A Visualization Technique for the Passage Rates of Unit Testing and Static Checking with Caller-Callee Relationships

Yuko Muto, Kozo Okano, and Shinji Kusumoto
Graduate School of Information Science and Technology
Osaka University
Suita, Japan
{y-mutoh, okano, kusumoto}@ist.osaka-u.ac.jp

***Abstract*—Software visualization has attracted lots of attention. The techniques fall into two categories: visualization of software component relationships and visualization of software metrics. We propose a new hybrid method based on both of the two categories. The proposed method visualizes coincidence between specification and implementation from two aspects: static checking and ordinal testing by test suites. Each of the verification is performed in a method or function basis (unit testing). In our method, each ratio of the coincidence is shown by pie charts which represent classes of the target software. Whole software is represented in a weighted digraph structure. We have prototyped a tool implemented our proposed method. We have evaluated the availability of the proposed method by applying the tool to two kinds of software: Warehouse Management Program, and a telephone directory management program. As a result, we conclude that the proposed method shows informative results.**

Keywords-component; unit testing; static checking; ESC/Java2; software quality; visualization

I. INTRODUCTION

Recently, visualization techniques for software play more important roles according to increase of the size of software.

The techniques fall into two categories: visualization of software component relationship and visualization of software metrics. The former approaches [1] often show program flows as PDG (Program Dependency Graph). The latter approaches include visualization of temporal sequence of software metrics which helps analysis of software development aspects [2].

The granularity of the visualization target varies from code segments to objects, class files, or libraries [3]. For object oriented programs, class is one of suitable granularity. Paper [4] shows several relationships among classes.

Some papers [3] and [5] have proposed visualization methods for components of software. Paper [3] also summarizes that visualization is performed in several views: static views which show abstract structure of programs, and dynamic views which depict dynamic traces of programs. Recently, quality of software becomes important. Few papers, however, provide visualization of the quality of software. Our approach overcomes such weakness.

ISO defines quality of software [6], consisting of six properties, functionality, reliability, usability, efficiency, maintainability and portability.

The functionality is a kind of metrics which defines if given software satisfies required properties. It requires that the software must implement the requirements. The functionality can be measured by ordinary unit testing, static checking and model checking. Ordinary unit testing tests a given module and its specification if the module satisfies the specification using sufficient amount of test suites. Ordinary unit testing is usually performed as an early step of software tests. A major drawback of ordinary unit testing is that the quality of results of the test sometimes depends on the quality of test suites used. If the coverage of the test suites is low, then some of properties cannot be tested.

On the other hand, static checking and model checking approaches do not require executing the source codes. These approaches check statically via source codes (or abstract model of the source code, which models behavior of the source code).One of the famous tools of static checking is ESC/Java2 [7]. Its input is Java program annotated with JML (Java Modeling Language) [8], [9], in DbC manner. It checks if the (behavior of the) source code satisfies the property described in JML. The quality of output also depends on the property itself as well as that of standard libraries used for ESC/Java. Other drawback of ESC/Java2 is that it is not easy to understand relationships among classes because its outputs are text-based.

Therefore, a hybrid approach is considered. For example, paper [10] provides a method which generates test suites using counter examples generated by ESC/Java2.

In this paper, we propose a new hybrid method based on both of the two categories. The proposed method visualizes coincidence between specification and implementation from two aspects: ordinary testing (by test suites) and static checking. Each of the verification is performed in a method or function basis (unit testing). In our method, the ratios of the coincidence are shown by pie charts which represent classes of the target software. Whole software is represented in a weighted digraph structure.

978-0-7695-4429-8/11 $26.00 © 2011 IEEE
DOI 10.1109/ISPAW.2011.54

The prototype tool runs as a plug-in of Eclipse, a famous framework for integrated develop environment for software. We have evaluated the availability of the proposed method by applying the tool to two kinds of software: Warehouse Management Program and a telephone directory management program. As a result, we conclude that the proposed method shows informative results.

