**Regular Paper**

# Effective Derivation of a Mapping of Variables in a Loop Structure

Kozo Okano†, Shinji Kusumoto‡, and Yukihiro Sasaki‡
†Faculty of Engineering, Shinshu University, Japan
‡Graduate School of Information Science and Technology, Osaka University, Japan
okano@cs.shibshu-u.ac.jp, kusumoto@ist.osaka-u.ac.jp

***Abstract*** - Static program analysis enables us to analyze a program without performing an actual execution run, but the analysis of loops is, however, difficult in general. In order to solve this problem, one of existing techniques derives a mapping between variables using regression analysis on data obtained by multiple executions of a program (run history). The technique is a kind of a hybrid approach and when we analyze a complicated loop using the technique, it may derive an incorrect mapping. Our new proposed technique overcomes this problem using recurrence relations. It first obtains a run history and then performs regression analysis on loop iterations and variables based on the run history. It finally derives a recurrence relation on the variables occurring in the loop body. Experiments confirm that it can derive useful mappings that we cannot derive by the existing technique.

***Keywords***: loops, static analysis, recurrence relation, run history, mapping

## 1 INTRODUCTION

In software engineering, especially in the maintenance phase of software, engineers need to understand software by reading or analyzing code. In such situations, program analysis methods can be helpful. Program analysis methods will produce an abstract summary of the behavior of a given fragment of code, usually a function, or a method (in object-oriented programming language). The abstract summary is usually in the form of a formal specification such as Java Modelling Language [1].

Program analysis methods are divided into two categories: (1) static program analysis types and (2) dynamic program analysis types. Static program analysis methods do not need to execute the target program while dynamic program analysis methods will.

Static program analysis methods use many concrete methods such as symbolic execution [2] and model checking [3]. Recently other approaches have emerged. *e.g.*, heuristic methods [4], and automatic predicate abstraction based methods, such as SLAM [5], BLAST [6]. Static program analysis methods using logic sometimes utilize SAT/SMT solvers [7]. SAT/SMT solvers are enhanced SAT solvers with background theories. A SAT solver is a simple solver for satisfiability problems on logical expressions over propositional variables (boolean variables). Some examples of background theories include decision problems on integer expressions and expressions over arrays and tuples (record types). Thus, SAT/SMT solvers can solve decision problems on programs. There are many SAT/SMT solvers including popular solvers are such as Z3 [8] and Yices [9].

For dynamic program analysis, Daikon [10] is a well-known tool. It derives program assertions from data obtained from execution logs of the target programs. The execution is usually performed many times in order to infer accurate assertions. Recently approaches based on regression analysis [11] have been proposed [12]. Le [12] proposed a method that derives a mapping from a family (or a set of sets) of variables to a set of variables before and after a target loop structure using regression analysis. It first executes the target loop multiple times with varying input values. Based on the data obtained by the execution, it then performs regression analysis. The analysis infers a mapping between variables before and after a target loop.

Therefore, it can easily analyze programs with loop structures. However, it can deal with only linear and quadratic mappings. Consequently, it cannot infer an exponent mapping which represents Fibonacci sequences.

Our proposed approach overcomes this problem as follows. First, we perform dynamic program analysis, and then we obtain data on the number $n$ of loop iterations and the program variables. Next, we perform regression analysis on the data and obtain a relation between $n$ and the program variables. Then we construct a recurrence formula on the program variables by static analysis on the loop body. The recurrence formula represents a relation between the program variables for the $n+1$-th and $n$-th loops. We can obtain their closed-form solution of the recurrence formula, which represents the program variables for $n$. By combining the closed-form solution and the results of the regression analysis, we obtain a final mapping representing the mapping between variables before and after a target loop.

The remainder of this paper is organized as follows. Section 2 presents disadvantages of the existing methods as preliminaries. Section 3 gives the proposed method. Sections 4 and 5 show experimental results and discussion, respectively. Finally, Section 6 summarizes this paper.

## 2 PRELIMINARIES

In general, static analysis approaches sometimes have trouble handling loop structures. In model checking, we overcome this problem by using several techniques such as loop unwinding and Craig interpolation for the approximation of loop invariants. In general, such techniques are not omnipotent due to memory limitations and calculating complexity. For this reason, some of the existing methods contrive sev-

eral methods, such as bounded unwinding [13], user-specified time-out mechanisms [14], and so on. In [15], the S2E tool utilizes Path Selection function which enables us to control the termination of a loop with multiple criteria. For example, PathKiller can stop loop iteration up to a user specified number. In another approach, Xie *et al.* [16] proposed a method which returns an *unknown* value for a variable that cannot be analyzed. Le [12] proposed a method based on regression analysis. It can derive a mapping between variables before and after a given loop structure, it claims that the method can derive more accurate mapping than others.

## 2.1 Regression Analysis

Regression analysis is an estimating method for the relationships among variables. It includes many techniques for modeling and analyzing several variables, when the focus is on the relationship between a dependent variable and one or more independent variables.

