Automated Test-Case Generation by Cloning

Mathias Landhäußer, Walter F. Tichy
Karlsruhe Institute of Technology
Karlsruhe, Germany
landhaeusser@kit.edu, tichy@kit.edu

***Abstract*—Test cases are often similar. A preliminary study of eight open-source projects found that on average at least 8 % of all test cases are clones; the maximum found was 42 %. The clones are not identical with their originals – identifiers of classes, methods, attributes and sometimes even order of statements and assertions differ. But the test cases reuse testing logic and are needed for testing. They serve a purpose and cannot be eliminated.**

We present an approach that generates useful test clones automatically, thereby eliminating some of the "grunt" work of testing. An important advantage over existing automated test case generators is that the clones include the test oracle. Hence, a human decision maker is often not needed to determine whether the output of a test is correct.

The approach hinges on pairs of classes that provide analogous functionality, i.e., functions that are tested with the same logic. *TestCloner* transcribes tests involving analogous functions from one class to the other. Programmers merely need to indicate which methods are analogs. Automatic detection of analogs is currently under investigation. Preliminary results indicate a significant reduction in the number of "boiler-plate" tests that need to be written by hand. The transcribed tests do detect defects and can provide hints about missing functionality.

***Keywords*-Automatic testing; Software testing; Testing; Test oracle**

I. Introduction

Before embarking on a project to generate tests by cloning, it is important to have an estimate of the potential of this technique. To this end, we performed a preliminary study of nine open source projects in Java: A library for parsing command line arguments (args4j), a library for logging (log4j), and seven libraries from the Apache Commons project. These projects were analyzed for test cases that were potential clones, i.e., could have been generated by cloning and adaptation. We chose JPlag [1], a tool that is used for plagiarism detection, to find potential test clones. The advantage of JPlag is not only that it ignores differences in identifiers, comments, and layout, but that it is also insensitive to reorderings. Irrelevant reorderings of declarations, statements, or assertions occur in practice and should not lead to the rejection of a test from the set of potential clones. (Other useful clone detectors are compared by Roy et al. [2].)

Data collection proceeded as follows. For each project, we extracted the test files and compared them pairwise with JPlag. Test files typically include several tests, so the pairing and counting of tests had to be done manually. Since cloning can go in both directions, we were careful to count only one direction. Due to the amount of data, we only analyzed file pairs that showed a similarity threshold above 50 %. The likelihood of clonable tests in files with a similarity below 50 % is by no means zero. The results in Tab. I thus underestimate the true number of highly similar test cases.[1] A more thorough study of the extent of potential test case cloning should be conducted. Nevertheless, on average at least 8 % of all tests could be generated by a suitable cloning technique, sometimes significantly more. An important benefit of the cloning approach is that it also clones the test oracle. Hence, a manual decision of whether the output of a test case is correct is often not needed (for exceptions see below).

Table I
Analysis of Cloned Test Cases.

Project	Version	Tests Total	Potential Clones	%
args4j	v2.0.20 (2011-12-12)	95	40	42 %
log4j	v1.2.16	583	106	18 %
collections	v4.0 (2011-12-23)	1085	61	6 %
configuration	v1.9 (2012-02-05)	1481	75	5 %
email	v1.3 (2012-01-11)	110	6	5 %
io	v2.2 (2012-01-23)	757	28	4 %
lang3	v3.2 (2012-01-26)	2098	130	6 %
primitives	v1.1 (2011-03-31)	808	102	13 %
Totals/Average		**7017**	**548**	**8 %**

The concept of *analogous functionality* is central for test case cloning. We say that pairs of software components exhibit analogous functionality, if they share an abstract specification. For example, when testing set functionality, a typical test enters some data elements and then checks whether they are present. Set functionailty occurs in all container data types, including for example `JFrame`. `JFrame` is a Java class that displays a window on a computer screen, but it is also a container for display items. The test logic of entering elements and checking their presence applies to all containers. Similarly, when testing conversion functions, a conversion followed by its inverse should return the input.

[1]For the last project, "primitives", JPlag cut off the list of results because there were too many files with 100 % similarity; we expect the actual number of clonable tests to be even higher for this library.

