DOI: 10.15439/2023F3165
ISSN 2300-5963 ACSIS, Vol. 37

Generation of Benchmark of Software Testing Methods for Java with Realistic Introduced Errors

Tomas Potuzak
0000-0002-8140-5178
Department of Computer Science and Engineering/ NTIS – New Technologies for the Information Society, European Center of Excellence, Faculty of Applied Sciences, University of West Bohemia
Univerzitni 8, 306 14 Plzen, Czech Republic
Email: tpotuzak@kiv.zcu.cz

Richard Lipka
0000-0002-9918-1299
NTIS – New Technologies for the Information Society, European Center of Excellence/Department of Computer Science and Engineering, Faculty of Applied Sciences, University of West Bohemia
Univerzitni 8, 306 14 Plzen, Czech Republic
Email: lipka@kiv.zcu.cz

***Abstract*—This paper deals with a benchmark of automated test generation methods for software testing. The existing methods are usually demonstrated using quite different examples. This makes their mutual comparison difficult. Additionally, the quality of the methods is often evaluated using code coverage or other metrics, such as generated tests count, test generation time, or memory usage. The most important feature – the ability of the method to find realistic errors in realistic applications – is only rarely used. To enable mutual comparison of various methods and to investigate their ability to find realistic errors, we propose a benchmark consisting of several applications with wittingly introduced errors. These errors should be found by the investigated test generation methods during the benchmark. To enable an easy introduction of various errors of various types into the benchmark applications, we created the Testing Applications Generator (TAG) tool. The description of the TAG along with two applications, which we developed as a part of the intended benchmark, is the main contribution of this paper.**

***Index terms*—Benchmark, software testing methods, automated test generation, application generation, Java code parsing, error introduction.**

I. Introduction

THE testing is a very important part of software development. It improves the probability of the correct functioning of an application, as it helps to uncover and fix errors unwittingly introduced into it during its development. As the manual creation of the tests is a lengthy and error-prone process, there is an intensive research on automated test generation methods for more than two decades (see, for example, [1] or [2]). In existing scientific papers, automated test generation is proposed or used on various testing levels. The lowest level is unit testing, which focuses on individual basic functional elements of the tested application, such as methods or functions. The middle level is the regression and integration testing focused on the correct cooperation of larger parts of the application. The highest level is the testing of the functionality of the entire application, its cooperation with its environment, and its adherence to its specification. At all levels, the expected advantages of the automated testing is the reduced time spent by programmers on the tests preparation and/or execution, decreased number of errors within the tests themselves and increased code coverage of the tested application. Nevertheless, there are also disadvantages. For example, it is inherently difficult to automatically verify whether the tested parts of the application give correct results, as it requires knowledge or generation of correct results. Another problem (discussed for example in [3]) is the combinatorial explosion. This means that the number of generated tests can be very high in order to cover all combinations of representative input values (e.g., values of tested method parameters). This can lead to long running times.

A different problem is the testing of automated test generation methods themselves. In many scientific papers, the method functioning is often demonstrated only on small examples (e.g., in [4] or [5]), from which the usability in a real project cannot be concluded. Although there are also methods tested on more realistic examples (e.g., in [6] or [7]), these examples are not mutually similar and do not enable direct comparison of the features of the methods.

It should also be noted that, from the pragmatic point of view, the most important feature of the method in a real project is the ability to find realistic errors of various types [8]. Nevertheless, in scientific papers, this feature is virtually never used as the metric for method assessing (with some exceptions, e.g., [9]). A quite common approach used for automated test generation methods evaluation is mutation testing [10]. Using this approach, several versions of the tested program (so-called mutants) with small changes imitating real errors introduced by mutation operators are generated. The quality of the automated test generation method can be then assessed by the number of mutants the method is able to identify [10]. However, a large portion of the papers uses only code coverage for the methods evaluation (e.g., [11] or

The work was supported by the Ministry of Education, Youth and Sports of the Czech Republic university specific research project SGS-2022-016 Advanced methods of data processing and analysis.

[12]) or other metrics, such as generated tests count, test generation time, or memory usage (e.g., in [13]).

In order to enable mutual comparison of various automated test generation methods and to investigate their ability to find realistic errors, we propose to create a benchmark consisting of several various applications with wittingly introduced errors. The numbers of the errors discovered and not discovered by each method can be then used for a direct assessment of the quality of each method. In order to enable an easy introduction of various errors of various types into the benchmark applications, we created a tool called Testing Applications Generator or TAG. The TAG is designed in and for Java language, similarly to many automated test generation methods (e.g., [4] or [14]). It enables to introduce errors of various types into the source codes of methods bodies of an application. The description of the TAG and its functioning and the first two applications, which we plan to utilize as a part of the benchmark of the automated test generation methods, are the main contributions of this paper.

The remainder of the paper is structured as follows. Section II briefly discusses the existing automated test generation methods. Section III is focused on related work. In Section IV, the TAG is described in detail. The two benchmark applications are described in Section V. The tests of the TAG and their results are described in Section VI. The conclusions and the future work are discussed in Section VII.

