selected. For each generated input value, its data generator is memorised and evaluated depending on the resulting fitness value of the test case. This means, that the set of assigned values increases with the number of utilisation. The actual evaluation of a generator is then calculated by the arithmetic mean of all assigned values. Generators which were able to produce input data that lead to better test cases, will get a higher value than generators for inadequate data. The probability for a data generator, to be selected again, depends on its evaluation.

Appendix B

Appendix B describes in detail the steps of the experiment done with RuTeG and presents the results. Section B.1 introduces the experiment. The test candidates are presented in Section B.2. The results are presented and discussed in Section B.3.

B.1 Experiment

In this experiment we test the developed tool on a number of different test candidates. We want to see whether the presented approach can be used to find possible test case scenarios. The experiment should show, if the tool is applicable and in which situations it is difficult to achieve full coverage. The test candidates are code examples that differ in their complexity and required input data. Each test candidate is tested multiple times, to obtain a good estimated result and to make sure that the data is consistent.

The outcome of this experiment is compared with the results obtained by a random test case generator. This is done to see whether search based software testing, and particularly the developed tool, shows improvements compared to random testing. The random test case generator produces test scenarios, without applying any heuristic search technique that modifies test cases towards better solutions. Arguments for the constructor and method invocations are created by selecting randomly one of the existing data generators. The sequence of method invocations is randomly chosen, to change the state of an object. Every test case is generated independent form previous test cases and thus test scenarios are not evolved.

Both test case generators start with the same initial condition. The Analyser is used to provide information about the class under test and its methods, such as the possible number of arguments for the constructor and method calls. After this, the generators start to deviate from each other. While RuTeG maintains a population of test individuals and learns about applicable type combinations, the quality of data generators and relevant method sequences, the random test generator, selects any type combination, data generator, and method sequence randomly.

Each test candidate is tested multiple times. A test run terminates when full code coverage is achieved, or when a predefined time is exceeded. This time varies between different test candidates, since they differ in their complexity of input data and code structure. However, the time constraint is the same regardless of the used test case generator.

B.2 Test Candidates

Test candidates are selected to cover different input data and complexity. They vary from classical code snippets, to more complex methods taken from the Ruby standard library and open source projects.

The test candidates are briefly presented in the following subsections. Table 7 shows the test candidates, their number of line of source code, and the cyclomatic complexity.

B.2.1 Triangle

The Triangle example is a short code snippet used in many testing papers that takes three numerical input values and determines, whether they can form a valid triangle or not. In case of a valid triangle, the method determines its type, whether it is a scalene, equilateral, or isosceles triangle.

B.2.2 ISBN Checker

The ISBN checker is a small tool that takes a string as input, and checks whether it is a valid ISBN10 or ISBN13 code. An ISBN[1] 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'. The ISBN code consists of five parts, namely the prefix element, the registration group element, the registrant element, the publication element, and the check digit. These parts can be separated with hyphens or spaces. Thus all the following ISBN codes are valid.

978-0-571-08989-5
978 0 571 08989 5
9780571089895

B.2.3 AddressBook

AdressBook is a simple application to organize contact information, such as addresses, e-mail addresses and phone numbers.

B.2.4 RBTree

RBTree is a sorted associative collection using Red-Black Tree as the internal data structure. It is a kind of a binary search tree that maintains additional information about colors, which is used to automatically balance itself whenever a node is inserted or deleted. The maximum height of the tree is 2 log(n + 1) which results in a search time of O(log n), where n is the number of nodes.

B.2.5 Bootstrap

Bootstrapping is one of many statistical resampling methods, and is often used when the theoretical distribution of the population is unknown. It can be applied to estimate properties of an approximated distribution, such as the arithmetic mean, standard deviation, or the confidence interval.

B.2.6 RubyStats

RubyStats is a statistics library which supports different kinds of distributions, such as binomial, beta and normal distributions, and provides some basic statistical functions.

B.2.7 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.

BFS is an algorithm to traverse a graph in breadth first order and to search for a given element. Similar is DFS, only that the graph is traversed in depth first order. The Floyd-Warshall algorithm can be used to find the shortest path within a weighted graph.

[1] http://www.isbn-international.org/en/manual.html

B.2.8 Ruby1.8

Ruby comes with a large standard library, providing a number of different classes and functions, especially for basic types. One such library is 'mathn', which provides mathematical functions for different numerical values, such as integers, real, rational and complex numbers.

Another Ruby standard library is 'matrx', which provides a class for representing a mathematical matrix, and contains methods for generating special-case matrices, such as zero, identity, diagonal, and singular matrices. It can be implemented to perform different arithmetic and algebraic operations, and to determine their mathematical properties, such as their traces, ranks, and determinates.

B.2.9 RubyChess

RubyChess is a stand-alone chess engine that comes with a graphical Ruby/Tk user interface, to play chess against the computer. It is a direct port from pythonchess 0.6.