The paper organized as follows. Chapter II briefly provides related work. Chapter III provides some definitions of words as preliminary. Chapter IV will show our proposed method. We give an overview of our prototype tool in Chapter V, following experimental results and discussion in Chapter VI and VII. Finally Chapter VIII concludes the paper.

II. RELATED WORK

A. Visualization

GraphTrace [4] has proposed a visualization method for OOP, to understand dynamic behavior of the program. The target language is OO Lisp. It has structural and behavioral views, which show tree views of class inheritance and method call structures using the source code and runtime execution information.

Paper [4] provides some case-study examples showing that visualization is useful. One of drawbacks of the method is that it uses only source code information and execution information, thus other information such as test coverages, cannot be obtained. Therefore, the method can provide only what the program implementor intends. Such a drawback is common among methods based on analysis of only products.

B. Combining Ordinary Unit Testing and Static Checking

Check 'n' Crash [10] combines ordinary unit testing and static checking. It automatically identifies faults as the following flows. ESC/Java2 produces some counter examples. Then test suites are automatically produced based on the counterexample, which are used in unit testing to identify faults. It is effective in a sense that it produces only suitable test suites for suspect faults.

It does not care points where the test suites are not generated. ESC/Java2 is neither sound nor complete, thus such points might have some serious faults. Therefore, it will miss some possibility that runtime execution causes errors, such as memory fault due to small capacity of main memory.

The above work performs ordinary unit testing after static checking. Paper [11] provides the opposite way. Tests cannot find corner case bugs. The method in [11] firstly performs testing to the target and obtains its coverages. Secondly it performs static checking on the complement part of the coverages. Thus, the static checking can be applied to a limited area of the target source code; it gains scalability. It, however, misses the bugs which are passed by the tests but are detected by static checking. It might still fail to detect some corner case bugs. For example, even the branch coverage does care the combination of branch conditions; while corner case bugs may detect at some specific values of variables which are not tested.

```
01: class Main {
02:   public static void main(String[] args) {
03:    Person p = new Person();                    // call Person
04:    p.setFullName("John Smith");                // call Person
05:    System.out.println(p.getFamilyName()); // call Person
06:   }
07: }
08: class Person {
09:   private String fullName = "";
10:   public Person() {}
11:   /*@ public behavior
12:      requires nm != null && !nm.equals("");
13:      ensures fullName.equals(nm); @*/
14:   public void setFullName(String nm) {
15:     fullName = nm;
16:   }
17:   public String getFamilyName() {
18:     return fullName.split(" ")[1];
19:   }
20: }
```

Figure 1. Main class and Person class with JML

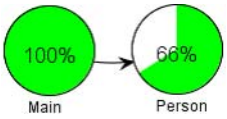

Figure 2. Visualization of static checking for Main class and Person class

III. PRELIMINARY

This chapter gives some definitions and explanation on unit testing and static checking.

A. Unit Testing

(Ordinary) unit testing is performed onto each module of given software. Conventionally the testing is performed by test suites. Famous metrics on unit testing includes statement coverage, branch coverage, condition coverage, and so on. These coverages are used for metrics for quality of testsuites themselves as well as that of results of unit testing.

JUnit is the de facto standard framework for unit testing. JCoverage [12] calculates some coverages including statement coverage. djUnit is a plug-in for Eclipse which exports coverage reports of JCoverage.

B. Static Checking

1) JML: JML (Java Modeling Language) [8], [9] is a specification language used for annotation into Java programming. Based on DbC (Design by Contract) [13], we can give assertions such as invariants, pre-conditions and post-conditions for a method.

Figure 1 shows an example of JML annotation. Person class has a *name* field and a setter method, *setName*. The fieldname must be always non-null, thus, the annotation of third line is given. Method *setName* has a pre-condition that *nm* is neither Null nor null String, thus, the annotation of fifth line is given. Also the ensure clause gives the post-condition which means that name has the same value to *nm*.