Regression analysis is usually based on a regression model. Regression models involve the following parameters and variables:

- The unknown parameters, denoted as $\beta$, which may represent scalars or vectors.
- The independent variables, $\mathbf{X}$.
- The dependent variables, $\mathbf{Y}$.

A regression model relates $\mathbf{Y}$ to a function of $\mathbf{X}$ and $\beta$.
$\mathbf{Y} \approx f(\mathbf{X}, \beta)$

In a formal manner, the approximation is typically formalized as $E(\mathbf{Y} \mid \mathbf{X}) = f(X, \beta)$. To carry out regression analysis, the form of the function $f$ must be specified in advance.

## 2.2 Segmented Symbolic Analysis (SSE)

The approach in [12] (SSE for short) uses approximation functions (regression models) shown in Table 1.

$$y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \beta_3 x_1 x_2 + \beta_4 x_1^2 + \beta_5 x_2^2$$
($x_1$ and $x_2 are$ the input and the output, respectively)

There are many cases in which the functions in Table 1 are not applicable. For example, the program for generating Fibonacci numbers in Listing 1 cannot apply the functions in Table 1.

Table 1: Transformation Table for Loops

| Model | example | values for $\beta_0 \sim \beta_5$ |
|---|---|---|
| Constant | $y = 0$ | 0 |
| Simple Linear | $y = x_1$ | 0 except for $\beta_1$ |
| Multiple Linear | $y = 2 \cdot x_1 + x_2$ | $\beta_4 = 0, \beta_5 = 0$ |
| Polynomial Linear | $y = x_1^2 + x_1 \cdot x_2$ if $x_1 > 0$ | |
| Piece-wise Linear | then $y = x_1$ else $y = 3$ | cannot be expressed |

Listing 1: Program for Fibonacci Numbers

```
public class TestFibo {
  public int cf(int n) {
    int current = 0;
    int prev = 1;
    int prevprev = 0;
    if(n > 0){
      for(int i = 0; i < n; i++) {
        current = prev + prevprev;
        System.out.print(current + " ");
        prevprev = prev;
        prev = current;
      }
      return current;
    } else {
      System.err.println(
        "Input is less than 1");
      return -1;
    }
  }
}
```

In general, it is hard to apply regression analysis on Fibonacci numbers because its closed-form solution is exponential to the number of loop iterations and it cannot be represented by the functions in Table 1.

SSE requires preparing many input patterns to output execution logs, which contain a large number of execution paths. Insufficient execution paths lessen the accuracy of the approximation. This is another disadvantage of the approach. In other words, the approach is not useful for a program with many branches in its loop structures, which makes the coverage of the test cases low.

# 3 OUR PROPOSED METHOD

In this section, we describe the differences between the existing method presented in [12] and our proposed method.

Our proposed method derives a mapping between variables before and after the target loop. The approach can easily deal with loop structures.

SSE uses mapping via regression analysis only. It loses precision in the approximation. Our proposed method uses regression analysis only to derive a mapping between arguments (of the given Java method) and the number of iterations of the target loop (of the given Java method). A relation between $n$, the number of iterations of the target loop, and the variables of the loop, is obtained by static analysis used in cooperation with an analysis tool for mathematics like Mathematica. This combination produces approximation with high accuracy.

## 3.1 Outline of Our Proposed Method

Here, we give an outline of our proposed method.

The inputs and the outputs of our method are summarized as follows.

- Input: a Java method with a single loop structure

- Output: an approximation of the loop structure in the method, if successfully generated, otherwise a failure is returned.

Note that for a method with multiple loop structures, we can divide the method into several methods where each method has a single loop structure.

For example in Listing 2, a method with multiple loop structure can be translated into several methods where each has only a single loop structure as shown in Listing 3.

In order to convert a nested loop structure into a simple loop structure, we use a new counter $vpc$ which indicates which loop is selected.

Listing 2: Multiple Loops Structure

```
public class MultipleLoops {
  public int ex(int n) {
    int local = 0;
    for (int i=0; i< n; i++)
      local += i;
    int x = local;
    int y = local*2;
    for (int j=0; j < n; j ++)
      for (int k=0; k < n; k ++) {
        x *= 3 + k;
        y += x + j;
      }
    return x + y;
  }
}
```

Listing 3: Method with a Single Loop Structure

```
public class MultipleLoops {

  public int ex1(int n) {
    int local = 0;
    for (int i=0; i< n; i++)
      local += i;
    return local;
  }

  public int ex2(int loc, int n) {
    int x = loc;
    int y = loc*2;
    int i = 0;
    byte vpc = 1;
    while (vpc < 2){
      switch (vpc) {
        case 1;
          if (j < n) {
            vpc = 2;
            j ++;
            k = 0;
          } else {
            vpc = 3;
          }
          break;
        case 2;
          if (k < n) {
            x *= 3 + k;
            y += x + j;
            k ++;
          } else {
            vpc = 1;
          }
          break;
      }
    }
    return x + y;
  }

  public int ex(int n) {
    int tmp = ex1(n);
    return ex2(tmp, n);
  }
}
```

Thus, for a method with a multiple loops structure, our proposed method is also applicable. For nested loops, it is known that such loops can be translated into a single loop. Therefore, in principle, this method is also applicable.