Test Candidate	**Methods**	**SLOC**	**CC**
Triangle			
	triangle_type	26	8
ISBN Checker			
	valid_isbn10?	18	7
	valid_isbn13?	13	6
AddressBook			
	add_address	10	3
RBTree			
	rb_insert	49	17
Bootstrap			
	bootstrapping	38	9
RubyStat			
	gamma	116	16
RubyGraph			
	bfs	39	12
	dfs	34	10
	warshall_floyd_shortest_paths	26	11
Ruby1.8			
	rank	56	13
	** (power!)	59	16
RubyChess			
	canBlockACheck	23	10
	move	111	26

Table 7: Test candidates for the experiment; Test Candidate, Methods, Source lines of code (SLOC), Cyclomatic Complexity (CC)

B.3 Results and Discussion

B.3.1 Triangle

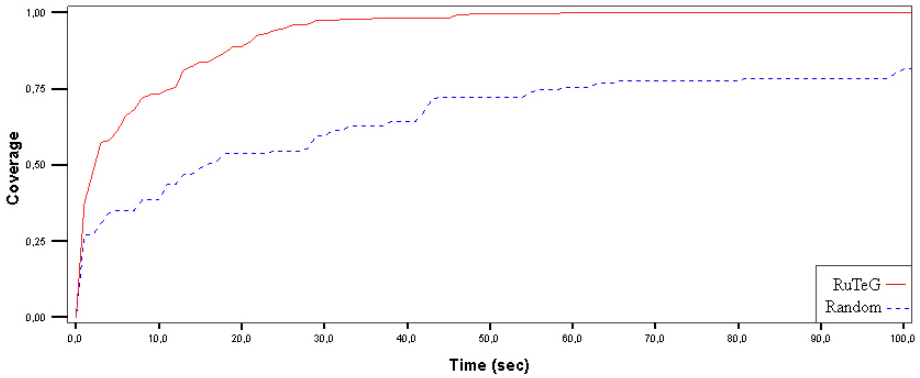


Figure 8: Line chart of the results for the Triangle test candidate - triangle_type

Figure 8 shows the results for the Triangle test candidate, where the average coverage is presented on the y-axis, and the time expressed in seconds on the x-axis. The solid line represents the experimental data obtained with RuTeG, while the dashed line is obtained with random testing.

From the line chart can be seen that after a short time RuTeG was able, to find test cases that achieved a code coverage of 50%. After this, the curve begins to flatten slightly until the upper limit is reached. In all test runs, RuTeG was able to achieve full coverage within the measured time. In case of the random test case generator, it was possible to cover relatively quickly the first 50%, whereas a difference can be observed already here between the two test case generators. Random testing required more time to find test scenarios in order to cover the first 50%. From this point, the line rises slowly until the average coverage of 81% is reached. Thus, the experimental data differ also in the final average coverage.

The reason of these results can be explained in the following way. RuTeG was able to eliminate individuals without numerical input values, since they were poorly evaluated. This leads already after few generations to a population with almost only numerical individuals. Through the combination of fitting individuals, and the generation of only numerical input values, it was possible to overcome the 80% mark. To achieve full coverage, a specific condition must be satisfied, namely three equal numerical input values such that an isosceles triangle can be formed. This is a fairly strict condition and difficult to generate with random testing. One reason for the difference in the required time between the two test case generators, may be because of the combination of three numerical input data. The random test case generator doesn't only have the difficulty to fulfill the strict condition, but also to produce a correct combination of input data.

B.3.2 ISBN Checker

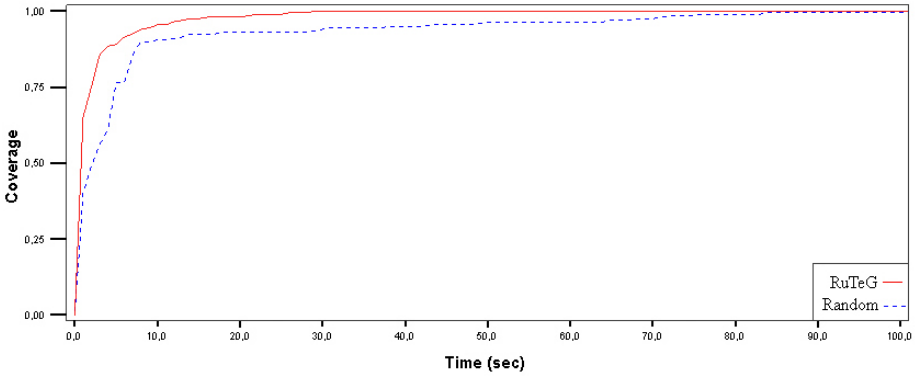


Figure 9: Line chart of the results for the ISBN Checker test candidate - valid_isbn10?

Beside of the standard String Data Generator, an additional ISBN Generator was used. The reason for this is that a random sequence of any length and any alphanumerical character, will most likely never succeed in producing a valid ISBN code. Therefore, the ISBN Generator creates values according to a predefined pattern. Thus, each character sequence has a length of either 10 or 13. A character can be any numerical value, while the last character, the check sum, is a number between 0 and 9 or 'X'.

Figure 9 shows the results of the experiment with the *valid_isbn10?* method from the ISBN Checker. From this line chart can be seen that the results are quite similar and that both test case generator were able to find valid input data, such that full code coverage is achieved. The observed results differ in the time required to find a valid ISBN code.

The method under test doesn't imply any strict conditions, nor requires any specific sequence of method invocations. That's probably the reason why both test case generators were able to succeed. In case of RuTeG, it became soon clear that the method under test requires a String as input data. Thus, the next step was to choose between the standard String generator and the ISBN generator. Also in this case, test scenarios got a better feedback using the ISBN generator than using the standard String generator, and therefore it was possible to focus on a specific set of the solution domain. On the other hand, the random test case generator is not able to reduce the domain of possibilities, which explains the difference in the time required to find valid input data.

Similar is the situation for the *valid_isbn13?* method, which has the same structure and conditions as the *valid_isbn10?* method, and differs only in the checksum calculation.