II. Automated Test Generation Methods

There is a large number of existing test generation methods, which are based on various technologies and use various inputs for their functioning (see [15] for details). Some examples are summarized in following subsections.

A. Commonly Used Technologies in Testing Methods

The test generation methods can be based on a single technology, but often employ multiple technologies. Control-flow-based methods utilize control flow diagrams created by static analysis, for example in [1]. The diagrams are used to generate tests covering all branches of the program, often in conjunction with random input data generation, as in [16].

Specification-based methods are somewhat similar to control-flow-based methods as the tests can be generated from diagrams utilized for the description of the application, such as UML diagrams, as in [17], [18]. Different forms of specification can be used as well, for example use case descriptions employed in [17], [19] or contracts employed in [20].

The search-based methods typically employ a search meta-heuristic to generate tests including genetic algorithms (e.g., in [4], [11]), particle swarm optimization (e.g., in [5]), or ant colony optimization (e.g., in [21]). The meta-heuristic is typically combined with a technology enabling to evaluate the found solutions, for example with control-flow diagrams (e.g., in [22]) or program instrumentation (e.g., in [23]).

Program-execution-based methods employ real or symbolic executions of the tested program for test generation. If the real program is executed, there is usually some form of code instrumentation, such as in [24] or [25]. Examples of symbolic-execution-based methods can be found in [26] or [27].

B. Commonly Used Inputs for Testing Methods

The aforementioned and other existing test generation methods use various primary inputs. In many cases, it is the source code of the tested program, as for example in [1], [4], [11], [26], or [27]. The source code does not have to be used directly. For example, in [1], the static analysis of source code is used for the generation of control-flow diagrams, which are in turn used for the generation of the tests. In [4], the source code is used for the determination of the method parameters, which are then used by a genetic algorithm.

The primary input can also be an instrumented execution of the program (e.g., in [24], [25]), or Java bytecode (e.g., in [12]). Yet another primary input can be a description of the tested program in some form, for example the UML diagram (e.g., in [17], [18]) or the contracts description (e.g., in [20]).

It should be noted however that, regardless of utilized primary input, the resulting generated tests are used for the testing of a real tested program (i.e., not its model nor description). That means that, although the source code and/or executable version of the program may not be necessary for the generation of the tests, it is required for the execution of the generated tests. The executed tests should then discover errors present in the program.

III. Related Work

As we are working on the creation of automated test generation methods benchmark, which would solve the difficulties of test generation method comparison (see Section I), we investigated the existing research in this area.

A. Benchmarks of Testing Methods

The benchmarks of testing methods are quite rare, but there are a few examples. A benchmark was used for a competition of Java unit tests generating tools (Java Unit Testing Tool Contest 2018) [28]. The benchmark consisted of 59 real-life Java classes from 7 open-source projects. The projects were selected randomly from a pool of GitHub repositories, which met predefined criteria, such as having enough stars, being able to be built by Maven, and containing JUnit 4 tests [28]. From the total number of 2 566 classes of the 7 projects, only classes with at least 1 method with at least 2 condition points were considered further. From them, 59 randomly selected classes were used as the benchmark [28]. For the selection of the projects, an unspecified script was used. For the class filtering, an extended CKJM library was used [28]. No intentional introduction of errors was reported in [28]. For the evaluation of the contesting tools, code coverage computed by the JaCoCo tool and mutation testing analysis performed by PIT tool were used [28].

A similar benchmark was used for the Java Unit Testing Tool Contest 2020 [29]. The selection of the projects was a bit different, the predefined criteria were being able to be built by Gradle or Maven and containing JUnit 4 tests [29]. In the end, 4 projects were selected. Only 1 094 classes with at least 1 method with at least 4 condition points were considered further. These classes were further filtered by trying to generate tests for them using Randoop tool with 10 seconds time budget for each class. Only the classes, for which at least one test was generated, were considered further. Using this filter, 382 classes remained. From them, 60 classes were randomly selected for the benchmark, while another 10 were selected based on the past experience [29]. For the class filtering, JavaNCSS tool was used [29]. Again, no intentional introduction of errors was reported in [29]. Similarly to [28], code coverage computed by the JaCoCo tool and mutation testing analysis performed by the PIT tool were used for the evaluation of contesting tools [29].

In [10], the creation of a repository of artifacts, usable for standardized evaluation of mutation-based testing methods, is described. The authors used a relational database as the storage of the artifacts and created import scripts for them. The basis of the repository is a set of Java classes taken from 4 open source projects from GitHub and from a set of simple Java programs. From the ca. 2 000 classes, ca. 50 000 test cases and ca. 195 000 mutants were generated using the existing EvoSuite and PIT tools, respectively. These test cases and mutants are also stored in the repository [10].

In [8], a benchmark testbed application with artificial error injection for the evaluation of testing methods is described. The application is the University Information System Testbed (TbUIS), a fictional, but functional university study information system, which includes students, teachers, management of exams, and related processes. It is a layered J2EE-JSP-Spring web application with relational database storage and object-relational mapping (ORM) using Hibernate. The application consists of 87 `.java` and 18 `.jsp` files with more than 10 000 lines of code in total. The TbUIS source code is highly covered by automated unit and frontend functional tests in order to reduce the number of errors introduced during the development of the application [8].

