A Visualization Technique for Unit Testing and Static Checking with Caller–Callee Relationships

Yuko Muto, Kozo Okano, 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 techniques fall into two categories: visualization of software component relationships and visualization of software metrics. In this paper, we propose a new hybrid method based on both categories. The proposed method visualizes the coincidence between specification and implementation from two aspects: static checking and ordinal testing by test suites. In our method, each ratio of the coincidence is shown by pie charts which represent classes of the target software. The whole software is represented in a weighted digraph structure. We have prototyped a tool to implement our proposed method. We have evaluated the utility of the proposed method by applying the tool to two kinds of software: a warehouse management program and a telephone directory management program. We conclude that the proposed method yields informative results.**

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

I. INTRODUCTION

Visualization techniques for software have recently been playing more important roles due to the increase in the size of software. Visualization techniques fall into two categories: the visualization of software component relationships and the visualization of software metrics. The former approach [1] often shows program flows as PDGs (Program Dependency Graphs). The latter approach includes the visualization of the temporal sequence of software metrics which helps in the analysis of software development [2].

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

Some papers [3] and [5] have proposed visualization methods software components. In [3] it is stated that visualization is performed in several views: static views which show the abstract structure of programs, and dynamic views which depict the dynamic traces of programs. Recently, the quality of software has become important. Few papers, however, provide a visualization of the quality of the software. Our approach overcomes this weakness.

ISO defines the quality of software [6] as consisting of six properties: functionality, reliability, usability, efficiency, maintainability, and portability. Functionality is a kind of metric which defines whether the software satisfies the required properties. It requires that the software must implement the requirements. Functionality can be measured by ordinary unit testing, static checking, model checking, or model based testing. Ordinary unit testing tests, using sufficiently many test suites, a given module relative to its specification to see whether the module satisfies the specification. 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 the results of such a test sometimes depends on the quality of the test suites used. If the coverage of the test suites is low, then some properties cannot be tested.

On the other hand, static checking and model checking do not require executing the source code. These approaches check the source code statically (or an abstract model of the source code which models its behaviour). One famous tool for static checking is ESC/Java2 [7]. Its input is a Java program annotated with JML (Java Modeling Language) [8], [9], in a DbC [13] manner. It checks whether the (behaviour of the) source code satisfies the property described in the JML. The quality of the output also depends on the property itself as well as that of the standard libraries used for ESC/Java. Another drawback of ESC/Java2 is that it is not easy to understand the relationships among classes because its outputs are text-based.

Model-based testing is yet another important approach. It needs a model to create test-suites. Recently, model-based testing with Spec Explorer has obtained a lot of attention [21]. Spec Explorer was developed by Microsoft Research. It uses spec# or AsmL [22] as the modeling language.

However, in this paper, we focus on classical unit testing and static analysis because unit testing is still a popular method and both static checking and model-based testing need a modeling language to describe specification of the target program.

Therefore, a hybrid approach is considered. For example, [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 categories. The proposed method visualizes the coincidence between specification and implementation from two aspects: ordinary testing (by test suites) and static checking. Each 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. The software as a whole is represented in a weighted digraph structure.

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: a warehouse management program and a telephone directory management program. We conclude that the proposed method yields informative results.

This paper is based on [23], strengthening related work and the experiment results section.

The contribution of our paper is as follows. 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. We show that such technique is useful to analyse the quality of a target program.

This paper is organized as follows. Chapter II briefly discusses related work. Chapter III provides some definitions of words as a preliminary. Chapter IV will describe our proposed method. We give an overview of our prototype tool in Chapter V, followed by experimental results and a discussion in Chapters VI and VII. Finally, Chapter VIII concludes the paper.

II. Related Work

There are many works related to our work.

A. Visualization

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

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

B. Combining Unit Testing and Static Cheking

Check'n'Crash [10] combines ordinary unit testing and static checking. The approach can automatically find some faults. 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 the sense that it produces only suitable test suites for suspected faults.

It does not cover 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 possibilities of runtime execution's causing errors, such as memory faults.

