SUMMARY We try to use a computer algebra system Mathematica as a test case generation system. In test case generation, we generally need to solve equations and inequalities. The main reason why we take Mathematica is because it has a built-in function to solve equations and inequalities. In this paper, we deal with both black-box testing and white-box testing. First, we show two black-box test case generation procedures described in Mathematica. The first one is based on equivalence partitioning. Mathematica explicitly shows a case that test cases do not exist. This is an advantage in using Mathematica. The second procedure is a modification of the first one adopting boundary value analysis. For implementation of boundary value analysis, we give a formalization for it. Next, we show a white-box test case generation procedure. For this purpose, we also give a model for source programs. It is like a control flow graph model. The proposed procedure analyzes a model description of a program.

key words: test case generation, equivalence partitioning, boundary value analysis, white-box testing, computer algebra system

1. Introduction

Software testing is a significant process in software development. There are many tools, for example JUnit, to manage existing test cases. However, there is no famous tool to generate test cases, though there are many research works concerning automatic test case generation. We guess the following reason why such a tool does not exist. There are many situations of software testing. So, we need various kinds of test cases. It is difficult to develop a unified test case generator which can deal with such many kinds of test cases. Consequently, whenever we want to generate test cases automatically, we are required to implement a generation procedure.

In this paper, we try to use a computer algebra system Mathematica as a system for test case generation. Mathematica makes implementation of a test case generation procedure easy. The most important reason why we take Mathematica is because it has a function to solve equations and inequalities. In general, a test case generation procedure needs to solve equations and inequalities. When we implement a procedure, we skip to develop such a solving mechanism by virtue of Mathematica. In this paper, we generate black-box test cases directly from a specification given by equations and inequalities. We need to find an instance (i.e. values) of the variables satisfying the specification. Mathematica Version 5 or later has a built-in function FindInstance which is appropriate for this purpose.

We also propose a white-box test case generation procedure. In white-box testing, we detect a candidate of an execution path in a model description of a source program. A candidate of a path has a condition to execute the path expressed by equations and inequalities. A test case generation procedure needs to find an instance of the variables satisfying the condition to execute the path.

In this paper, first, we show two black-box test case generation procedures described in Mathematica. The first one is based on equivalence partitioning. Mathematica explicitly shows a case that test cases do not exist. This is an advantage in using Mathematica. The second procedure is a modification of the first one adopting boundary value analysis. For implementation of boundary value analysis, we give a formalization for it.

Next, we show a white-box test case generation procedure. For this purpose, we also give a model for source programs. It is like a control flow graph model. The proposed procedure analyzes a model description of a program.

There are some other reasons why we use Mathematica. Mathematica is a functional programming language and it has a built-in function Map to transform a list. Using this function, we can make a generation procedure compact. The last reason is that Mathematica can be used for specification and modeling of target software. As we have mentioned, black-box test cases are generated from a specification given by equations and inequalities. These are described in Mathematica. On the other hand, in white-box test case generation, a model description of a source program is given by a list in Mathematica. In test case generation, both the input sources and the procedures are described in the same language. This is also an advantage in using Mathematica.

We have intended that the proposed generation procedures do not depend on target software. We need to check applicability of our procedures by many case studies. In this paper, we show one case study. We take “the triangle problem” introduced in Myers’ famous textbook of software testing [9]. We generate ten test cases of Myers’ example. Five cases are generated by the first black-box testing procedure based on equivalence partitioning. Two of the rests are generated by the second black-box testing procedure based on boundary value analysis. Remaining three cases are generated by the white-box test case generation procedure. To show that our procedures do not depend on target software, we give another example after all procedures are introduced.

Manuscript received July 21, 2009.
Manuscript revised November 10, 2009.

The author is with the Graduate School of Information Science and Engineering, Tokyo Institute of Technology, Tokyo, 152-8552 Japan.

a) E-mail: satoshi@cs.titech.ac.jp
DOI: 10.1587/transinf.E93.D.1006

Copyright © 2010 The Institute of Electronics, Information and Communication Engineers
This paper is organized as follows. First, in Sect. 2, we explain an overview of a testing process using Mathematica. Next, in Sect. 3, we describe Myers’ triangle problem. After introducing the computer algebra system Mathematica in Sect. 4, we show two black-box test case generation procedures in Sect. 5. A white-box test case generation procedure is shown in Sect. 6. To show these procedures do not depend on a problem, we give another example in Sect. 7. In Sect. 8, we explain some related works. Finally, in Sect. 9, we conclude the paper with discussions on future works.


In Fig. 1, we show an overview of a testing process using Mathematica. This process includes a phase to specify software and a phase to write a source program. Each node in the figure is a “program” which appears in the process. A rectangle with edges contains an artifact concerning target software. A rectangle with round corners is a test case generator or a testing tool. A rectangle with a color is a Mathematica program. A white rectangle is a program written in another language. A source program of target software is written for example in Java. There is a testing tool for example JUnit for a Java program.