B.3.3 AddressBook

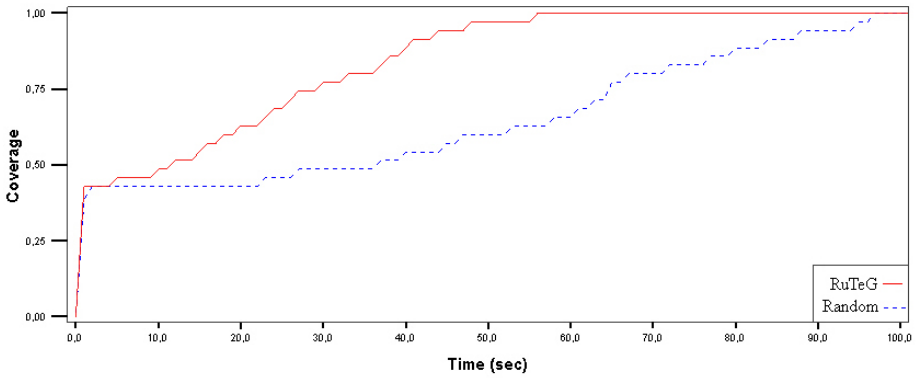


Figure 10: Line chart of the results for the AddressBook test candidate - add_address

Figure 10 shows the results collected for the *add_address* method from the AddressBook test candidate. Both test case generators were able to find test cases in order to achieve full coverage. The outcome differs in the time. RuTeG was quicker in finding a possible solution than the random test case generator.

Both test case generators were able to find solutions to achieve full statement coverage. This is generally because of the simplicity of the method under test. The main objective was to see if it is possible to generate data for a specific type without a predefined data generator. The method under test requires an object of type Person and a String. Thus, for user defined classes, which are part of the project and as such a possible input type candidate, but don't have an explicit defined data generator, an object is created by using the same method as for the class under test. This means that the Anaylser collects information about the user defined class, which is used to generate a constructor call. On the other hand, if a data generator for the user defined class is present, then only the generator will be used instead of the general method, to produce a possible input object. However, an explicit definition of a data generator for a used defined class is certainly a better choice, since the default creation of an object is quite generally and may not be very efficient.

B.3.4 RBTree

Figure 11 presents the results for the Red Black Tree test candidate, where *rb_insert* is the method under test. From this line chart it is possible see that both test case generators have similar results at the beginning. As the time progresses, the difference becomes larger. A coverage of 50% was achieved already after the first few test cases. After that, the curves begin to flatten, whereas the line of RuTeG is steeper than the line of the random test case generator. RuTeG was able to find test scenarios to achieve full coverage, while random testing achieved an average coverage of 88%.

The values inserted into the tree can be of any type, as long as they are comparable. This means, that possible types are for example strings and nu-

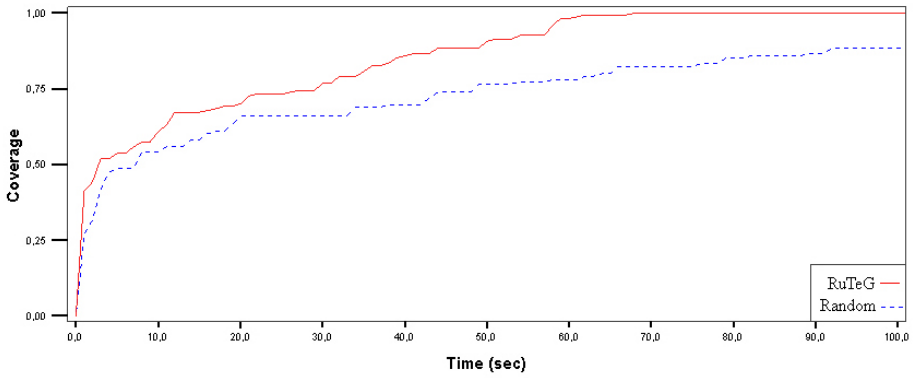


Figure 11: Line chart of the results for the RBTree test candidate - rb_insert

meric values. However, to cover the complete method under test, a sequence of insertions is required, by using types that are comparable among themselves. As long as there is only one value, there will never be a rotation within the tree. Therefore, the advantage of RuTeG compared to the random test case generator, is that individuals with an adequate method sequence will most likely remain within the population. The sequence can be extended through the combination of other individuals or through the mutation phase, which hopefully leads then to the solution. On the other hand, the random test case generator starts every time from scratch.

B.3.5 Bootstrap

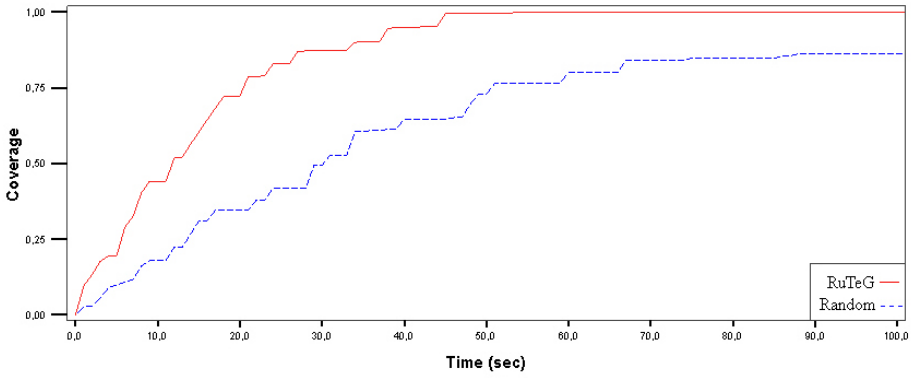


Figure 12: Line chart of the results for the Bootstrap test candidate - bootstrap

Figure 12 shows the results for the Bootstrap test candidate, where *bootstrapping* is the method under test. A difference between the two test case generators can already be observed at the beginning. RuTeG has a steeper curve compared to random testing, which means that it was able to find quicker solutions. Also in the final average coverage there is a difference observable. RuTeG could find test scenarios to achieve full coverage, whereas the random

test case generator achieved an average coverage of 86%.

