Retrofitting Automatic Testing Through Library Tests Reusing

Lei Ma[*], Cheng Zhang[†], Bing Yu[‡], Jianjun Zhao[§]
[*] Harbin Institute of Technology [†] University of Waterloo [‡] Waseda University [§] Kyushu Univetsity
malei@hit.edu.cn c16zhang@uwaterloo.ca uhyou.yu@akane.waseda.jp zhao@ait.kyushu-u.ac.jp

***Abstract*—Test cases are useful for program comprehension. Developers often understand dynamic behavior of systems by running their test cases. As manual testing is expensive, automatic testing has been extensively studied to reduce the cost. However, without sufficient knowledge of the software under test, it is difficult for automated testing techniques to create effective test cases, especially for software that requires complex inputs.**

In this paper, we propose to reuse existing test cases from the libraries of software under test, to generate better test cases. We have the observation that, when developers start to test the target software, the test cases of its dependent libraries are often available. Therefore, we propose to perform program analysis on these artifacts to extract relevant code fragments to create test sequences. We further seed these sequences to a random test generator GRT to generate test cases for target software. The preliminary experiments show that the technique significantly improves the effectiveness of GRT. Our in-depth analysis reveals that several dependency metrics are good indicators of the potential benefits of applying our technique on specific programs and their libraries.

***Index Terms*—Automated test generation, software reuse, program analysis, random testing**

I. Introduction

Dynamic information, which exhibits during program executions, is useful for program comprehension. The cognitive study on program behavior by analyzing execution traces is a common approach to understanding the program [1]. As an important way to directly convey software information to human, visualization techniques represent program execution traces graphically, which is helpful to understand program dynamic behaviors [2], especially for distributed applications that contain multiple concurrent running processes and perform asynchronous communications over the network [2]. Analysis of execution traces is also heavily used by other program comprehension techniques, such as architecture reconstruction [3], [4], behavioral pattern property mining [5], [6], and feature location [7]. The effectiveness of these program comprehension techniques directly rely on the accuracy and comprehensiveness of the execution traces used.

In practice, test cases are often used to execute programs such that the parts of interest of the programs are analyzed for program comprehension. However, it is labor-intensive to manually create and maintain high quality test cases. To be useful for program comprehension, the test cases are expected to create diverse program states and cover many code areas of interest, and it is widely accepted that obtaining higher test coverage is essential to cover more diverse program states.

Over the last decades, automated testing techniques are extensively studied to abridge the gap between testing theory and practical tool automation. Nowadays, there are a few automated testing tools that can obtain reasonable code coverage on real-world applications [8], [9], [10]. However, it is still challenging for these tools to obtain high code coverage without using extra knowledge from the software under test (SUT), especially on programs that require complicated inputs or method invocation protocols, where understanding such input structures and invocation protocols is essential. Several studies [11], [12], [13] have shown that understanding the SUT and reusing relevant artifacts (e. g. code samples from public code bases) can provide useful information on how the software is used, which is helpful to improve code coverage of automatic testing. Unfortunately, these artifacts are not always available when the SUT is being tested. For example, for a newly developed program, which is to be thoroughly tested, its test suites are still under development, and there is probably no sample code in any public codebase.

In this paper, we propose a technique to mine extra knowledge from the libraries, on which the SUT is dependent, to improve the effectiveness of automatic testing on the SUT itself. Our key observation is that comprehensive test suites of the libraries are often available at the time when we start to test the SUT, and the library test suites often create input objects that are useful to generate test cases for the SUT. In our technique, we first perform static analysis on the SUT to discover all external types and identify those libraries used by the SUT. For each library, we analyze the source code of its test cases. More specifically, we perform a simplified variant of static program slicing to extract relevant statement sequences. Then, we transform the extracted sequences into well-formed methods, so that they can be compiled and executed. We integrate the generated methods into the testing process of the random test generator GRT [14] to enable GRT to create object states that are difficult to generate randomly. The evaluation on five real-world applications shows that our technique improves the average statement coverage and branch coverage by 7.0% and 9.3%, respectively. Considering GRT has been highly optimized [9], [15], the improvement brought by the extra knowledge extracted from library tests is noticeable. Our investigation also shows that simple metrics of the dependences between the SUT and its libraries can be main factors that influence the effectiveness of our technique.

