New Metrics for Program Specifications Based on DbC

Kozo Okano†, Yuko Muto†, Yukihiro Sasaki†, Takafumi Ohta†, Shinji Kusumoto†, and Kazuki Yoshioka†

†Graduate School of Information Science and Techlology, Osaka University, Japan
{okano, t-ohta, kusumoto}@ist.osaka-u.ac.jp

Abstract - For realizing dependable and maintainable software, Design by Contract (DbC) is a useful approach. DbC utilizes constraints as contracts between the caller and the callee routines in programs. Verifiers for the programs are able to check whether the given source code satisfies the given constraints. However, it is difficult to measure the exhaustiveness for a specification, *i.e.*, how well the constraints cover the ideal specification for the source code. This paper proposes Variable Coverage, a simple set of metrics to check the exhaustiveness of specification for source code in Java and other object-oriented programming languages. The proposed coverage observes the occurrence of variables in the constraints, such that the variables are also used in the target method/-constructor. We applied the metrics to three actual programs to evaluate the ability of Variable Coverage to find variables that should have been referred in specifications as important variables. As a result, the shortage of JML annotations found in the target programs shows the usefulness of the proposed metrics.

Keywords: DbC, Coverage, Specification,Testing, Metrics

1 INTRODUCTION

Formal methods [1], which are mathematical techniques for the specification, development and verification of software and hardware systems, have attracted much attention because they are said to play important roles for designing software, especially since the size of software has increased. The larger the program size, the more frequently the software testing misses the corner case. Formal methods can perform exhaustive checking. In various industries, such as public transportation systems, real and large programming projects have been successful by using formal methods [2]. Formal methods are classified into three technologies: deductive methods, model checking, and model-based simulation or testing.

Design by Contract (DbC) [3] is a well-known approach for clarifying the responsibility between callers and callees. Java Modeling Language (JML) [4]–[6] is a specification language for Java based on DbC. A program based on DbC can be verified with static checking and runtime checking. For example, ESC/Java2 [7] and jml4c [8] are such tools for Java. As an example in another language, Spec# [9] is a superset of C#, and the static checker for Spec#, developed by Microsoft, uses Boogie [10].

However, it is difficult to determine whether the specification is well-written (exhaustive). If the specification is low exhaustive, the correctness of the program is not clear. In runtime checking, as an example, a runtime checker produces a violation when the source code and its specification do not match. No violation is reported by the runtime checkers if the code has no specification, because no constraint has been specified. Consequently, we cannot do anything about the quality of the source code.

Some papers have studied coverage metrics for hardware verification. Chockler et al. [11] summarized coverage metrics for simulation-based verification, such as code coverage and assertion coverage. To generate a test efficiently, Moundanos et al. [12] proposed functional coverage as the amount of control behaviors covered by a test suite using abstraction techniques. Nevertheless, few coverage metrics can be applied to general-purpose programming languages at the implementation level. This includes Java and JML.

In this paper, we propose Variable Coverage as coverage metrics for formal specifications at the implementation level. Variable Coverage consists of the coverage for the pre-condition, post-condition, assignable and invariant.

In experiments, we used a prototype that measures the Variable Coverage of three kinds of programs. As a result, we found a shortage of JML annotations in the target programs. The result shows the usefulness of our proposed metrics.

The paper is organized as follows. Section 2 provides definitions of important terms and Section 3 introduces related work. Section 4 shows our proposed method, Variable Coverage, followed by experiments and discussion in Sections 5 and 6, respectively. Finally, Section 7 concludes this paper.

2 PRELIMINARIES

This section provides some concepts and definitions.

2.1 Design by Contract

Design by Contract (DbC) was proposed by Bertrand Meyer [3]. In DbC, suppliers (callee routines) and clients (caller routines) make contracts with each other. The clients should satisfy the pre-conditions, and the suppliers should satisfy the post-conditions under the pre-conditions. This mechanism makes it easier to identify bugs.

Some programming languages support DbC as a standard, and others have a specification language that is separate from the core grammar of the language. Eiffel [13] supports DbC as the standard. C# and Java have no standard contract system but some specification languages are proposed separately. Spec# [9] is a superset of C# to describe contracts. For Java, JML [4] is the de-facto standard specification language.

2.2 Constraints

The pre-condition for a routine (method) is a set of Boolean constraints. It should be `true` prior to the routine execution.

Clients are responsible to meet the pre-condition.

The post-condition for a routine is a set of Boolean constraints. It should be `true` after the routine execution, provided that its associated pre-condition holds. Suppliers are responsible to meet the post-condition under the pre-condition.

The routine is permitted to assign values to only the variables specified in Assignable. The constraints help the developers to detect side effects.

Invariant is a set of Boolean constraints. Invariants should be always `true`. Depending on the target of a constraint, invariants are divided into class invariant and loop invariant. This paper deals with only the class invariant.

2.3 Java Modeling Language

JML is a specification language based on DbC for Java. JML supports the pre-condition, post-condition, assignable and invariant. We explain them through class `BankAccount`, an account for a customer of a bank, as an example.

