CATTO: Just-in-time Test Case Selection and Execution

Dario Amoroso d'Aragona*, Fabiano Pecorelli*, Simone Romano†, Giuseppe Scanniello†,
Maria Teresa Baldassarre‡, Andrea Janes§, Valentina Lenarduzzi¶
dario.amorosodaragona@tuni.fi, fabiano.pecorelli@tuni.fi, siromano@unisa.it, gscanniello@unisa.it
teresa.baldassarre@uniba.it, andrea.janes@unibz.it, valentina.lenarduzzi@oulu.fi
*Tampere University, Finland, †University of Salerno, Italy, ‡University of Bari, Italy
§Free University of Bozen-Bolzano, Italy, ¶University of Oulu, Finland

***Abstract*—Regression testing wants to prevent that errors, which have already been corrected once, creep back into a system that has been updated. A naïve approach consists of re-running the entire test suite (TS) against the changed version of the software under test (SUT). However, this might result in a time- and resource-consuming process; e.g., when dealing with large and/or complex SUTs and TSs. To avoid this problem, Test Case Selection (TCS) approaches can be used. This kind of approaches build a temporary TS comprising only those test cases (TCs) that are relevant to the changes made to the SUT, so avoiding executing unnecessary TCs. In this paper, we introduce CATTO (Commit Adaptive Tool for Test suite Optimization), a tool implementing a TCS strategy for SUTs written in Java as well as a wrapper to allow developers to use CATTO within IntelliJ IDEA and to execute CATTO just-in-time before committing changes to the repository. We conducted a preliminary evaluation of CATTO on seven open-source Java projects to evaluate the reduction of the test-suite size, the loss of fault-revealing TCs, and the loss of fault-detection capability. The results suggest that CATTO can be of help to developers when performing TCS. The video demo and the documentation of the tool is available at: https://catto-tool.github.io/**

***Index Terms*—Software testing, test case selection, regression testing**

I. INTRODUCTION

Regression testing is the activity that prevents the arise of errors that have already been corrected once they are reintroduced to a system that has been modified [1]. *Retest-all* is a naïve approach, which consists in re-running the entire test suite (TS) against the changed software version under test (SUT) [2]. The problem with Retest-all is that the re-execution of the entire TS might result in a time- and resource-consuming process, especially when the SUT and its TS grow in size and/or complexity. Moreover, there are Agile development practices, like test-driven development [3], which leverage continuous regression testing, requiring the developer to execute the TS several times during a development session. If the execution of the TS demands too much time or too many resources, the developer is likely not to execute regression tests as many times as the Agile development practice would require. To tackle this problem, researchers have devised several strategies, which can be grouped into three main groups: test suite minimization (or reduction), Test Case Selection (TCS), and test case prioritization [1]. Both test suite minimization and TCS strategies seek to reduce the size of the TS that will be re-executed against the SUT. To that end, test suite minimization strategies remove redundant/obsolete test cases (TCs), either temporarily or permanently, from the TS. On the other hand, TCS strategies temporarily remove TCs that are not modification-aware. In other words, TCS strategies build a temporarily TS by selecting a subset of TCs (from the original TS) that are relevant to the changes made to the SUT, so avoiding executing those TC that do not exercise the changed parts. Finally, TC prioritisation strategies concern the identification of an "ideal" ordering of TCs maximizing some desirable properties (e.g., early fault detection).

In this paper, we introduce CATTO (Commit Adaptive Tool for Test suite Optimization), a tool implementing a TCS strategy for SUTs written in Java that selects TCs to be re-executed by comparing the call graphs of the two versions of the SUT (i.e., the versions before and after some changes are made to the SUT). Unlike other TCS tools (e.g., SPIRITuS [4], Pythia [5], or TestTube [6]), CATTO determines a link between source code and corresponding test cases using a generated call graph (see Sect. III). To allow developers to execute CATTO just-in-time before committing changes to the repository, we implemented a plugin for IntelliJ IDEA [7] that wraps its functionalities.