We limit the class of the input Java method as follows because we will use SAT/SMT solvers in later analysis stages. The allowed types are bool, byte, short, int, float, double, and arrays of them. The proposed method cannot deal with String. For control structures, it allows for the use of `if`, `for`, `while` statements.

Through of the section, we use the following variables:
$\boldsymbol{x}$: arguments of the given (target) method
$\boldsymbol{y}$: variables which a user want to analyze
$n$: the number of iterations of the loop

Figure 1 shows the architecture of a tool proto-type of our proposed method.

The procedure is summarized as follows.

**Step 1:** Add proper print statements to the target source code in order to store execution logs on $\boldsymbol{y}$ and others.

**Step 2:** Execute the program with varying $\boldsymbol{x}$ and obtain a sufficient number of execution logs.

**Step 3:** Analyze the logs and obtain the execution paths, relations between $\boldsymbol{x}$ and the execution paths, and a record on $n$.

**Step 4:** Using regression analysis, infer the relation between $\boldsymbol{x}$ and $n$.

**Step 5:** As a recurrence relation, obtain a relation between $\boldsymbol{y}_s$ at the entry point and $\boldsymbol{y}_e$ at the exit point of the loop body.

**Step 6:** Solve the closed-form of the recurrence relation obtained in **Step 5**.

**Step 7:** Finally, calculate an expression representing a relation between $\boldsymbol{y}$, $\boldsymbol{x}$, and $n$ by integrating **Step 3**, **Step 4**, and **Step 6**.

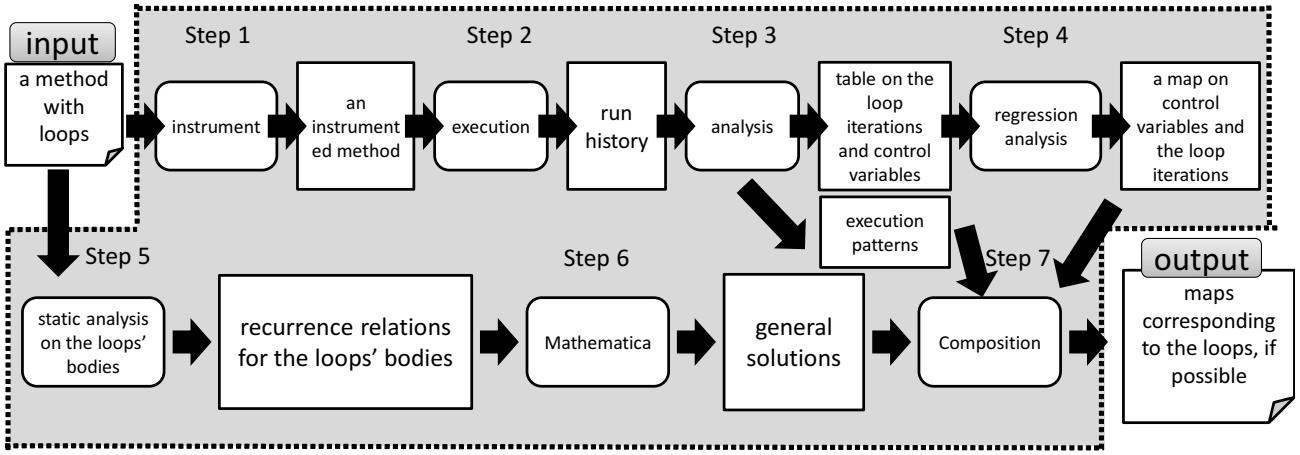



Figure 1: Tool overview

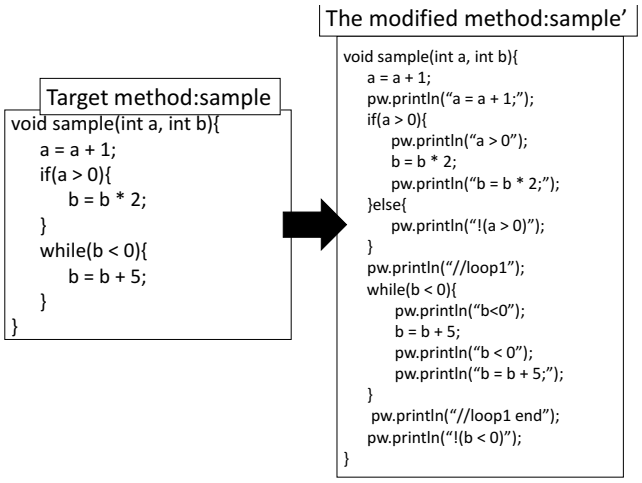



Figure 2: Print statement instrument for obtaining execution logs

In the following subsections, we will explain **Steps 1, 3, 4, 5, 6, and 7** which are important steps of our proposed method.