That paper performed ordinary unit testing after static checking. The opposite way is used in [11]. Tests cannot find corner case bugs. The method in [11] firstly performs testing relative to the target and obtains its coverages. Secondly it performs static checking on the complementary part of the coverages. Thus, the static checking can be applied to a limited area of the target source code; and it gains in 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 cover the combination of branch conditions; while corner case bugs may be detectable only for 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 behaviour
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

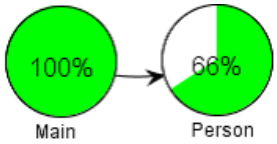

Figure 2. Visualization of static checking for Main class and Person class

III. Preliminary

This chapter gives some definitions and explanations of unit testing and static checking.

A. Unit Testing

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

B. Static Checking

1) JML: JML (Java Modeling Language) [8], [9] is a specification language used for annotation in Java programming. Based on DbC (Design by Contract) [13], we can assert 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 field *name* must be always non-null, thus, the annotation of the third line is given. Method *setName* has a pre-condition that *nm* is neither Null nor null String, thus, the annotation of the fifth line is given. Also the ensure clause gives the post-condition which means that name has the same value as *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. The verification is performed by translating the source code and associated JML annotation into a logical expression, and evaluating by a theorem prover called simplify [24].

It requires the Java source code and outputs a result as text messages which say pass or fail for each property and each method. If it reports fail, its counter-example also generated. Such approaches are also taken in Spec#, a C# based language with specification annotation features. A translator translates Spec# program code into a logical language designed for general programs, Boogie2 [20]. The translated language is then evaluated by the SMT (Satisfiability Modulo Theories) solver, Z3 [19].

IV. Our Proposed Method

This chapter describes our method.

A. Overview

Figure 2 presents the result of static checking for Figure 1. The caller–callee relation of the given target program is shown in a digraph, where each node and each edge represent, respectively, a class and a caller–callee relation. Each node also represents a pie-chart which gives the 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 the caller–callee relation instead of the 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 to ours.

B. Definition of Passage Rate Metrics

Here, we have to think of 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) those for the results of static checking. In this paper, we focus on the metrics for (1), (3) and (4). We discuss the metrics for (2) in Chapter VII.

1) Passage Rate for quality of the test suites: We adopt also statement coverage as a passage rate of unit testing. The reason is that 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.

2) Passage Rate for Results of Unit Testing: We adopt also statement coverage as a passage rate of the result of unit testing. More precisely, we define Passage Rate for as the number of test suites passed divided by the total number of test suites. The test suites passed should be defined as the test suites executed whose results satisfy a reference level which is prepared in advance. Currently, for simplicity, the tool regards the metrics for (1) and (2) as being the same.

3) 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 by

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

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

4) Some discussion of metrics for the quality of the test suites and for the results of ordinary unit testing: As a result, we adopt the same statement coverage as passage rates for both metrics: that for the quality of the test suites and that for the results of ordinary unit testing. It would be a good idea to give different metrics for the two qualities. One of the ideas is that for the quality of the test suites, we define the statement coverage based on a syntactical calculus but the real passage rate. Such a definition would produce different values against test suites with random behaviour or dynamic binding. However, in this paper, we use the same statement coverage.

C. Definition of the Caller–Callee Relation

If method m1 appears in 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 A calls B. Let n_{AB} be the number of calls such that class A calls class B. We say A calls B n_{AB} times. The following explains the caller–callee relation and the number. In Figure 1, Main class calls the constructor of Person class in line 3, method *setFullName* in line 4, and method *getFamilyName* in line5. Thus, Main class calls Person class three times.

V. Implementation

Here, we give simple descriptions of 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 it as a plug-in. We use libraries MASU and JUNG as part of the tools. MASU provides general metrics measurement and a program analysis library [1]. We use MASU in order to analyse 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.

Currently, the tool only deals with: (3) metrics for the results of ordinary unit testing, and (4) those for the results of static checking.