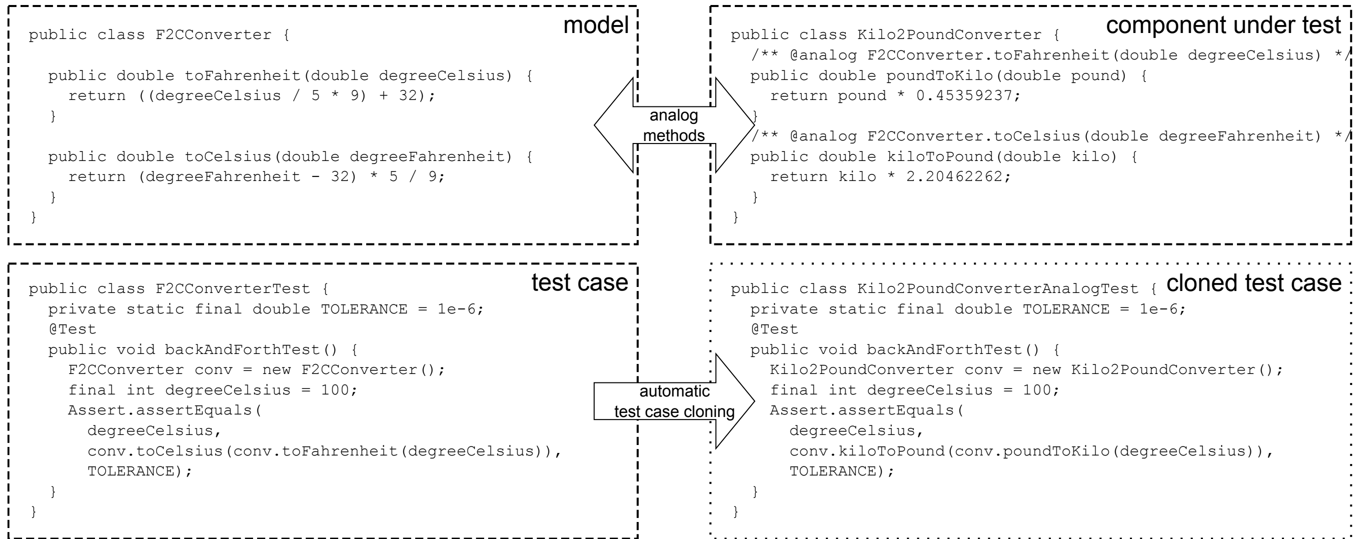

Figure 1. Adaption of Test Cases to the New Component Under Test.

This pattern could be transferred from, say, a Fahrenheit-to-Celsius converter to any other unit converter.

Testing by cloning means identifying analogous software components and adapting the test cases of one of the analogs (called the *model*) to be used on the other (the *component under test* or *CuT*). Consider the example in Fig. 1. There are two converters: one converts degrees Fahrenheit to Celsius and back, the other one pounds and kilograms. The weight conversion functions are annotated to indicate that they have analogs in the temperature converter. With this information, we can transcribe the test case of the temperature converter, including its built-in oracle, for the weight converter.

This approach is not limited to functions and their inverses. For example, the Java class `JMenu` has methods to add, remove, and enumerate menu items. The analogs for these methods can be found in the class `LinkedList`. If we have tests for linked lists, they can be transcribed automatically for `JMenu`. Of course, this test set is not sufficient, because it fails to test the special properties of menus. But it still reduces the tester's work.

Note that this technique also transfers the test oracle. It is well known that test oracles are difficult to construct automatically [3], [4]. Furthermore, our technique does not require algebraic specifications as in metamorphic testing [5], [6]. This approach can also reduce test writing in the following situation, often occurring in test-driven development: If one implements a class following a known interface, tests of a pre-existing implementation of the same interface can be applied to the new class.

Not all tests can be cloned without help from the programmer. For example, if we have a test that checks whether 120 degree Fahrenheit is converted to the right Celsius value, then this test does not apply to weight conversions immediately. But this test can still be re-used, if the programmer is willing to adapt the constants, which is less work then rewriting the entire test.

Test cases need to be transcribed, not merely copied. For instance, the identifiers of a model that appear in a test case need to replaced consistently with the corresponding CuT identifiers. If parameter lists differ, parameters need to be reordered or additional ones supplied. Parameter types may differ, which means that values of the correct types need to be generated for the tests.