## 3.2 Step 1

In order to store execution logs, we add print statements to the target source code (Figure 2).

This step is similar to the Instrument step of Daikon [10], a famous tool for detecting invariants of programs.

In Fig. 2, `pw` is an instance of `PrintWriter` class. `pw` uses its `println` method to output log to a text file. The statement is inserted after assignment statement of the original code. For a control statement, such as an if-statement, it is also inserted to output the information on the condition of the path. For example it will output "$a > 0$" for the path in which the condition holds. For a loop structure, it outputs start and end markers as shown in Fig. 2, as well as the information on the condition.

JDT [17] is used to implement such an instrument.

## 3.3 Execution paths

Many paths are considered with regard to the conditions in the loop structure. Thus, we have to enumerate every pattern of the paths.

In general, a path in a loop structure is defined as a sequence of sentences executed for a given a concrete set of values of variables in the loop structure.

We call any path enumerated "an execution path (in the loop)." In general, the number of "if-statements" is $i$, and then there are $2^i$ execution paths at most.

## 3.4 Step 3-1

Here, we obtain a relation between $\boldsymbol{x}$ of the given method and the statements in the loop body.

The execution logs contain records on $\boldsymbol{x}$ and the number of occurrences of the execution path.

We explain this more precisely using the example in Fig. 3.

Let us assume that the upper left code is the target method. The method contains two execution paths as shown in Fig. 3.

We can see from the figure, that if the argument $i$ is 25, then Execution Path2 (EXP2) occurs twice and EXP1 occurs three times.

Figure 3 shows only three tuples, but, we actually obtain these tuples with more than 100 executions varying the values of $i$. The same value 100 is used in the existing method. Many studies including [18] have proposed how to generate efficient values of the inputs (arguments).

## 3.5 Step 3-2

Next we confirm the order of the executing paths. Figure 3 shows the situation in which EXP2 is first executed twice and then EXP1 is executed three times.

Let us assume that in general executions of a loop body, each execution path occurs repeatedly and successively.

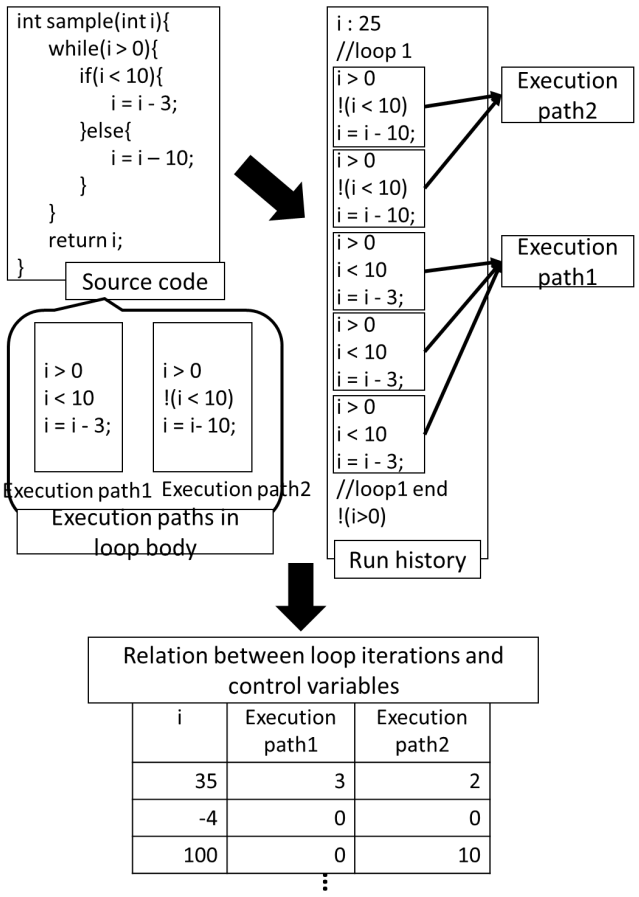



Figure 3: Analysis of Run history

> **Execution path assumption**: For any values of the variables, if an instance of an execution path occurs more than twice, then the execution paths occur successively.

Under this assumption, we can abstract this sequence as the following regular expression.

$$(EXP2)^*(EXP1)^* \tag{1}$$

At **Step 3**-2, we obtain such regular expressions on the execution patterns.

When we recall Fig. 3 we observe that in the sequence, EXP2 is first executed twice and followed by three executions of EXP1. We call such a pattern an execution pattern.

We enumerate every execution pattern from the execution logs. For each execution pattern, we abstract the constants representing the number of occurrences with the Kleene closure symbol $*$. For example, three occurrences of the execution path EXP1 is abstracted as $(EXP1)^*$.

For a loop, we can obtain a set of execution patterns.

For simplicity, hereafter we consider execution patterns in a form of $(EXP1)^*(EXP2)^* \cdots (EXPn)^*(n > 0)$. For other cases, we return failure of analysis.