CATTO was specifically developed with the intention to focus on fast feedback during the development cycle, preferring technologies that are not resource intensive, also accepting a lower accuracy of the tool. We conducted a preliminary evaluation of CATTO on seven open-source Java projects to evaluate the reduction of the test-suite size, the loss of fault-revealing TCs, and the loss of fault-detection capability.

II. CATTO COMPONENTS

CATTO consists of two main components: CATTO CORE and CATTO INTELLIJ IDEA. CATTO CORE contains the application logic, CATTO INTELLIJ IDEA is a plugin for IntelliJ IDEA to provide the features of CATTO CORE during development.

As shown in Fig. 1 (on the right hand side) CATTO CORE performs six sequential steps, described as follows.

1) **Dynamic classes loading**: the tool loads the classes of the current and previous version of the project and their

TABLE I
SELECTION CRITERIA DESCRIPTION.

Operation	Selected Test Case
Production Method	
– added or modified	The respective test methods
– deleted	Test methods that covered it
– deleted in a hierarchy	All tests methods covering the actual production methods in hierarchy
Constructor of a production class	
– added, modified or deleted	All test methods covering the production methods in that class
Static field of a production class	
– added, modified, or deleted	All test methods covering the production methods in that class
Test Method	
– added	The new test method
– modified	The modified test method
Static field, constructor or fixture method of a test class	
– modified	All test methods in that test class

dependencies. We rely on SOOT, a third-party component, to transform byte-code in an intermediate representation (called "JIMPLE" [8]), necessary to have a normalized representation of the code that is easier to compare.

2) **Instrumentation**: Once all classes are loaded, CATTO CORE dynamically creates a Java test class for each actual test class, adding to each test method a call to the fixture methods if they are declared in the class itself or in a parent class (e.g., *setUp()* and *tearDown()*). This step is required to ensure the creation of a reliable call graph.
3) **Call graph creation**: Once the fake test classes are created, CATTO CORE uses these as a starting point for the generation of the call graph of the current version of the project. The generation of the call graph starts from the test methods and goes through the production code, mapping in this way each production method to its corresponding test methods.
4) **Code Analysis**: Then, CATTO CORE analyzes the two versions of the project searching for code changes and marking all the methods and class modified. CATTO CORE marks a method or class as changed according to the operations described in Tab. I.
5) **Test Case Selection**: The methods marked in the previous step are used by CATTO CORE to select the test methods to execute. For each marked method, the corresponding test methods are selected. Tab. I describes all the criteria the tool adopts to perform TCS.
6) **Test Case Execution**: Finally, when the test methods are selected, CATTO CORE executes them and displays the results, as well as the error stack trace in case of failure.

The component CATTO INTELLIJ IDEA extends the functionality of CATTO CORE integrating it in IntelliJ IDEA. This integration not only provides a user interface of CATTO CORE within a development environment, but also (1) catches the commit event and prepares CATTO CORE to execute the analysis before the code is committed; (2) retrieves the previous version of the project (the version of the project at the time of the last executed commit); (3) builds the current version of the project; (4) runs CATTO CORE; (5) returns the test execution summary to the user and asks them whether to proceed with the commit or not; (6) saves the committed version of the project in a hidden folder for future analyses. Figure 1 (left hand side) shows the order in which these steps take place and how the CATTO CORE steps are integrated within the CATTO INTELLIJ IDEA to perform TCS.

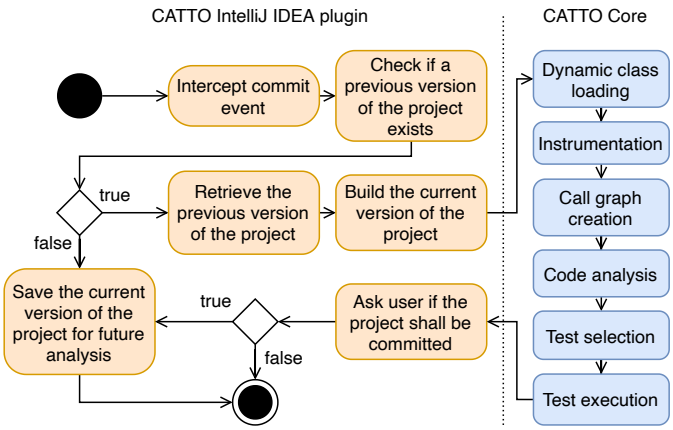


Fig. 1. Activity diagram of CATTO and the interaction between its parts

III. THE CATTO ARCHITECTURE

Figure 2 depicts the architecture of our tool in form of a UML component diagram also depicting the relevant packages. CATTO CORE relies on SOOT as a third-party component and CATTO INTELLIJ IDEA exploits the *IntelliJ IDEA Platform SDK API*. Due to space limitations, we discuss only the architecture of CATTO CORE, being it the core of the system.