To introduce errors into the TbUIS application, the Error seeder application is used. It operates on the bean (i.e., class) level. Each bean of the TbUIS application can be replaced by a version with introduced error or errors. The errors, which shall be introduced, are selected from a predefined set. The resulting version of the TbUIS application with the beans with introduced errors can be then compiled and used as a part of the benchmark of testing methods [8].

B. *Assessing and Comparability of Testing Methods*

The diversity of examples, on which the functionality of the automated test generation methods is demonstrated in scientific literature (see Section I), is mentioned in several review papers. A thorough review paper [30] deals with search-based test generation methods. One of the conclusions is that there is a lack of standardized rigorous way to assess and compare various methods. Moreover, it is pointed out that, while many of the test-generating methods can achieve high code coverage, it is not clear whether the tests are actually able to find errors in the source code [30].

Similarly, the review paper [31], which is focused on mutation testing, concludes that the experimental material used in the papers describing various test generation methods is typically non-standardized, lacks reusability, and is rarely available to be shared to support further experiments [31]. One of the conclusions of the review paper [32] focused on search-based and mutation testing methods is that the comparability of the automated methods is difficult [32].

IV. Testing Applications Generator

In order to address the difficult comparability of the automated test generation methods, we decided to create a benchmark, which would consist of several various applications with wittingly introduced errors. The number of the errors discovered and not discovered by each automated test generation method can be then used for a direct assessment of the quality of each method. Nevertheless, since various methods can be focused on specific types of applications and/or errors, the creation of a single benchmark application with hardwired errors would be of limited usefulness. Hence, we created a prototype implementation of the Testing Applications Generation (TAG) tool. The TAG enables to introduce errors of various types into the source codes of imported applications. The TAG is inspired by the TbUIS [8] (see Section III.A), but is different in many ways (see below).

A. *Usage of TAG*

The TAG is a Java desktop application with a graphical user interface (GUI) enabling to import multiple Java applications. The entire project can be imported (see Section IV.C), but the source codes are required. The source code files of each imported application are parsed and the entire structure of packages, classes, interfaces, and other code artifacts are stored down to the level of individual methods.

Each method has a single imported body, but additional copies of the body can be created on user request. The user can then introduce one or multiple errors into each copy (see Section IV.D for details). All the created copies are stored.

In order to export an application with selected introduced errors, the user then only selects the method bodies containing the required errors and the application is created in a selected folder. The exported application can be used as a part of the benchmark of automated test generation methods, as it contains known introduced errors and, inevitably, other errors already present in the application prior its import.

So, the TAG is distantly similar to the Error seeder of the TbUIS (see Section III.A). However, unlike the Error seeder, the TAG is not designed for a single application. Multiple applications can be stored and virtually any Java application with source codes can be imported. There are no requirements for a specific technology, such as Spring, the applica-

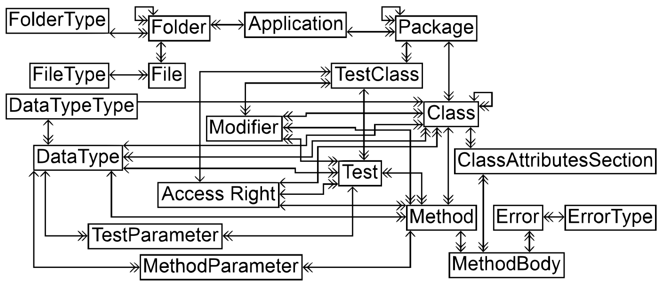


Fig. 1 Scheme of the TAG data model

tion must be only compilable by a standard Java compiler (currently version 11). The introduced errors are also not limited to a predefined set. Lastly, the Error seeder operates at the Java beans level (i.e., an entire bean is replaced with a faulty one), while the TAG operates at the method level (i.e., a method body is replaced with a faulty one).

B. Data Model

All the data utilized by the TAG are stored in a relational database using ORM via Hibernate. The scheme of the data model is depicted in Fig. 1. The database enables to save the entire structure of an application project including folder structure with various types of files (e.g., libraries, resources, documentation, or build scripts) and the package structure with classes, interfaces, and other source code artifacts. The files other than source code files are stored only as type, name, content, and parent folder. On the other hand, the folders representing package structure are also stored as packages and the contained source code files are parsed and stored as classes, interfaces, methods, and their bodies. The content of the class outside any method (i.e., typically attributes), so-called *class attributes section* are stored as well. For each method, multiple bodies can be stored and, for each class, multiple class attributes sections can be stored. The test classes (or test cases in JUnit terminology) containing individual tests are parsed as well. However, each test (corresponding to a method of the test class) has only one body and each test class has only one class attributes section. The reason is that the introduction of errors into tests is not expected.