Up to this point, we have assumed that programmers indicate the analogs. We are also researching techniques that find analogs automatically. So far, we have experimented with information retrieval techniques on identifiers. More sophisticated natural language processing techniques that analyze comments are currently under investigation.

II. Approach

This section describes the concepts underlying *TestCloner*, our prototype for automated testing by cloning. It works with the Java programming language, Javadoc documentation, and JUnit test cases. The following subsection explains the process of test case transcription given programmer-provided analogs. The second one discusses research into automatically finding suitable analogs.

A. Defining Model-CuT Relationships

TestCloner loads available classes using Javaparser [7] and generates an AST based on the Eclipse Modeling Framework [8]. The AST also records the analogs marked by `@analog` annotations. Any class that contains `@analog` links is considered a CuT, for which test cases should be generated. The class to which the annotation points is called

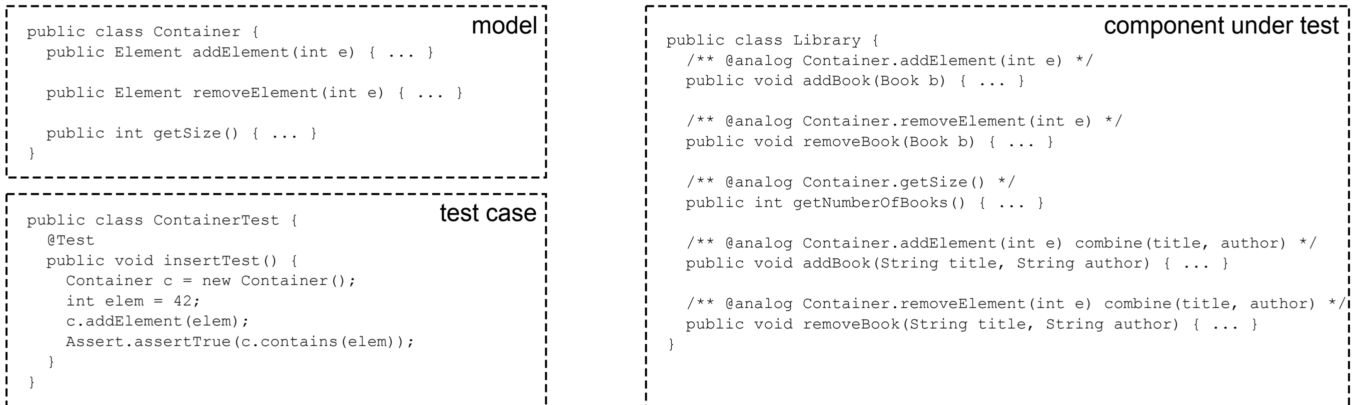

Figure 2. Library Example.

a model, since it provides test patterns. A CuT can have several models. A model can be a CuT itself with respect to a different model.

The mapping of a CuT method to its analog is annotated by providing the fully qualified name of the model class followed by the signature of the model method. Differences among CuT and model methods in terms of parameter number, type, and order must be treated specially (see below). Return types only matter if return values are actually used in test cases. If *TestCloner* cannot find a match for a given annotation, it emits a warning and proceeds with the next annotation.

A given CuT may provide the same model functionality more than once. For example, `MultiConverter` provides methods for converting temperatures among the scales Celsius, Fahrenheit, Kelvin, and Reaumur. Only consistent pairs of X-to-Y and Y-to-X converters should be mapped to the methods of `F2CConverter`. This aspect is taken into account by including group numbers in annotations. The group number indicates which CuT methods should be mapped to model methods as a set. Multiple setter/getter pairs are a frequent instance of this situation. Without groups, *TestCloner* cannot determine consistent mappings.

B. Test Case Transcription

Before the actual transcription process, *TestCloner* identifies applicable test cases. For all model classes, *TestCloner* loads the corresponding JUnit test cases and checks their applicability to CuTs. A model's test case is applicable to a CuT if the mapping is complete with respect to the test case; i.e. every member that is accessed in a test case must have an analog in the CuT. If not, *TestCloner* cannot clone the test case and proceeds with the next one.