CATTO CORE is composed by four main packages: $project$, $code.analyzer$, $test.selector$, $test.runner$.

The $project$ package manages a single version of the project and contains all the information about it (e.g., dependencies). It allows distinguishing between the current and the previous version of a project, setting up SOOT differently depending on the case, in particular creating the call graph only for the current version of the project.

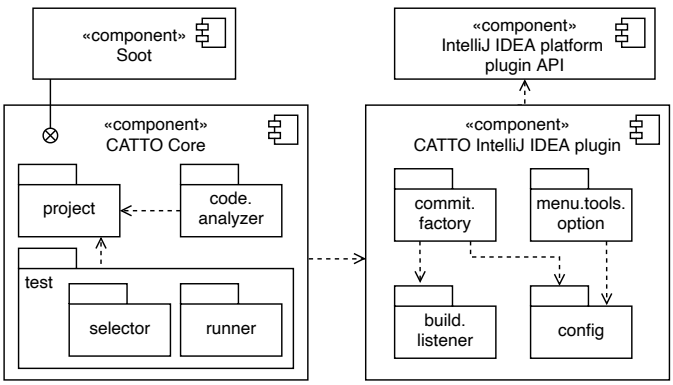


Fig. 2. CATTO architecture at the package level

The $code.analyzer$ package provides change analysis features between two different code versions. To allow a truthful comparison between two versions of the same method, CATTO checks the name of the package and the header of the method. When the package name and the header are equal, CATTO compares the body of the methods marking the method as modified if there are some changes in the body. When the header of the method and the package are different, the method is marked as new if does not belong to any class of the previous version, otherwise, it is marked as deleted. By using the byte-code and transforming this into a SOOT normalized intermediate representation, CATTO is also able to ignore cosmetic changes in the code.

The $test.selector$ package analyzes the call graph to find the tests to select. To minimize the required time to analyze the call graph, CATTO utilizes multi-threading and performs the search starting from the marked methods, going up the graph until a test method is found. For each test method, the corresponding sub-call graph is analyzed; the above mentioned search strategy allows the tool to analyze only those sub-call graphs containing a marked method.

The $test.runner$ package allows running the selected test methods. To allow Junit to launch the test methods, the project and its dependencies are loaded dynamically and added to the JAVA classpath. The use of JUnit [9] to execute the test methods ensures that all and only the necessary fixture methods (e.g., methods tagged with `@Before` or `@After`) are executed before and after the selected test methods.

IV. TYPICAL USAGE SCENARIO

This section describes how developers can use CATTO in a typical scenario. Lets suppose Alice, a developer, is adding a new feature to a system. After concluding the work, she will commit the changes. At this point, CATTO intercepts the commit event, selects and runs only the test methods that cover the newly implemented methods (see step (1) and (2) in Figure 3) and shows the results of the executed test methods in the console (see step (3) in Figure 3). If some tests fail, CATTO notifies the failure to Alice (see step (4) in Figure 3), thus, she can choose whether to commit anyway or to fix the issue(s) before proceeding with the commit.

Now suppose that Alice continuous her work and after a while somebody notices a bug in the new feature. Alice finds out what has to be changed, updates the code, adds testing code to cover it and commits the changes. CATTO, intercepts the commit, selects the new test methods and the test methods that cover the changes in the code, runs these, and, as before, shows the output in the console and notifies Alice about the results of the regression testing.

In summary, CATTO encourages Alice to practice continuous testing intercepting git [10] commits and visualizing tests results directly within the IDE.