Besides the structures of the imported applications, the database also contains all necessary code lists, such as access rights, modifiers, file and folder types, or error types. Some of them, for example the access rights and modifiers are expected to hold constant sets of values. To the others, such as file and folder types or error types, new values can be added as needed. The database also contains all the errors, which are introduced into the applications.

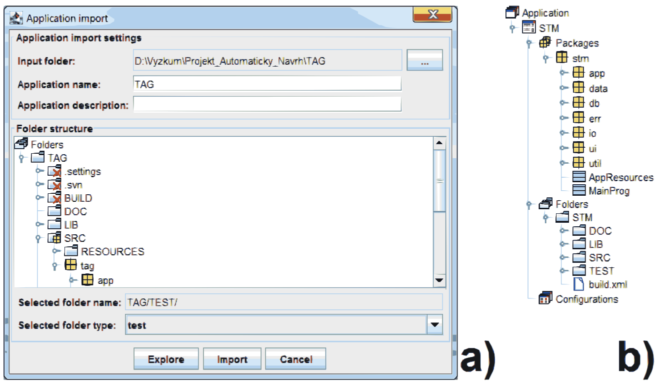
C. Application Import

For each application, its entire project can be imported including the folder structure, source codes, resources files, libraries, build scripts, and other files. Nevertheless, only the source codes are required. The contents of other files are only stored into the database, while the contents of source code files (i.e., `.java` files) are parsed down to the method body level, but not further. That means that the content of the body of a method is not parsed and is stored as a text segment. Similarly, a class attributes section (containing mainly attributes) is stored as a text segment. On the other hand, the headers of classes and methods are parsed including method parameters, return values, type parameters, and so on. Test classes are parsed and stored similarly to normal classes.

During the import, the content of the selected folder with the imported application is explored and displayed as a tree to the user, who must mark the source code and tests subfolders. He or she can also choose the type of other folders or mark some not required folders as ignored (see Fig. 2a). Then, the import including the parsing of the source code files is performed automatically. There are no specific requirements for the folder structure, it is possible to import Eclipse or Maven/Gradle styles or customized structures.

If there is a problem during parsing a source code file, the import is not stopped. Instead, the source code file is stored into the database as a general file with its entire content "as is" (similarly to, for example, a resource file). The import then resumes with the next source code file. This way, one (or multiple) file with a parsing error does not hinder the entire import. It is not possible to introduce errors into the files, whose parsing failed (unless the application is edited later directly using the GUI of the TAG), but the other correctly parsed files are not negatively affected. The parsing error can be caused by syntax errors or by using an unexpected construction, such as constructions added to newer versions of Java. Currently, the parser is set to Java 11.

Once the application is imported, its folder and package structures are displayed as a tree (see Fig. 2b). It is possible to display the details of its individual items and add/edit/delete them. Theoretically, it is possible to create the entire application by adding its individual items one by one (i.e., without the import), but this approach would be lengthy and error-prone and it is not recommended. The TAG is no Integrated Development Environment (IDE), its editing capabilities are intended only for little changes, which might be necessary during the introduction of the errors (see Section IV.D) or during other minor adjustments of the application (e.g., correction of the failed parsing – see above).


Application import

Application import settings

Input folder:	D:\Vyzkum\Projekt_Automaticky_Navrh\TAG
Application name:	TAG
Application description:	

Folder structure

Selected folder name:	TAG/TEST/
Selected folder type:	test

Fig. 2 Application import (a) and imported application structure (b)

D. Error Introduction

The errors introduction is manual in the sense that the user must manually edit a method body or a class attributes section, to which he or she intends to insert the error. The added error should be also added to the list of errors. Each specific error has its type, name, description and corresponding method body and/or class attributes section. This way, each method body and class attributes section can be described by the errors it contains. This enables easy selection of intended method bodies and class attributes sections during the export of the application (see Section IV.E).

Because, within a class, errors can be introduced into various bodies of the same or of various methods and also into various class attributes sections, it is possible that some of the method bodies would not be compatible with some of the class attributes sections. Hence, the information, which bodies are compatible with which class attributes sections, is stored and then utilized during export (see Section IV.E)

The types of the errors can be selected from a list and the user can readily add new types. Currently, the list is largely empty, since the common and realistic error types determination is the part of our future work (see Section VII).

E. Application Export

Each application can be exported in multiple versions, with different errors or entirely free of artificially introduced errors. The errors present in the application during the import (i.e., unwittingly introduced during application development) will be of course present. Each version is represented by so-called *configuration*. The configuration is created by specifying, which method body shall be used for each method with multiple method bodies and which class attributes section shall be used for each class with multiple class attributes sections. The stored information about the compatible method bodies and class attributes sections is used for checking whether only compatible method bodies and class attributes sections are used together. There can be multiple configurations per application. All available configurations are displayed as part of the application tree (see Fig. 2b).

The version of the application corresponding to the configuration can be exported to a selected folder. The export is automatic. The generation of the folder and package structure is straightforward. The `.java` files are generated from the classes and their contents in folders corresponding to their packages. For this purpose, the user can specify several features of the code style, such as the usage of spaces or tabulators for the indentation. The other files are only created in corresponding folders and filled with their content stored in the database.