Figure 1 is the source code of class `BankAccount` with JML.

```
1  public class BankAccount {
2
3    private int balance;
4    // @invariant balance >= 0;
5
6    // @ensures balance == 0;
7    // @assignable balance;
8    public BankAccount() {
9      this.balance = 0;
10   }
11
12   // @requires amount >= 0;
13   // @requires balance >= amount;
14   // @ensures  balance == \old(balance) -
         amount;
15   // @assignable balance;
16   public void withdraw(int amount) {
17     this.balance -= amount;
18   }
19
20   // @requires amount >= 0;
21   // @ensures  balance == \old(balance) +
         amount;
22   // @assignable balance;
23   public void deposit(int amount) {
24     this.balance += amount;
25   }
26
27   // @ensures \result == balance;
28   // @assignable \nothing;
29   public int getBalance() {
30     return this.balance;
31   }
32
33   // @pure
34   public void inquiry() {
35     System.out.println("Balance is " + this.
           balance);
36   }
37
38 }
```

Figure 1: Source Code of Class BankAccount with JML

Pre-conditions Keyword `@requires` expresses the pre-condition. In Fig. 1, methods `withdraw` and `deposit` have pre-conditions in lines 12,13 and 20.

Post-conditions Keyword `@ensures` expresses the post-condition. The constructor and methods `withdraw`, `deposit` and `getBalance` have post-conditions. Line 6 in Fig. 1 means that field `balance` is 0 after the instance creation.

Assignables `@assignable` expresses the assignable. The following `@assignable` classes, fields that can be assigned, are listed. If every field is prohibited to be assigned, then `@assignable \nothing` is used, as in line 28 in Fig. 1. To shorten the code, `@pure` is equivalent to `@assignable \nothing`.

Invariants The JML description of invariants is `@invariant`. Also, attribute `a` with `@non_null` is equivalent to `@invariant a != null`. In Fig. 1, line 4 is an invariant clause that means field `balance` must be 0 or more at any time.

2.4 Global Variables

Generally, the term "global variables" is not used in object-oriented programming language. In this paper, as a matter of convenience, we define global variables as follows.

Definition 2.1 (Global Variables)
When a method m is a member of class c, a global variable g is defined as:

- g is not a member of c, and
- g is visible from m.

Figure 2 shows an example of a global variable. The variable `font` of class `Config` is a global variable for method `draw`.

```
1  public class Config {
2    public static Font font;
3  }
4
5  public class Customer {
6    public void draw(Graphics g) {
7      g.setFont(Config.font);
8      g.drawString("An example for a global
           variable", 10, 10);
9      ...
10   }
11 }
```

Figure 2: An Example of a Global Variable

3 RELATED WORK

This section introduces some of the work related to this paper.

3.1 Program Verification

ESC/Java [14], an Extended Static Checker for Java, was the practical usable checker among the early verifiers. Currently, its successor version, ESC/Java2 [15] is widely used and supports JML2.

Also supporting the newer Java, Mobius [16] has attracted increased attention as a program verification environment (PVE) that includes static checkers, runtime checkers, and verifiers. It is provided as an Eclipse [17] plug-in. ESC/Java2 is also integrated into Mobius.

3.2 Verification Coverage

Coverage metrics for formal verification are called verification coverage primarily in the hardware field. Verification coverage falls into two categories: syntactic coverage and semantic coverage [11]. As syntactic coverage, code coverage for model-based simulation is the metric derived from software testing [18]. The ratio of executed code during a simulation is code coverage. As simple coverage, line coverage is the code of a block without control transition.

Coverages depending on a control flow graph (CFG) are branch coverage, expression coverage, and path coverage.

Semantic coverage is categorized into assertion coverage and functional coverage. Assertion coverage is the measuring method by which users determine variables to observe. Assertion coverage measures what assertions are covered with a given set of input sequences [11].

To generate a test suite to be analyzed, Moundanos et al. [12] proposed functional coverage, which is the amount of control behavior covered by a test suite using abstraction techniques.

3.3 Assertion Density

Assertion density is the number of assertions per line of code [19]. Without sufficient assertion density, the full benefits of assertions are not realized. Assertions must be verified for behaviors as design intents, *i.e.*, statements of properties.

4 VARIABLE COVERAGE

This section defines Variable Coverage, our proposed method.

4.1 Motivation

Formal verification checks the consistency between source code and its contracts based on the Class Correctness formula. Chockler et al. [11] stated that "Measuring the exhaustiveness of a specification in formal verification has a similar flavor as measuring the exhaustiveness of the input sequences in simulation-based verification for hardware." To apply this idea to software, the input sequences of a method/constructor correspond to variables. Consequently, we propose a coverage metric that observes variables.

4.2 Policies

We propose a set of metrics that supports these policies:

1. Our metric checks all variables as input and output.
   It is oriented with verification coverage.

2. Our metric is simple.
   The execution of measuring the coverage requires a relatively short time. The metric targets developers who describe assertions in JML. Our metric should be checked for a short time on a frequent basis when the developers want to conduct checks.

3. Our metric uses only static information.
   Using only static information (source code and JML) without an execution trace enables the measurement of coverage for a part of incomplete code.

4.3 Constraints Development Process with Variable Coverage

Quickly measuring Variable Coverage (hereafter, VC) enables a high frequency of measurements. Implementators can improve the constraint descriptions by the iterative process:

Step 1 Implementators describe assertions.

Step 2 VC is measured.

Step 3 Iterate Step 1 if the implementators do not find all of their assertions.