A. Input

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

B. XML documents

Here, we give overview of the XML document. Figure 3 shows the format of an XML document. The class tag corresponds to a Java class. It contains the Package name, class simple name, coverage, as well as information on methods. The method tag corresponds to a Java method. The attributes tag is used for displaying the information.

```
<?xml version="1.0" encoding="utf-8"?>
<class>
 <packageName> package name </packageName>
 <simpleName> class id </simpleName>
 <methodCount> the number of methods
  </methodCount>
 <coverage> coverage </coverage>
 <methods>
  <method id="1">
   <name> method id </name>
   <parameter> arg1,arg2,... </parameter>
   <attributes>
    <attribute>
    <title> attribute title </title>
    <value> value of attribute </value>
    </attribute>
   </attributes>
  </method>
  <method id="2">
  ...
  </method>
 </methods>
</class>
```

Figure 3. XML format

C. Views

Figure 4 is a screenshot of the Tool. It has Main View for showing the digraph and Method View for showing detailed method information.

Main view in Figure 4 shows the caller–callee relations by edges. The thickness of the edge expresses the number of calls. For example, if class A calls class B *n* times and class A calls class C m times, then the edge between A and B is thicker than that between A and C provided that $n > m$.

Main View supports several features, such as a filtering feature. Using the filtering feature, we can select a class and its related classes. Figure 5 shows an example. From the Main View, if we select Request Class and perform the filter feature, then the view below is obtained. The view only shows Request class and its related classes. The related classes are selected by caller–callee relation metrics.

Figure 6 shows the architecture of the tool. Each of the XML files is generated by its specific script which calls an adequate tool, djUnit or ESC/Java2.