(A) and (B) are test case generation procedures for black-box testing and for white-box testing, respectively. By the procedure (A), test cases are generated only if a specification of software satisfies the requirement for an input of (A). Similarly, by the procedure (B), test cases are generated only if the model description of a source program of software satisfies the requirement for an input of (B). In this paper, we show examples of (A) and (B) with the requirements for inputs of these.

The numbers 1, 2, 3, 4 and 5 show the order of developing the artifacts. The artifacts with a number in parentheses, (1), (3) and (4), are created by the developer. The artifacts with a number in angles, (2) and (5), are automatically generated by the procedures (A) and (B), respectively. First, the developer writes (1), a specification of software, in Mathematica. Then, applying the procedure (A) to the specification, (2), black-box test cases are generated. Next, the developer writes (4), a source program of software. Proceeding this, (3), a model description of the program should be made. In this paper, the model description is given by a list in Mathematica. Then, applying the procedure (B) to the model description, (5), white-box test cases are generated. Finally, using a testing tool, for example JUnit, the source program is tested by both the black-box test cases and the white-box test cases.

3. Myers’ Triangle Problem

Myers’ triangle problem is to generate test cases for the following program[9]. “The program reads three integer values from a card. The three values are interpreted as representing the lengths of the sides of a triangle. The program prints a message that states whether the triangle is scalene, isosceles, or equilateral.”

Let \( (a, b, c) \) be a 3-tuple of integer variables. We denote a test case by an instance of \((a, b, c)\). Myers enumerated the following fourteen test cases for a solution of this problem. Note that the last one itself is not a test case, but it is a requirement for a test case.

(1) A valid scalene triangle: for example, \((2, 3, 4)\)
(2) A valid equilateral triangle: \((1, 1, 1)\)
(3) A valid isosceles triangle: \((3, 3, 4)\)
(4) Valid isosceles triangles with all three permutations of the values of the case \((3, 3, 4)\): \((3, 4, 3), (4, 3, 3)\)
(5) One side has a zero value: \((0, 3, 2)\)
(6) One side has a negative value: \((-1, 4, 2)\)
(7) Three integers greater than zero such that the sum of two is equal to the third: \((1, 2, 3)\)
(8) At least three test cases from all permutations of the values of the case \((7)\): \((1, 2, 3), (1, 3, 2)\)
(9) Three integers greater than zero such that the sum of two is less than the third: \((1, 2, 4)\)
(10) At least three test cases from all permutations of the values of the case \((9)\): \((1, 2, 4), (1, 4, 2)\)
(11) All sides are zero: \((0, 0, 0)\)
(12) A test case specifying noninteger values: \((2, 5, 3.5)\)
(13) A test case specifying wrong number of values: \((1, 2)\)
(14) For each test case, specify the expected output from the program in addition to the input values.

Using Mathematica, we generate test cases with the numbers \((1), (2), \cdots, (9)\) and \((10)\). Each test case includes the output value which Myers mentions in (14). We exclude the test case \((11)\), because it is a sub case of \((5)\). (The case \((11)\) seems to be the case that the variables are defined but values are not substituted for these.) We assume that all input values must be syntactically valid. That is, we exclude the cases \((12)\) and \((13)\), because they are not syntactically valid.

4. Preliminaries to Use Mathematica

In this section, we describe some of the grammar and some
A built-in function FindInstance gives an instance of variables satisfying a given expression, if it exists. FindInstance returns an empty set if any instance does not exist. Specifically, FindInstance[expr, vars, dom, n] finds an instance of variables vars that makes a statement expr be True. The optional third argument dom specifies the domain which the instance belongs. dom is chosen from Complexes(default), Reals, Integers and Booleans. The optional fourth argument n specifies the number of the instances. The default value is one.

Let us consider the triangle problem. Assume that three integers \((a, b, c)\) are the lengths of the three sides of a triangle. Then, the following is the equivalent condition: \(a+b > c\) and \(b+c > a\) and \(c+a > b\). (We do not need the condition that three integers are positive. See the next section.) We regard this as a specification of a triangle. Here, we show a Mathematica program to find an instance of \((a, b, c)\) satisfying the specification, and the output from Mathematica. In this paper, we use a broken line to separate a program from paragraphs. If there is an output of a program, we also use a broken line to separate it from the program.

```
FindInstance[{a + b > c, b + c > a, c + a > b}, {a, b, c}, Integers]

\{a -> 2, b -> 2, c -> 2\}
```

To clarify the applicability of test case generation with Mathematica, we restrict an expression expr of the first argument of FindInstance within a known soluble class. A system of equations and inequalities is called indeterminate, if the number of the equations in a system is fewer than the number of the variables in the system. A Diophantine system is an indeterminate system of polynomial equations and inequalities with integer coefficients and variables. Mathematica can solve any quantifier-free linear Diophantine system with an arbitrary number of variables [15]. By FindInstance, Mathematica can also find an instance of variables satisfying such a system. We think the class of such systems is practical. At least, any expression appearing in the triangle problem belongs to this class.