If *TestCloner* identifies an applicable test case, *TestCloner* transcribes it so that the test uses the CuT instead of the model. Therefore it replaces model identifiers with CuT identifiers and performs parameter adaptations. After transcription, the applicable tests of a model class are provided to the programmer as a new class containing JUnit test cases which can be run immediately. If a cloned test fails, the programmer has to decide whether this a true failure of the CuT or whether the cloned test needs to be adapted or discarded. We identified the following issues when cloning test cases.

1) Transcribing parameters: Test data is typically provided as parameters. If the parameter types of CuT and model methods are compatible, *TestCloner* simply reuses the parameters from the model's test case. If not, *TestCloner* generates a new parameter. So far, *TestCloner* generates primitive types randomly and calls default constructors for objects. More sophisticated test data generators will be added later. In particular, we are investigating whether the setup of test fixtures can be transcribed.

2) Handling extra parameters: A related problem arises when a CuT method uses multiple parameters instead of a single one. An example is the pair of analogous functions `Container.add(element)` and `Library.addBook(title, author)`; see Fig. 2. The problem is that in the library class, the addition of a new book is not in terms of a single object, but instead of two parameters. (We took the class `Library` from reference [9] and modified it slightly for this example.) There are two ways to handle this situation. The programmer can provide a convenience method that takes a `Book` as a parameter and calls the original `addBook` method. The other technique is to indicate that both parameters should be treated as a single object using the `combine` annotation. *TestCloner* then produces two appropriate values wherever the model method is passed a single one. Both of these techniques are illustrated in Fig. 2; of course, only one is actually needed.

3) Handling duplicates: The transcription of parameters that are used more than once in a given test case also needs careful handling. For example, in Fig. 2 the test case uses the variable `elem` twice. If the developer declares a variable for a parameter and uses it more than once, the transcription

Table II
ANALYSIS OF CLONED TESTS.

Component under Test	Model	Analog Methods	Testcases					Test LOC Generated
			Model	Applicable	Generated	Invalid	Fault Revealing	
MultiConverter	F2CConverter	2*2	5	5	10	4	0	80 (181.8 %)
UniqueBoundedStack	BUStack	10	31	20	20	2	0	241 (51.9 %)
UniqueBoundedStack	Stack	8	23	12	12	2	0	139 (38.3 %)
Library	LinkedList	5	13	13	13	0	8	249 (51.2 %)
JMenu	GetterSetter	2	1	1	1	0	0	16 (100 %)
JMenu	LinkedList	8	31	21	21	0	4	405 (83.3 %)
BubbleSort	QuickSort	1	13	13	13	0	0	132 (100 %)
Totals		**38**	**117**	**85**	**90**	**12**	**8**	**1262 (63.4 %)**

process simply uses the variable in the generated test case. If we encounter a test case that uses different variables with equal content, the transcriber also must generate equal content for the new parameters. The same applies if the programmer passed the same literal constant multiple times.

4) Handling return values: As stated above, return types are ignored when analyzing analog mappings. However, when transcribing, return values become important. If a model method returns void, we can safely ignore any returned value of the analogous CuT method. If the return type of a model method is non-void, *TestCloner* needs to check whether the returned value is actually used in applicable test cases. If not, no further action is needed. If it is used elsewhere, e.g. as a parameter, *TestCloner* checks the compatibility of the returned type and the parameter's type. A similar analysis is needed if both return types are non-void and not identical.

5) Mapping attributes: Test cases may read and write attributes of the model class. If model and CuT provide those attributes, our test transcriber simply uses them. But what is to be done if the attributes are missing from the CuT? So far, we decided not to add analogy mappings for attributes but instead rely on (private) setter and getter methods. These methods allow flexible mappings, but they may have to be added by the programmer.

C. Software Maintenance

During maintenance, test cases of the model may change or additional ones may be added. The new test cases may simply be transcribed for the CuTs. The question is what to do with modified test clones. With some bookkeeping, the original clone could be found and replaced with the new one. However, this may not always be appropriate, because the specification of the methods involved may diverge. This is clearly an area for future research.

D. Test Coverage

Estimating the test coverage of cloned tests in a CuT is impossible. To achieve a prescribed coverage, programmers should use the usual tools and write additional tests until the desired coverage is reached.