The exported application can be directly used as a part of a benchmark. If its compilation is required, for example for the generation or execution of the tests by the benchmarked test generation methods, it can be performed using a standard build script, which is usually present. The contemporary prototype version of the TAG cannot perform the compilation automatically, but it is part of our future work. It is currently possible to import end export `.class` files with bytecode. However, without automatic compilation, the exported `.class` files do not correspond to exported `.java` files if some errors were wittingly introduced using the TAG. Hence, the manual compilation after the export is recommended.

V. Benchmark Applications

Concurrently with our work on the implementation of the TAG, we are working on our own benchmark for test generation methods. This work consists of two main branches – creation or selection of the applications used for the benchmark and selection of the artificially introduced errors into these applications.

The selection of the errors is part of our future work (see Section VII). Regarding the benchmark applications, we decided to create new applications rather than using existing projects, similarly to the TbUIS (see Section III.A). The main reason is the consequent full control over these applications enabling us, among other things, to prepare and perform their very thorough testing.

Besides the thorough testing, there were several other requirements for the applications:

- Usage of relational database
- Usage of ORM
- Usage of web services
- Usage of file input and output
- Usage of command line interface (CLI)
- Optionally usage of third party libraries
- Optionally usage of simple GUI for debugging purposes and simple data input/output.

Based on these requirements, the resulting applications should use various common technologies and there should be an opportunity to introduce errors of very different types (such as a database error versus a web service error). The GUI was not considered essential, since the test generating methods are usually not focused on GUI testing. It is also not considered necessary for all the resulting applications to meet all the requirements.

Currently, there are two applications, which were developed by two of our bachelor students (see Acknowledgment section). The applications are two parts (frontend and backend) of a single system – a school agenda of an elementary or a high school. Both parts are described in Sections V.A and V.B. During the development of both applications, approximately half of the development time was devoted to the unit, integration, and functional testing to limit the number of errors unwittingly introduced during the development. Even then, the applications are expected to contain some errors. However, these errors are likely to be discovered during the usage of the applications as a part of the benchmark sooner or later. Once an unwittingly introduced error is discovered, it will be only documented and its discovery in the further usages of the benchmark will be observed.

A. Benchmark Application 1 – School Agenda Backend

Benchmark application 1 (BA1 – School Agenda Backend) manages data of elementary or high school agenda including students, teachers, classrooms, absences, and so on. It is a realistic application in sense that it would be utilizable for a real school, but some aspects may be missing in its data model. Additionally, there is only a basic HTTP authentication for the access of the web service.

The BA1 is a standard Java application with a layered architecture utilizing Spring Boot. The data are stored in a relational database using ORM via Hibernate. There is no GUI, the only interface of the application is the REST (Representational State Transfer) web service. The data transferred using the web service is in JSON format. Besides the Java Core API classes, the application utilizes several third-party libraries, such as Jackson, Hibernate, Spring boot, or JUnit among others.

The source code of the application consists of 77 classes in 10 packages with a total length of 328 kB. The source code of the unit and integration tests consists of 57 test classes with 655 tests with a total length of 448 kB. Functional testing consisting of 111 scenarios was performed manually.

B. Benchmark Application 2 – School Agenda Frontend

Benchmark application 2 (BA2 – School Agenda Frontend) provides the user interface for the school agenda. It communicates with the BA1 using the REST web service. It provides the JavaFX GUI for manual management of the data and CLI for bulk data import and export.

The BA2 is a standard Java application with a layered architecture. All the data are acquired from the BA1 REST web service, there is no direct access to the database. The data can be imported and exported from and into `.json` and `.xml` files. The application utilizes Jackson and JUnit among other libraries.

The source code of the application consists of 141 classes in 44 packages with a total length of 489 kB. The source code of the unit and integration tests consists of 38 test classes with 524 tests with a total length of 239 kB. Functional testing consisting of 437 scenarios was performed manually.

VI. Tests and Results

The prototype implementation of the TAG was tested in order to verify its ability to import and parse and export the applications, which are intended for the benchmark.

A. Testing Environment and Applications

The tests were performed on a standard notebook. Its hardware consists of dual-core Intel i5-6200U at 2.30 GHz, 8 GB of RAM, 250 GB SSD, and 500 GB HDD with 7 200 RPM. All the imports and exports were performed using the 500 GB HDD (not the 250 GB SSD). The installed software was Windows 7 SP1 64bit, and Java 11 (64 bit).

Five applications were used for testing. Each application was represented by a single folder with the entire project. The applications were developed using various IDEs and build tools leading to various folder structures. Also, the applications were developed by four different authors leading to different Java code styles and different utilized Java versions. All these features increase the variability of the applications and hence improve the quality of testing.