The method under test doesn't require any previous initialization to achieve full coverage, thus the method sequence of the test scenario is irrelevant. However, the method requires a number of parameters. The first argument is an array of sample data, followed by a number of numerical values that affect the behaviour of the method. The difference between the two results can be explained of RuTeG's capability to reduce the domain of possible input type combinations, while the other test case generator chooses any combination randomly. Hence, RuTeG can focus on more problem specific data which leads to a quicker solution.

B.3.6 RubyStat

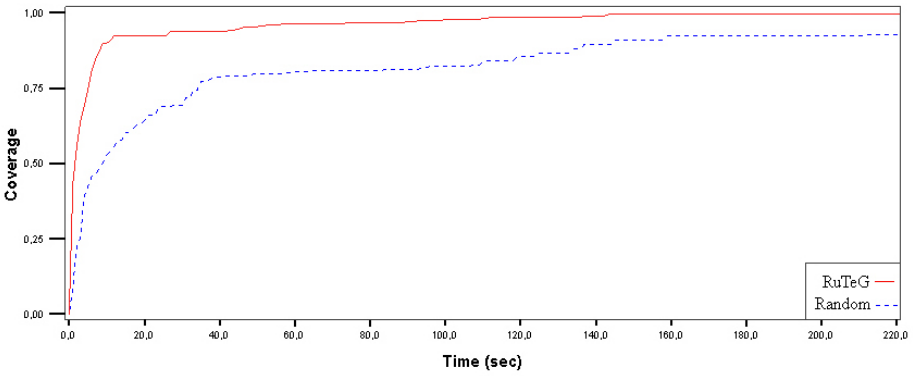


Figure 13: Line chart of the results for the RubyStat test candidate - gamma

Figure 13 presents the result of the experiment with the *gamma* method from the RubyStat test candidate. RuTeG and the random test case generator were able to cover much code within a short time. However, finding test cases to cover the remaining code required some more time. The average code coverage achieved by RuTeg was 98%, while the random test case generator achieved an average coverage of 92%.

One parameter of the method call is used in an equation that results in a small ϵ value. This ϵ is then tested in a nested condition. While RuTeG couldn't cover this part in few exceptional cases, the random test case generator failed almost ever. The reason for that is probably that RuTeG was able to maintain individuals that satisfied the precondition and thus after several combinations and mutations to find a value, such that the condition for ϵ was fulfilled. The random test case generator however, lost track of such potential input values and therefore it was harder to cover the specific code portion.

B.3.7 RubyGraph

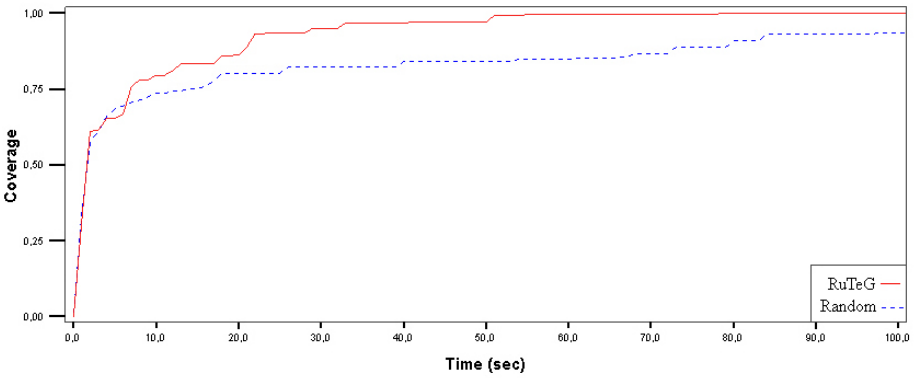


Figure 14: Line chart of the results for the RubyGraph test candidate - bfs

An additional data generator was defined for the graph library, to construct a directed graph. The generator selects two vertices from a small given domain and forms with these an edge. This is repeated multiple times, to obtain a set of random size with different edges. Thus, the result of the generator is a random graph.

Figure 14 shows the results of the experiment, where *bfs* is the method under test. Both test case generator show a similar curve and differ only at the end. With RuTeG it was possible to cover all lines of code, while the random test case generator achieved an average code coverage of 93%.

The method traverses a graph in breadth first order. This is also possible if the graph is not connected. In that situation, only vertices that are connected with the starting point are traversed. Thus, there is no specific condition required in order to cover a certain portion of the code. The only critical condition to achieve full coverage, is to find a given vertex within the graph. This is probably the main reason why both test case generator achieved a high code coverage.

Similar is the situation for the *dfs* method, which has the same structure and conditions as the *bfs* method, only that the graph is traversed in depth first order instead of breadth first order.

Figure 15 shows the empirical result with the *warshall_floyd_shortest_paths* as method under test. Both test case generators show similar results and were able to achieve full code coverage. There is no major difference observable between the two generators.

The method under test applies the warshall-floyd algorithm to find the shortest path between two vertices. Like in the previous case of the RubyGraph test candidate, there is no specific method sequence required or strict condition present.