2) ESC/Java2: ESC/Java2 [14] is a static checking tool which verifies whether the source code satisfies the annotation described in JML for each method. In theory, neither soundness nor completeness is guaranteed, however it efficiently finds bugs in normal usage. It is one of the useful tools in the sense of a light-weight formal approach. It supports Java version1.4. Some of major libraries have been annotated in JML or built-in.

It requires a Java source code and outputs a result as text messages which describes passed or failed for each property and each method. If it reports failed, its counter-example also generated.

IV. OUR PROPOSED METHOD

This chapter describes our method.

A. Overview

Figure 2 visualizes the result of static checking for Figure 1. Caller-callee relation of the given target program is shown in a digraph, where each node and each edge represent a class and the caller-callee relation, respectively. Each node also represents a pie-chart which gives a passage rate of the corresponding class. The passage rate is evaluated based on unit testing and static checking. The weight of an edge corresponds to the number of method calls relating to the classes. We use caller-callee relation instead of class hierarchy relation used typically in class diagram, because in this paper, we focus on modular verification/testing, where properties of classes or methods and their relations are important. Of course, such a structure can be visualized using a similar way of ours.

B. Definition of Passage Rate Metrics

Here, we have to think the following four kinds of metrics: (1) metrics for the quality of the test suites, (2) metrics of the quality of assertions, (3) metrics for the results of ordinary unit testing, and (4) that for the results of static checking. In the paper, we focus on the metrics for (3) and (4) only. We discuss the metrics for (1) and (2) in the later.

1) Passage Rate for Results of Unit Testing: We adopt statement coverage as a passage rate of unit testing. The reason why is the following: statement coverage is simple and easy to calculate; the value of branch coverage generated by djUnit is different to the original value; and condition coverage is not supported by JCoverage.

2) Passage Rate for Results of Static Checking: Let Mpassed(A) and M(A) be the number of passed methods in a class A, and the total number of methods in a class A, respectively. The passage rate of static checking for the class A is defined as:

$$Cs(A) = Mpassed(A) / M(A)$$

We give an example for the metrics using Figure 1. From the output by ESC/Java2, we can infer that constructor and method *setFullName* are valid, however method *getFamilyName* is not valid. Therefore Mpassed(Person)=2 and M(Person)=3, respectively. Cs(Person) is calculated as 66%.

C. Definition of Caller-Callee Relation

If some method m1 is appeared in a method m2 as a method call statement, we say m2 calls m1. If a method in class A calls some method in class B, we say class A calls class B. Let nAB be the number of every call, such that class A calls class B. We say class A calls class B nAB times. The following explains the caller-callee relation and the number. In Figure 1, Main class calls constructor of Person class in line 3, *setFullName* method in line 4 and *getFamilyName* method in line5. Thus, Main class calls Person class three times.

V. IMPLEMENTATION

Here, we give simple descriptions on our prototyped tool. The tool is implemented as a plug-in of Eclipse. The size of the program is about 2000 LOC without comments, with 14 packages and 33 classes. The program is mainly written in Java 1.6, developed on Eclipse Galileo. We use PDE (Eclipse Plug-in Development Environment) in order to implement as a plug-in. We use some libraries MASU and JUNG as part of the tools. MASU provides general metrics measurement and program analysis library [1]. We use MASU in order to analyze caller-callee relation of the given program. JUNG, Java Universal Network/Graph Framework, is a graph visualization library [15]. We use it to draw the output digraph.

A. Input

The inputs of the tool are the directory of the target source codes, XML generating scripts, and location of XML files. The tool requires that the target source codes are written in Java version 1.4. The version restriction is due to the restriction of ESC/Java2. XML generating scripts are replaceable according to the metrics.

B. Views

Figure 3 is screenshot of the Tool. It has Main View for showing the digraph and Method View for showing detail method information.