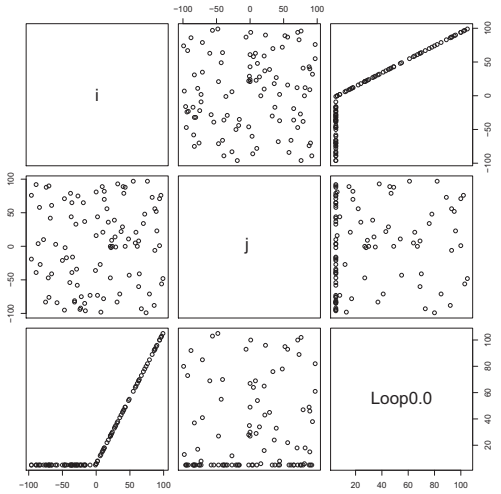



Figure 4: Relation between loop iterations and control variables

## 3.6 Step 4

Here, we describe how to derive a relation between $\boldsymbol{x}$ and the number of occurrences of an execution path (which is obtained at **Step 3-2**).

We use R [19], a regression analyzer.

Figure 4 shows relations between arguments (integers $i$ and $j$) and an execution path named "loop." Loop0.0 stands for "loop."

A plot located in the first row, second column in Fig. 4 shows a relation between $i$ and $j$.

In a similar way, plots in the first row, third column, and in the second row, third column show relations between $i$ and the number of executions of the "loop," and between $j$ and the number of executions of the "loop," respectively.

The plot in the first row, second column in Fig. 4 indicates that there is no correlation between $i$ and $j$ due to their randomness.

Additionally, we find that there is no correlation between $j$ and the number of loop executions. However, there is a strong correlation between $i$ and the number of loop executions. For the case where $i$ is negative, the number of executions of "loop" becomes 0. For the case on $i > 0$, the number of executions of "loop" becomes $i$.

Such a relation can be obtained using the regression analysis for each execution path.

For example, let $i$ and $j$ be arguments.

Let $a_0, a_1, a_2 \cdots$ be coefficients.

The following model (expression) can be used in regression analysis.

$$n = a_0 + a_1 i + a_2 j + a_3 ij + a_4 i^2 + a_5 j^2 \tag{2}$$

For Fig. 4, we obtain a result in which $a_1$ equals 1 and the other coefficients are 0. Thus, we obtain a relation $n = i$.

Note that $n$ is the number of iterations. Thus, it does not have a negative value. We assume that $n = 0$ when $n < 0$ for later analysis steps.

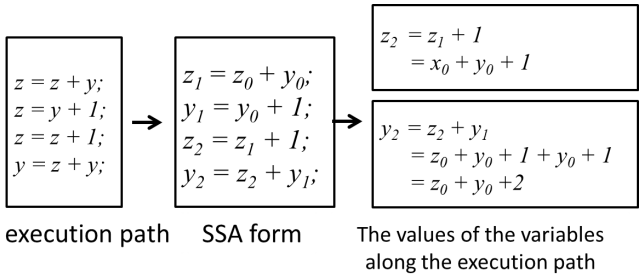



Figure 5: Conversion of Execution int SSA form

For the case of failure of regression analysis, we return analysis failure.

## 3.7 Step 5

Here, for each execution path $i$, we derive a relation between variables $\boldsymbol{y}_0^i$ and $\boldsymbol{y}_k^i$, where $\boldsymbol{y}$ is a vector of variables appearing in the execution path $i$. The suffixes are the same as the SSA form explained bellow.

First, a series of assignment statements of the execution path into SSA (static Single Assignment) form [20]. In SSA form every variable can appear in at most one assignment. In order to satisfy this condition, an original variable is, in general, divided into several variables when it is involved in several assignment statements. For such a case, the divided variables are distinguished by their own suffixes.

For example, an execution path can be translated into the SSA form shown in Fig. 5. The execution path uses variable $z$ and $y$. Their corresponding variables for the first assignment, are represented as $z_0$ and $y_0$. At line 1, $z+y$ is assigned for $z$. In such a case, variables $z_0$ and $z_1$ are used. In a similar way, at line 3, $z_2$ is used. The suffixes play a role to distinguish variable $z$ at different positions.

Next, using the SSA forms, we derive a recurrence relations on the variables.

In Fig. 5, the first values of the variable $z$ and $y$ are represented as $z_0$ and $y_0$. The final values are represented as variables $z_2$ and $y_2$. Using the SSA form, we can infer that $z_2$ equals $z_1 + 1$ and $z_1$ equals $z_0 + y_0$. Thus, $z_2$ and $y_2$ equal $z_0+y_0+1$ and $z_2+y_1$, respectively. Because $y_1$ equals $y_0+1$, we infer that $y_2$ equals $z_0 + y_0 + 2$.