We call such an iteration "Quick VC revise."

4.4 Definition of Variable Coverage

VC consists of four kinds of metrics: coverage for the pre-condition, post-condition, assignable and invariant. Tables 1 and 2 show the VC metrics for a single constraint and multiple constraints, respectively.

4.4.1 The Coverage for Pre-conditions

Pre-conditions should check all input variables, *i.e.*, parameter of the method, attributes and global variables referred in the method. Thus, the coverage for pre-conditions consists of Parameters Coverage and Referred Attributes Coverage.

Definition 4.1 (PrPC)
Let $P(m)$ and $P_{held-by-pre}(m)$ be a set of parameters defined in method m and held by a pre-condition in method m, respectively. Equation (1) defines $PrPC(m)$, Parameters Coverage for pre-conditions of method m.

$$PrPC(m) = \frac{\mid P_{held-by-pre}(m) \mid}{\mid P(m) \mid} \tag{1}$$

In Fig. 3, both $\mid P_{held-by-pre}(m) \mid = \mid \{age\} \mid = 1$ and $\mid P(m) \mid = \mid \{name, age\} \mid = 2$ hold. Hence, we have $PrPC(m) = 1/2$.

Definition 4.2 (PrAC)
Let $A_{referred}(m)$ and $A_{held-by-pre}(m)$ be a set of attributes referred in method m and held by the pre-condition in method

Table 1: Variable Coverage (single constraint)

Coverage Name	Constraint	Target Variables	Measuring Unit
PrPC	Pre-Condition	Parameters	Method
PrAC		Refered attributes	Method
PrGC		Refered global variables	Method
PoRC	Post-Condition	Return value	Method
PoAC		Assigned attributes	Method
PoGC		Assigned global variables	Method
AAC	Assignable	Assigned attributes	Method
IAC	Invariant	Attributes	Class

Table 2: Variable Coverage (multiple constraints)

Coverage Name	Constraint	Target Variables	Measuring Unit
PrIAC	Pre-condition + invariant	Refered attributes	Method
PoIAC	Post-condition + invariant	Assigned attributes	Method

```
1 //@ requires age >= 0;
2 // no requires holds 'name'
3 public Customer(String name, int age){
4   this.name = name;
5   this.age = age;
6 }
```

Figure 3: An Example to Explain Parameters Coverage for Pre-condition

m, respectively. Equation (2) defines $PrAC(m)$ as the Referred Attributes Coverage for pre-conditions of method m.

$$PrAC(m) = \frac{\mid A_{held-by-pre}(m) \mid}{\mid A_{referred}(m) \mid} \tag{2}$$

Definition 4.3 (PrGC)
Let $G_{referred}(m)$ and $G_{held-by-pre}(m)$ be a set of global variables referred in method m and held by the pre-condition in method m, respectively. Equation (3) defines $PrGC(m)$ as the Referred Global Variables Coverage for pre-conditions of method m.

$$PrGC(m) = \frac{\mid G_{held-by-pre}(m) \mid}{\mid G_{referred}(m) \mid} \tag{3}$$

4.4.2 The Coverage for Post-conditions

Post-conditions observe output variables that have an effect outside of the method, such as the return value, attributes and global variables assigned in the method. Hence, the coverage for a post-condition is composed of Return Value Coverage, Assigned Attributes Coverage and Assigned Global Variables Coverage.

Definition 4.4 (PoRc)
Equation (4) defines $PoPC(m)$ as the Parameters Coverage for post-conditions of method m.

$$PoRC(m)=\begin{cases} 1 & \text{(return value is held by post-condition)} \\ 0 & \text{(otherwise)} \end{cases} \tag{4}$$

Definition 4.5 (PoAC)
Let $A_{assigned}(m)$ and $A_{held-by-post}(m)$ be a set of attributes assigned in method m and held by the post-condition in method m, respectively. Equation (5) defines $PoAC(m)$ as the Assigned Attributes Coverage for post-conditions of method m.

$$PoAC(m) = \frac{\mid A_{held-by-post}(m) \mid}{\mid A_{assigned}(m) \mid} \tag{5}$$

Definition 4.6 (PoGC)
Let $G_{assigned}(m)$ and $G_{held-by-post}(m)$ be a set of global variables assigned in method m and held by the post-condition in method m, respectively. Equation (6) defines $PoGC(m)$ as the Assigned Global Variables Coverage for post-conditions of method m.

$$PoGC(m) = \frac{\mid G_{held-by-post}(m) \mid}{\mid G_{assigned}(m) \mid} \tag{6}$$

4.4.3 The Coverage for Assignables

Assignable constraints are written on methods or constructors. Some variables are assigned in the method or constructor, including attributes that have their scope outside of the method. Thus, the coverage for assignables includes Assigned Attributes Coverage.

Definition 4.7 (AAC)
Let $A_{assigned}(m)$ and $A_{held-by-asgn}(m)$ be a set of attributes assigned in method m and held by the assignable in method m, respectively. Equation (7) defines $AAC(m)$ as the Assigned Attributes Coverage for the assignable of method m.

$$AAC(m) = \frac{\mid A_{held-by-asgn}(m) \mid}{\mid A_{assigned}(m) \mid} \tag{7}$$

4.4.4 The Coverage for Invariants

Class invariants are described in a class. The variables owned by the classes are attributes. Hence, coverage for invariants has Attributes Coverage for invariants.