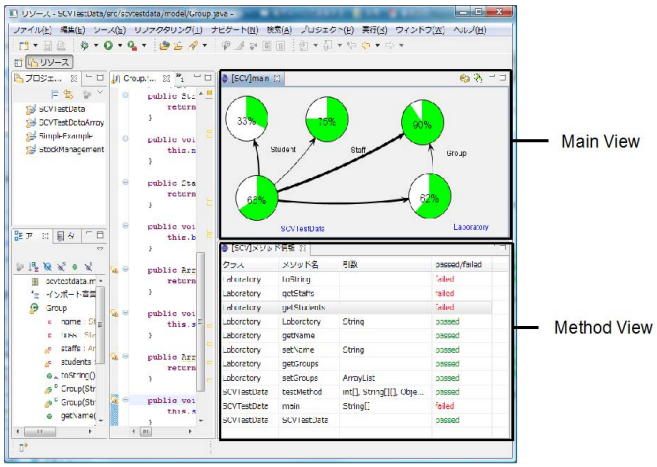

Figure 3. Screenshot

VI. EXPERIMENTS

In order to evaluate our proposed method, we apply the tool to two programs.

A. The Evaluation Approach

We apply our tool to the following two programs.

1) Targets: We use two programs, one is Warehouse Management Program, and the other one is Personal Telephone Directory.

Warehouse Management Program is implemented in Java1.4. The program has seven classes of about 400 LOC except JML annotations and test suites have seven classes of 200 LOC. The program and its JML annotations were written by an undergraduate student in order to verify the usefulness of JML annotations and ESC/Java2 in [16]. We have written its test suites to use them in this paper.

Personal Telephone Directory is also written in Java1.4. It has five classes of about 260 LOC except JML annotations and test suites have ten classes of 800 LOC. Its original program is an assignment for an undergraduate exercise. A member of teaching staff in our university wrote it and its test suites to reference. We reused the core of the program and test suites. In this paper, we added JML annotations to it.

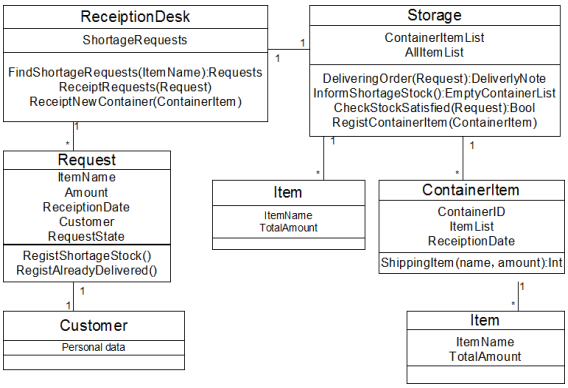

Figure 4. Class constitution of Warehouse Management Program

2) Condition: We reserve an assumption. Assumption 1: We assume that Warehouse Management Program has valid JML annotations with poor test suites, whereas Personal Telephone Directory has poor JML annotations with enough test suites.

In fact, the former assumption is guaranteed by Paper [16], and the latter has 800 LOC of test suites to 260 LOC of source codes.

B. Warehouse Management Program

Warehouse Management Program consists of seven classes: ContainerItem, Customer, Item, ReceptionDesk, Request, Storage and StockState. Figure 4 shows the UML diagram of the program.

Because the program already has JML annotation with checked, we just add test suites for the unit testing. The test suites only check constructors and setter/getter methods. Thus, the quality of the test suites is low. Though the Storage class has fields named containerlist and allitemlist and their getter methods, we didn't describe their test suites, because setter methods for the fields are not implemented in the class.

C. Personal Telephone Directory

Personal Telephone Directory has the following five classes: AddressBook, Entry, NameComparator, TelComparator and MailDomainComparator.

Personal Telephone Directory has enough test suites, thus, we regard that the program is valid from the view of unit testing. On the other hand, JML annotation is not given enough.

D. Results

Figures 5 and 6 show the digraphs represented unit testing and static checking, respectively for Warehouse Management Program. Figures 7 and 8 show the digraphs represented unit testing and static checking, respectively for Personal Telephone Directory.