II. Related work

Creating input objects with desirable states is essential to increase code coverage and bug detection ability of automatic testing techniques. However, it can be quite challenging when testing programs that require complicated input objects. Extensive research has been done on automatic software testing and several techniques show that reusing relevant knowledge of SUT is helpful to improve testing code coverage.

MSeqGen [11] mines the frequently used method sequence patterns from public codebases. It assumes that public codebases contain code usage samples of SUT, which often does not apply to programs that are under development without public releases. OCAT [12] and Palus [13] mine knowledge from the existing tests of the SUT itself to improve automatic testing. Palus first performs dynamic analysis on the provided test cases to train method sequence models, and then uses static analysis to identify method relevance based on field accesses. Both results are used to guide run-time test generation. OCAT adopts object capture-and-replay techniques, where object states are captured from running sampled test cases by serialization technique [16], and then used as input to support further testing. Both OCAT [12] and Palus [13] assume that the test cases of the SUT already exist. However, such test cases are often missing or incomplete during the development phase or at the early stage of testing.

Differing from existing techniques, our technique is based on the observation that test cases of libraries are often available when we test the SUT. Unlike sequence patterns and input objects extracted from SUT tests (and their executions), test cases of libraries do not directly provide knowledge on how the SUT is used. Our technique explores the possibility of leveraging such indirect knowledge (contained in library tests) to improve the effectiveness of automatic testing on the SUT.

III. Approach

Our technique consists of two main steps. In the first step, we extract useful statement sequences from existing test cases of libraries. Then, in the second step, we integrate the extracted sequences into the process of a random test generator to improve its effectiveness. Prior to these two steps, we analyze the code of the SUT to identify all the libraries which the SUT depends on and obtain the test cases of the libraries.

A. Extracting Statement Sequences from Library Tests

When we start testing an SUT, the libraries used by the SUT are often well tested and their test suites are generally available. These test suites contain test cases that cover various kinds of usage scenarios of the libraries. The tested scenarios probably include those used by the SUT, and they can create library class objects which are more useful than randomly generated ones. From library test cases, we extract statement sequences in the test code so that library objects can be generated by executing these sequences.

A main design decision of this step is to choose which library objects to extract. In general, every state-changing statement can result in one or more new object states. For example, an assignment to a field changes the state of the object containing that field and a method call probably changes the state of the receiver object or its argument object(s). However, many objects cannot be explicitly obtained or accessible, unless we instrument the test code to capture them at run-time [12]. In this preliminary study, we perform static analysis on library tests and focus on objects created by three kinds of statements that are commonly useful for reusing: 1) for each `new` statement, we try to obtain the object newly created; 2) for each method call whose return type is not `void`, we try to obtain the returned object; 3) for each method call whose return type is `void`, we try to obtain the receiver object if it exists. We do not capture all the objects used as method call arguments, to avoid extracting too many method sequences that may be less useful for reusing. In the example below, our technique captures all the three states of the object `list`, where the first object is newly created and the other two are from the method call on receiver objects.

```
List<Integer> list = new List<Integer>();
list.add(1);
list.add(2);
```

We analyze the test code to identify all relevant instances of the three kinds of statements whose extracted object types are compatible with a desired type in SUT. Every statement with a `new` statement, object-returning method call, or method call on a receiver object, is used as the *last* statement in a statement sequence to be extracted. Starting from the last statement s, we perform backward static program slicing to identify all the statements noted as $\{s_1, s_2, ..., s_n\}$ that are relevant to s. Then, we synthesize a new method, using the statement sequence $\{s_1, s_2, ..., s_n, s'\}$ as the method body and the type of the object extracted from s as the return type. The last statement s' is derived form of s. In cases that we need the method-return object, we directly add a `return` keyword for s. For example, if the original s is `new Foo(1)`, then we create s' as `return new Foo(1)`. In the case that we need the receiver object of s, we create s' by adding a return statement after s, to return the receiver object. For example, we add the statement `return list` at the end of the example above.

After generating the method, we add it into the enclosing test class of the test method where the statement sequence is extracted. In this way, we can avoid compilation errors, such as those caused by accesses to private fields of the test class.