4.2 Defining a Function

Here, we describe how to define a function in Mathematica. Take the triangle problem. We define four functions below.

```
x1[a_, b_, c_] = a > 0 && b > 0 && c > 0
x2[a_, b_, c_] = a + b > c && b + c > a && c + a > b
x3[a_, b_, c_] = a == b || b == c || c == a
x4[a_, b_, c_] = a == b && b == c
```

Here, assume that a, b and c are integer variables. Each function returns True if the following respective condition is satisfied, and returns False otherwise. x1: All values of the variables are positive. x2: The condition for the lengths of the three sides of a triangle. x3: Two variables have the same value. x4: All three variables have the same value.

As FindInstance, the name of a built-in function starts from a capital letter. On the other hand, the name of a user defined function must start from a small letter. When we define a function, we do not need to specify the types of the parameters and the return value. In the above, the types of the parameters \(a\), \(b\) and \(c\) can be integer or real. The type of the return value of the function becomes Boolean. The parameters of a function are enclosed by the brackets. Notice that a parameter in the left-hand side of each function is shown as \(a\), for example. We should write a parameter in such a way, if we want to give an arbitrary argument for the parameter. On the contrary, if we write a parameter in the left-hand side simply as \(a\), the valid argument is only the variable \(a\).

In Mathematica, to evaluate several expressions sequentially, we input the expressions separated with the semicolons. In case that we write several expressions in the right-hand side of a function, we use also the semicolons for separation. And then, all expressions in the right-hand side are enclosed by the parentheses. In case we need local variables \(x, y, \ldots\) in the right-hand side of a function, we define the function as Module[\(\{x, y, \ldots\}\), expr]. In this case, expr can also be sequential expressions separated by the semicolons. The return value of a function is set by Return[expr]. A function \(f\) with lazy evaluation has the form such as \(f[\ldots] := \text{expr}\). The symbols = and := are used for assignment. The symbol == is used for equality.

4.3 Control Structure

If[condition, t, f] returns \(t\) if condition evaluates to True, and returns \(f\) if condition evaluates to False. The third argument \(f\) can be omitted. Switch[expr, form1, value1, form2, value2, \ldots] evaluates expr, then compares it with each of the forms in turn. It returns the result of evaluation of valuei, corresponding to the first match found. For[start, test, incr, body] executes start, then repeatedly
evaluates body and incr, until test fails to give True.

4.4 List and Its Manipulation

In Mathematica, a list is a fundamental data structure. A list has the form such as \{e1, e2, \ldots\}. This list is expressed also by List[e1, e2, \ldots]. AppendTo[expr, elem] appends an expression elem to a list expr as its last element. Join[e1, e2] joins two lists e1, e2. Flatten[expr] flattens a nested list expr.

The i-th element of a list expr is denoted by expr[[i]]. The number of the elements of a list expr is given by Length[expr]. In an ordinary language, the index of an array begins from 0. However, the index of a list begins from 1 in Mathematica. So, in case that we use For statement to deal with a list expr, For[i = 1, i <= Length[expr], i++, \ldots] is a typical usage.

Instead of using For statement, we can use Map function. For a list expr with the form List[e1, e2, \ldots], Map[f, expr] returns List[f[e1], f[e2], \ldots]. For a list expr with the form List[e1, e2, \ldots], Apply[f, expr] returns f[e1, e2, \ldots]. Suppose that expr is a list List[11, 12, \ldots] and each of 11, 12, \ldots is also a list. By choosing an element ei from each of ei, we have a list List[e1, e2, \ldots]. Distribute[expr, List] returns the list consisting of all possible lists generated by the above manipulation. For example, Distribute,List[{1, 2}, {3, 4}], List] returns \{{1, 2, 3}, {1, 4}, {2, 3}, {2, 4}\}.

A pure function is a function with no name. Let body be the definition of a function. Then the pure function is denoted by body &. In body, the formal parameter is expressed by #. For example, Map[# + 1 &, {1, 2, 3}] returns \{2, 3, 4\}.

4.5 Term, Position, Replacement and Rule

In Mathematica, each expression has a term structure. About notions such as term, position and subterm, refer for example to [1]. Head[expr] returns the root symbol of expr. Position[expr, pattern] returns the set of positions which pattern occur in expr. ReplacePart[expr, new, pos] returns the result of the replacement of the subterm of expr at each position in pos with a term new. In Mathematica Ver.5 or earlier, ReplacePart is used as the above [16]. In Ver.6 or later, the usage is different [14]. A rule lhs \rightarrow rhs rewrites each occurrence of lhs to rhs. expr /. rules applies a list of rules rules to an expression expr.

4.6 Other Functions and Notation of Expressions

Subsets[expr] returns the list of the power set of a list expr. Complement[e0, e1] returns the complement of a set e1, in case that a set e0 is the universe.