Definition 4.8 (IAC)
Let $A(c)$ and $A_{held-by-inv}(c)$ be a set of attributes owned by class c and held by the invariants in class c, respectively. Equation (8) defines $IAC(c)$ as the Attributes Coverage for invariants of class c.

$$IAC(c) = \frac{\mid A_{held-by-inv}(c) \mid}{\mid A(c) \mid} \tag{8}$$

4.4.5 The Coverage for Pre-conditions and Invariants

Definition 4.9 (PrIAC)
Let us assume that Class c owns method m. Also, let $A_{referred}(m)$, $A_{hold-by-pre}(m)$, and $A_{hold-by-inv}(c)$ be a set of attributes referred in method m, held by the pre-condition in method m, and held by invariants in class c, respectively. Equation (9) defines $PrIAC(m)$ as the Referred Attributes Coverage for pre-conditions and invariants of method m.

$$PrIAC(m) = \frac{PrIACNR(m)}{\mid A_{referred}(m) \mid} \tag{9}$$

where $PrIACNR(m) =$
$\mid A_{referred}(m) \cap (A_{held-by-pre}(m) \cup A_{held-by-inv}(c)) \mid$

4.4.6 The Coverage for Post-conditions and Invariants

Definition 4.10 (PoIAC)
Let us assume that Class c owns method m. Let $A_{assigned}(m)$, $A_{hold-by-post}(m)$, and $A_{hold-by-inv}(c)$ be a set of attributes referred in method m, held by the post-condition in method m, and held by the invariants in class c, respectively. Equation (10) defines $PoIAC(m)$ as the Assigned Attributes Coverage for post-conditions and invariants of method m.

$$PoIAC(m) = \frac{PoIACNR(m)}{\mid A_{assigned}(m) \mid} \tag{10}$$

where $PoIACNR(m) =$
$\mid A_{assigned}(m) \cap (A_{held-by-post}(m) \cup A_{held-by-inv}(c)) \mid$

4.4.7 Ignored Variables

Constants are ignored when measuring the coverage because such variables do not affect the communication among methods. For example, in Java, the variables described by `final` modifier are ignored.

5 EVALUATION

This section gives our experimental evaluations and the results.

5.1 Overview

We performed experiments using our prototype tool to evaluate our proposed coverage metrics. We measured (1) execution times, and (2) numeric results of our proposed coverage. Here is the experimental environment; HP Z800 Workstation (Xeon E5607 dual core 2.27 GHz, 2.26 GHz and main memory 32 GB), Windows 7 Professional for 64 bits with Service Pack 1 and Java Version 1.7.

5.2 Target Programs

We applied our approach to three programs: The Warehouse Management Program (WMP) [20], HealthCard (HC) [21], [22], and the Syllabus Management System for a university (SMS). Table 3 summarizes the target programs including the size of the programs and available assertion types of each program.

Table 3: Target Programs

Target Program	N	Available JML Assertions
WMP	53	requires,ensures, assignable,invariant
HC	197	requires,ensures,assignable
SMS	562	requires,ensures

N = The number of target methods and constructors

WMP was developed by an ex-member of our research group. This program has requires, ensures, assignables and invariants, and they all passed the static checker, ESC/Java2.

HC is a medical appointment application written as a master's thesis by Ricardo Rodrigues at the University of Madeira. The application is based on JavaCard, the platform of IC card devices. In general, the embedded systems need stricter quality because it is difficult to update their software. HC has two versions: a running version and a JML version. We utilize the JML version as the experimental target because the JML version contains more JML descriptions than does the running version. The HC program has no `@invariant` in JML because `model` is used instead of `@invariant`. Thus, in this evaluation, Attributes Coverage for invariants is not measured.

SMS is implemented in Java by a software company as an educational resource for the IT Specialist Program Initiative for Reality-based Advanced Learning (IT Spiral), a national educational project lead by MEXT. Members of our research group added only pre-conditions and post-conditions in JML to the system, and the system produced no violations by jml4c, a runtime checker.

We added the standard libraries (*e.g.*, `java.lang.Object`) with JML descriptions [15] to the target programs. Thus, the contracts of the superclass or interface are added to the class that inherits a class or implements an interface. For example, the contracts of `java.lang.Object#toString()` are added to all the `toString()` methods. Additionally, the results of coverage do not include the methods of the standard libraries. Furthermore, we excluded abstract classes, interfaces, test classes and the main method because they should not necessarily have contracts.

The JML annotations on each experimental target are described based on judgements of each developer. Hence, there is no common policy for describing annotations between the experimental targets.

5.3 Results of Execution Times

Table 4 shows the results of the execution times. We measured three execution times for each program and show their average in the table.

Table 4: Execution Times

Target Program	Execution Time
WMP	9.3 sec
HC	16.0 sec
SMS	14.0 sec

5.4 Results of Variable Coverage

Tables 5, 6, and 7 show the results of the coverages for pre-conditions, for post-conditions, and for assignables, respectively.