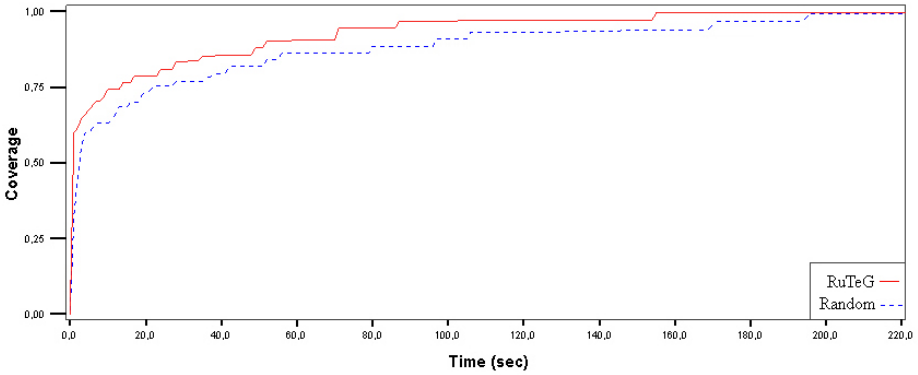


Figure 15: Line chart of the results for the RubyGraph test candidate - warshall_floyd_shortest_paths

B.3.8 Ruby1.8

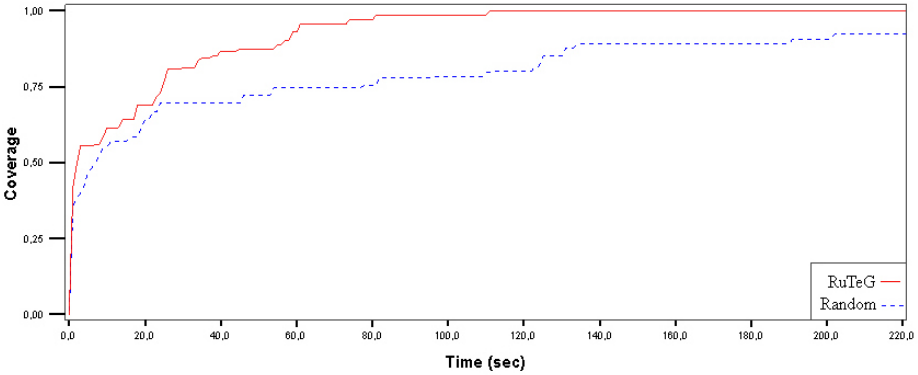


Figure 16: Line chart of the results for the Matrix test candidate - rank

Figure 16 shows the result obtained using the *rank* method from the matrix standard library. It can be observed that the results at the beginning are similar for both test case generators. After a code coverage of 70%, the two curves start to deviate, where the dashed line from random testing becomes flatter than the curve of RuTeG. The final average coverage achieved by the random test case generator is 92%.

The *rank* method doesn't require any parameters, but a proper initialization through the constructor. A matrix is represented as an array of numerical arrays. The difference between the two test case generators may be, because in case of RuTeG it was possible to disqualify individuals that were not correctly initialized, while many test scenarios produced by the random test case generator led to exceptions.

Figure 17 shows the results from the exponential operation for rational numbers, which is defined in the mathn standard library. A difference between the results can be observed already at the beginning. RuTeG was able to cover

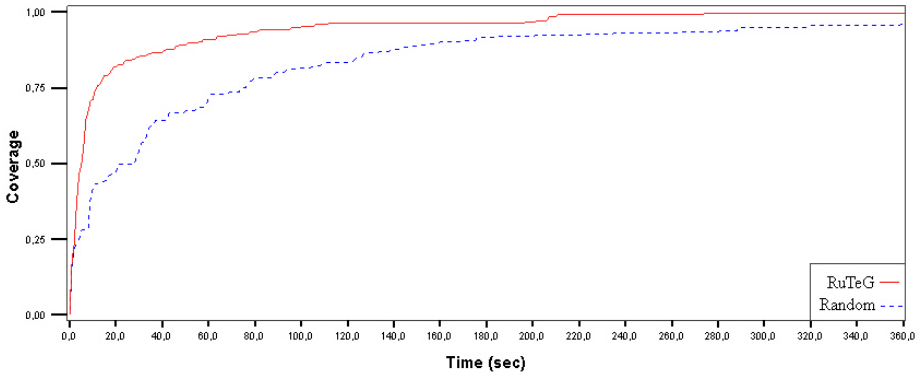


Figure 17: Line chart of the results for the Mathn test candidate - ** (power!)

more code within a shorter time than random testing. In both situations, the curves become flatter at the end. While RuTeG could achieve full code coverage, the random test case generator has an average coverage of 96%.

The exponent, which is the parameter of the method under test, can be an integer, real or rational number, which is handled differently by the algorithm. RuTeG tries to examine many different type combinations at the beginning, and then focus towards better solutions. This may be one reason because of the high coverage at beginning. As the time progresses, the random test case generator catches up with RuTeG, but has still a lower average coverage at the end.