The obtained equations can be represented as the following recurrence relations:

$$z[n+1] = z[n] + y[n] + 1,\ y[n+1] = z[n] + y[n] + 2 \quad (3)$$

Here, $z[0]$ and $y[0]$ stand for the seed values, i.e., $z_0$ and $y_0$ for the variables $z$ and $y$. Symbols $z[n]$ and $y[n]$ stand for the general term of $z$ and $y$ obtained by repeating $n$-times of the SSA form.

## 3.8 Step 6

Here, we solve the recurrence relation.

For example, we can obtain the following recurrence relation from an SSA form in Fig. 5.

$$z[n+1] = z[n] + y[n] + 1,\ y[n+1] = z[n] + y[n] + 2 \quad (4)$$

Let us assume that the seed values of $z$ and $y$ are $z_0$ and $y_0$, respectively. We can obtain a closed-form solution for the recurrence relation using Mathematica[21], as follows.

$$z[n] = \frac{1}{2}(-4 + 3 \cdot 2^n + 2^n y_0) \quad (5)$$

$$y[n] = \frac{1}{2}(-2 + 3 \cdot 2^n + 2^n z_0) \quad (6)$$

It is difficult to obtain such a complex expression using the existing method [12].

## 3.9 Step 7

Here we integrate the obtained analysis results in the previous steps and generate the final mapping.

Let us assume that the execution pattern is $(EXP1)^*(EXP2)^* \cdots (EXPk)^*$, and that the number of execution times of EXP$i$ is $m_i$.

$\boldsymbol{y}_0^i$ stands for the initial values at the entry of EXP$i$

Let $\boldsymbol{y} = F_i(\boldsymbol{y}_0^i, n)$ be the mapping obtained at **Step 6**. Let $G_i(\boldsymbol{x})$ be the mapping obtained at **Step 4**, where $m_i = G_i(\boldsymbol{x})$.

Let us consider the following cases.

1. $k = 1$ holds
2. $k > 1$ and $\forall i \exists c : 0 < i \leq k, (G_i = c\boldsymbol{x}$ or $G_i = c)$ holds
3. $\forall i, j : 0 < i, j \leq k, G_i = G_j$ holds
4. otherwise

For the first three cases, we can obtain the result as follows.

For case (1), the final mapping is $\boldsymbol{y} = F_1(\boldsymbol{y}_0^1, G_1(\boldsymbol{x}))$.

For cases (2) and (3), the final mapping is $\boldsymbol{y} = F_k(\cdots F_2(F_1(\boldsymbol{y}_0^1, G_1(\boldsymbol{x})), G_2(\boldsymbol{x})), \ldots, G_k(\boldsymbol{x}))$.

For case (4), we conclude that the mapping cannot be generated and failure is returned.

We shows an example for the case (2):

Let $(EXP1)^*(EXP2)^*$ be the execution pattern.

Let us consider a situation where when EXP1 is executed $n$ times, then the value of variable $y$ increases by $n$, and if EXP2 is executed $n$ times. Then, the value of variable $y$ increases by $10n$.

In such a case, equations $F_1(y_0^1, n) = y_0^1 + n$ and $F_2(y_0^2, n) = y_0^2 + 10n$ holds. Additionally let us assume that when $i > 0$ EXP1 and EXP2 are executed $i$ and 1 times, respectively; and that when $i \leq 0$ EXP1 and EXP2 are not executed.

By integrating all of the above, we can obtain the final mapping $y = y_0^1$ for $i \leq 0$, and $y = y_0^1 + x + 10$ for $i > 0$.

# 4 EXPERIMENTS

The setup of the experiments is summarized as follows.

- OS: Windows 7 Enterprise 64bit
- CPU: Intel Xeon E5-2609 2.40GHz $\times$ 2
- Memory: 48.0GB
- Java: JRE7
- R: version 3.0.2
- Mathematica 9.0.0
- Z3: z3-4.3.0

## 4.1 Overview of the experiment

The research questions are summarized as follows.

RQ1 Mapping quality: Is the obtained mapping accurate?

RQ2 SAT/SMT applicability: Is the obtained mapping applicable to SAT/SMT solvers?

RQ3 Range of Capability: Can more types of mapping be obtained from as compared to the existing method?

We use Z3 for criteria for RQ2

Table 2 summarizes the target programs and each program contains loop structures.

## 4.2 Results

Tables 3 and 4 show the execution times and results of our experiments.

For RQ2, a ✓ mark means that the output values, that are obtained from the mapping using random inputs varying from 0 to 200, have relative errors within 10% against the true values. A $\times$ mark stands for other cases.

# 5 DISCUSSION

## 5.1 RQ1

With Mathematica, we can correctly derive closed-form solutions of the recurrence formulae obtained from the programs. For some programs not shown in Table 2, we cannot derive their closed-form solutions. The reasons are that 1) Mathematica cannot deal with them, and 2) in general, not every recurrence relation has a closed-form solution. For such cases, the existing methods also cannot be applied.

