Improvement of a Visualization Technique for the Passage Rate of Unit Testing and Static Checking and its Evaluation

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 have already proposed a 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 the 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.

In this paper, we propose Priority Layout to emphasize important classes , and implemented our method into a tool. We have evaluated time in finding bug at source code and test cases between using Priority Layout, ISOM Layout and uncomplicated tables instead of graphs. As a result, time in finding bug at source code and test cases by proposed graph are a half of it using table.

***Keywords*-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.

ISO defines quality of software [5] including functionality which means if given software satisfies required properties. The functionality can be measured by ordinary unit testing, static checking and model checking. Ordinary unit testing tests for 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 code. These approaches check statically via source code (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[6]. Its input is Java program annotated with JML (Java Modeling Language) [7], [8], in DbC manner. It checks if the (behaviour 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/Java2. Other drawback of ESC/Java2 is that it is not easy to understand its outputs for especially novices because outputs are text based. Relationships among classes should be visualized.

Therefore, a hybrid approach is considered. In paper[9], we have proposed a hybrid method based on both of the two categories. The 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 the 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. The prototype tool runs as a plug-in of Eclipse. 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.

GraphTrace[4] has proposed a visualization method for OOP, to understand dynamic behavior of the program. The target language is OO LispD. 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.

Some papers [3] and [10] 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 a weakness.

Check'n'Crash[11] 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 counter example, 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.

The above work performs ordinary unit testing after static checking. Paper [12] provides the opposite way. Tests cannot find corner case bugs. The method in [12] 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 could be detected by static checking alone.

In Paper [13], authors provide and evaluate quality of contracts by means of mutation testing. From their observation, it is important to support developers by tool such as providing quality of assertion because there are some differences among individuals. They develop a tool which generate mutants from test target program for Java and C. They defined mutation coverage depending on lower and upper bound of completeness of JML assertions to evaluate contracts quality. The upper bound is the number of detected mutants divided by the number of non-equivalent mutants, and the lower bound is the number of detected mutants divided by the number of all mutants. They evaluate quality of contracts in the following process: generating random test cases, executing the test cases into target program, generating meta-mutants by their tool and executing the test cases into them, computing upper and lower bound from difference in results between the target program and mutants.

Comparing with using mutations, our approach has an advantage in execution time. Our approach needs just an execution of unit testing and static checking without advanced preparing such as generating mutants.

In this paper, we propose Priority Layout to emphasize important classes, and implemented our method into a tool. We have evaluated time in finding bug at source code and test cases between using Priority Layout, ISOM Layout and uncomplicated tables instead of graphs. As a result, time in finding bug at source code and test cases by proposed graphs are a half of using table.

The paper is organized as follows. Section II provides the definitions of some words as preliminary. Section III will show our proposed method. We give an overview of our prototype tool in Section IV, followed by experiments and discussion in Section V and VI. Finally Section VII concludes the paper.

II. Preliminary

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

A. Unit Testing

(Ordinary) unit testing is performed onto each module of a given software. Conventionally the testing is performed by test suites.

Well-known metrics on unit testing includes statement coverage, branch coverage, condition coverage, and so on. These coverages are used for metrics for quality of test suites themselves as well as that of results of unit testing.

JUnit, JCoverage, djUnit and others are tools for unit testing. JUnit is the de facto standard framework for unit testing for Java. Its test case codes are described in TestCase class; it executes the class in order to perform unit testing. The tool indicates whether a test method passes or fails, i.e., whether the test target method called by the test method makes expected behavior or not. JCoverage[14] enumerates coverages. It does not depend on specific tools such as JUnit. It creates coverage reports on specified metrics and granularity. djUnit is an assistant tool to integrate JUnit and JCoverage as eclipse plugin. It requires test cases like JUnit, and exports coverage reports according to JCoverage [1].

```
01: class Main {
02:   public static void main(String[] args) {
03:     Person p = new Person();
04:     p.setFullName("John Smith");
05:     System.out.println(p.getFamilyName());
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: }
```

Fig. 1. Main class and Person class with JML

B. Static Checking

JML (Java Modeling Language)[7], [8] is a specification language used for annotation into Java programming. Based on DbC (Design by Contract)[15], 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 $fullName$ field and a setter method, $setFullName$. Method $setFullName$ has a pre-condition that nm is neither Null nor null String, thus, the annotation of line 12 is given. Also the ensure clause at line 13 gives the post-condition which means that $fullName$ has the same value to nm.

ESC/Java2[16] 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 version 1.4.

[1]However, it uses modified version of JCoverage; the value of the report sometimes differs from the original one.

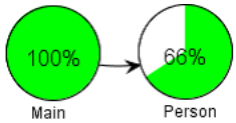

Fig. 2. Visualization of static checking for Main class and Person class

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 is also generated.