We use static program slicing for sequence extraction, because we intend to conservatively include all statements that are necessary to create the object at the last statement. It is worth noting that we use a simplified variant of traditional slicing algorithm, in that, our algorithm only considers data dependences (ignoring control dependences). As reported in a previous study [17], most unit test cases have straight-line code and thus data dependences usually suffice to identify relevant statements. More importantly, we tend to avoid complex control structures in the extracted statement sequences, because long running sequences (e. g., infinite loops) usually result in useless objects or run-time exceptions which are harmful to automatic test generators.

B. Integration with Random Test Generator

After statement sequences are extracted from library tests and transformed into executable methods, we integrate them into the random test generator GRT, which is one of the state-of-the-art automatic test generators [15], [9]. GRT shares the common workflow of feedback-directed random testing [8] and guides each step of test generation through orchestrated program analysis. To test an SUT, GRT extracts all the target *methods under test* (MUTs) from the SUT and iteratively tests the MUTs by creating new test sequences and maintaining an *object pool* that stores those successfully executed sequences. In each testing iteration, GRT creates a new test sequence seq by concatenating a randomly selected method $m(T_1, T_2, \ldots, T_n) : T_r$ from the set of MUTs, and its corresponding inputs with types $T_1, T_2, \ldots, T_n$ from the object pool, to test m. Upon successful execution, GRT stores the sequence seq back to the object pool as inputs for further iterations of test generation. GRT incrementally generates more test sequences to test SUT until the time limit hits.

We enhance GRT to incorporate the statement sequences extracted from library tests. Although the sequences are in the form of executable methods, they may still rely on the `setUp()` methods in the original library test classes to set up the testing environment. Therefore, besides loading extracted methods into GRT through reflection, we also identify their dependent `setUp()` methods and concatenate them into new sequences, to ensure that the corresponding `setUp()` method is executed before each extracted method. Before the GRT main test iteration starts, we run all these sequences inside the GRT test execution framework to obtain the run-time object states of library classes. We only keep object states with successful execution that have no exceptions or errors. In this way, we convert the extracted sequences into the object states which can be used by GRT.

To obtain more relevant object states, we further perform purity analysis and use methods that have side effects to mutate the object states. To store the objects, we create a separate object pool in GRT and store all these newly created objects into the pool. At run-time, GRT randomly selects input objects from the new object pool as well as its original object pool. Using a separate pool generally avoids the interference of newly introduced objects on the main test generation iterations of GRT, as the pollution of the object pool (i. e., the pool is full of weakly related objects) can cause additional overhead of the testing procedure [13].

IV. Preliminary Evaluation

Using the enhanced version of GRT, we have performed a set of experiments to investigate two research questions:

RQ1: Is the proposed technique useful to improve code coverage of automatic testing techniques?

RQ2: What factors can affect the effectiveness of our technique?

The experiments used five popular programs selected from MVNRepoistory [18]. As shown in Table I, the programs vary in size, from 4K to 50K Non-commenting Lines of Code (NLOC). Columns 3 to 5 give the number of classes, instructions, and branches measured by JaCoCo [19]. Depending on their intended usage and design, the dependency of SUTs on their libraries varies greatly as shown in Table II. Column 3 shows the number of dependencies of SUT on its libraries. We count a dependency if an SUT class references a library class. Columns 4 and 5 measure the number of unique SUT classes using library classes, and the number of library classes used by SUT, respectively. For instance, in ASM-tree, there are 72 dependencies among SUT classes and libraries. Among the 43 SUT classes, 36 of them use 10 library classes collectively.

TABLE I
SUBJECT PROGRAMS USED IN THE STUDY.

Software (version)	NCLOC	#Class (*.class)	#Insn.	#Bran.
ASM-tree (5.1)	4,345	4,805	524	43
Apache Http-client (4.5.2)	19,772	53,158	4,913	463
Codehaus Wstx-asl (3.2.2)	36,621	80,238	12,086	248
Jackson-databind (2.7.3)	54,040	111,634	14,087	571
Joda-time (2.9.2)	28,624	65,327	6,580	246
Sum	143,402	310,357	38,190	1,571

In running GRT and its enhanced version, we allocated a global time budget of 3,000 seconds for each subject. To counteract the non-deterministic behavior of GRT, we ran each setting five times. All experiments were run on a computer cluster. Each cluster node ran a GNU/Linux system running Linux kernel 3.5.0 on a 16-core 1.4 GHz AMD 64-bit CPU with 48 GB of RAM, with Oracle's Java VM (JVM) version 1.7.0_65, using up to 4 GB for the JVM.