V. TOOL VALIDATION

The validation was performed using seven Java open source systems (denoted below as S1–S7): *Apache Commons IO* [11], *Apache Commons Beanutils* [12], *Apache Commons Codec* [13], *DBCP* [14], *JXPath* [15], *JFreeChart* [16] and *JGap* [17].

We selected these systems because they are Java projects, are open-source, belong to different domains, and have non-trivial test suites.

The validation was performed using SMUG [18], a mutation generator tool. SMUG introduces mutation in the code considering the whole system, thus all the classes. We are aware that the choice of using synthetic changes may not reflect the real world. However, several studies in the past relied on artificial mutation to evaluate the performance of test suite optimization tools [4]. To simulate source code changes, for each of the seven considered projects, we created 30 mutated versions using SMUG (using the default configuration). While creating the mutated versions, SMUG executes the test methods tracking the failures. We validated CATTO by considering the original version as the previous version and the $nth-$mutated version as the current version. As an outcome, we expected that all the failed test methods would be selected. For each pair (original and mutated version) we calculated the following metrics, referring to the fault revealing selected test methods as X and the fault revealing test methods as Y:

- **Test Suite Reduction (TSR)**: the percentage of reduction of the test suite, calculated as the proportion between the number of test methods in the original test suite and the number of test method in the selected test suite. The higher the value, the smaller the selected test suite. Therefore, the most desirable value is 1.
- **Inclusiveness. (I)**: the capability to select all the fault-revealing TCs. $I = \frac{X}{Y},\ if\ Y \neq 0, 1\ if\ Y = 0$. The most desirable value is 1.

For each system we calculated the average value of each metric obtained comparing the original and each mutated version.

The results reported in Table II show that in two systems the mean of the TSR is more than 0.85, in one is more than 0.6, in two is more than 0.3 and in two is below 0.2. For the Inclusiveness, in all the systems the mean is more than 0.7, with two cases where is more than 0.95.

TABLE II
VALIDATION RESULTS

	S1	S2	S3	S4	S5	S6	S7
TSR	0.63	0.31	0.90	0.86	0.25	0.40	0.09
I	0.88	0.84	0.87	0.78	0.71	0.97	0.96

VI. RELATED WORK

Researchers have proposed several Test Case Selection (TCS) approaches based on various techniques. Integer programming [19], [20], data-flow analysis [21], [22], [23], [24], graph walking [25], [26], [2], [27], textual difference and information retrieval [4], [28], [5], modification detection [6], and firewall [29], [30] are just some of the techniques that the TCS approaches available in the literature rely on.

Fischer et al. [19], [20] introduced one of the earliest TCS approaches. This approach used integer programming to