And[e1, e2] and Or[e1, e2] return the conjunction and the disjunction of e1 and e2, respectively. Not[expr] returns the negation of expr. These three are represented also by e1 && e2, e1 | | e2 and !expr, respectively. Similarly, an equation and inequalities e1 == e2, e1 > = e2, e1 > e2, e1 <= e2 and e1 < e2 are represented also by Equal[e1, e2], GreaterEqual[e1, e2], Greater[e1, e2], LessEqual[e1, e2] and Less[e1, e2], respectively.

Scan[f, expr] is similar to Map[f, expr]. Map[f, expr] shows all intermediate results of the evaluation of expr, but Scan[f, expr] shows only the final result.

Print[expr] writes an expression expr in the standard output. At last, in Mathematica, a comment is enclosed by (* and *).

5. Black-Box Test Case Generation

5.1 Equivalence Partitioning

We give a test case generation procedure based on equivalence partitioning [9].

Suppose that a set S of conditions is given. Here, a condition is given by a set of equations and inequalities. There is a space defined by the set V of the variables appearing in S. Instances of V make a class. The largest class is the set of all instances of V. We call this the domain of S. By a condition x ∈ S, the domain is partitioned into two classes, the valid and the invalid equivalence classes of the condition x, respectively. The valid equivalence class of x is specified by x itself; the invalid equivalence class of x is specified by ~x, the negation of the condition x. The equivalence partitioning P of the domain with respect to the set S of conditions is a set of classes defined as follows: P = \{∧x∈S. x′ | x′ ∈ [x, ~x]\}.

Figure 2 shows a Mathematica program to make the equivalence partitioning P w.r.t. the set S of conditions, and to generate test cases by the equivalence partitioning P. Let V be the set of the variables appearing in S. In the program, the mathematical sets S, V and P are expressed by the list s, v and p, respectively. The inputs of the procedure are the lists s and v, and the output is the list tc of test cases. An element of the list xp denotes the set {~x, x} for an element x of S. Note that, by using functions Map and Distribute, we have implemented a compact procedure.

We give the requirement for a condition in S. Consequently, we give the requirement for an input of the procedure. A linear equation or inequality with integer coefficients and variables is a condition. Each of the negation of a condition, a conjunction of conditions and a disjunction of conditions is also a condition. With this requirement, each
element in the equivalence partitioning $P$ w.r.t. $S$, becomes a linear Diophantine system, hence an instance of variables satisfying an element in $P$ is computed by Mathematica.

Recall the triangle problem. We give the set $\{x_1, x_2, x_3, x_4\}$ of conditions for three integers $V = \{a, b, c\}$ to specify a triangle and its property. ($x_1$) All three integers are positive, i.e. $a > 0 \land b > 0 \land c > 0$. ($x_2$) For each pair of two integers, the sum of these is larger than another integer, i.e. $a + b > c \land b + c > a \land c + a > b$. ($x_3$) Two of three integers have the same value, i.e. $a = b \lor b = c \lor c = a$. ($x_4$) All three integers have the same value, i.e. $a = b = c$. We give the following lists $s$ and $v$ to the procedure in Fig. 2.

$$\begin{align*}
x_1[a, b, c] &= a > 0 \& \& b > 0 \& \& c > 0 \\
x_2[a, b, c] &= a + b > c \& \& b + c > a \& \& c + a > b \\
x_3[a, b, c] &= a = b \lor b = c \lor c = a \\
x_4[a, b, c] &= a = b \& \& b = c \\
s &= \{x_1[a, b, c], x_2[a, b, c], x_3[a, b, c], x_4[a, b, c]\} \\
v &= \{a, b, c\}
\end{align*}$$

We show the obtained test cases in Table 1. In Table 1, for example, $x_1 = 1$ means that the condition $x_1$ holds true; $x_1 = 0$ means that $x_1$ does not hold.

Note that when the function `FindInstance` returns the empty set, a kind of theorem proving is done. For example, each case satisfying $(x_1, x_2) = (0, 1)$ has no instance. This means that $x_2$ implies $x_1$, that is, if $a + b > c \land b + c > a \land c + a > b$, then $a, b$ and $c$ must be positive. We need the test case $\{-139, -139, -16\}$ in Table 1, because the program must not return the output “isosceles” for this case. One may try to find a test case satisfying $a + b > c \land b + c > a \land c + a > b$ for non-positive integers, because the program must not return the output “scale” or “isosceles” or “equilateral” for this case, similarly. However, such a test case does not exist. Mathematica explicitly shows it. This is an advantage in using Mathematica.

We have other two cases which have no test case; $(x_3, x_4) = (0, 1)$ and $(x_1, x_2, x_3, x_4) = (1, 0, 1, 1)$. The former is trivial. The latter means that if three integers are equal and all are positive, then these must construct a triangle. Recall that three integers $(a, b, c)$ construct a triangle if and only if $a + b > c$ and $b + c > a$ and $c + a > b$.

