Variable Coverage: A Metric to Evaluate the Exhaustiveness for Program Specifications Based on DbC

Yuko Muto†, Yukihiro Sasaki†, Takafumi Ohta†, Kozo Okano†, 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 dependability and maintainability of software, Design by Contract is one of useful notions, which utilizes constraints as contracts between the caller and the callee routines in programs. Some verifiers for programs are able to check whether given source code satisfies given constraints. It is, however, hard to measure the exhaustiveness for specification, *i.e.*, how much constraints cover ideal specification for the source code. This paper proposes Variable Coverage, a simple set of metrics to check the exhaustiveness of specification with source code for Java and other Object-Oriented programming languages. The proposed coverage observes occurrence of variables in constraints, such that the variables are also used in the target method/constructor. We applied the metrics to three concrete programs in order to evaluate that Variable Coverage is able to help to find variables which should have been referred in specifications as important variables. As a result, we found some shortage of JML annotations in target programs, which shows the usefulness of the proposed metrics.

Keywords: DbC, Coverage, Specification,Testing, Metrics

1 Introduction

Formal methods[1], 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 as increasing the size of software. The larger program sizes, the more frequently software testing misses corner-case. On the other hand, formal methods have potential for exhaustive checking. In the industry, some real large projects succeeded ussing formal methods, such as the public transportation systems[2]. Formal methods are classified into three technologies: deductive methods, model checking, model-based simulation or testing.

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

It is, however, hard to determine whether the speficication is well-written (exhaustive) or not. If the specification is low exhaustive, the correctness of the program is not clear. Take runtime checking as an example. A runtime checker produces a violation when source code and its specification do not meet. No violation is reported by runtime checkers if the code has no specification because there is not any constraint to check. Consequently we cannot anything about the quality of the source code.

Some papers have studied coverage metrics for hardware verification. Paper[11] summarizes some coverage metrics for simulation-based verification such as code coverage, assertion coverage. In order to generate test efficiently, Paper [12] has proposed functional coverage as the amount of control behaviors covered by the test suite using abstraction techniques. Nevertheless, there are few coverage metrics which can be applied to general purpose programming languages at the implementation level such as 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 pre-condition, post-condition, assignable and invariant.

We have experimented it to apply the three kinds of programs using a prototype which measures Variable Coverage. As a result, we found some shortage of JML annotations in the target programs, which shows the usefulness of our proposed metrics.

The paper is organized as follows. Section 2 provides the definitions of some words as preliminary, and Section 3 mentions the related works. Section 4 will show 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 the definition of as preliminaries.

2.1 Design by Contract

Design by Contract (DbC) is a notion proposed by Bertrand Meyer [3]. In DbC, suppliers (caller routine) and clients (callee routine) make contracts each other. Clients should satisfy the pre-conditions, and suppliers should satisfy the post-conditions under pre-conditions. This mechanism makes it easy to identify bugs.

Some programming languages support DbC as standard, others have the specification language separately from the core grammer of the language. Eiffel[13] supports DbC as 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, Java Modeling Language[4] is the de-facto standard specification language.

2.2 Constraints

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 pre-condition.

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

The routine is permitted to assign values to only the variables specified in Assignable. The constrains provide to detect side effects for developers.

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

2.3 Java Modeling Language

Java Modeling Language (JML) is a specification language based on Design by Contract for Java. JML supports pre-condition, post-condition, assignable and invariant. We explain them through class `BankAccount`, an account for customer of a bank, as example.

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

Pre-conditions Keyword `@requires` is used to express the pre-condition. In Fig. 1, methods `withdraw` and `deposit` have pre-conditions at lines 12,13 and 20.

Post-conditions Keyword `@ensures` means 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 instance creation.

Assignables `@assignable` is used to express assignable. The following `@assignable` clases, fields which can be assigned are listed. If every field is prohibited to be assigned, describe `@assignable \nothing` like line 28 in Fig. 1. Furthermore, `@pure` is equivalent to `@assignable \nothing`; this is used to make it short.