Table 5: Results of Coverage for Pre-conditions

Target Program	PrPC	PrAC	PrIAC
WMP	99.17%	9.09%	96.97%
HC	79.22%	46.24%	NA
SMS	41.82%	2.77%	NA

Table 6: Results of Coverage for Post-conditions

Target Program	PoRC	PoAC	PoIAC
WMP	100.00%	94.12%	100.00%
HC	84.11%	48.39%	NA
SMS	99.68%	99.38%	NA

Table 8 shows the results of the coverage of invariants for WMS.

6 DISCUSSION

This section discusses the experimental results and the threats to validity.

6.1 Warehouse Management Program

The following method does not cover Parameter Coverage for pre-conditions:

```
StockManagement.Request#
Request(java.lang.String, int,
   StockManagement.Customer, java.util.Date,
   byte).
```

We found that parameter `rqst` is not covered by `requires` in the source code of the constructor `Request`. The byte-type parameter `rqst` means the request state instead of Enum, as `SHORTAGE=0`, `SATISFYED=1`, `DELIVERED=2`, `WAIT=3`. Therefore, the constraints of class `Request` in JML are lacking because the attribute `rqst` must be any of 0 to 3.

Table 6 shows that every return value is held by its post-conditions. No problem was found when we read the source code and JML.

The following method does not cover the Assigned Attributes Coverage for post-conditions:

```
StockManagement.ReceiptionDesk#
   ReceiptionDesk().
```

Developers who described the source code and JML seemed to recognize the shortage of post-conditions, because the comment "ensures are included in invariants" is in the source code (Fig. 4).

Table 7: Results of Coverage for Assignables

Target Program	AAC
WMP	100.00%
HC	41.94%
SMS	NA

Table 8: Results of Coverage for Invariants (WMP)

Class Name	P	IAC
ContainerItem	3 / 3	100.00%
Customer	3 / 3	100.00%
Item	2 / 2	100.00%
ReceiptionDesk	2 / 2	100.00%
Request	4 / 6	66.67%
StockState	NA	NA
Storage	3 / 3	100.00%

P=The number of attributes held by invariants / the Number of attributes

```
1 //ensures are included in invariants.
2 //@ public behavior
3 //@   assignable requestList, storage;
4 public ReceiptionDesk() {
5   requestList = new LinkedList();
6   storage = new Storage();
7 }
```

Figure 4: Constructor ReceptionDesk That Is Not Covered by Post-conditions

In the source code of class `ReceptionDesk` (Fig. 5), attributes `requestList` and `storage` are held by invariants.

Also, the result of Assigned Attributes Coverage for post-conditions and invariants is 100%. Even if Assigned Attributes Coverage for post-conditions is low, we can conclude that the source code does not have a problem because the value of Assigned Attributes Coverage for post-conditions is high. Hence, VC helps us to clarify that the source code does not have a problem.

As in the case of class `ReceptionDesk`, it is difficult to know the reason why post-conditions are omitted in a general case. One solution is the designer should describe a comment or some keyword when the post-conditions are included in the class invariants.

Table 7 shows that all assigned attributes are held by assignables. Therefore, we can see that every assignable is described correctly in WMP.

Table 8 shows that Attributes Coverage for invariants of most of classes is 100%, but the coverage of class `Request` is 66%. Class `Request` has six attributes, but two of them are not held by invariant constraints. We found that attributes `deliveringDate` and `requestState` in class `Request` are the cause. `deliveringDate` is defined as the `java.util.Date` type field, which is the date of delivery. Any field of type `java.util.Date` except for `deliveringDate` in class `Request` has a constraint "the

```
public class ReceiptionDesk {
  private /*@ spec_public non_null @*/ List
      requestList;
  private /*@ spec_public non_null @*/ Storage
      storage;
  //@ public invariant \typeof(requestList) ==
      \type(Request);
    ...
}
```

Figure 5: Invariants in Class ReceptionDesk

Table 9: Extracted Results of Coverage for HC

T	N	PrPC	PrAC	PoRC	PoAC	AAC
(1)	197	79.22 %	46.24 %	84.11%	48.39%	41.94 %
(2)	38	82.61%	42.86 %	88.89%	NA	NA

T=The Type of Targets
N=The Number of Targets
(1):All methods and constructors
(2):Except for constructors, setters and getters

field is not null." Thus, the implementor has no insight into the constraints of `deliveringDate` because `deliveringDate` can be null before delivery. The same is true with respect to field `requestState`. Figure 6 shows our recommended revised version of constraints based on the results.

```
1  public class Request implements Comparable {
2    private /*@ spec_public non_null @*/ Date
         receiptionDate;
3    private /*@ spec_public non_null @*/ String
         itemName;
4    private /*@ spec_public @*/ int amount;
5    private /*@ spec_public non_null @*/
         Customer customer;
6
7    private byte requestState;
8    private Date deliveringDate;
9    //@invariant
10     (requestState != delivered &&
           deliveringDate == null) ||
11     (requestState == delivered &&
           deliveringDate != null);
12   ...
13 }
```

Figure 6: Class Request with Recommended JML Revisions

6.2 HealthCard

From the manual inspection we conclude that the JML assertion for HC is described in the following way. No constructors have a JML description because the JML description is on the interface. Setters and getters have no JML description. We discuss constructors, setters and getters later. Table 9 lists the results of HC except for constructors, setters and getters.