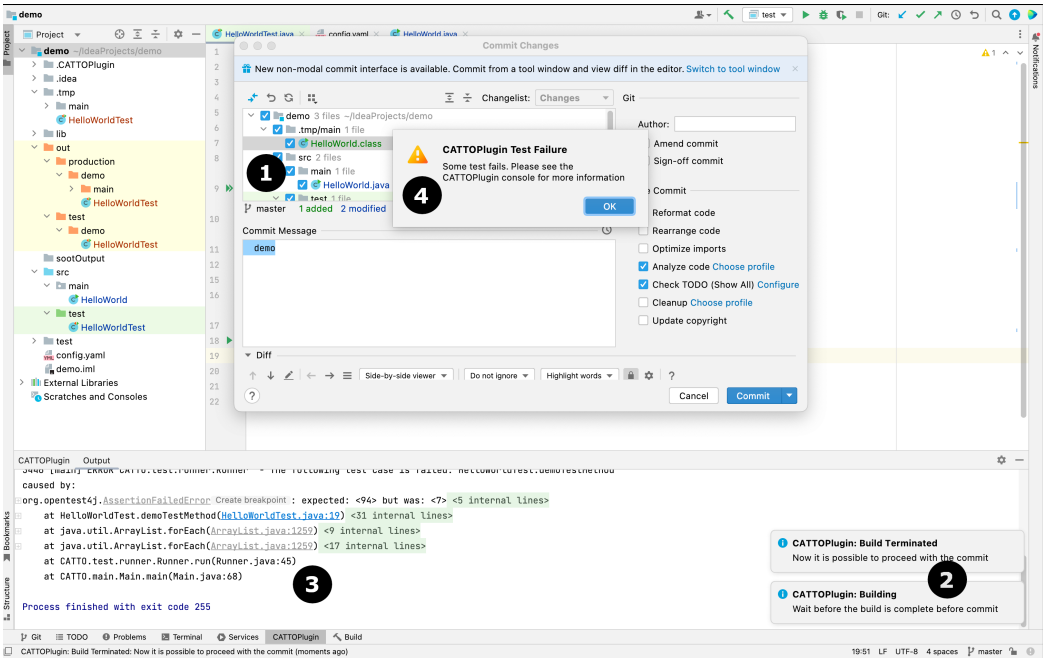


Fig. 3. Screenshot of an interaction with the CATTO INTELLIJ IDEA

represent the TCS problem in the context of Fortran code. Harrold and Soffa [21] applied data-flow analysis as a test case selection criterion. Similarly, Taha et al. [22] provided a test case selection framework based on an incremental data-flow analysis method. Gupta et al. [23] used program slicing techniques to find definition-use pairs that were impacted by a code change. Wong et al. [24] integrated a data-flow selection strategy with coverage-based minimisation and prioritisation.

Rothermel and Harrold proposed TCS approaches based on graph walking. First, they focused on graph walking of control dependence graphs [25]. Later, they extended their approach by using program dependence graphs for intra-procedural selection, and system dependence graphs for inter-procedural selection [26]. They further extended their work by relying on control flow [2] and inter-procedural control flow graphs [27].

Other approaches rely on the use of textual difference and information retrieval. Volkolos and Frankl [5], [28] presented Pythia, a TCS approach based on the textual difference between the source code of the two versions of the SUT. Romano et al. [4] presented SPIRITuS, an information retrieval-based TCS approach that uses method code coverage information and a vector space model to select test cases to be run. Chen et al. [6] introduced TestTube, a testing framework that selects test cases based on modification detection. TestTube tracks the execution of test cases to create links between test cases and the program components it exercises. Finally, Gligoric et al. [29], [30] proposed Ekstazi, a firewall-based approach that selects test cases based on the changes to their dependent files.

VII. CONCLUSION AND FUTURE WORK

In this paper, we present CATTO (Commit Adaptive Tool for Test suite Optimization), a tool implementing a TCS strategy for Java programs that selects TCs to be re-executed by comparing the call graphs of the two versions of the SUT (i.e., the versions before and after a change).

CATTO encourages the Agile practice of continuous testing intercepting git commits and providing tests results directly within the IDE. This relieves the developer of the burden of constant context switching between development and testing. In a Lean context, this tool represents the application of Jidoka, a concept used in Lean Manufacturing that suggests to "stop the line" (intended is the production line) in case a problem is detected [31]. Jidoka has the goal of automating quality assurance processes and to avoid unnecessary rework: it does this by stopping the entire production until it is clear how to fix an occurring problem. This strategy is not perfect for everyone, but in context where the cost of fixing an issue is higher than producing code, it is worth consideration. CATTO implements the concept of "stopping the line" in two ways: it automatically verifies (tests) the source code change (this cannot be deactivated) and—in case of a problem—warns the user. In the future, the strategies for regression testing will be extended—adding also test case prioritization, to increase the efficiency of the testing tool and to reduce testing time—as well as the way users are notified about the problem: from unobtrusive warnings to more aggressive strategy like to forbid a commit in case of a failing test case. Finally, we also plan to validate our tool via an in-vivo assessment involving real software developers.

REFERENCES

[1] S. Yoo and M. Harman, "Regression testing minimization, selection and prioritization: A survey," *Softw. Test. Verif. Reliab.*, vol. 22, no. 2, 2012.

[2] G. Rothermel and M. J. Harrold, "A safe, efficient regression test selection technique," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 6, no. 2, 1997.

[3] H. Erdogmus, G. Melnik, and R. Jeffries, "Test-driven development," in *Encyclopedia of Software Engineering*. Taylor & Francis, 2010.