E. Discussion

1) Unit Testing: Warehouse Management Program: In Figure *5*, thick arcs show that the source class calls many methods in the sink class. By observing the arcs, we can estimate the number of stabs needed to unit testing.

In general, every terminal node (class) has high values of passage rate. It shows that such a class tends to be a typical Java bean, thus they have only simple setter/getter methods.

2) Static Checking: Warehouse Management Program: Figure *6* shows that every class has high passage rate. Let's look at precisely the caller-callee relation and the result of static checking. For example class ReceptionDesk has passage rate of 100%. It seems that the class has perfect high quality and no problem. The class calls the following classes: Storage (87%), ContainerItem (88%), Request (75%), Customer (90%).

The value in parentheses shows the passage rate of the corresponding class. If class Request has some bugs, then it might affect the quality of ReceptionDesk. We must calculate the passage rate which includes the passage rate of caller classes.

3) Comparison between Unit testing and Static checking: Warehouse Management Program: Classes Customer, Request, Item and StockState have high passage rate in both of unit testing and static checking. These classes have codes satisfying their specification well. Thus the quality of the class is also high.

On the other hand, classes ReceptionDesk, Storage and ContainerItem are with low passage rate of unit testing while that high passage rate of static checking.

Thus, we can conclude that the unit testing is not enough performed. In fact, test suites for the classes are only those of setter/getter methods. Though the quality of unit testing is low, the classes have high quality because static checking is passed.

4) Comparison between Unit testing and Static checking: Personal Telephone Directory: We discuss the results in Figures *7* and *8*.

First, let's consider two classes AddressBook and Entry, both of which have high passage rates in unit testing and static checking. We conclude that these classes are in high quality.

Next, we consider classes with high passage rate in unit testing and low passage rate in static checking. Classes NameComparator and TelComparator are pertinent.

We conclude that the JML specification is too restrictive or ESC/Java2 cannot enough prove the correctness of given assertion. These classes implement java.util.Comparator interface. Though the library used in ESC/Java2 includes annotation of java.util.Comparator, the annotations are very general and weak. Moreover, neither NameComparator nor TelComparator does have adequate annotation. Thus, the quality of static checking results is low. We also conclude that the quality of these classes is high due to the passage rate of unit testing.

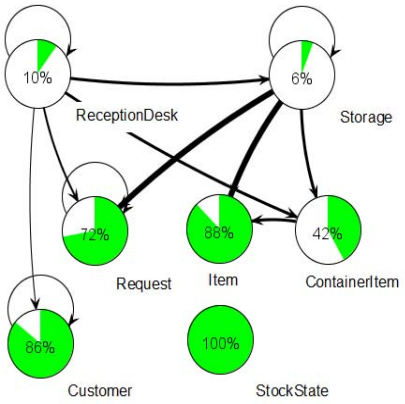

Figure 5. Result of unit testing in Warehouse Management Program

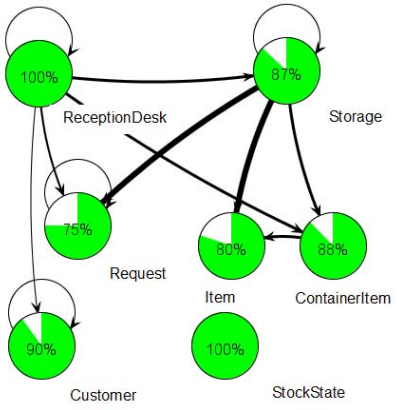

Figure 6. Result of static checking in Warehouse Management Program

F. Threats to Validity

Here, we simply summarize threats to validity. As external threats to validity, we can enumerate the following items: 1. The size of the target programs is not so large, 2. The categories of the target programs are the same, and 3. The correctness of JML specification itself is not tested enough. For 1 and 2, to handle large size programs in huge range of categories, we need more programs with JML annotations. Today, Java programs with JML are not popular; it is not easy task. Several researches provide methods automatically produce JML annotations [17]. Such techniques might help to resolve the problem. For 3, we have already discussed it.