According to Table 9, the following methods have no preconditions with their parameters even though they are neither setters/getters nor constructors:

- `commons.CardUtil#byte[] clone(byte[])`
- `commons.CardUtil#void cleanField(byte[])`
- `commons.CardUtil #boolean validateObjectArrayPosition (java.lang.Object[], short)`
- `commons.CardUtil #short countNotNullObjects (java.lang.Object[])`

The parameters of the methods are array type, and any caller or any callee does not guarantee that each of the parameters is not null. We found the shortage of JML descriptions by applying Variable Coverage. In addition, the methods do not check whether the parameters inside are null. `NullPointerException` is thrown when the parameter array is `null`. The result shows that these methods have potential bugs.

Also, this program has a method with comments in natural language instead of JML constraints. Figure 7 shows the source code of method `validateObjectArrayPosition` of class `CardUtil`. Line 1 in the figure indicates that the developers know the lack of JML descriptions. We consider, as future work, that we could infer contracts from useful comments.

```
1  //Returns false if position points to a null
       value or if position is out of bounds.
2  //@ assignable \nothing;
3  public /*@ pure @*/ static boolean
       validateObjectArrayPosition (Object[]
       array, short position) {
4    if(position < 0 || position >=
         countNotNullObjects(array))
5      return false;
6    else
7      return true;
8  }
```

Figure 7: Comments Instead of Contracts

For Referred Attributes Coverage for pre-conditions, the results of 23 methods are not full coverage. The results of 8 of 23 of the methods with `toString` are eliminated because their source code has the comment, "Testing code."

For the other 15 methods, we explain the method `validateAllergyPosition`. It does nothing other than call utility method `validateObjectArrayPosition` of class `CardUtil` (Fig. 8).

```
1  public boolean validateAllergyPosition(short
       position){
2    return CardUtil.validateObjectArrayPosition(
         this.allergies, position);
3  }
```

Figure 8: Source Code of Method validateAllergyPosition

It is preferable that contract violations are produced in a previous step than over a later step because it is easier to identify bugs. Thus, methods `validateAllergyPosition`

and validateVaccinePosition should be written with more JML descriptions.

For Return Value Coverage for post-conditions, analogous with pre-conditions, the following methods have no post-conditions even though they have neither setters/getters nor constructors:

- commons.CardUtil#byte[] clone(byte[])
- commons.CardUtil #short countNotNullObjects (java.lang.Object[])
- commons.CardUtil #boolean validateObjectArrayPosition (java.lang.Object[],short)

The JML descriptions of the methods can be improved. For method clone, we recommend the post-condition @ensures \result != null. Also, we recommend the post-condition for method validateObjectArrayPosition idea in Figure 9, based on its comment "//Returns false if position points to a null value or if position is out of bounds."

```
/*@ ensures
    (\result == false)  ==>
    (array == null ||
     position <= 0 || position >=
         countNotNullObjects(array))
  @*/
```

Figure 9: Recommended Post-condition of Method validateObjectArrayPosition

For method countNotNullObjects, we suggest @ensures \result >= 0;.

About the Assigned Attributes Coverage for post-conditions, the result is not available because no methods assign attributes.

No methods assign attributes except constructors, setters and getters.

In general, constructors and setters tend to change the attributes. Although every getter does not change the attributes, its return value is used by other methods. To guarantee the behavior of the class, constructors, setters and getters should have JML descriptions.

We recommend that developers describe the JML description of constructors, setters and getters, as in Figure 10. For setters, developers should write pre-conditions that mean that the parameters equal the attributes assigned. For getters, developers should write post-conditions that means that the return value equals the attributes returned.

6.3 Syllabus Management System

The parameters of 207 methods are not held by pre-conditions; 144 of them are setters, and 63 are others. As an instance of setters, Figure 11 shows the source code of method setJugyouKamoku of class JikanwariJugyouKamokuDTO.
When parameter jugyouKamoku is null, the attribute jugyouKamoku is set to null.

```
public class Person {
  private String name;

  //@requires name != null;
  //@ensures this.name == name;
  public Person(String name) {
    this.name = name;
  }

  //@requires name != null;
  //@ensures this.name == name;
  public void setName(String name) {
    this.name = name;
  }

  //@ensures \result == this.name;
  //@assignable nothing;
  public String getName() {
    return this.name;
  }

}
```

Figure 10: Recommended Source Code with JML of Setter and Getter

If method setJugyouKamoku is called again, the null reference occurs at line 2. Thus, the pre-condition should have the jugyouKamoku != null constraint for parameter jugyouKamoku.

```
1 //@ ensures  this.jugyouKamoku.equals(
      jugyouKamoku);
2 public void setJugyouKamoku(final JugyouKamoku
       jugyouKamoku) {
3   this.jugyouKamoku = jugyouKamoku;
4 }
```

Figure 11: An Example for Setter of SMS

Only the following method does not have full coverage for Return Value Coverage for post-Conditions:
service.UserServiceImpl#
boolean authenticate(java.lang.String,
java.lang.String, entity.UserKubun)
The method authenticate of class UserServiceImpl returns true or false depending on its parameters. We found no post-condition in its source code, but whether constraints are needed or just forgotten is difficult to distinguish. Therefore, for such a method, we recommend writing explicit contracts to single out oversights:
ensures \result == true|false;.