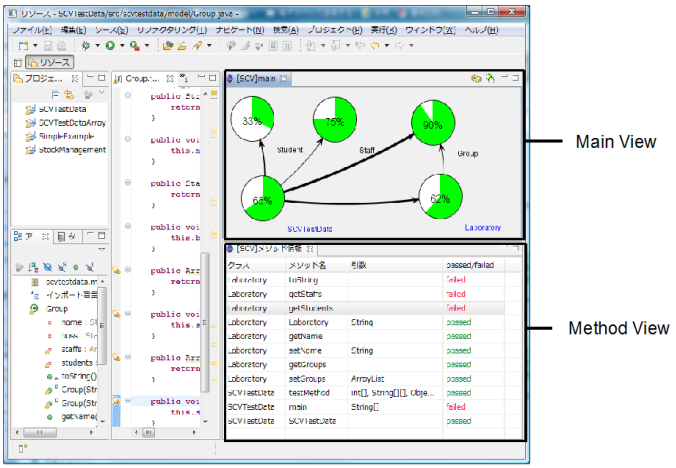

Figure 4. Screenshot

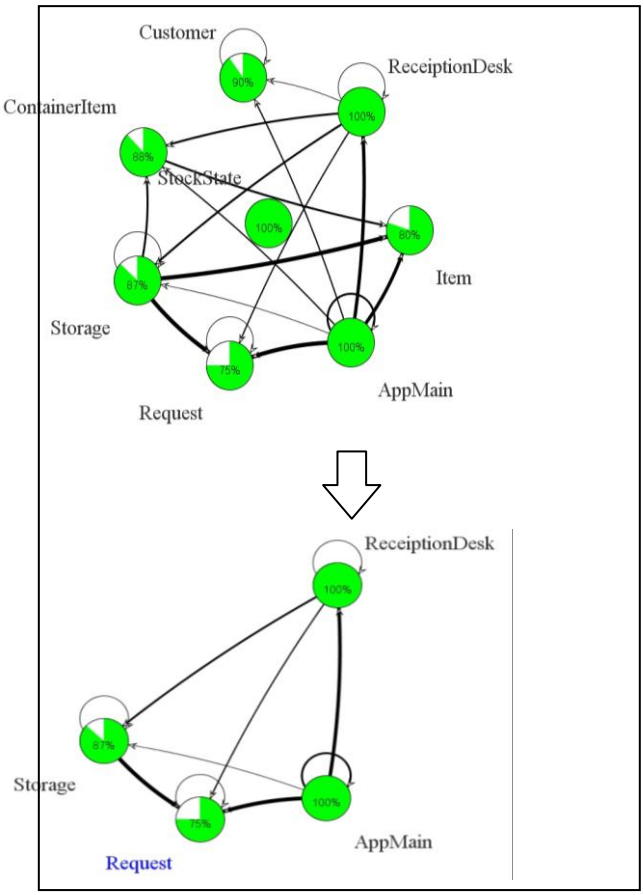

Figure 5. Filtering feature

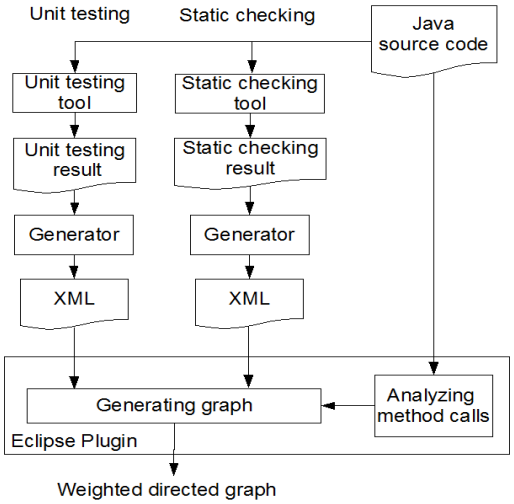

Figure 6. Architecture

VI. EXPERIMENT

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

A. The Evaluation Approach

We applied our tool to the following two programs.

1) Targets: We use two programs, one is a warehouse management program, and the other one is a personal telephone directory.

The warehouse management program is implemented in Java1.4. The program has seven classes of about 400 LOC except for JML annotations and the 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.

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

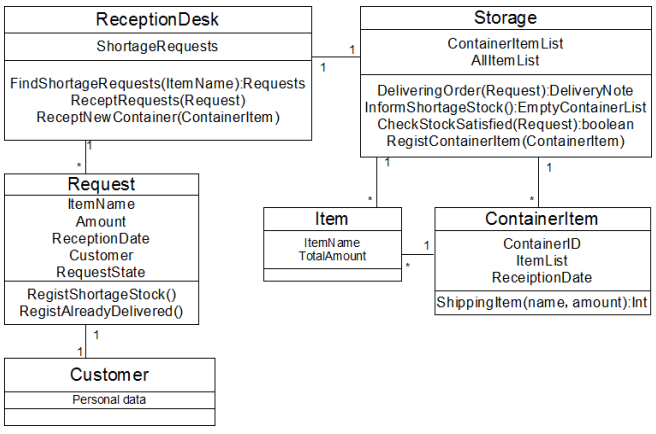

Figure 7. Class constitution of the warehouse management program

2) Condition: We make an assumption. Assumption 1: We assume that the warehouse management program has valid JML annotations with poor test suites, whereas the personal telephone directory has poor JML annotations with sufficient test suites.

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

B. Warehouse Management Program

The warehouse management program consists of seven classes: ContainerItem, Customer, Item, ReceptionDesk, Request, Storage and StockState. Figure 7 shows the UML diagram of the program. The program manages stock items of a warehouse of a liquor shop. Inputs are lists of container items and lists of request orders; while outputs are empty container lists and lists of shipping orders. The management has to decide the outputs according to the current status of warehouse.

Because the program already has JML annotation with checking, we just added 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

The personal telephone directory has the following five classes: AddressBook, Entry, NameComparator, TelComparator, and MailDomainComparator.

The program manages a personal telephone directory. It has sorting features by three kinds of keys.

The personal telephone directory has sufficient test suites, thus, we regard the program is valid from the point of view of unit testing. On the other hand, the JML annotation is not sufficient.

D. Results

Figures 8 and 9 show the digraphs representing unit testing and static checking, respectively, for the warehouse management program. Figures 10 and 11 show the digraphs representing unit testing and static checking, respectively, for the personal telephone directory.