III. Our Proposed Method

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 represents 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. Additionally, we defined Priority Layout to help find important classes.

A. Definition of Passage 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 contracts, (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 (1),(3) and (4). We discuss the metrics for (2) later.

1) Passage Rate for Results of Unit Testing: We adopt statement coverage as a coverage 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 from the original value; and condition coverage is not supported by JCoverage.

Our approach utilizes djUnit because it covers both of (1) and (3). The coverage report by djUnit contains only code coverage data without acceptance of test methods, however, developers can recognize the acceptance of test method because JUnit view attached by djUnit provides that information.

2) Passage Rate for Results of Static Checking: Let $M_{passed}(A)$ 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 coverage of static checking for the class A is defined as:

$$C_s(A) = M_{passed}(A)/M(A). \quad (1)$$

We give an example for the metrics using Figure 2. From the output by ESC/Java2, we can infer that constructor and the method $setFullName$ are valid, however method $getFamilyName$ is not valid. Therefore $M_{passed}(Person) = 2$ and $M(Person) = 3$, respectively. $C_s(Person)$ is calculated as 0.66.

B. Definition of Caller-Callee Relation

If some method m_1 appears statically in a method m_2 as a method call statement , we say m_2 calls m_1. If a method in class A calls some method in class B, we say class A calls class B. Let n_{AB} be the number of every call, such that class A calls class B. We say class A calls class B n_{AB} times.

In Figure 1, Main class calls constructor of Person class in line 3, $setFullName$ method in line 4 and $getFamilyName$ method in line 5. Thus, Main class calls Person class three times.

C. Priority Layout

We propose Priority Layout to emphasize important classes. Figure 3 demonstrates Priority Layout. Some graph layouts are proposed to avoid overlapping nodes and edges, such as Circle Layout and ISOM Layout[17]. ISOM Layout is effective for dense graphs because it is based on a competitive learning strategy. Our approach requires a layout based on characteristics of classes or caller-callee relationships because the more important classes should be attracted more.

1) Importance: Let $G = (V, E)$ be a graph, where V and E denote sets of nodes and edges, respectively. Edge e_{ij} denotes an edge from node v_i to v_j. Let w be weight of an edge. $w(e_{ij}) = 1$ if caller-callee relationship of from node v_i to v_j exists. Otherwise, we let $w(e_{ij})$ be 0. $IN(v_i)$ is defined as a group of directed edges end with v_i. Weight of node v_i is defined as the sum of a weight of a directed edge e_{ki} end with v_i. Importance is the weight of node. Also, importance of v_i equals the number of callees which are called by v_i.

Definition 3.1 (Importance):

$$w(v_i) = \sum_{e_{ki} \in IN(v_i)} w(e_{ki}) \quad (2)$$

2) Layout: Classes with higher importance are allocated in the more superior region of the graph. When classes have the same importance, they are in a row at regular intervals.

IV. Implementation

Here, we give simple descriptions on our prototyped tool. Please refer [9] for details. 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.

Figure 4 shows the screen shot of the tool. The tool has Main View and Method View. Main View displays the digraph. Here, a node corresponds to a class. The coverage is displayed as a pie chart. If some class calls some other class many times, the corresponding edge is drawn in thick style. Method View displays detail method information on the selected class at Main View.

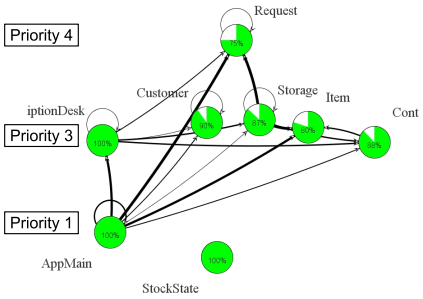

Fig. 3. Example for Priority Layout

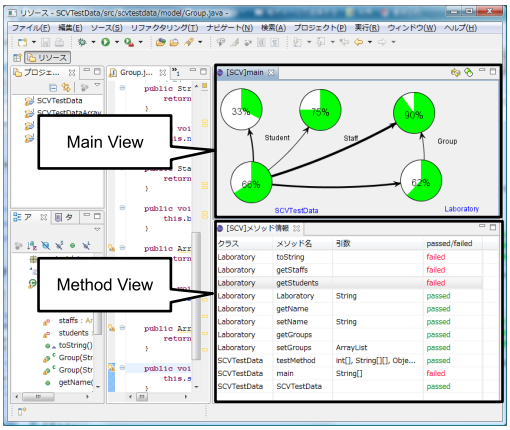

Fig. 4. Screen shot

V. Experiments

In order to evaluate our proposed method, we asked ten examinees to use our tool.

A. GQM approach

We adopt GQM approach[18] to evaluate our tool. Figure 5 shows the result of applying GQM. From the result we choose two metrics:

- Response time to find bugs at source code
- Response time to find defects of test cases

In addition, questionnaire includes:

Q1 How much do you think comparing two graphs helps you find bug?