For Assigned Attributes Coverage for post-conditions, the result of the following method is not held by post-conditions:
entity.Soshiki # void add(entity.Soshiki)
Figure 12 shows the source code of the method add of class Soshiki. The post-condition in line 3 calls the getter method getKaiSoshiki. From the source code of the getter (Figure 13), the getter just returns the attribute kaiShoshiki without changing it. We recommend using ensures this.kaiSoshiki.contains(soshiki); instead of line 3.

Calling the setter of the attribute in the methods is the same as assigning the attribute. For example, line 5 in Figure 14 is

```
1 //@ requires soshiki != null;
2 //@ ensures this.getKaiSoshiki().contains(
      soshiki);
3 public void add(final Soshiki soshiki) {
4   if (getKaiSoshiki() == null) {
5     this.kaiSoshiki = new LinkedHashSet<
          Soshiki>();
6   }
7   soshiki.setJouiSoshiki(this);
8   getKaiSoshiki().add(soshiki);
9 }
```

Figure 12: Added Source Code of Method of Class Soshiki

```
1 //@ ensures (this.kaiSoshiki != null) ? (this.
      kaiSoshiki.size() == \result.size()) && (
      \forall Soshiki s; this.kaiSoshiki.
      contains(s); \result.contains(s)) : \
      result == null;
2 // anotation OneToMany(cascade = CascadeType.
      ALL, targetEntity = Soshiki.class,
      mappedBy = "jouiSoshiki")
3 public Set<Soshiki> getKaiSoshiki() {
4   return this.kaiSoshiki;
5 }
```

Figure 13: Source Code of Method getKaiSoshiki of Class Soshiki

equivalent to assigning the attribute `SESSION`. Assigned Attributes Coverage should also be extended to a target calling the setter of the attribute.

```
1 public static Session currentSession() {
2   Session s = SESSION.get();
3   if (s == null) {
4     s = SESSION_FACTORY.openSession();
5     SESSION.set(s);
6   }
7   return s;
8 }
```

Figure 14: Example of the Unmonitored Case of Assigning an Attribute

6.4 The cost of writing additional annotations

The case studies revealed that the proposed metrics could contribute to detect shortage of JML annotations. However, it may require a certain amount of effort to fix the shortage of the annotations. We believe that the amount of annotations is important to evaluate the safety of software systems. We also propose our method considering the scalability. It should be easy to calculate for real software. However, redundant annotations should not increase the safety even though they require a certain amount of effort to be described. Hence, it is necessary to judge whether the shortage of annotations detected by the proposed metrics really required to be fixed.

7 CONCLUSION

This paper proposed Variable Coverage, a set of metrics for the exhaustiveness of specification with source code based on Design by Contract. Our proposed coverage observes variables depending on constraints. We applied our approach to three programs to evaluate the ability of Variable Coverage to find variables that should have been referred in specifications as important variables. As a result, we found a shortage of JML annotations in the target programs, and this shows the usefulness of our proposed metrics.

Future work includes inferring the constraints. The first approach is suggesting constraints from the comments in the source code. The second approach is using the modifiers of a method; static methods should not have assignable clauses except for static variables. This means no attributes are permitted to be assigned, because static methods do not change the internal state, (*i.e.*, attributes). Such a modifier helps to generate helpful assertions.

8 FUTURE WORK

In order to reduce the verification time, we proposed the metrics that only consider the amount of annotations. However, we plan to make the proposed metrics take the quality of annotations into account. Currently, the proposed metrics consider only the amount of annotations. They do not consider the quality of the annotations. Hence, the high values of the proposed metrics may not be directly linked to the safety of software systems. Therefore, it will improve the usefulness of the proposed metrics to consider the quality of annotations. We add these explanations in Section 8 as our future work.

ACKNOWLEDGMENTS

This work is being conducted with the support of Grants-in-Aid for Scientific Research C(21500036) and S(25220003).

REFERENCES

[1] E.M. Clarke, and J.M. Wing, “Formal Methods: State of the Art and Future Directions,” ACM Computing Surveys, Vol.28, No.4, pp.626–643 (1996).

[2] J.R. Abrial, “Formal Methods in Industry,” Proceeding of the 28th international conference on Software engineering - ICSE ’06, pp.761–768 (2006).

[3] B. Meyer, “Applying ‘Design by Contract’,” IEEE Computer, Vol.25, No.10, pp.40–51 (1992).

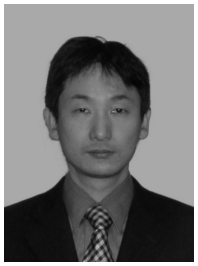
[4] G.T. Leavens, A.L. Baker, and C. Ruby, “JML: A Notation for Detailed Design,” Behavioral Specifications of Businesses and Systems, pp.175–188 (1999).

[5] P. Chalin, P.R. James, and G. Karabotsos, “JML4: Towards an Industrial Grade IVE for Java and Next Generation Research Platform for JML,” Proceeding VSTTE ’08 Proceedings of the 2nd international conference on Verified Software: Theories, Tools, Experiments, eds. N. Shankar, and J. Woodcock, Vol.5295, pp.70–83 (2008).

[6] D.R. Cok, “OpenJML: JML for Java 7 by extending OpenJDK,” Proceeding NFM’11 Proceedings of the Third international conference on NASA Formal methods, pp.472–479 (2011).