[4] S. Romano, G. Scanniello, G. Antoniol, and A. Marchetto, "Spiritus: A simple information retrieval regression test selection approach," *Information and Software Technology*, vol. 99, 2018.

[5] F. I. Vokolos and P. G. Frankl, "Pythia: A regression test selection tool based on textual differencing," in *Reliability, quality and safety of software-intensive systems*. Springer, 1997.

[6] Y.-F. Chen, D. S. Rosenblum, and K.-P. Vo, "Testtube: A system for selective regression testing," in *International Conference on Software Engineering*. IEEE, 1994.

[7] JetBrains s.r.o., "IntelliJ IDEA," https://www.jetbrains.com/idea/.

[8] R. Vallee-rai and L. Hendren, "Jimple: Simplifying java bytecode for analyses and transformations," Sable Research Group, McGill University, Tech. Rep. Sable Technical Report 1998-4, 1998.

[9] JUnit contributors, "JUnit," https://junit.org.

[10] GIT contributors, "GIT," https://git-scm.com.

[11] The Apache Software Foundation., "Apache Commons IO," https://commons.apache.org/proper/commons-io.

[12] ——, "Commons BeanUtils," https://commons.apache.org/proper/commons-beanutils/.

[13] ——, "Apache Commons Codec," https://commons.apache.org/proper/commons-codec.

[14] ——, "The DBCP Component," https://commons.apache.org/proper/commons-dbcp.

[15] ——, "The JXPath Component," https://commons.apache.org/proper/commons-jxpath.

[16] D. Gilbert, "JFreeChart," https://www.jfree.org/jfreechart.

[17] K. Meffert, "JGAP," https://sourceforge.net/projects/jgap/.

[18] S. Romano and G. Scanniello, "Smug: a selective mutant generator tool," in *2017 IEEE/ACM 39th International Conference on Software Engineering Companion*, 2017.

[19] K. F. Fischer, "A test case selection method for the validation of software maintenance modifications," in *International Computer Software and Applications Conference*. IEEE, 1977.

[20] K. Fischer, F. Raji, and A. Chruscicki, "A methodology for retesting modified software," in *National Telecomm. Conference B-6-3*, 1981.

[21] M. J. Harrold and M. Souffa, "An incremental approach to unit testing during maintenance," in *Conf. on Softw. Maintenance*. IEEE Computer Society, 1988.

[22] A.-B. Taha, S. M. Thebaut, and S.-S. Liu, "An approach to software fault localization and revalidation based on incremental data flow analysis," in *International Computer Softw. & Applications Conf.* IEEE, 1989.

[23] R. Gupta, M. J. Harrold, and M. L. Soffa, "An approach to regression testing using slicing." in *Conf. on Softw. Maintenance*, vol. 92, 1992.

[24] W. E. Wong, J. R. Horgan, S. London, and H. Agrawal, "A study of effective regression testing in practice," in *International Symposium On Software Reliability Engineering*. IEEE, 1997.

[25] G. Rothermel and M. J. Harrold, "A safe, efficient algorithm for regression test selection," in *Conf. on Soft. Maintenance*. IEEE, 1993.

[26] ——, "Selecting tests and identifying test coverage requirements for modified software," in *Int. Symp. on Softw. testing and analysis*, 1994.

[27] G. Rothermel, M. J. Harrold, and J. Dedhia, "Regression test selection for c++ software," *Software Testing, Verification and Reliability.*, vol. 10, no. 2, 2000.

[28] F. I. Vokolos and P. G. Frankl, "Empirical evaluation of the textual differencing regression testing technique," in *International Conf. on Soft. Maintenance (Cat. No. 98CB36272)*. IEEE, 1998.

[29] M. Gligoric, L. Eloussi, and D. Marinov, "Ekstazi: Lightweight test selection," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 2, 2015.

[30] ——, "Practical regression test selection with dynamic file dependencies," in *Proceedings of the 2015 International Symposium on Software Testing and Analysis*. ACM, 2015.

[31] J. P. Womack and D. T. Jones, *Lean Thinking: Banish Waste and Create Wealth in Your Corporation*, 2nd ed. Free Press, 2003.