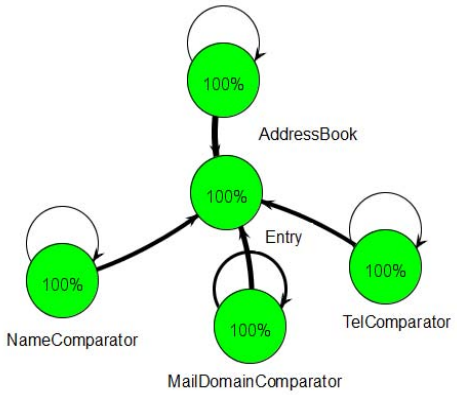

Figure 7. Result of unit testing in Personal Telephone Directory

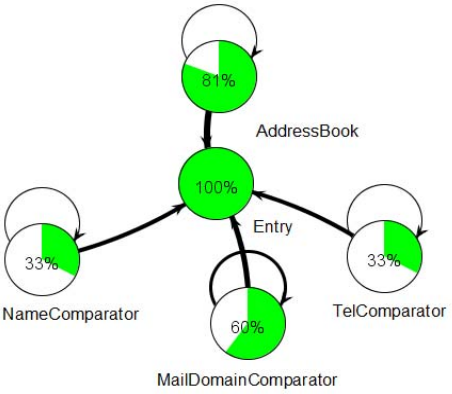

Figure 8. Result of static checking in Personal Telephone Directory

VII. DISCUSSION

Here, we discuss two reminder metrics in chapter IV.

A. Passage Rate Metrics

1) Metrics for the quality of the JML assertions: We need metrics to specify the quality of given JML statements. We have researched past papers, however, we find no suitable existing coverage or metrics for JML. Thus, we devise a new metric, called Variable Coverage. In general, assertions are conditions on program variables. For example, pre-condition and post-condition assert that parameters and return value (and/or some field variables) of the method meet the conditions, respectively. In a similar way, Class Invariant asserts invariant conditions for field variables during the

object is alive. Hence, we can regard the ratio of variables with its condition as coverage. We consider Variable Coverage for a method as a metric on its parameters, return value and related field variables. Variable Coverage consists of Parameter Coverage, Return Value Coverage and Field Variables Coverage. These coverages are used in combination. For example, for a typical post-condition, Return Value Coverage and Field Variable Coverage are used.

Parameter Coverage is the ratio by the number of used parameters in the pre-condition to that of all parameters.

Return Value Coverage means whether post-condition holds return value or not. The result must be 0% or 100%.

Field Variable Coverage is the ratio by the number of used field variables in conditions to that of all field variables. Field variables are classified into mutable and immutable in the method. If a variable must change, post-condition would use the variable. For the other variables, Pure or Invariant should hold them.

2) Metrics for the quality of the test suites: Unfortunately JCoverage measures only passed statements when it calculates the statement coverage. Thus, the result of the statement coverage by JCoverage contains both aspects of the quality of test cases and the quality of testing result. In order to measure purely the quality of the test suites, we can use other coverage tool such as Open Code Coverage Framework[18].

VIII. CONCLUSION

This paper proposed a visualization method for software quality from multiple aspects. We developed a prototype tool of our method as a plug-in of Eclipse, and performed evaluation through some examples. The results show that we can evaluate the quality of software in more details by the proposed method. Additionally, in a preliminary experiment we had, some examinees said "This visualization method is more effective than reading program only or viewing simple table in order to find bugs".

Future work includes researching and evaluating what we described in chapter VII, quality of the test suites and JML. Visualizing based on other kind of structure such as a class diagram is also considered. Furthermore, we will try to find bugs automatically using the passage rate and caller-callee relationships.

ACKNOWLEDGMENT

This work is being conducted as a part of Stage Project, the Development of Next Generation IT Infrastructure, supported by Ministry of Education, Culture, Sports, Science and Technology, as well as Grant-in-Aid for Scientific Research C (21500036).