Table III
ANALYSIS OF ANALOG DETECTION.

Component under Test	Model	Analog Detection	
		Precision	Recall
MultiConverter	F2CConverter	–	0 %
UniqueBoundedStack	BUStack	100 %	60 %
UniqueBoundedStack	Stack	100 %	62.5 %
Library	LinkedList	100 %	80 %
JMenu	GetterSetter	100 %	100 %
JMenu	LinkedList	100 %	62.5 %
BubbleSort	QuickSort	100 %	100 %
Totals		**100 %**	**67.6 %**

E. Finding Analogs Automatically

Up to now, we have assumed that programmers provide the analogs. Even with this assumption, *TestCloner* can save work. An interesting extension would be to detect analogs automatically. Automatic detection would reduce the testing load on programmers even further and potentially provide greater test coverage. We present several options for finding mappings between a CuT and existing models.

A potential source for analogies are method and parameter names, because analogous concepts are often expressed with the same vocabulary. For example, the method for adding items to JMenu is called `add`, the same as in many other container types. *TestCloner* presently uses a word similarity approach, because exact matches cannot be expected. This approach works as follows.

We assume that methods begin with verbs. *TestCloner* therefore isolates the verbs and retrieves their synonyms from WordNet [10]. This would yield, for instance, `append` as synonym for `add`. For every pair of CuT method and potential model method, a similarity score is computed. In particular, this score is high if both verbs appear in the same synonym set. Generally speaking, the score depends on how far two words are separated in the WordNet graph, for instance by Hypernym/Hyponym edges. Since not all methods are named by a single verb, we split identifiers, taking into account CamelCase, underscores, and other separators. The mapper then tries to identify a similar verb in the model and the CuT method and checks whether extra words appear as parameters or as part of the identifier

(as in `addBook(...)` and `add(Book b)`). The mapper also takes mapping groups into account. For example, if the `Library` contains a set of methods for adding and removing books and another set for adding and removing customers, the mapper creates a group for both subsets. Further improvements are expected when larger knowledge bases such as Cyc [11] and more sophisticated splitters (such as the one described by Enslen et al. [12]) are integrated in *TestCloner*.

At present, we are also investigating information retrieval methods applied to Javadoc comments and even the implementations of methods. More sophisticated NLP methods may be applicable. An advanced option is identifying idioms in program code. For instance, if a method adds a parameter value to an array or other container structure, then this may be a hint to test for set functionality.

III. Preliminary Results

To evaluate testing by cloning, we constructed a case study benchmark; see Tab. II. The left column lists the CuTs. The converter and sorter classes are based on textbook examples. `Library` and `UniqueBoundedStack` are taken from reference [9]. `JMenu` is from the Java Swing package. The column "Analog Methods" indicates how many methods were annotated as analogs. The "Testcases" column shows number of tests provided by the model, number of applicable tests, number of generated tests, and number of generated tests that fail. The following column provides the lines of test code generated by *TestCloner*. The percentage in parentheses is the reuse ratio, i.e. the ratio of generated to model test code. In three cases, the reuse ratio is perfect. In one case, the ratio is over 100%, because the same test code was reused multiple times. This is because the `MultiConverter` provides two groups of methods which we can map to the model (this is also indicated in the first column with the entry $2 * 2$), c.f. Section II-A. The reuse ratios for `UniqueBoundedStack` and `Library` are low, because the corresponding models inherit additional methods from `Container`, whereas the CuTs do not. Consequently, not all tests are applicable. Perhaps this is an indication that the missing methods should be added (i.e. a specification defect).

Cloned tests may fail for two reasons. First, they may fail because of a defect in the CuT. In this case, the clone revealed a defect. Test clones may also fail because they are invalid, i.e. they do not apply for the CuT or their oracle is incorrect. In this case, the cloned test needs to be corrected.