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

2.4 Global Variables

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

```
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

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. A variable `font` of class `Config` is a global variable for method `draw`.

3 Related Work

This section introduces some works related to this paper.

3.1 Program Verification

ESC/Java[16], an Extended Static Checker for Java, is the practical usable checker among early verifiers. Currently, its successor version, ESC/Java2[17] is widely used, and it supports JML2.

Supporting the newer Java, Mobius[18] attracts rising attention as a program verification environment (PVE), including static checkers, runtime checkers and verifiers. It is provided as an Eclipse[19] plug-in. ESC/Java2 is also integrated into Mobius.

```
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

3.2 Verification Coverage

Coverage metrics for formal verification are called verification coverage in mainly the field of hardware. Verification coverage falls into two categories: syntactic coverage and semantic coverage[11]. As syntactic coverage, code coverage for model-based simulation is the metrics derived from software testing[20]. The ratio of executed code during the simulation is code coverage. As simple coverage, line coverage, the code of block without control transition.

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

For semantic coverage, there are functional coverage and assertion coverage. Assertion coverage is the measuring method which users determine variables which to observe. The assertion coverage measures what assertions are covered with a given set of input sequence[11].

In order to generate test suite and analyze it, paper [12] proposed functional coverage which is the amount of control behavior covered by the test suite using abstraction techniques.

3.3 Assertion Density

Assertion density is the number of assertions per line of code[21]. Without sufficient assertion density, the full benefits of assertions are not realized. Assertions must be verified, which are behaviors as design intents, *i.e.*, statements for properties.

4 Variable Coverage

This section defines Variable Coverage, our proposed method.

4.1 Motivation

Formal verification checks consistency between source code and its constracts based on Class Correctness formula. Paper[11] also states 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." Applying the idea to software, the input sequences of a method/constructor correspond to variables. Consequently, we propose a coverage metric which observes variables.

4.2 Policies

We propose a set of metrics which supports these policies:

1. Our metric checks all variables as input and output.
   It is oriented from verification coverage.
2. Our metric is simple.
   The execution of measuring the coverage requires enough short time. The metric targets developers who describe assertions in JML. Our metric should be checked in short time on a frequent basis when they want to check.
3. Our metric uses only static information.
   Using only static information (source code and JML) without execution trace enables to measure coverage for a part of incomplete code.

4.3 Constraints Development Process with Variable Coverage

Quickly measuring Variable Coverage (VC in short) enables to high-frequently measure it. Implementators can improve constraints description by the iterative process:

Step1 Implementators describe assertions

Step2 VC is measured

Step3 Iterate Step1 if implementators find lack of their assetions

We call such an iteration "Quick VC revise."

4.4 Definition of Variable Coverage

VC consits of four kinds of metrics, including coverage for pre-condition, post-condition, assignable and invariant. Tables 1 and 2 show 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 refered in the method. Thus, the Coverage for Pre-conditions consists of Parameters Coverage, and Refered 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 pre-condition in method m, repsctively. Equation (1) defines $PrPC(m)$, Parameters Coverage for pre-conditions of method m.

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

For Fig. 3, $\mid P_{held-by-pre}(m) \mid = \mid \{age\} \mid = 1$, 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 pre-condition in method m, respectively. Equation (2) defines $PrAC(m)$, Referred Attributes Coverage for pre-conditions of method m.

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

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 (multi-constraint)

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

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 pre-condition in method m, respectively. Equation (3) defines $PrGC(m)$, 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 which affect the outside of method, *i.e.*, return value, attributes and global variables assigned in the method. Hence, the coverage for post-condition is composed of Return Value Coverage, Assigned Attributes Coverage and Assigned Global Variables Coverage.

Definition 4.4 (PoRc)
Equation (4) defines $PoPC(m)$, 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 post-condition in method m, respectively. Equation (5) defines $PoAC(m)$, 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 post-condition in method m, respectively. Equation (6) defines $PoGC(m)$, 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

Constraints assignable are written on methods or constructors. Some variables are assigned in the method or constructor, among them attributes have the scope of method outside. Thus, coverage for assignable 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 assignable in method m, respectively. Equation (7) defines $AAC(m)$, Assigned Attributes Coverage for 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 classes are attributes. Hence, coverage for invariants has Attributes Coverage for invariant.