Q2 Comparing between Priority Layout and others(ISOM Layout or table), which do you like?

Examinees must answer the questions in one to five point. For instance, for Q1, answering score 1 means that our approach never help finding a bug, answering score 5 means it can help finding a bug. For Q2, answering score 1 means that Priority Layout is worse than ISOM Layout or simple tables, answering score 5 means that Priority Layout is better than others.

B. Methodology

We experimented in the following process. First, we explained how to use our tool to examinees and they practice exercise by our tool. After that, they try to find bugs in source code and its test cases for two problems and answer the time to solve them and some questionnaires.

The below enumeration details the way to find bugs using our tool.

1) Display unit testing result into view.
2) Display static checking result into view.
3) Find buggy method comparing unit testing and static checking
4) Reason why the method is failed in static checking by reading specification
5) Find shortage line of test cases

There are ten examinees, who are members of our laboratory studying software engineering. They consist of a Ph.D candidate, six master students and three undergraduates. They understand Java grammar, unit testing and static checking. All of them utilize Eclipse usually.

We gave some items to examinees: Java source code with sufficient JML, test cases and specification written in Japanese.

We implemented Multiple Table View to compare our graph propose and simple method. Multiple Table View contains class names, passage rates of unit testing and static checking. It does not have caller-callee relationship data.

Table I shows assigning examinees to groups. Group A and B compare between ISOM Layout and Priority Layout, and Group C and D compare between Priority Layout and table instead of graphs.

TABLE I
Group

Group	Method used in Problem 1	Method used in Problem 2
A	ISOM Layout	Priority Layout
B	Priority Layout	ISOM Layout
C	Table	Priority Layout
D	Priority Layout	Table

C. Problems

Target programs to solve are based on Warehouse Management Program[19]. We altered its source code and test cases to be buggy. Warehouse Management Program consists of seven classes: ContainerItem, Customer, Item, ReceptionDesk, Request, Storage and StockState. Its test suites sizes 800 LOC.

addItem method of AppMain has a bug. addItem method calls constructor of Item class without checking parameters. On the other hand, in Figure 6, the constructor of Item class has a pre-condition that parameter name must not be null and amount must be greater than zero. Thus, addItem method violates the pre-condition of constructor of Item class. addItem method can behave unanticipatedly and cause troubles when it is executed.

D. Result and Observation

Table II shows the result of time to detect buggy method, excluding wrong answer and who cannot find the bug within deadline. Time required to find the bug using Priority Layout is the shortest, followed by using ISOM Layout and using simple tables in this order. Using Priority Layout reduces the

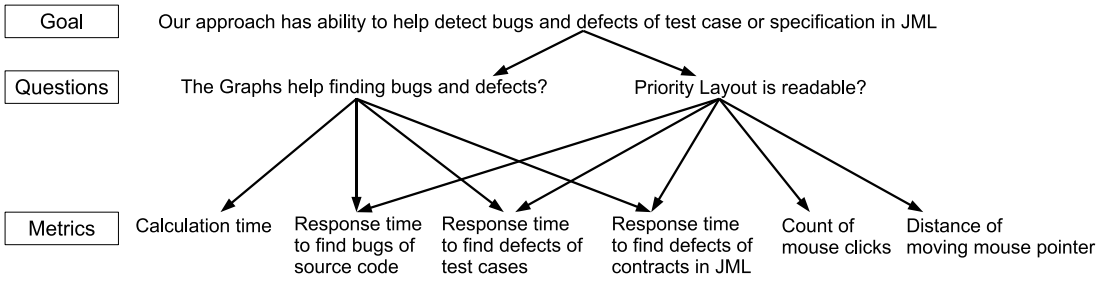

Fig. 5. Result of applying GQM Model

TABLE II
RESPONSE TIME TO DETECT BUGGY METHOD

Method	Avg. of Time (min)
ISOM Layout	12.9
Priority Layout	10.79
Table	21.48

TABLE III
RESPONSE TIME TO DETECT INSUFFICIENT TEST CASE

Method	Avg. of Time (min)
ISOM Layout	13.66
Priority Layout	14.9
Table	26.39

```
public class Item {
  ...
  //@requires name != null && amount >= 0;
  public Item(String name, int amount) {
    ...
  }
  ...
}
```

Fig. 6. Item

time by 16 % of ISOM Layout, and using both of graphs result half the time of using table. Thus, for this example program, usefulness of our proposed method was observed about finding the bug in source code.