In the following we analyze instances where cloned tests failed, in order to illustrate test effectiveness. The failures in the converter class come from constants that do not carry over from one pair of units to another. Fixing the constants is sufficient. Tests of `UniqueBoundedStack` fail because they overflow the bound (which the models do not have). The only change necessary is the negation of the test oracles, because these cases are not failures: a bounded stack cannot store elements beyond the bound. Four tests concerning `Library` fail due to the insertion of `null` elements which are permitted in `LinkedList` but not in `Library`. The same is the case for `JMenu`. In both cases the programmer must decide whether to change the implementation to allow `null` or to simply negate the test oracles. Three further tests of `Library` fail because of actual defects in the code. A final one fails because the library counts books instead of copies of books. In total, four cloned tests revealed defects, eight identified a missing case (handling of `null`), and eight cloned tests needed to be adapted slightly (but did not reveal any defects). 70 tests succeeded.

Detection of analogies is still in a state of flux. We searched for matches of all CuT methods in all model classes. The precision and recall numbers are shown in Tab. III, but note that the number of models is small. The precision figures show that when matching verbs, all detected analogs were the right ones, but recall is not perfect. The converter classes have a recall of 0 % because their methods do not contain verbs. The name of the class would provide a good clue and will be used in the future.

IV. Conclusions

Preliminary results appear promising. Automated testing by cloning does detect defects. Although the method does not eliminate testing effort entirely, reusing test cases saves work, in particular work of the tedious kind. Much research remains to be done before this approach becomes practical. Transcribing the setup of test fixtures is an open problem. Additional benchmark examples need to be examined, which may lead to extensions of the transcription process. Data-flow analysis may make transcriptions safer. More sophisticated test patterns are also needed. For instance, can tests for design patterns such as Observer or Composite be written once and then transcribed for applications of the patterns? Finding models automatically would help eliminate annotations; an intermediate step would be to annotate analogies among classes instead of methods. Finally, a realistic evaluation on large benchmarks should be performed once the technology matures. This evaluation should answer the question whether this approach actually saves work.

References

[1] L. Prechelt, M. Philippsen, and G. Malpohl, "Finding plagiarisms among a set of programs with JPlag," *Journal of Universal Computer Science*, vol. 8, no. 11, pp. 1016–1038, 2002.

[2] C. K. Roy, J. R. Cordy, and R. Koschke, "Comparison and evaluation of code clone detection techniques and tools: A qualitative approach," *Science of Computer Programming – Special Issue on Program Comprehension (ICPC 2008)*, vol. 74, no. 7, pp. 470–495, 2009.

[3] W. E. Howden, *Software Testing and Validation Techniques*, 2nd ed. New York: IEEE Computer Society Press, Jun. 1981, ch. Introduction to the Theory of Testing, pp. 16–19.

[4] E. J. Weyuker, "On testing non-testable programs," *The Computer Journal*, vol. 25, no. 4, pp. 465–470, 1982.

[5] Z. Q. Zhou, D. H. Huang, T. H. Tse, Z. Yang, H. Huang, and T. Y. Chen, "Metamorphic testing and its applications," in *Proceedings of the 8th International Symposium on Future Software Technology (ISFST 2004)*, 2004, pp. 346–351.

[6] T. Y. Chen, T. H. Tse, and Z. Zhou, "Fault-based testing in the absence of an oracle," in *Int. Computer Software and Applications Conf.*, 2001, pp. 172–178.

[7] J. V. Gesser, "Javaparser – Java 1.5 Parser and AST," http://code.google.com/p/javaparser/, accessed: 03/21/2012.

[8] Eclipse Modeling Framework, http://eclipse.org/emf/, accessed: 03/21/2012.

[9] P. D. Stotts, M. Lindsey, and A. Antley, "An informal formal method for systematic JUnit test case generation." in *XP/Agile Universe*, ser. LNCS, D. Wells and L. A. Williams, Eds., vol. 2418. Springer, 2002, pp. 131–143.

[10] C. Fellbaum, Ed., *WordNet: An Electronic Lexical Database*. Cambridge, MA: MIT Press, 1998.

[11] Cycorp Inc., "ResearchCyc," http://research.cyc.com/, accessed: 03/21/2012.

[12] E. Enslen, E. Hill, L. Pollock, and K. Vijay-Shanker, "Mining source code to automatically split identifiers for software analysis," in *6th IEEE Int. Working Conf. on Mining Software Repositories, 2009, MSR '09*, May 2009, pp. 71–80.