Here, we check whether each test case of Myers is included. (1) A scalene triangle is in the case $(x_1, x_2, x_3, x_4) = (1, 1, 0, 0)$. (2) An equilateral triangle is in $(1, 1, 1, 1)$. (3) An isosceles triangle is in $(1, 1, 1, 0)$. (6) The case that contains a non-positive number is in $(0, 0, 0, 0)$. (9) The case that positive three numbers do not construct a triangle is in $(1, 0, 0, 0)$. Other cases are not still generated. Note that the test case $[0, 0, 0]$ satisfies (5) of Myers’ description. However, the procedure has generated it without the intention that it contains zero.

As Myers says, a test case needs information of the output value. We use the following constants to show the equivalence class which an instance of $(a, b, c)$ belongs: equilateral triangles (400), isosceles triangles (300), scalene triangles (200), and the class that any instance does not construct a triangle (100). We define a function to determine the class as follows.

$$\begin{align*}
type[a, b, c] &= If[x_2[a, b, c], \\
&\quad If[x_3[a, b, c], If[x_4[a, b, c], 400, 300], 200], 100]
\end{align*}$$

5.2 Boundary Value Analysis

The procedure introduced in the previous subsection is very simple, because we want to clearly explain an advantage in using Mathematica. That is, Mathematica explicitly shows a class which does not have a test case. Here, we want to modify the procedure. Take a condition $x$ or its negation $\neg x$ in a set $S$ of conditions. Let us call each of these simply a condition. If it is expressed by a disjunction of several expressions, the equivalence class of each expression should be considered. Note that, in this case, an instance may belong to more than one class. That is, this is not a partition. This is called “path sensitizing” in the part of cause-effect graph in Myers’ book [9].

Let us consider boundary value analysis [9]. Take an inequality for example $a \leq 0$. This can be considered as the disjunction of $a < 0$ and $a = 0$. In this case, we regard the expression $a = 0$ as a “boundary”. Here, we introduce a terminology. Take inequalities $a < 0$ and $a \leq 0$. In the rest of this paper, we call the former a strict inequality and the latter a non-strict inequality, respectively. If a condition contains an inequality, one of the valid equivalence class and the invalid equivalence class is defined by a non-strict inequality. As we have seen, we can regard a non-strict inequality as the disjunction of the strict inequality and the equation. Boundary value analysis is a sub case of path sensitizing, because a non-strict inequality is expressed by a disjunction.

Here, we formalize path sensitizing containing bound-
ary value analysis. Recall the equivalence partitioning $P$ of the domain w.r.t. the set $S$ of conditions, $P = \{\wedge_{x \in S} x' \mid x' \in (x, \sim x)\}$. Here, we modify the set $(x, \sim x)$ for a condition $x \in S$. Take a condition $x$ or $\sim x$. If it contains a non-strict inequality, we transform it into the disjunction of the strict inequality and the equation. Next, for the obtained expression, compute the disjunctive normal form (DNF). The reason why we need the DNF is that we are considering path sensitizing.

To obtain a DNF in Mathematica, we use a built-in function LogicalExpand. However, actually we should use LogicalExpand before the transformation of a non-strict inequality into the disjunction of the strict inequality and the equation, because Mathematica sometimes does not expand the negation of an expression. Take $x_l(a, b, c) = a > 0 \land b > 0 \land c > 0$ of the triangle problem. Checking $y_1 = x_l(a, b, c)$, we have only $y_1 = \neg(a > 0 \land b > 0 \land c > 0)$. We have $\text{LogicalExpand}[y_1] = a \leq 0 \lor b \leq 0 \lor c \leq 0$, as we desire.

Here, we give a procedure $\text{RepAll}(x)$ for a condition $x$ to transform its non-strict inequality into the disjunction of the strict inequality and the equation. First, we compute the DNF of the condition by LogicalExpand. Suppose that the DNF is expressed by $c_1 \lor c_2 \lor \cdots \lor c_n$. Take a conjunctive clause $c_i(1 \leq i \leq n)$ in the DNF. If it contains non-strict inequalities, we compute the expression obtained by the following replacement. Replace each occurrence of a non-strict inequality with one of the strict inequality and the equation. Put the all possible replacements for the conjunctive clause $c_i$ into the set $\text{rep}(c_i)$. For example, if the clause $c_i$ is $f \leq 0 \land g \geq 0$, the obtained set $\text{rep}(c_i)$ is $\{f < 0 \land g > 0, f < 0 \land g = 0, f = 0 \land g > 0, f = 0 \land g = 0\}$.

Next, we compute the set $\text{rep}(c_1) \cup \text{rep}(c_2) \cup \cdots \cup \text{rep}(c_n)$. Note that each element of the obtained set is also a conjunctive clause. By connecting all elements of the set by disjunction, we obtain a DNF $\text{RepAll}(x)$.