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 invariants in class c, respectively. Equation (8) defines $IAC(c)$, Attributes Coverage for invariant of

class c.

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

4.4.5 The Coverage for Pre-conditions and Invariants

Definition 4.9 (PrIAC)
Let 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 pre-condition in method m, and held by invariants in class c, respectively. Equation (9) defines $PrIAC(m)$, Referred Attributes Coverage for pre-conditions and invariants of method m.

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

, 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 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 post-condition in method m, and held by invariants in class c, respectively. Equation (10) defines $PoIAC(m)$, Assigned Attributes Coverage for post-conditions and invariants of method m.

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

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

5 Evaluation

This section gives the experimental evaluation and the results.

5.1 Overview

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

5.2 Target Programs

We apply our approach to three programs: Warehouse Management Program (WMP)[22], HealthCard (HC)[23], [24], and Syllabus Management System for a university (SMS). Table 3 summerizes the target programs including the size of programs and available assertion types which the program has.

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 is developped by an ex-member of our research group. This program has all of `requires`, ensures, assignables and invariants, and they are whole passed by the static checker, ESC/Java2.

HC is a medical appointment application which is written as a master thesis work by Joao Pestana Ricardo Rodrigues from University of Madeira. It is based on JavaCard, the platform of IC card devices. In general, the embeded systems need more strictly quality because it is hard to update their software. The HealthCard has two versions: running version and JML version. We utilize JML version as experimental target because JML version contains more JML description than running version. HealthCard program has no `@invariant` in JML because `model` is used instead of `@invariant`. Thus, in this evaluation, Attributes Coverage for Invariant are not measured.

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

We add the standard libraries (*e.g.*, `java.lang.Object`) with JML descriptions[17] into target programs. Thus, into the class which inherits a class or implements a interface, the contracts of its superclass or interface are added. For example, the contracts of `java.lang.Object#toString()` are added into all methods `toString()`. As well, 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 have necessarilly contracts.

5.3 Results of Execution Times

Table 4 shows the results of execution times. We measured threr excecution times for each program; it shows the average of them.

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 coverages for pre-conditions, for post-conditions, and for assignable, 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 coverage of invariant for Warehouse Management System.

6 Discussion

This section discuses 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 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`. Threfore, constraints of class `Request` in JML lack because its 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.

For Assigned Attributes Coverage for Post-condition, the following meshod does not cover it:
`StockManagement.ReceiptionDesk# ReceiptionDesk().`

Developers who described the souce code and JML seemd to recognize the shortage of post-condition, because there is a comment “ensures are included in invariants” in the source code (Fig. 4).

Table 7: Results of Coverage for Assignable

Target Program	AAC
WMP	100.00%
HC	41.94%
SMS	NA

Table 8: Results of Coverage for Invariant (Warehouse Management Program)

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 tequestList, storage;
4 public ReceiptionDesk() {
5   tequestList = new LinkedList();
6   storage = new Storage();
7 }
```

Figure 4: Constructor ReceptionDesk which 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-condition and Invariant is 100%. Even if Assigned Attributes Coverage for Post-condition is low, we can conclude that the source code does not have the problem because the value of Assigned Attributes Coverage for Post-condition is high. Hence, VC helps us to clarify that source code has no problem.

Like the case of class `ReceptionDesk`, it is hard 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 rightly in Warehouse Management Program.

Table 8 shows that Attributes Coverage for Invariant of most of classes have 100%, but the coverage of class `Request` is 66%. Class `Request` has six attributes but the two of all are not held by invariant constraints. We found that attributes `deliveringDate` and `requestState` in class `Request`, are the cause. `deliveringDate` is defined as `java.util.Date` type field which means the date of de-

```
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 HealthCard

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

livering. Any field of type `java.util.Date` except for `deliveringDate` in class `Request` has a constraint "the field is not null." Thus, the implementor has no idea about the constraints of `deliveringDate` because `deliveringDate` can be null before delivering. The same is true respect to field `requestState`. Figure 6 shows our suggested revised version of constraints for them 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 JML We Suggest

6.2 HealthCard

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

According to Table 9, the following methods have no precondition with their parameters though they are 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 dose 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 their parametera are null or not in their body. `NullPointerException` is thrown when the parameter array is `null`. It shows that these methods have potential bugs.