E. Discussion

1) Unit Testing: Let us discuss the unit testing results of each program.

a) Warehouse Management Program: In Figure 8, 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 for unit testing. Every terminal node (class) has high values of passage rate. This shows that such a class tends to be a typical Java bean, thus they have only simple setter/getter methods.

b) Personal Telephone Directory: Sufficient test suites are given, the passage rates of every class are all 100%. Entry class is called from every other class; thus, its quality affects the whole the program. Developers should look carefully at

Entry class. Visualization of such information is useful for developers.

2) Static Checking: Let us discuss two graphs generated by static checking.

a) Warehouse Management Program: Figure 9 shows that every class has a high passage rate. Let's look at precisely the caller–callee relation and the result of static checking. For example, class ReceptionDesk has a passage rate of 100%. It seems that the class has perfect quality and no problems. The class calls the following classes: Storage (87%), ContainerItem (88%), Request (75%), and 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 including the passage rate of calling classes.

b) Personal Telephone Directory: NameComparator, MailDomainComarator and TelComparator have the same function. Therefore their behaviours are also the same, although the implementation of comparator is different. However, the passage rates are not identical: 33% and 60%. The reason is that MailDomainComparator has two private methods which are passed while the others have one. Therefore, the passage rate of MailDomainComparator becomes 3/5 = 60%, while others are 1/3 = 30%.

Therefore, we have to take note that such figures do not correctly indicate the quality. We have to consider the difference in importance between private methods and public methods. I.e., it might be a good idea to calculate the passage rate on public methods only.

3) Comparison between Unit testing and Static checking: Let us consider the results of unit testing and static checking.

a) Warehouse Management Program: The classes Customer, Request, Item, and StockState have high passage rates in both unit testing and static checking. These classes have codes satisfying their specifications well. Thus the quality of the class is also high.

On the other hand, the classes ReceptionDesk, Storage, and ContainerItem have low passage rates of unit testing yet high passage rates of static checking.

Thus, we can conclude that unit testing is not enough performed. In fact, the 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.

b) Personal Telephone Directory: We discuss the results in Figures 10 and 11. First, let's consider the classes AddressBook and Entry, both of which have high passage rates in unit testing and static checking. We can conclude that these classes are of high quality.

Next, we consider classes with a high passage rate in unit testing and a 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 satisfactorily prove the correctness of a given assertion. These classes implement the java.util.Comparator interface. Though the library used in ESC/Java2 includes annotations of java.util.Comparator, the annotations are very general and weak. Moreover, neither NameComparator nor TelComparator has adequate annotation. Thus, the quality of the static checking results is low. We also conclude that the quality of these classes is high due to the passage rate of unit testing.

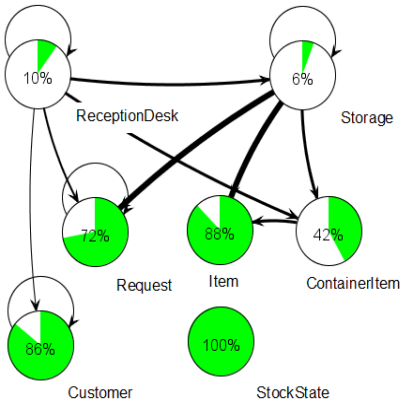

Figure 8. Result of unit testing in the warehouse management program

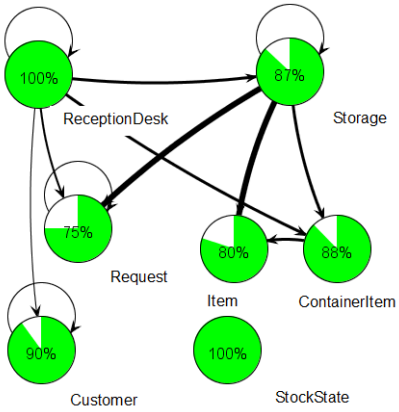

Figure 9. Result of static checking in the warehouse management program

F. Threats to Validity

Here, we simply summarize the 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 the JML specification itself is not tested enough. For 1 and 2, to handle large programs in a huge range of categories, we need more programs with JML annotations. Today, Java programs with JML are not popular: it is not an easy task. Several papers provide methods that automatically produce JML annotations, such as Daikon [17]. Such techniques might help to resolve the problem. Daikon is a tool to generate assertions by executing the target program with test suites. Daikon has a lot of assertion templates and from the trace of variables to check, it infers suitable assertions. We have already discussed 3.