Suppose that the set $S$ of conditions is given. Take a condition $x \in S$ and compute $\text{RepAll}(x)$ and $\text{RepAll}(\sim x)$. Let $ps(x) = \text{RepAll}(x) \lor \text{RepAll}(\sim x)$. This is the modification of the set $(x, \sim x)$ in the equivalence partitioning $P$. Let us use the same name $P$ for the set of path sensitizing w.r.t. the set $S$ of conditions: $P = \{\wedge_{x \in S} x' \mid x' \in ps(x)\}$.

Figure 3 shows a Mathematica program to do the above path sensitizing for a given set of conditions, and to generate test cases by the path sensitizing. Let $V$ be the set of the variables appearing in $S$. In the program, the mathematical sets $S, V$ and $P$ are expressed by the list $s, v$ and $p$, respectively. The inputs of the procedure are the lists $s$ and $v$, and the output is the list $tc$ of test cases. An element of the list $ps$ denotes the set $ps(x)$ for an element $x$ of $S$.

The requirement for a condition in $S$ is the same as the procedure in the previous subsection.

For a conjunctive clause $c$, let $\text{rep}_p(c)$ and $\text{rep}_z(c)$ be the sets of all possible replacements concerning the inequalities with $\leq$ and those with $\geq$, respectively. In the Mathematica program, for a conjunctive clause $e$, reple and repge are functions to obtain the sets $\text{rep}_p(e)$ and $\text{rep}_z(e)$, respectively. $\text{repall}$ is a function which implements $\text{RepAll}(x)$ for a condition $x$. Let $e$ be the list representing $x$. First, $\text{repall}$ computes the DNF $e_1$ of $e$. Next, $\text{repall}$ transforms the DNF $e_1$ into the list of the conjunctive clauses. Then, for each clause $c_i$ in the list, $\text{repall}$ computes the set $\text{rep}(c_i)$. First, by using reple, it replaces all inequalities with the symbol $\leq$. Next, by using repge, it replaces all inequalities with the symbol $\geq$.

We give the list $s$ and $v$ in the previous subsection to the procedure in Fig. 3. Below we show the intermediate list $ps$.

```
\begin{verbatim}
reple[e_] := (
  ep = Position[e, LessEqual];
  eps = Subsets[ep];
  lis = Map[ReplacePart[
    ReplacePart[e, ep, Complement[ep, #]] & , eps];
  Return[lis]
)

repge[e_] := (
  ep = Position[e, GreaterEqual];
  eps = Subsets[ep];
  lis = Map[ReplacePart[
    ReplacePart[e, ep, Complement[ep, #]] & , eps];
  Return[lis]
)

repall[e_] := (
  e1 = LogicalExpand[e];
  Switch[Head[e1],
    Or, 11 = e1 /. {Or -> List},
    And, 11 = {e1},
    Not, 11 = {e1}];
  12 = Apply[Join, Map[reple[#] & , 11]];  
  13 = Apply[Join, Map[repge[#] & , 12];
  Return[13]
)

ps = Map[Join[repall[!, #], repall[#]] &, s]  
p = Distribute[ps, List]

tc = Map[FindInstance[#, v, Integers] &, p]
\end{verbatim}
```

Fig. 3 The procedure based on boundary value analysis.
6. White-Box Test Case Generation

In the previous section, we give a specification of a triangle to satisfy the symmetry of the variables. The test cases (4), (8) and (10) of Myers are not generated, because these cases are required to have an order of the values of the variables. If we have a mechanism to divide the cases according to the order of the values of variables (e.g. $a < b < c$, $b \leq a < c$, ... for the triangle problem) in the generation procedure, such test cases can also be generated. It is a future work to give such a mechanism in the procedure.

In this paper, to generate the test cases (4), (8) and (10), we take another approach. We give an example of a source program which contains a sorting mechanism of the variables. Then, those test cases are covered by the white-box test cases.

6.1 An Example of a Source Program

Assume that three integers $a$, $b$ and $c$ satisfy $a \leq b \leq c$. In this case, $a$, $b$ and $c$ are the lengths of the three sides of a triangle if and only if $a + b > c$. We can use this if a sorting mechanism exists. The following is an example Java program for the triangle problem.

```
public class Triangle {
    private int a, b, c;

    public Triangle(int a, int b, int c) {
        this.a = a; this.b = b; this.c = c;
    }

    public int type() {
        if (a > b) {
            if (a > c) {a1 = a; b1 = c; c1 = b;}
            else {a1 = a; b1 = b; c1 = c;}
        } else if (b > c) {
            if (b > a) {a2 = b; b2 = a; c2 = c;}
            else {a2 = b; b2 = b; c2 = c;}
        } else if (a > b) {
            if (a > c) {a3 = a; b3 = b; c3 = c;}
            else {a3 = a; b3 = a; c3 = c;}
        } else if (a2 + b3 < c1) r = 1000;
        else if (a2 + b3 == c1) r = 1000;
        else if (a2 == b3) return r;
    }
}
```

6.2 The Model Description for the Source Program

A developer should make a design of a method in a program, before developing the method. For example, a control flow graph [2] is used for a design of a method. The design model used in this paper is simpler than the control flow graph model in [2]. It has only "If-statements" for control. In Fig. 4, we show a part of the model description with graphical representation, for the method type() of the example program. Only the former part of the method, that is, the part of sorting is shown.