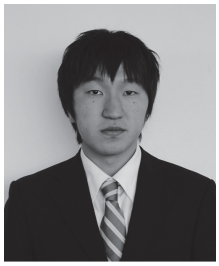
[7] D.R. Cok, and J.R. Kiniry, "ESC/Java2: Uniting ESC/-Java and JML," International Workshop on Construction and Analysis of Safe Secure and Interoperable Smart Devices CASSIS 2004, Vol.3362, pp.108–128 (2004).
[8] A. Sarcar, "A New Eclipse-Based JML Compiler Built Using AST Merging," 2010 Second World Congress on Software Engineering, pp.287–292 (2010).
[9] M. Barnett, K.R.M. Leino, and Schulte, "The Spec# Programming System: An Overview," Construction and Analysis of Safe, Secure, and Interoperable Smart Devices, eds. G. Barthe, L. Burdy, M. Huisman, J.L. Lanet, and T. Muntean, Vol.3362, pp.49–69 (2005).
[10] M. Barnett, B.Y.E. Chang, R. DeLine, B. Jacobs, and K.R.M. Leino, "Boogie: A Modular Reusable Verifier for Object-Oriented Programs," 4th International Symposium, FMCO 2005, Vol.4111, pp.364–387 (2006).
[11] H. Chockler, O. Kupferman, and M. Vardi, "Coverage Metrics for Formal Verification," International Journal on Software Tools for Technology Transfer, Vol.8, No.4-5, pp.373–386 (2006).
[12] Dinos Moundanos, J.A. Abraham, and Y.V. Hoskote, "Abstraction Techniques for Validation Coverage Analysis and Test Generation," IEEE Transactions on Computers, Vol.47, No.1, pp.2–14 (1998).
[13] B. Meyer, Eiffel : The Language (Prentice Hall Object-Oriented Series), Prentice Hall (1991).
[14] C. Flanagan, K.R.M. Leino, M. Lillibridge, G. Nelson, J.B. Saxe, and R. Stata, "Extended Static Checking for Java," ACM SIGPLAN Notices, Vol.37, No.5, pp.234 (2002).
[15] KindSoftware, "ESC/Java2" .
[16] J. Kiniry, P. Chalin, C. Hurlin, B. Meyer, and J. Woodcock, "Integrating Static Checking and Interactive Verification: Supporting Multiple Theories and Provers in Verification," in VERIFIED SOFTWARE: THEORIES, TOOLS, EXPERIMENTS, eds. B. Meyer, and J. Woodcock, Vol.4171 of Lecture Notes in Computer Science, pp.153–160 (2008).
[17] E. Foundation, "Eclipse" .
[18] S. Tasiran, and K. Keutzer, "Coverage Metrics for Functional Validation of Hardware Designs," IEEE Design & Test of Computers, Vol.18, No.4, pp.36–45 (2001).
[19] H. Foster, D. Lacey, and A. Krolnik, Assertion-Based Design, SpringerSecond edition (2004).
[20] M. Owashi, K. Okano, and S. Kusumoto, "Design of Warehouse Management Program in JML and Its Verification with ESC/Java2 (in Japanese)," The Transactions of the Institute of Electronics, Information and Communication Engineers D, Vol.91, No.11, pp.2719–2720 (2008).
[21] R.M.S. Rodrigues, "JML-Based formal development of a Java card application for managing medical appointments," University of Madeira (2009).
[22] R.M.S. Rodrigues, "HealthCard" .

(Received December 10, 2013)
(Revised July 8, 2014)



Kozo Okano received his BE, ME, and PhD degrees in information and computer sciences from Osaka University in 1990, 1992, and 1995, respectively. Since 2002, he has been an associate professor at the Graduate School of Information Science and Technology of Osaka University. In 2002, he was a visiting researcher at the Department of Computer Science of the University of Kent in Canterbury, England. In 2003, he was a visiting lecturer at the School of Computer Science of the University of Birmingham, England. His current research interests include formal methods for software and information system design. He is a member of IEEE_CS, IEICE of Japan, and IPS of Japan.

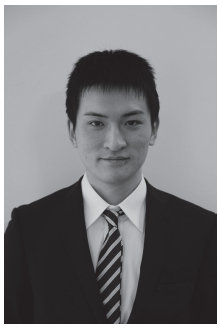
Yuko Muto received her BI and ME degrees from Osaka University in 2010 and 2012, respectively. She now works at Microsoft Ltd.



Yukihiro Sasaki received his BI degree from Osaka University in 2012. He is now a master's course student at Osaka University. His research interests include automatic test case generation, especially for the dynamic generation of assertion.

Takafumi Ohta received his BI degree from Tohoku University in 2013. He is now a master's course student at Osaka University. His research interests include bug identification using concolic execution.

Shinji Kusumoto received his BE, ME, and DE degrees in information and computer sciences from Osaka University in 1988, 1990, and 1993, respectively. He is currently a professor at the Graduate School of Information Science and Technology of Osaka University. His research interests include software metrics and software quality assurance techniques. He is a member of the IEEE, the IEEE Computer Society, IPSJ, IEICE, and JFPUG.



Kazuki Yoshioka received his BI and ME degrees from Osaka University in 2011 and 2013, respectively. He now works at Hitachi Ltd.