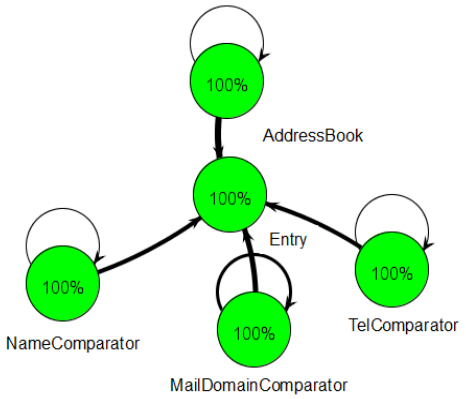

Figure 10. Result of unit testing on personal telephone directory

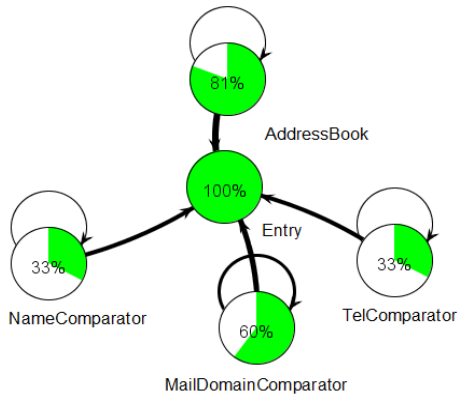

Figure 11. Result of static checking on personal telephone directory

VII. DISCUSSION

Here, we discuss two remaining 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 devised a new metric, called Variable Coverage. In general, assertions are conditions on program variables. For example, pre-condition and post-condition assert that the parameters and return values (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, *for a given assertion*, we can regard the ratio between the number of its *used* variables and the number of *all instance variables and parameters* as coverage, called Variable Coverage. 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 of 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 of 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 suites 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 in multiple aspects. We developed a prototype tool of our method as a plug-in of Eclipse, and evaluated it through some examples. The results show that we can evaluate the quality of software in more detail by the proposed method. Additionally, in a preliminary experiment we had, some examinees said “This visualization method is more effective than reading the program only or viewing a simple table in order to find bugs”.

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

ACKNOWLEDGMENT

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

We also thank Dr. Takashi Ishio of Osaka University for providing the 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 edition). 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: 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: 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: 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: 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, 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 edition) 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: IEEE Computer Society, 2010, pp. 262–26.

[19] C. M. Wintersteiger, Y. Hamadi, L. M. de Moura, "Efficiently solving quantified bit-vector formulas," In Proc. of FMCAD 2010: pp. 239–246, 2010.

[20] K. R. M. Leino and P. Rümmer, "A polymorphic intermediate verification language: Design and logical encoding," In Lecture Notes in Computer Science, vol. 6015, pp. 312–327, 2010

[21] M. Veanes, C. Cambell, et. al., "Model-based testing of object-oriented reactive systems with Spec Explorer," In Formal Methods and Testing, Lecture Notes In Computer Science, vol. 4949, pp. 39–76, 2008.

[22] M. Veanes, N. Bjorner, Y. Gurevich, and W. Schulte, "Symbolic bounded model checking of abstract state machines," in International Journal Software Informatics, vol. 3, no. 2–3, pp. 149–170, June 2009.

[23] Y. Mutoh, K. Okano, S. Kusumoto, "A visualization technique for the passage rate of unit testing and static checking with caller–callee relationships," Proc. of International Conference on Advanced Software Engineering 2011, to appear, May 2011.

[24] D. Detlefs, G. Nelson, and J. B. Saxe, "Simplify: A theorem prover for program checking," Journal of the ACM, vol. 52, no. 3, pp. 365–473, 2005.