We give a formal definition of the model for source programs as follows. A Program is a Block. A Block is a set of Statement's. The Statement's are executed sequentially. A Statement is a Substitution-statement or a If-statement. A Substitution-statement consists of a Variable in the left-hand side, the equality symbol, and an Expression in the right-hand side. In this case, an Expression is substituted with a Variable. An If-statement consists of the following three. The first is a Condition. The second is a Block executed if a Condition becomes true. The third is a Block executed if a Condition becomes false.

We use Mathematica for a model description. A model description has a tree structure. A tree structure can be described by a list in Mathematica. A BNF representation of the model is as follows.

```
<Program> ::= <Block>
<Block> ::= "{" <Statement> { "," <Statement> } "}"
<Statement> ::= <Substitution-statement> | <If-statement>
<Substitution-statement> ::= <Variable> "==" <Expression>
<If-statement> ::= "{" <Condition> ","
<Block> ::= "" <Block> "}"
```

The following is the model description of the method.

```
if (b3 == c1) r = 400;
else r = 300;
else
    if (b3 == c1) r = 300;
    else r = 200;
return r;
```

---

Fig. 4 The model description for the method type() in the program.

type() in the example program. We add three lists \( v \), \( \text{invar} \) and \( \text{outvar} \), to show the sets of all variables, the input variables and the output variables, respectively.

\[
\begin{align*}
\text{prog} &= \{
\{ & a_0 > b_0, \{ a_1 = b_0, b_1 = a_0 \}, \\
& \{ a_1 = a_0, b_1 = b_0 \} \}, \\
\{ & b_1 > c_0, \{ b_2 = c_0, c_1 = b_1 \}, \\
& \{ b_2 = b_1, c_1 = c_0 \} \}, \\
\{ & a_1 > b_2, \{ a_2 = b_2, b_3 = a_1 \}, \\
& \{ a_2 = a_1, b_3 = b_2 \} \}, \\
\{ & a_2 \leq 0, \\
& \{ r = 100 \}, \\
& \{ & a_2 + b_3 < c_1, \\
& \{ r = 100 \}, \\
& \{ & a_2 + b_3 = c_1, \\
& \{ r = 100 \}, \\
& \{ & a_2 = b_3, \\
& \{ & b_3 = c_1, \{ r = 400 \}, \{ r = 300 \} \}, \\
& \{ & b_3 = c_1, \{ r = 300 \}, \{ r = 200 \} \} \} \} \} \} \\
v &= \{ a_0, a_1, a_2, b_0, b_1, b_2, b_3, c_0, c_1, c_2, r \} \quad \text{invar} = \{ a_0, b_0, c_0 \} \quad \text{outvar} = \{ r \}
\end{align*}
\]

Note that the following: we assume that a source program is correctly translated into the model description. For a Java program, the Java byte code can be used for an alternative to the model description [8].

### 6.3 White-Box Test Case Generation Procedure

A white-box test case is an input for a source program which reveals an execution path in the model description of the program. A test case generation procedure should detect such a path with the condition to execute the path.

In Fig. 5, we give a white-box test case generation procedure. The inputs of the procedure are the following four: a model description \( \text{prog} \) and the lists \( v \), \( \text{invar} \) and \( \text{outvar} \) for the sets of all variables, the input variables and the output variables, respectively.

The requirement for \( \text{prog} \) is as follows. It must satisfy the syntactic definition of a model description given in the above. Furthermore, after all substitutions denoted by the instances of \textit{Substitution-statement} are completed, each instance of \textit{Condition} becomes an expression which satisfies the following two: (1) it is a linear Diophantine system and (2) each variable in it is an element of \( \text{invar} \). These conditions are satisfied for example in the following case. Each instance of \textit{Condition} is originally a linear Diophantine system and the right-hand side of each instance of \textit{Substitution-statement} is a linear combination of variables within the input variables and the already assigned variables.

Two functions, \( \text{analyzeblock} \) and \( \text{analyzeif} \) detect all execution paths in a given model description. These are defined using mutual recursion. \( \text{analyzeblock} \) and \( \text{analyzeif} \) deal with a list representing a \textit{Block} and a list representing an \textit{If-statement}, respectively. Each of these returns a list consisting of the conditions to execute a path in the model description.

Note that we use “exhaustive path testing” [9] for the coverage criterion. The reason why we take this criterion is that the number of the paths in the model description is very small in this case. In the model description of a larger program, there are many paths so that we can hardly enumerate all of these. In such a case, we should take statement coverage or branch coverage [9]. To give other procedures based on these criteria is a future work.