First two applications were the BA1 (see Section V.A) and the BA2 (see Section V.B). Second two applications were taken from a project focused on traffic assignment problem – the Dynamic Traffic Assignment (DTA) and Static Traffic Modeler (STM). The last application was the TAG itself. The features of the applications are summarized in Table I. The "Folder structure" describes the folder structure of the project. Three applications utilize a Maven/Gradle-based structure while the two remaining utilize an Eclipse-based style. That does not mean that the same-based structures are identical, there are slight variations. The "Size to import" shows the total size of the folders and files, which are not ignored during the import. The "`.java` to import count" is the number of `.java` files contained in the imported folders and the "Others to import count" is the number of all other files contained in the imported folders.

B. Tests Description

The tests were performed the same way for each application. In the TAG, the application import was started and the root folder of the application project was selected as the input folder. Then, the structure of the project was explored and displayed as a tree. The tester marked the source and test folders and also the ignored folders. The ignored folders were the project settings folders, build and output folders, and version control folders. The application was then automatically imported. No errors were introduced into the imported application. Rather, the imported application was exported to a different folder without any functional changes.

Several parameters were observed – the number of folders and files, the number of packages and classes, the number of unsuccessfully parsed files (see Section IV.C), the import time, and the export time. Because of the time measurement, import and export of each application were performed four times. First time measurement was discarded, as it was significantly higher than the others, because the data from the disk was not present in the cache. Three other time measurements were averaged. Although only three measurements do not offer significant precision, it is enough to get a good idea of how long the export and import approximately last, which is the main purpose of the time measurement. The other parameters did not change between attempts, since the import and export are both deterministic.

Table I Features of the Applications Used for the Testing

Feature	BA1	BA2	DTA	STM	TAG
Folder structure	Maven/Gradle			Eclipse	
Size to import [kB]	835	2 856	899	9 945	11 415
`.java` to import count	134	179	78	305	62
Others to import count	25	408	97	34	39

Since the introduction of errors, which is the purpose of the TAG, was not part of these tests, the exported application should be identical to its imported counterpart. To determine this, the exported application was compiled and executed and manual functional testing of randomly chosen functionalities was performed. Moreover, all unit and aggregation tests present in the application were executed. Direct comparison of the imported and exported (i.e., generated) source code was not performed since there are non-functional differences, such as different indentation, empty lines, methods order, and so on.

C. Tests Results

The results of the testing are summarized in Table II. It can be observed that the import and export times are quite similar for a single application, with the export time being slightly higher in all instances. The times are also quite low, under a second in four of five applications, and under 2.5 seconds in the case of the STM. As such, the import and export times do not pose any problem for the TAG usage. The times seem to be influenced mainly by the number of parsed (and generated) `.java` files.

The parsing of `.java` files during the import works very well. There were no parsing errors in two applications, namely the BA1 and DTA. There were 5 files (2.9%), which were not parsed correctly, for the BA2, 13 files (3.6%) for the STM, and 10 (7.6 %) for the TAG. The parsing errors were caused by Java 14 `record` construction in the case of the BA2. In all other cases, the parsing error was caused by the usage of methods with type parameters (e.g., `<T> void foo(T t)`), which our parser currently does not support. This setback will be corrected as part of our future work (see Section VII). The unsuccessfully parsed files were stored as general files with their entire contents (see Section IV.C) and were correctly recreated similar to resource or library files during the export. The counts of these files were added to "Files count" row in Table II. After this adjustment, the numbers of actually imported files precisely correspond to the expected numbers of imported files (compare Table II "Files count" row and Table I "Others to import count" row).

The testing of the exported applications as described in Section VI.B was performed for all applications successfully, no errors unwittingly introduced by the TAG were found. This indicates that even the unsuccessfully parsed `.java` files during the import do not pose a problem as long as their number is low enough. A high number of unsuccessfully parsed files (e.g., 50 %) would significantly reduce the amount of source code, to which an error can be intentionally introduced. This, in turn, would reduce the usefulness of the application as a part of the benchmark.

TABLE II RESULTS OF THE APPLICATIONS EXPORT AND IMPORT

Feature	BA1	BA2	DTA	STM	TAG
Packages count	10	44	5	24	8
Classes count	134	174	84	363	132
Folders count	27	183	41	50	21
Files count	25	413	97	47	49
Unsuccessfully parsed files count	0	5	0	13	10
Import time [ms]	688	725	717	2 231	582
Export time [ms]	757	833	841	2 477	664

VII. Conclusion and Future Work

In this paper, we described the proposal for a benchmark of automated test generation methods consisting of realistic applications with artificially introduced errors. We focused primarily on the TAG tool, which enables the error introduction into imported applications and the export of multiple versions of multiple applications with various sets of introduced errors. The tests of the prototype implementation of the TAG were also described and its ability to import and export application was demonstrated using five applications. We also described first two applications, which are planned to be part of the benchmark.

In our future work, we will continue to work on the implementation of the TAG. These works include updating the parser to include newer Java constructions and the methods with type parameters. We also plan to improve user experience by automatically analyzing the structure of the imported folder and presetting all the types of files and folders to the correct types. The automatic compilation of the exported applications will be added as well.