Also, there is a method with comments in natural language instead of constraints in JML. 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 it is possible to 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-condition, the results of 23 methods are not full coverage. For methods `toString` ocuppying 8 of 23, they are eliminated from the results because their source code have the comments, "Testing code."

One behalf of other 15 methods, we explain a method `validateAllergyPosition`. It does nothing other than calling 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 at previous step than later because it makes easy to identify bugs. Thus, methods `validateAllergyPosition` and `validateVaccinePosition` should be written more JML descriptions.

For Return Value Coverage for Post-condition, in analogy with pre-conditions, the following methods have no post-conditions in spite they are 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 suggest the post-conditions `@ensures \result != null`. Also, we have the idea like Figure 9 for method `validateObjectArrayPosition`, from 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: Post-condition of Method validateObjectArrayPosition which We Recommend

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

About Assigned Attributes Coverage for Post-condition, the result is not available because there are no methods which assign to the attributes.

There are no methods which assign the attributes except for 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. In order to guarantee the behavior of the class, constructors, setters and getters should have JML desciriptions.

We recommend for developers to describe the JML desciption of constructors, setters and getters like Figure 10. To setters, developers should write pre-condition which means that parameters equals attributes assigned. To getters, developers should write post-condition which means that return value equals 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`.

```
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 this.name;
  public String getName() {
    return this.name;
  }

}
```

Figure 10: Source Code with JML of Setter and Getter We Recommend

When parameter `jugyouKamoku` is `null`, the attribute `jugyouKamoku` is set to `null`.

If method `setJugyouKamoku` are called again, the null reference is occured at line 2. Thus, pre-condition should have the constraints for parameter `jugyouKamoku` which means `jugyouKamoku != null`.

```
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 Syllabus Management System

Only the following method does not have full coverage for Return Value Coverage for Post-Condition:
`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; It is hard to distinguish from forgetting constraints. Therefore, for such a method, we recommend to write explicitly these contract to alternate from oversights:
`ensures \result == true|false;`.

For Assigned Attributes Coverage for Post-condition, the result of the below 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`. Post-condition at Line 3 calls getter method `getKaiSoshiki`. From The source code of the getter (Figure 13), the getter just returns the attribute `kaiShoshiki` without changing it. We recommend to describe

`ensures this.kaiSoshiki.contains(soshiki);` instead of line 3.

```
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: Source Code of Method add 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