## 5.2 RQ2

There are mappings that are not applicable to SAT/SMT solvers. For example, for Newton, our method derives an expression $\sqrt{a}\cosh(2^n \cosh^{-1}\sqrt{a})$. Z3 cannot deal with the expression.

For such a case concolic testing [22], [23] might be a solution for further analysis.

## 5.3 RQ3

The existing method, in principle, cannot derive a correct mapping for a Fibonacci generator. The existing method approximates it as quadratic equations. It, however, has large relative errors for a large input. Thus, advantage of our proposed method is confirmed.

## 5.4 Execution Times

A large proportion of execution times are occupied by Mathematica computation. Particularly, solving the closed-form solutions is highly time consuming.

## 5.5 Other Discussions

The proposed method in this work uses regression analysis for obtaining a mapping between the number of loop iterations and arguments. In general, obtaining a relation between variables before and after the target loop is a complex task. For this reason, relative errors arising from regression analysis become small. Consequently, our approach has the advantage for cases in which (1) a mapping between the number of loop iterations and arguments is linear or quadratic, and (2) a relation between variables before and after the target loops is complex.

## 5.6 Limitation of the Methods

The closed-form solution is obtained using Mathematica in this work; thus the ability of obtaining the solution depends on Mathematica.

We assume that the patterns of execution paths are in the restricted form shown earlier in the execution path assumption. If the code violates the assumption, then the proposed method cannot be applied.

## 5.7 Treats to Validity

The programs used in the experiments are small. The number of the programs is also small. The results, however, show that the proposed method can be applicable to programs that cannot be handled by the existing methods.

The class of variable types for variable is restricted, mainly due to the limitations of SAT/SMT solvers.

# 6 CONCLUSION

This paper proposed a new method for inferring a mapping between variables before and after a given loop structure. The experimental results show that our proposed method can derive complex mapping, which the existing methods cannot successfully derive.

Our future work includes the application of concolic testing on our derived mappings, in order to perform efficient and further analysis.

## Acknowledgments

The research is also being partially conducted as Grant-in-Aid for Scientific Research C (16K00094) and S(25220003).

Table 2: Target Programs

| Program | What to process | LOC | Number of loop structures | Number of if statements |
|---|---|---|---|---|
| Fibonacci | Fibonacci numbers | 20 | 1 | 1 |
| Newton | calculation of square root by Newton method | 30 | 1 | 1 |
| DrawPict | draw pictures | 39 | 1 | 1 |
| DrawPara | draw parabola | 77 | 3 | 10 |
| Summation | calculation of summation | 20 | 1 | 1 |
| Power | calculation of power series | 20 | 1 | 1 |

Table 3: Execution Times (sec.)

| Program | Step 1 | Step 2 | Step 3 | Step 4 | Step 5 | Steps 6 and 7 |
|---|---|---|---|---|---|---|
| Fibonacci | 2.6 | 0.62 | 0.1 | 4.4 | 4.4 | 0.72 |
| Newton | 2.3 | 0.36 | 0.0 | 2.4 | 1.3 | 0.64 |
| DrawPict | 2.7 | 0.59 | 0.0 | 3.0 | 2.2 | 0.75 |
| DrawPara | 2.5 | 1.1 | 0.0 | 2.9 | 2.9 | 0.65 |
| Summation | 2.4 | 0.57 | 0.05 | 2.4 | 2.4 | 0.66 |
| Power | 2.3 | 0.38 | 0.0 | 3.5 | 3.5 | 0.75 |

| Program | total |
|---|---|
| Fibonacci | 8.4 |
| Newton | 5.8 |
| DrawPict | 7.0 |
| DrawPara | 7.2 |
| Summation | 6.0 |
| Power | 6.9 |

Table 4: Experimental Results

| Program | the existing method | our method | RQ2 |
|---|---|---|---|
| Fibonacci | × | ✓ | × |
| Newton | × | ✓ | × |
| DrawPict | ✓ | ✓ | ✓ |
| DrawPara | × | ✓ | ✓ |
| Summation | ✓ | ✓ | ✓ |
| Power | × | ✓ | × |

Funding from Mitsubishi Electric Corporation is gratefully acknowledged.

## REFERENCES

[1] G. T. Leavens, A. L. Baker, and C. Ruby: "JML: a Java modeling language," Formal Underpinnings of Java Workshop (at OOPSLA'98) pp.404–420 (1998).

[2] P. Godefroid, N. Klarlund, and K. Sen: "DART: directed automated random testing," ACM SIGPLAN Notices, Vol.40, No.6, pp.213–223 (2005).

[3] W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda: "Model checking programs," Automated Software Engineering, Vol.10, No.2, pp.203–232 (2003).

[4] P. Cousot: "Proving program invariance and termination by parametric abstraction lagrangian relaxation and semidefinite programming," Proceedings of the 6th International Conference of VMCAI 2005, Vol. 3385, pp.1–24, Lecture Notes in Computer Science (2005).