A. Code Coverage

Table II lists the averaged code coverage results. Columns 8 and 11 are the instruction and branch coverage of the original GRT. Columns 9 and 12 are the instruction and branch coverage of the proposed technique. Columns 10 and 13 are the relative improvement of the proposed technique compared with GRT. The data shows that overall our technique has positive effects, improving code coverage of GRT on four out of the five subjects (except Joda-time) statistically significantly (calculated by Mann-Whitney U Test, at $p < 0.05$ level) with coverage improvement from 2.4% to 24.3%.

On Joda-time, we cannot observe improvement, which is consistent with the data of sequence extraction from its library tests. From column 5 of Table II, we can see that Joda-time only uses two of its library classes. Columns 6 and 7 show that our technique does not extract any statement sequences with these two types. Therefore, the technique has no effects on GRT in testing Joda-time, and the negligible difference of coverage is caused by random factors.

In summary, our answer to *RQ1* is that, *in general, our technique is effective to improve the code coverage of automatic testing tools. Its effectiveness may vary with different subjects. In particular, it is ineffective if the library tests do not provide useful objects.*

TABLE II
DEPENDENCY METRICS OF STUDIED SUBJECTS AND EVALUATION RESULTS.

Software	#Class	#Depend. SUT and Lib	#Uniq. SUT Referencer	# Uniq. Lib Referencee	Mined Seq.		Instr. Coverage (%)			Branch Coverage (%)		
					Num.	#Types	GRT	Prop.	Impr.	GRT	Prop.	Impr.
ASM-tree	43	72	36	10	16	6	66.2	67.8	2.4	57.3	59.0	3.0
Apache Http-client	463	1,602	427	132	486	58	65.0	71.2	9.5	53.3	58.9	10.5
Codehaus Wstx-asl	248	359	123	53	305	33	54.2	63.3	16.8	41.6	51.7	24.3
Jackson-databind	571	1,072	399	105	554	21	58.0	61.7	6.4	43.1	46.6	8.1
Joda-time	246	37	24	2	0	0	86.7	86.8	0.1	76.4	76.7	0.4
Sum (Avg. for Cov.)	1,571	3,142	1,009	302	1,361	118	66.0	70.2	7.0	54.3	58.6	9.3

B. Discussion on Dependency Metrics

From Table II, we can see that the number of extracted sequences and their types from library tests (columns 6 and 7) can be a good indicator of the effectiveness of our technique. The more sequences with more required types we can extract from library tests, it is more probable that we can improve the automatic testing on SUT. The number of unique SUT referencers (column 4) is another factor to show how the SUT relies on its libraries. Given the same set of extracted sequences, the more SUT classes depending on libraries, the more probable reusing library tests would help. Moreover, the number of unique library referencees (column 5) describes how many unique library classes are used by SUT. Statement sequences in library tests that can generate these types are actually useful to the testing of SUT. Thus, if the test cases of libraries test these types more extensively, our technique is more likely to improve the code coverage.

In addition to metrics shown in Table II, the effectiveness of our technique could be affected by several other factors. On the SUT side, a potentially influential factor is whether or not the code branches that are difficult to cover by automatic tools are related to library objects. On the library side, the quality of library tests can be crucial. Since it is common that the development of libraries and their tests is oblivious to particular SUTs that may depend on them, we expect high quality library tests, which test most usage scenarios and have high code coverage, to be more useful to our technique.

V. CONCLUSION

Many program comprehension techniques perform analysis on execution traces to obtain cognitive information. Improving automatic testing to cover diverse program states can be very useful for program comprehension. This paper explores the usefulness of reusing library tests to improve the automatic testing. The results of our study on five real-world programs are positive and show that the dependency metrics of SUT and its libraries can be useful to indicate the effectiveness of our technique. Our future work includes (1) applying our technique to automatic testing tools based on symbolic execution, (2) extending our technique to other testing platforms [20] .

ACKNOWLEDGMENT

This work was supported by the Fundamental Research Funds for the Central Universities, and the National High-tech R&D Program of China (863 Program) 2015AA020101, 2015AA020108.