We will also add common and realistic types of errors, which will be then introduced into the applications for their usage as the part of the benchmark. We plan to semi-automatically process publicly available contents of bug tracking tools to determine the common types of errors in developed and maintained applications and their frequency of occurrence. Then, we will utilize the obtained information to introduce realistic errors into our benchmark applications and finish the benchmark of automated test generation methods. Both the resulting benchmark and the TAG applications are planned to be made public, once they are finished.

Acknowledgment

The authors wish to thank to Martin Dub and Martin Jakubašek, who created the BA1 and BA2 applications, respectively, as part of their bachelor thesis.

References

[1] N. Gupta, A. P. Mathur, and M. L. Soffa, "Generating test data for branch coverage," in Proceedings ASE 2000 - Fifteenth IEEE International Conference on Automated Software Engineering, Grenoble, September 2000, https://doi.org/10.1109/ASE.2000.873666

[2] P. Fröhlich and J. Link, "Automated Test Case Generation from Dynamic Models," in ECOOP '00: Proceedings of the 14th European Conference on Object-Oriented Programming, Cannes, June 2000, pp. 472–491, https://doi.org/10.1007/3-540-45102-1_23

[3] B. S. Ahmed, K. Z. Zamli, W. Afzal, and M. Bures, "Constrained Interaction Testing: A Systematic Literature Study," in IEEE Access, vol. 5, November 2017, pp. 25706–25730, https://doi.org/10.1109/ACCESS.2017.2771562

[4] Z. J. Rashid and M. F. Adak, "Test Data Generation for Dynamic Unit Test in Java Language using Genetic Algorithm," in 2021 6th International Conference on Computer Science and Engineering

(UBMK), Ankara, September 2021, pp. 113–117, https://doi.org/10.1109/UBMK52708.2021.9558953

[5] R. J. Cajica, R. E. G. Torres, and P. M. Álvarez, "Automatic Generation of Test Cases from Formal Specifications using Mutation Testing," in 2021 18th International Conference on Electrical Engineering, Computing Science and Automatic Control (CCE), Mexico City, November 2021, https://doi.org/10.1109/CCE53527 .2021.9633118

[6] H. Homayouni, S. Ghosh, I. Ray, and M. G. Kahn, "An Interactive Data Quality Test Approach for Constraint Discovery and Fault Detection," in 2019 IEEE International Conference on Big Data (Big Data), Los Angeles, December 2019, pp. 200–205, https://doi.org/10.1109/BigData47090.2019.9006446

[7] A. Alsharif, G. M. Kapfhammer, and P. McMinn, "DOMINO: Fast and Effective Test Data Generation for Relational Database Schemas," in 2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST), Västerås, April 2018, pp. 12–22, https://doi.org/10.1109/ICST.2018.00012

[8] M. Bures, P. Herout, and S. A. Bestoun, "Open-source Defect Injection Benchmark Testbed for the Evaluation of Testing," in Proceedings of the 13th IEEE International Conference on Software Testing, Validation and Verification (ICST), Porto, October 2020, pp. 442–447, https://doi.org/10.1109/ICST46399.2020.00059

[9] M. Kelly, C. Treude, and A. Murray, "A Case Study on Automated Fuzz Target Generation for Large Codebases," in International Symposium on Empirical Software Engineering and Measurement (ESEM), Porto de Galinhas, Septemmber 2019, https://doi.org/10.1109/ESEM.2019.8870150

[10] A. V. Pizzoleto, F. C. Ferrari, and G. F. Guarnieri, "Definition of a Knowledge Base Towards a Benchmark for Experiments with Mutation Testing," in SBES '21: Proceedings of the XXXV Brazilian Symposium on Software Engineering, Joinville, September 2021, pp. 215–220, https://doi.org/10.1145/3474624.3477060

[11] S. Varshney and M. Mehrotra, "A differential evolution based approach to generate test data for data-flow coverage," in 2016 International Conference on Computing, Communication and Automation (ICCCA), Greater Noida, April 2016, pp. 796–801, https://doi.org/10.1109/CCAA.2016.7813848

[12] J. Zhang, S. K. Gupta, and W. G. Halfond, "A New Method for Software Test Data Generation Inspired by D-algorithm," in 2019 IEEE 37th VLSI Test Symposium (VTS), Monterey, April 2019, https://doi.org/10.1109/VTS.2019.8758641

[13] H. V. Tran, L. N. Tung, and P. N. Hung, "A Pairwise Based Method for Automated Test Data Generation for C/C++ Projects," in 2022 RIVF International Conference on Computing and Communication Technologies (RIVF), Ho Chi Minh City, December 2022, https://doi.org/10.1109/RIVF55975.2022.10013824

[14] M. Motan and S. Zein, "Android App Testing: A Model for Generating Automated Lifecycle Tests," in 2020 4th International Symposium on Multidisciplinary Studies and Innovative Technologies (ISMSIT), Istanbul, October 2020, https://doi.org/10.1109/ISMSIT50672.2020.9254285

[15] T. Potuzak and R. Lipka, "Current Trends in Automated Test Case Generation," in FedCSIS 2023, September 2023, to be published