We also thank to Dr. Ishio for providing Personal Telephone Directory program and its test suites.

REFERENCES

[1] T. Miyake, Y. Higo, S. Kusumoto, and K. Inoue, "Masu: A metrics measurement framework for multiple programing languages [in japanese]," The IEICE transactions on information and systems (Japanese edetion). D, vol. 92, no. 9, pp. 1518–1531, 2009.

[2] S. Morisaki and K. Matsumoto, "Toward optimized collection and visualization of software metrics for progress sharing in offshore software development project," In Proc. of the 2nd Workshop on Accountability and Traceability in Global Software Engineering (ATGSE2008), pp. 3–4, 2008.

[3] W. Lowe, M. Ericsson, J. Lundberg, and T. Panas, "Software comprehension - integrating program analysis and software visualization," 2002.

[4] M. F. Kleyn and P. C. Gingrich, "Graphtrace–understanding objectoriented systems using concurrently animated views," in OOPSLA '88: Conference proceedings on Object-oriented programming systems, languages and applications. New York, NY, USA: ACM, 1988, pp. 191–205.

[5] A. Gonzalez, R. Theron, A. Telea, and F. J. Garcia, "Combined visualization of structural and metric information for software evolution analysis," in IWPSE-Evol '09: Proceedings of the joint international and annual ERCIM workshops on Principles of software evolution (IWPSE) and software evolution (Evol) workshops. New York, NY, USA: ACM, 2009, pp. 25–30.

[6] ISO, "Software engineering-product quality-part 1 : Quality model," ISO/IEC : 9126-1:2001, 2001.

[7] P. Chalin, "Early detection of jml specification errors using esc/java2," SAVCBS '06: Proceedings of the 2006 conference on Specification and verification of component-based systems, pp. 25 32, 2006.

[8] C. Yoonsik and P. Ashaveena, "Specifying and checking method call sequences of java programs," Software Quality Journal, vol. 15, no. 1, pp. 7–25, March 2007.

[9] L. Burdy, M. Huisman, and M. Pavlova, "Preliminary design of bml: A behavioral interface specification language for java bytecode," In Fundamental Approaches to Software Engineering (FASE 2007), pp. 215–229, 2007.

[10] C. Csallner and Y. Smaragdakis, "Check 'n' crash: combining static checking and testing," in ICSE '05: Proceedings of the 27th international conference on Software engineering. New York, NY, USA: ACM, 2005, pp. 422–431.

[11] V. Vipindeep and P. Jalote, "Efficient static analysis with path pruning using coverage data," in WODA '05: Proceedings of the third international workshop on Dynamic analysis. New York, NY, USA: ACM, 2005, pp. 1–6.

[12] jcoverage ltd., "Jcoverage," http://www.jcoverage.com/.

[13] B. Meyer, Object-oriented software construction (2nd ed.). Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1997.

[14] C. Flabagan, K. Rustan, M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata, "Extended static checking for java," Proc. of the ACM SIGPLAN 2002, pp. 234–245, 2002.

[15] JO'Madadhain, D. Fisher, S. White, and Y. Boey, "The jung (java universal network/graph) framework," UC Irvine Information and Computer Science, Tech. Rep., 2003.

[16] M. Owashi, K. Okano, and S. Kusumoto, "Design of warehouse management program in jml and verification with esc/java2 [in japanese]," The IEICE transactions on information and systems (Japanese edetion) D, vol. 91, no. 11, pp. 2719–2720, 2008.

[17] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin, "Dynamically discovering likely program invariants to support program evolution," IEEE Transactions on Software Engineering, vol. 27, pp. 99–123, 2001.

[18] K. Sakamoto, H. Washizaki, and Y. Fukazawa, "Open code coverage framework: A consistent and flexible framework for measuring test coverage supporting multiple programming languages," in Proceedings of the 2010 10th International Conference on Quality Software, ser. QSIC '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 262–26.