B.3.9 RubyChess

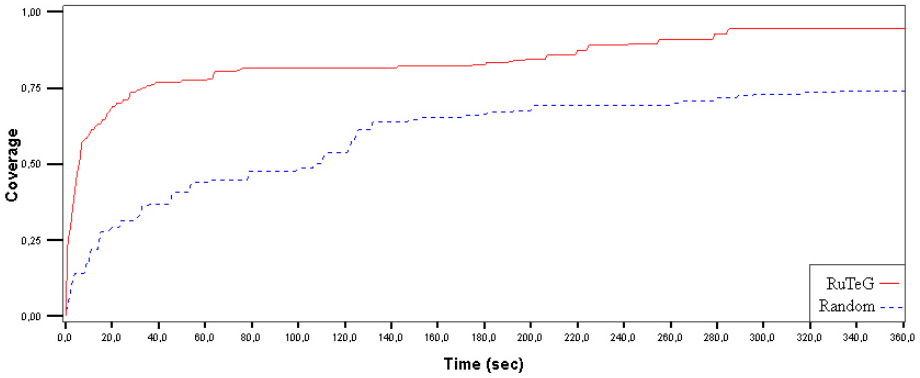


Figure 18: Line chart of the results for the RubyChess test candidate - canBlockACheck

Figure 18 shows the empirical result from the RubyChess test candidate, where *canBlockACheck* is the method under test. RuTeG, compared to the random test case generator, was able to find quicker solutions that covered more code. After a code coverage of 75%, the curve increases only slowly. Both test case generators failed to achieve full coverage. However, the average code

coverage achieved by RuTeG is higher than the coverage achieved by random testing.

The method under test required a previous initialisation of the chessboard and an attackmap. Thus an important factor for the code coverage, is an adequate sequence of method calls. The argument passed to the method under test is used to specify a certain position on the board. The reason why RuTeG was able to cover more code, was probably because individuals that were able to set up a possible chess situation, remained in the population. While the method sequence was kept or slightly modified, the data passed to these method invocations changed with the time, and thus it was possible to generate different scenarios. However, both generators failed to cover code, that was deeply nested with a combination of multiple and strict conditions.

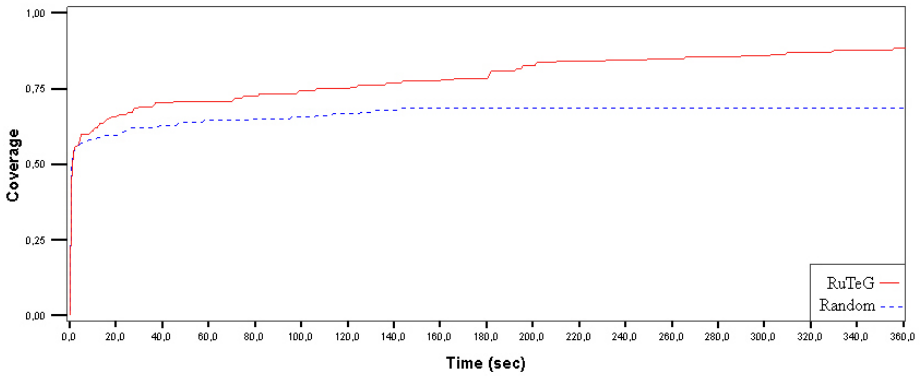


Figure 19: Line chart of the results for the RubyChess test candidate - move

Figure 19 presents the results from the RubyChess test candidate, where *move* is the method under test. From this line chart can be seen, that both data generators were able to cover already at the beginning more than 50% of the code. However, the random test case generator hardly found new test scenarios such that the code coverage increased, while on the other hand, the curve of RuTeG increases slowly with the time. Also in this case, both test case generators failed to achieve full code coverage.

The method under test requires a specific sequence of method invocations to bring the object to a certain state. Thus, the difference between the two results may be explained similar as in the previous case. The behaviour of the method under test was mainly depended on the internal state of the object, rather than the input value for the argument.

Appendix C

Appendix C concludes the thesis with the discussion (Section C.1) of the key findings with respect to the central research question, highlights the strengths and limitation of the study, and presents some possible direction of future work (Section C.2).

C.1 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*** in which the outcome was compared with the results of a random test case generator. 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 be observed while applying RuTeG on the RubyTK 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 evolve data generators automatically such that better input values are produced.

RuTeG is able to test methods with arguments that must be specified, arguments that have a default value, and arguments of variable length. An unsolved problem is still the generation of code blocks, which is a portion of context related code. This code depends strongly on the implementation, and is difficult to generate automatically. The current version of RuTeG generates an empty code block if required. This however doesn't guarantee the generation of successful test scenarios.

If we compare the implementation of RuTeG and the random test case generator, then it is definitely the latter which stands out in its simplicity, because there is no heuristic attempt to search for better solutions. The core of RuTeG is the Test Case Generator, which implies the GA. Together with the algorithm specific functionalities, such as the selection, combination and mutation, but also the definition of the individual and the fitness function, the size comprises 780 SLOC. Not included are the Analyser, Test Case Executor, and the different data generators, which remain the same for both test case generators.

The applicability and efficiency of the tool was tested in the experiment on 14 ***test candidates***. Some test candidates are code snippets that were often used in many testing papers [25, 18, 12]. Other test candidates are taken from the Ruby standard library and open source projects, because we wanted to test the tool on more realistic and complex code examples.

The Ruby Standard library consists of a number of classes, which were used to search for complex test candidates. However, methods with the highest

cyclomatic complexity are parsers. A parser may have many control structures to respond differently for each keyword, but can not really be considered as a challenging test candidate. Other methods with a relatively high cyclomatic complexity have basic types as arguments. Hence, it was difficult to find test candidates that met our expectations. Therefore we extended our search to open source projects to find test candidates with different complexity and input data.