[16] S. Poulding and J. A. Clark, "Efficient software verification: Statistical testing using automated search," in IEEE Transactions on Software Engineering, vol. 36, no. 6, February 2010, pp. 763–777, https://doi.org/10.1109/TSE.2010.24

[17] L. Bao-Lin, L. Zhi-shu, L. Qing, and C. Y. Hong, "Test Case Automate Generation from UML Sequence diagram and OCL expression," in 2007 International Conference on Computational Intelligence and Security (CIS 2007), Harbin, December 2007, pp. 1048–1052, https://doi.org/10.1109/CIS.2007.150

[18] Meiliana, I. Septian, R. S. Alianto, Daniel, and F. L. Gaol, "Automated Test Case Generation from UML Activity Diagram and Sequence Diagram using Depth First Search Algorithm," in Procedia Computer Science, vol. 116, October 2017, pp. 629–637, https://doi.org/10.1016/j.procs.2017.10.029

[19] M. Zhang, T. Yue, S. Ali, H. Zhang, and J. Wu, "A Systematic Approach to Automatically Derive Test Cases from Use Cases Specified in Restricted Natural Languages," in LNCS, vol. 8769, 2014, pp. 142–157, https://doi.org/10.1007/978-3-319-11743-0_10

[20] D. Xu, W. Xu, M. Tu, N. Shen, W. Chu, and C. H. Chang, "Automated Integration Testing Using Logical Contracts," in IEEE Transactions on Reliability, vol. 65, no. 3, November 2016, pp. 1205–1222, https://doi.org/10.1109/TR.2015.2494685

[21] H. Sharifipour, M. Shakeri, and H. Haghighi, "Structural test data generation using a memetic ant colony optimization based on evolution strategies," in Swarm and Evolutionary Computation, vol. 40, June 2018, pp. 76–91, https://doi.org/10.1016/j.swevo.2017.12.009

[22] T. Shu, Z. Ding, M. Chen, and J. Xia, "A heuristic transition executability analysis method for generating EFSM-specified protocol test sequences," in Information Sciences, vol. 370–371, November 2016, pp. 63–78, https://doi.org/10.1016/j.ins.2016.07.059

[23] S. Khor and P. Grogono, "Using a genetic algorithm and formal concept analysis to generate branch coverage test data automatically," in Proceedings. 19th International Conference on Automated Software Engineering, 2004, Linz, September 2004, pp. 346–349, https://doi .org/10.1109/ASE.2004.1342761

[24] C. Fetzer and Z. Xiao, "An automated approach to increasing the robustness of C libraries," in Proceedings International Conference on Dependable Systems and Networks, Washington D.C., June 2002, pp. 155–164, https://doi.org/10.1109/DSN.2002.1028896

[25] H. Tanno, X. Zhang, T. Hoshino, and K. Sen, "TesMa and CATG: Automated Test Generation Tools for Models of Enterprise Applications," in 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering, Florence, May 2015, pp. 717–720, https://doi.org/10.1109/ICSE.2015.231

[26] T. Su, G. Pu, B. Fang, J. He, J. Yan, S. Jiang, and J. Zhao, "Automated Coverage-Driven Test Data Generation Using Dynamic Symbolic Execution," in 2014 Eighth International Conference on Software Security and Reliability (SERE), San Francisco, June 2014, pp. 98–107, https://doi.org/10.1109/SERE.2014.23

[27] L. Hao, J. Shi, T. Su, and Y. Huang, "Automated Test Generation for IEC 61131-3 ST Programs via Dynamic Symbolic Execution," in 2019 International Symposium on Theoretical Aspects of Software Engineering (TASE), Guilin, July 2019, https://doi.org/10.1109/TASE.2019.00004

[28] U. R. Molina, F. Kifetew, and A. Panichella, "Java Unit Testing Tool Competition: Sixth round," in SBST '18: Proceedings of the 11th International Workshop on Search-Based Software Testing, Gothenburg, May 2018, pp. 22–29, https://doi.org/10.1145/3194718.3194728

[29] X. Devroey, S. Panichella, and A. Gambi, "Java Unit Testing Tool Competition: Eighth Round," in ICSEW'20: Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops, Seoul, June 2020, pp. 545–548, https://doi.org/10.1145/3387940.3392265

[30] S. Ali, L. C. Briand, H. Hemmati, and R. K. Panesar-Walawege, "A systematic review of the application and empirical investigation of search-based test case generation," in IEEE Transactions on Software Engineering, vol. 36, no. 6, August 2009, pp. 742–762, https://doi.org /10.1109/TSE.2009.52

[31] A. V. Pizzoleto, F. C. Ferrari, A. J. Offutt, L. Fernandes, and M. Ribeiro, "A Systematic Literature Review of Techniques and Metrics to Reduce the Cost of Mutation Testing," in Journal of Systems and Software, vol. 157, November 2019, https://doi.org/10.1016/j.jss.2019.07.100

[32] R. Jeevarathinam and A. S. Thanamani, "A survey on mutation testing methods, fault classifications and automatic test cases generation," in Journal of Scientific and Industrial Research (JSIR), vol. 70, no. 2, February 2011, pp. 113–117.