Table III summarizes the time required to find the bug in test cases. On the other hand, time to find the bug in test cases, there is little difference between using Priority Layout and ISOM Layout. The reason should be that our proposed technique visualizes classes instead of methods. Let T_t be time to find the bug in test cases, T_c be time to find buggy test class, and T_m be time in find buggy test method. These variables have the following relation:

$$T_t = T_c + T_m.$$

Finding bugs in test cases occurs after finding bugs in source code because we had an instruction to do so. A class and its test class have a one-to-one correspondence in the target program, Hence,

$$T_c = 0$$

and

$$T_t = T_m$$

hold. T_m must be similar between using Priority Layout and ISOM Layout because none of layout provide to help find method.

The average of answers are 3.5 and 4.3 for Q1 and Q2, respectively. The result of Q1 shows that it is useful to compare between graphs of unit testing and graphs of static checking. Table II and the result of Q2 conclude that Priority Layout improves visibility of the graphs for developers.

Since Priority Layout reduces response time in finding bugs at source code, the examinees realize that Priority Layout is better than another layout.

The followings are some comments from the examinees:

- The graph makes it easy to understand dependency of classes
- It enables the users to easily realize classes to focus
- It takes much time to understand the program structure
- Visualizing dependency of methods is required
- It must provide reason why methods were failed by static checking.

To quicken finding bugs, last two comments demonstrate the need more information on methods.

E. Threats to Validity

1) Internal Validity: There is little difference in ability among the groups. The examinees learned software development because they are members of software design laboratory. They are divided into four groups evenly mixed their grades.

Among the examinees, there are few habituation for the target program and our tool because they read the target program for the first time and they have never utilized the tool.

2) External Validity: The universality of the program is low because we use small size program in the experiment. However, the program is a typical to manage inventories, it includes the essence of similar kind of real systems. Consequently, our tool is useful in these types of programs.

Answers of the questionnaire have low generality because they are subjective data. Also, the reliability of the results for Q1 and Q2 is low. This is caused by the reason: although the scores in the questionnaires are ordinary scale, the data is averaged.

Hence, we need to perform more experiment. The example includes comparing the response time for large size program between using our tool and using eclipse without our tool.

VI. Discussion

We need metrics to specify the quality of given JML statements. We have researched past papers, however, we find few 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 while 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.

VII. Conclusion

In this paper, we improved and evaluated the visualization method for software quality using unit testing and static checking. We proposed Priority Layout, and performed evaluation using examinees. The results show that our approach contributes to reduce the time in finding bugs at source code and test cases.

Future work includes improving information on methods to find easily actual bug location. Furthermore, we will research and evaluate what we said in section VI, quality of the test suites and annotations in JML.

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).

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 edition). D*, vol. 92, no. 9, pp. 1518–1531, 2009. [Online]. Available: "http://ci.nii.ac.jp/naid/110007361123/"

[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. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.100.3818

[4] M. F. Kleyn and P. C. Gingrich, "Graphtrace—understanding object-oriented 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] ISO, "Software engineering-product quality-part 1 : Quality model," *ISO/IEC : 9126-1:2001*, 2001. [Online]. Available: http://ci.nii.ac.jp/naid/10024638597/

[6] P. Chalin, "Early detection of jml specification errors using esc/java2," in *SAVCBS '06: Proceedings of the 2006 conference on Specification and verification of component-based systems*. New York, NY, USA: ACM, 2006, pp. 25–32.

[7] 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. [Online]. Available: http://dx.doi.org/10.1007/s11219-006-9001-4

[8] L. Burdy, M. Huisman, and M. Pavlova, "Preliminary design of bml: a behavioral interface specification language for java bytecode," in *Proceedings of the 10th international conference on Fundamental approaches to software engineering*, ser. FASE'07. Berlin, Heidelberg: Springer-Verlag, 2007, pp. 215–229. [Online]. Available: http://portal.acm.org/citation.cfm?id=1759394.1759418

[9] Y. Muto, K. Okano, and S. Kusumoto, "A visualization technique for software quality using unit testing and static checking with caller-callee relationship," in *Proceedings of International Conference on Advanced Software Engineering*, 2011.

[10] 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.

[11] 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.

[12] 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.

[13] T. Knauth, C. Fetzer, and P. Felber, "Assertion-driven development: Assessing the quality of contracts using meta-mutations," in *Proceedings of the IEEE International Conference on Software Testing, Verification, and Validation Workshops*. Washington, DC, USA: IEEE Computer Society, 2009, pp. 182–191. [Online]. Available: http://portal.acm.org/citation.cfm?id=1547559.1548252

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

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

[16] 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.

[17] K. T, "The self-organizing map," *Proceedings of the IEEE*, pp. 1464–1480, 2002.

[18] V. Basili, "Using measurement to build core competencies in software," *Seminar sponsered by Data and Analysis Center for Software*, 2005.

[19] 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 edition) D*, vol. 91, no. 11, pp. 2719–2720, 2008. [Online]. Available: http://ci.nii.ac.jp/naid/110007380947/