Apart from the test candidates mentioned in the experiment, we applied RuTeG on other classes and methods from the Ruby standard library. For some methods it was never possible to achieve full code coverage, even after repeating the tests several times. After analysing the reason why it failed to cover specific portions of the code, we could locate some errors. This was due to wrong computations and the usage of undefined variables, which results in exceptions and thus to uncovered code. This or similar cases show when and how the application can help to improve the quality of the code. An open problem is still the so called test oracle problem. This means that it is not possible to test the result of a test case according to the formal specification.

In this study we wanted to identify which characteristics are typical for a ***dynamic programming language*** and how they affect the automatic generation of test cases. We selected Ruby for the implementation of our tool. 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 doesn't matter, where parts of a class are defined, as long they are available when the object is created. Thus, methods can be defined in different modules and included within a class. All these methods are available during runtime. RuTeG makes use of this ability to search for available methods that may change the internal state of an object. RuTeG identifies also the kind of arguments, whether its specification is required or if they have a default value associated. Arguments can also have a variable length or require a code block. This information is collected and available at runtime.

Another characteristic, that many dynamic programming languages have in common, is "duck typing". Objects are described by what they can or can not do, instead of being associated to a specific type. This makes it difficult to identify the input data for method invocations, also because an argument can be used in different ways. Often methods behave differently, depending on the argument's current type. RuTeG presents a possible approach to classify such applicable type combinations and to disqualify inappropriate types.

Throughout the experiment it was possible to identify different kinds of ***complexity***. One concerns the input type of data. Basic types are easier to generate than objects that consists of multiple values. In the latter case, a single value or a combination of values can be decisive to satisfy a given condition. Changing one value may modify the entire structure or meaning of an object. This was observable for the RubyGraph test candidate. If we consider for example a cyclic graph, then the removal of a single edge may result in a completely different graph and thus have an affect to the executed code.

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, such that a specific order is required. An example, in which the method sequence plays an important role, is the RubyChess test candidate.

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.

It is important to ensure the correctness of the empirical results and to avoid a misleading conclusion. *Statistical conclusion validity* is related to the reliability of the observed results. 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. On the other hand, RuTeG learns during the search process to distinguish between better and weaker data generators and applicable type combinations, and can therefore focus on more promising solutions.

Furthermore, it should also be mentioned that we applied only GA as a heuristic search algorithm to generate possible test cases. We do not know how other search techniques perform, such as hill climbing, simulated annealing or tabu search, to name but a few. Even if they cannot achieve a higher coverage, it may be possible that they find different solutions quicker.

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 generalizability. 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. Furthermore, RuTeG applies the ParseTree to collect some of the relevant information at runtime, which is a Ruby tool that presents the code in an abstract syntax tree using S-expressions. The collected information can probably be obtained also in other dynamic programming languages, but in a different way. 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. Therefore we cannot be sure whether the sample is representative of the Ruby code, but we can use the results as an indicator.

C.2 Conclusion

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 could find test scenarios that cover all the code only in 4 of 14 cases. The statistical significance of the difference in the results was tested by a student's t-test. A difference was also observable in the time required to find possible test cases, where RuTeG could find solutions quicker than the random test case generator.

There are different kinds of complexity that have a major effect on the generation of tests cases. These are self-defined types, complex and compound data structures, input data with a small solution space, and the method sequence to change the internal state of an object.

The complexity of method sequences is a sensitive factor in the automatic generation of test cases. The more complex the method sequence becomes, the more difficult it is to find possible solutions. Very complex and dependable method sequences may not occur frequently, but in such situations, both test case generators will most likely fail to achieve high code coverage.

The goal of RuTeG is to find test scenarios in order to cover as much code as possible. However, code coverage is not a very strong coverage criterion. A possible future step would be aiming for branch or condition coverage. The reason why RuTeG supports only code coverage, is because there is no tool available for Ruby that measures branch or condition coverage. Therefore, for code coverage it was possible to fall back on existing tools.

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.

References

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

[2] Ayari, K., Bouktif, S., and Antoniol, G. Automatic mutation test input data generation via ant colony. In *GECCO '07: Proceedings of the 9th annual conference on Genetic and evolutionary computation* (New York, NY, USA, 2007), ACM, pp. 1074–1081.

[3] Baresel, A., and Sthamer, H. Evolutionary testing of flag conditions. In *GECCO* (2003), pp. 2442–2454.

[4] Baresel, A., Sthamer, H., and Schmidt, M. Fitness function design to improve evolutionary structural testing. In *GECCO '02: Proceedings of the Genetic and Evolutionary Computation Conference* (San Francisco, CA, USA, 2002), Morgan Kaufmann Publishers Inc., pp. 1329–1336.

[5] Bouchachia, A. An immune genetic algorithm for software test data generation. In *HIS '07: Proceedings of the 7th International Conference on Hybrid Intelligent Systems (HIS 2007)* (Washington, DC, USA, 2007), IEEE Computer Society, pp. 84–89.

[6] Cheon, Y., and Kim, M. A specification-based fitness function for evolutionary testing of object-oriented programs. In *GECCO '06: Proceedings of the 8th annual conference on Genetic and evolutionary computation* (New York, NY, USA, 2006), ACM, pp. 1953–1954.

[7] Claessen, K., and Hughes, J. Quickcheck: a lightweight tool for random testing of haskell programs. In *ICFP '00: Proceedings of the fifth ACM SIGPLAN international conference on Functional programming* (New York, NY, USA, 2000), ACM, pp. 268–279.

[8] Díaz, E., Tuya, J., and Blanco, R. Automated software testing using a metaheuristic technique based on tabu search. In *ASE* (2003), pp. 310–313.