[5] T. Ball and S.K. Rajamani: "The slam project: Debugging system software via static analysis," Proceedings of the 29th ACM SIGPLAN-SIGACT POPL'02, pp.1–3 (2002).

[6] T.A. Henzinger, R. Jhala, R. Majumdar, G.C. Necula, G. Sutre, and W. Weimer: "Temporal-safety proofs for systems code," Proceedings of the 14th International Conference on Computer Aided Verification, CAV 2002, pp.526–538 (2002).

[7] A. Biere, M. Heule, H. Van Maaren, and T. Walsh: "Handbook of Satisfiability," IOS press (2009).

[8] L. deMoura and N. Bjørner: "Z3: An efficient smt solver," Proceedings of Tools and Algorithms for the Construction and Analysis of Systems 2008, Vol.4963, pp.337–340, Lecture Notes in Computer Science (2008).

[9] B. Dutertre: "Yices 2.2," Computer-Aided Verification (CAV'2014), Vol.8559, pp.737–744, Lecture Notes in Computer Science (2014).

[10] M.D. Ernst, J. Cockrell, W.G. Griswold, and D. Notkin: "Dynamically discovering likely program invariants to support program evolution," IEEE TSE, Vol.27, pp.1–25 (2001).

[11] M. Younger: "Handbook for Linear Regression," Duxbury Resource Center (1979).

[12] W. Le: "Segmented Symbolic Analysis," Proceedings of the 2013 International Conference on Software Engineering, pp.212–221 (2013).

[13] D.R. Cok and J.R. Kiniry: "Esc/java2: Uniting esc/java and jml: Progress and issues in building and using esc/java2 and a report on a case study involving the use of esc/java2 to verify portions of an internet voting tally system," Proceedings of Construction and Analysis of Safe, Secure and Interoperable Smart Devices: International Workshop, CASSIS 2004, Vol.3362, pp.108–128, Lecture Notes in Computer Science (2005).

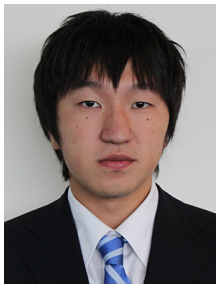
[14] C. Cadar, D. Dunbar, and D. Engler: "KLEE: Unassisted and Automatic Generation of High-coverage Tests for Complex Systems Programs," Proceeding of the 8th USENIX Conference on Operating Systems Design and Implementation, pp.209–224 (2008).

[15] V. Chipounov, V. Kuznetsov, and G. Candea: "S2E: A Platform for In-vivo Multi-path Analysis of Software Systems," Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems, pp.265–278 (2011).

[16] Y. Xie, A. Chou, and D. Engler: "ARCHER: Using Symbolic, Path-sensitive Analysis to Detect Memory Access Errors," Proceedings of the 9th European Software Engineering Conference Held Jointly with 11th ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp.327–336 (2003).
[17] "Eclipse Java development tools (JDT)," (accessed 2015-05-05). http://www.eclipse.org/jdt/.
[18] K. Kobayashi, Y. Sasaki, K. Okano, and S. Kusumoto: "Automated assertion generation using PDF and SMT-Solver," IEICE Transaction on Information and Systems, JD, Vol.96, No.11, pp.2657–2668 (2013) (In Japanese).
[19] "the R statistical package," (accessed 2015-05-05). http://cran.r-project.org/.
[20] E. Clarke, D. Kroening, and F. Lerda, "A Tool for Checking ANSI-C Programs," Proceedings of the 10th International conference on Tools and Algorithms for the Construction and Analysis of Systems, pp.168–176 (2004).
[21] "Mathematica: Wolfram Research," (accessed 2015-05-05). https://www.wolfram.com/.
[22] K. Sen, D. Marinov, and G. Agha: "CUTE: A Concolic Unit Testing Engine for C," SIGSOFT Software Engineering Notes, Vol.30, No.5, pp.263–272 (2005).
[23] R. Majumdar and K. Sen: "Hybrid Concolic Testing," Proceeding of the 29th International Conference on Software Engineering, pp.416–426 (2007).

(Received October 20, 2017)
(Revised December 27, 2017)

**Kozo Okano** received his BE, ME, and PhD degrees in Information and Computer Sciences from Osaka University in 1990, 1992, and 1995, respectively. From 2002 to 2015, he was an Associate Professor at the Graduate School of Information Science and Technology of Osaka University. In 2002 and 2003, he was a visiting researcher at the Department of Computer Science of the University of Kent in Canterbury, and a visiting lecturer at the School of Computer Science of the University of Birmingham, respectively. Since 2015, he has been an Associate Professor at the Department of Computer Science and Engineering, Shinshu University. His current research interests include formal methods for software and information system design. He is a member of IEEE, IEICE, IPSJ.



**Yukihiro Sasaki** received his BI and MI degrees from Osaka University in 2012 and 2014, respectively. He currently works for Mitsubishi Electric Corporation. His research interests include dynamic generation of assertion for programs.

**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.