Calling the setter of the attribute in the methods is the same as assigning the attribute. For example, line 5 at Figure 14 is equivalent to assigning the attribute `SESSION`. Assigned Attributes Coverage should be extend to target calling the setter of the attribute additionally.

```
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 for the Unmonitored Case of Assigning to An Attribute

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 some variables depending on constraints. We applied our approach to three programs in order to evaluate that Variable Coverage is able to help to find variables which should have been referred in specifications as important variables. As a result, we found some shortage of JML annotations in target programs, which shows the usefulness of our proposed metrics.

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

Acknowledgments

This work is being conducted as a part of Grant-in-Aid for Scientific Research C(21500036) and S(25220003).

REFERENCES

[1] Edmund M. Clarke and Jeannette M. Wing. Formal Methods: State of the Art and Future Directions. *ACM Computing Surveys*, 28(4):626–643, December 1996.

[2] Jean-Raymond Abrial. Formal Methods in Industry. In *Proceeding of the 28th international conference on Software engineering - ICSE '06*, page 761, New York, New York, USA, May 2006. ACM Press.

[3] Bertrand Meyer. Applying 'Design by Contract'. *IEEE Computer*, 25(10):40–51, 1992.

[4] Gary T Leavens, Albert L Baker, and Clyde Ruby. JML: A Notation for Detailed Design. *Behavioral Specifications of Businesses and Systems*, pages 175–188, 1999.

[5] Patrice Chalin, Perry R. James, and George Karabotsos. JML4: Towards an Industrial Grade IVE for Java and Next Generation Research Platform for JML. In Natarajan Shankar and Jim Woodcock, editors, *Proceeding VSTTE '08 Proceedings of the 2nd international conference on Verified Software: Theories, Tools, Experiments*, volume 5295 of *Lecture Notes in Computer Science*, pages 70–83, Berlin, Heidelberg, October 2008. Springer.

[6] David R. Cok. OpenJML: JML for Java 7 by extending OpenJDK. In *Proceeding NFM'11 Proceedings of the Third international conference on NASA Formal methods*, pages 472–479, April 2011.

[7] David R. Cok and Joseph R Kiniry. ESC/Java2: Uniting ESC/Java and JML. In *International Workshop on Construction and Analysis of Safe Secure and Interoperable Smart Devices CASSIS 2004*, volume 3362 of *Lecture Notes in Computer Science*, pages 108–128. Springer, 2004.

[8] Amritam Sarcar. A New Eclipse-Based JML Compiler Built Using AST Merging. In *2010 Second World Congress on Software Engineering*, pages 287–292. IEEE, December 2010.

[9] Mike Barnett, K. Rustan M. Leino, and Schulte. The Spec# Programming System: An Overview. In Gilles Barthe, Lilian Burdy, Marieke Huisman, Jean-Louis Lanet, and Traian Muntean, editors, *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*, volume 3362 of *Lecture Notes in Computer Science*, pages 49–69, Berlin, Heidelberg, 2005. Springer.

[10] Mike Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. Boogie: A Modular Reusable Verifier for Object-Oriented Programs. In *4th International Symposium, FMCO 2005*, volume 4111 of *Lecture Notes in Computer Science*, pages 364–387, 2006.

[11] Hana Chockler, Orna Kupferman, and Moshe Vardi. Coverage Metrics for Formal Verification. *International Journal on Software Tools for Technology Transfer*, 8(4-5):373–386, April 2006.

[12] Dinos Moundanos, Jacob A. Abraham, and Yatin V. Hoskote. Abstraction Techniques for Validation Coverage Analysis and Test Generation. *IEEE Transactions on Computers*, 47(1):2–14, 1998.

[13] Bertrand Meyer. *Eiffel : The Language (Prentice Hall Object-Oriented Series)*. Prentice Hall, 1991.

[14] Charles Antony Richard Hoare. An Axiomatic Basis for Computer Programming. *Communications of the ACM*, 12(10):576–580, October 1969.

[15] Bertrand Meyer. *Object-Oriented Software Construction (2nd Edition)*. Prentice Hall, 2 edition, 2000.

[16] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended Static Checking for Java. *ACM SIGPLAN Notices*, 37(5):234, May 2002.

[17] KindSoftware. ESC/Java2.

[18] Joseph Kiniry, Patrice Chalin, Clément Hurlin, Bertrand Meyer, and Jim Woodcock. Integrating Static Checking and Interactive Verification: Supporting Multiple Theories and Provers in Verification. In Bertrand Meyer and Jim Woodcock, editors, *VERIFIED SOFTWARE: THEORIES, TOOLS, EXPERIMENTS*, volume 4171 of *Lecture Notes in Computer Science*, pages 153–160. Springer, Berlin, Heidelberg, 2008.

[19] Eclipse Foundation. Eclipse.

[20] Serdar Tasiran and Kurt Keutzer. Coverage Metrics for Functional Validation of Hardware Designs. *IEEE Design & Test of Computers*, 18(4):36–45, 2001.

[21] Harry Foster, David Lacey, and Adam Krolnik. *Assertion-Based Design*. Springer, second edition, May 2004.

[22] Masayuki Owashi, Kozo Okano, and Shinji 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*, 91(11):2719–2720, 2008.

[23] Ricardo Miguel Soares Rodrigues. JML-Based formal development of a Java card application for managing medical appointments. *University of Madeira*, 2009.

[24] Ricardo Miguel Soares Rodrigues. HealthCard.