[9] Feldt, R. An interactive software development workbench based on biomimetic algorithms. Tech. rep., Department of Computer Engineering, Chalmers University of Technology, Gothenburg, Sweden, 2002.

[10] Feldt, R., Torkar, R., Gorschek, T., and Afzal, W. Searching for cognitively diverse tests: Towards universal test diversity metrics. In *ICST '08: 1st IEEE International Conference on Software Testing, Verification and Validation* (2008).

[11] Harman, M., Hassoun, Y., Lakhotia, K., McMinn, P., and Wegener, J. The impact of input domain reduction on search-based test data generation. In *ESEC-FSE '07: Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering* (New York, NY, USA, 2007), ACM, pp. 155–164.

[12] Harman, M., and McMinn, P. A theoretical & empirical analysis of evolutionary testing and hill climbing for structural test data generation. In *ISSTA '07: Proceedings of the 2007 international symposium on Software testing and analysis* (New York, NY, USA, 2007), ACM, pp. 73–83.

[13] Inkumsah, K., and Xie, T. Evacon: a framework for integrating evolutionary and concolic testing for object-oriented programs. In *ASE '07: Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering* (New York, NY, USA, 2007), ACM, pp. 425–428.

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

[15] Korel, B. Automated software test data generation. *IEEE Transaction on Software Engineering 16*, 8 (1990), 870–879.

[16] Majumdar, R., and Sen, K. Hybrid concolic testing. In *ICSE '07: Proceedings of the 29th International Conference on Software Engineering* (Washington, DC, USA, 2007), IEEE Computer Society, pp. 416–426.

[17] McCarthy, J. Lisp: a programming system for symbolic manipulations. In *ACM '59: Preprints of papers presented at the 14th national meeting of the Association for Computing Machinery* (New York, NY, USA, 1959), ACM, pp. 1–4.

[18] McMinn, P. Search-based software test data generation: a survey. *Software Testing, Verification and Reliability 14*, 2 (2004), 105–156.

[19] McMinn, P., Harman, M., Binkley, D., and Tonella, P. The species per path approach to searchbased test data generation. In *ISSTA '06: Proceedings of the 2006 international symposium on Software testing and analysis* (New York, NY, USA, 2006), ACM, pp. 13–24.

[20] McMinn, P., and Holcombe, M. Evolutionary testing of state-based programs. In *GECCO '05: Proceedings of the 2005 conference on Genetic and evolutionary computation* (New York, NY, USA, 2005), ACM, pp. 1013–1020.

[21] McMinn, P., and Holcombe, M. Evolutionary testing using an extended chaining approach. *Evolutionary Computing 14*, 1 (2006), 41–64.

[22] Michael, C., and McGraw, G. Automated software test data generation for complex programs. In *ASE '98: Proceedings of the 13th IEEE international conference on Automated software engineering* (Washington, DC, USA, 1998), IEEE Computer Society, p. 136.

[23] Michael, C. C., McGraw, G., and Schatz, M. A. Generating software test data by evolution. *IEEE Transactions on Software Engineering 27*, 12 (2001), 1085–1110.

[24] Oster, N., and Saglietti, F. Automatic test data generation by multi-objective optimisation. In *SAFECOMP* (2006), J. Grski, Ed., vol. 4166 of *Lecture Notes in Computer Science*, Springer, pp. 426–438.

[25] Pargas, R. P., Harrold, M. J., and Peck, R. Test-data generation using genetic algorithms. *Software Testing, Verification & Reliability 9*, 4 (1999), 263–282.

[26] Sen, K. Concolic testing. In *ASE '07: Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering* (New York, NY, USA, 2007), ACM, pp. 571–572.

[27] Sen, K., Marinov, D., and Agha, G. Cute: a concolic unit testing engine for c. In *ESEC/FSE-13: Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering* (New York, NY, USA, 2005), ACM, pp. 263–272.

[28] Stinckwich, S., and Ducasse, S. Introduction to the smalltalk special issue. *Computer Languages, Systems & Structures 32*, 2-3 (2006), 85–86.

[29] Thomas, D., Fowler, C., and Hunt, A. *Programming Ruby. The Pragmatic Programmer's Guide.* Pragmatic Programmers, 2004.

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

[31] Tracey, N., Clark, J., Mander, K., and McDermid, J. An automated framework for structural test-data generation. In *ASE '98: Proceedings of the 13th IEEE international conference on Automated software engineering* (Washington, DC, USA, 1998), IEEE Computer Society, p. 285.

[32] Tracey, N., Clark, J., McDermid, J., and Mander, K. A search-based automated test-data generation framework for safety-critical systems. In *Systems engineering for business process change: new directions* (New York, NY, USA, 2002), Springer-Verlag New York, Inc., pp. 174–213.

[33] Wappler, S., and Lammermann, F. Using evolutionary algorithms for the unit testing of object-oriented software. In *GECCO '05: Proceedings of the 2005 conference on Genetic and evolutionary computation* (New York, NY, USA, 2005), ACM, pp. 1053–1060.

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

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

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

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

[38] Wegener, J., Baresel, A., and Sthamer, H. Evolutionary test environment for automatic structural testing. *Information and Software Technology Special Issue on Software Engineering using Metaheuristic Innovative Algorithms 43*, 14 (2001), 841–854.

[39] Windisch, A., Wappler, S., and Wegener, J. Applying particle swarm optimization to software testing. In *GECCO '07: Proceedings of the 9th annual conference on Genetic and evolutionary computation* (New York, NY, USA, 2007), ACM, pp. 1121–1128.

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