REFERENCES

[1] B. Cornelissen, A. Zaidman, A. van Deursen, L. Moonen, and R. Koschke, "A systematic survey of program comprehension through dynamic analysis," *IEEE Trans. on Softw. Eng.*, vol. 35, no. 5, pp. 684–702, 2009.

[2] L. C. Briand, Y. Labiche, and J. Leduc, "Toward the reverse engineering of uml sequence diagrams for distributed java software," *IEEE Trans. Softw. Eng.*, vol. 32, no. 9, pp. 642–663, Sep. 2006.

[3] C. Riva and J. V. Rodriguez, "Combining static and dynamic views for architecture reconstruction," in *Proc. of the 6h Eur. Conf. on Softw. Main. and Reeng.*, Washington, USA, 2002.

[4] C. Riva and Y. Yang, "Generation of architectural documentation using xml," in *Reverse Engineering, 2002. Proceedings. Ninth Working Conference on*, 2002, pp. 161–169.

[5] J. Yang, D. Evans, D. Bhardwaj, T. Bhat, and M. Das, "Perracotta: Mining temporal api rules from imperfect traces," in *Proc. of the 28th Intl. Conf. on Softw. Engg.*, Shanghai, China, 2006, pp. 282–291.

[6] D. Lo, S.-C. Khoo, and C. Liu, "Mining past-time temporal rules from execution traces," in *Proc. of the WODA*, Seattle,, 2008, pp. 50–56.

[7] N. Wilde and M. C. Scully, "Software reconnaissance: Mapping program features to code," *Journ. of Softw. Main.*, vol. 7, no. 1, pp. 49–62, 1995.

[8] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball, "Feedback-directed random test generation," in *Proc. of the 29th Int. Conf. on Softw. Eng. (ICSE)*, Minnesota, USA, 2007, pp. 75–84.

[9] L. Ma, C. Artho, C. Zhang, H. Sato, J. Gmeiner, and R. Ramler, "GRT: Program-analysis-guided random testing," in *IEEE/ACM Int. Conf. on Auto. Softw. Eng. (ASE'15)*, Lincoln, USA, 2015, pp. 212–223.

[10] G. Fraser and A. Arcuri, "Evosuite: Automatic test suite generation for object-oriented software," in *Proc. of the 19th ACM SIGSOFT ESEC/FSE*, Szeged, Hungary, 2011, pp. 416–419.

[11] S. Thummalapenta, T. Xie, N. Tillmann, J. de Halleux, and W. Schulte, "Mseqgen: Object-oriented unit-test generation via mining source code," in *Proc. of the 7th ESEC/FSE*, The Netherlands, 2009, pp. 193–202.

[12] H. Jaygarl, S. Kim, T. Xie, and C. K. Chang, "Ocat: Object capture-based automated testing," in *Proceedings of the 19th International Symposium on Software Testing and Analysis*, Trento, Italy, 2010, pp. 159–170.

[13] S. Zhang, D. Saff, Y. Bu, and M. D. Ernst, "Combined static and dynamic automated test generation," in *Proc. of the 2011 Intl. Symp. on Softw. Testing and Analysis*, Toronto, Canada, 2011, pp. 353–363.

[14] L. Ma, C. Artho, C. Zhang, H. Sato, J. Gmeiner, and R. Ramler, "GRT: An automated test generator using orchestrated program analysis," in *IEEE/ACM Int. Conf. on Auto. Soft. Eng.(ASE'15)*, 2015, pp. 842–847.

[15] L. Ma, C. Artho, C. Zhang, H. Sato, M. Hagiya, Y. Tanabe, and M. Yamamoto, "GRT at the SBST 2015 tool competition," in *The 8th Int. Workshop on SBST*, Florence, Italy, 2015, pp. 48–51.

[16] XStream, "http://x-stream.github.io/."

[17] K. Rubinov, "Automatically generating complex test cases from simple ones," Ph.D. dissertation, Università della Svizzera italiana, 2013.

[18] MVNRepository, "http://mvnrepository.com/popular."

[19] JaCoCo v0.6.4, "http://www.eclemma.org/jacoco/."

[20] B. Yu, L. Ma, and C. Zhang, "Incremental web application testing using page object," in *2015 IEEE International Workshop on Hot Topics in Web Systems and Technologies (HotWeb'15)*, 2015, pp. 1–6.