By executing \( \text{analyzeblock}[\text{prog}] \), all execution paths in the model description are detected. Here, we give one important attention. When Mathematica reads the model description \( \text{prog} \), it evaluates all the expressions in \( \text{prog} \) immediately. After this, even if we execute \( \text{analyzeblock}[\text{prog}] \), the paths in \( \text{prog} \) cannot be detected correctly. To avoid this, a built-in function \text{HoldForm} must be applied to each expression in \( \text{prog} \). \text{HoldForm} is a function to prohibit the evaluation of an expression temporarily. For example, the expression \( a_0 > b_0 \) is modified to the expression \text{HoldForm}[a_0 > b_0].

By executing \( \text{analyzeblock}[\text{prog}] \) for the modified

```mathematica
analyzeblock[1_] := Module[{idata, data},
    idata = Map[(Length[#] == 3, 
    analyzeif[#, {#}] &), l];
    data = Distribute[idata, List];
    Return[data]
]

analyzeif[1_] := Module[
    {truecond, falsecond, data},
    truecond = analyzeblock[1[[2]]];
    falsecond = analyzeblock[1[[3]]];
    truecond = Map[Join[1[[1]], #] &, truecond];
    falsecond = Map[Join[1[[1]], #] &, falsecond];
    data = Join[truecond, falsecond];
    Return[data]
]

res = analyzeblock[prog]
fi = Map[FindInstance[ReleaseHold[
    Flatten[##], v, Integers] &, res]
case = {} 
Scan[(If[Length[#] == 1, AppendTo[case,
    {invar, outvar} /. ##[[1]]] &), fi] &]
```

Fig. 5 The white-box test case generation procedure.
prog, we obtain 56 execution paths in prog. For example, assume that \(a0 \leq b0 \leq c0\) and these construct a scalene triangle. For the execution path with this assumption, we obtain the following list of conditions.

\[
\begin{align*}
\{& a0 \rightarrow 4, a1 \rightarrow 4, a2 \rightarrow 4, b0 \rightarrow 6, \\
& b1 \rightarrow 6, b2 \rightarrow 6, b3 \rightarrow 6, c0 \rightarrow 8, \\
& c1 \rightarrow 8, c2 \rightarrow 0, r \rightarrow 200\}
\end{align*}
\]

Using \texttt{FindInstance}, we find an instance of the variables satisfying the above conditions. Here, we must apply a built-in function \texttt{ReleaseHold} to the list of the conditions, to remove prohibition of evaluation introduced by the function \texttt{HoldForm}.

\[
\begin{align*}
\text{res} &= \text{analyzeblock}[\text{prog}] \\
\text{Flatten}[\text{res[[56]]}]
\end{align*}
\]

\[
\begin{align*}
\{& ! a0 > b0, a1 == a0, b1 == b0, ! b1 > c0, \\
& b2 == b1, c1 == c0, ! a1 > b2, a2 == a1, \\
& b3 == b2, ! a2 <= 0, ! a2 + b3 < c1, \\
& ! a2 + b3 == c1, ! a2 == b3, ! b3 == c1, \\
& r == 200\}
\end{align*}
\]

First, we show test cases by equivalence partitioning.

\[
\begin{align*}
\text{x1[a_, b_, c_, \_\_]} &= a + b == 10 c + d \\
\text{x2[a_, b_, c_, \_\_]} &= 0 <= a <= 9 && \\
& 0 <= b <= 9 && c >= 0 && 0 <= d <= 9 \\
\text{s} &= \{\text{x1[a, b, c, d]}, \text{x2[a, b, c, d]}\} \\
\text{v} &= \{a, b, c, d\}
\end{align*}
\]

By the program in Fig. 5, all white-box test cases are generated. Here, a test case is a pair of the list of the input values and the list of the output values. Below we show only the test cases concerning (4), (8) and (10) of Myers’ test cases.

\[
\begin{align*}
\{& \ldots, \{7,3,1\},\{100\}, \ldots, \{4,1,3\},\{100\}, \\
& \ldots, \{3,1,7\},\{100\}, \ldots, \{3,1,4\},\{100\}, \\
& \{3,1,3\},\{300\}, \ldots, \{3,7,1\},\{100\}, \\
& \ldots, \{3,3,1\},\{300\}, \ldots, \{1,4,3\},\{100\}, \\
& \ldots, \{1,3,3\},\{300\}, \ldots\}
\end{align*}
\]

Finally, we translate each test case with a Mathematica representation into the test case with a JUnit representation.

\[
\begin{align*}
i &= 31 \\
\text{public void test31()} \\
\text{\{Triangle \_tri = new Triangle(4,6,8); \}} \\
\text{\assertEquals(200,\_tri.type());}
\end{align*}
\]

Note that the following. In the source program and the model description, we used many variables to show the length of a side of a triangle. For example, we used \(b0, b1, b2\) and \(b3\) for the length of a side. However, in an actual development, one may use only one variable \(b\) instead of \(b0, b1, b2\) and \(b3\). To generate test cases correctly, we need referential transparency, so that we have done such a variable renaming in a source program. It is obvious that a test case generation procedure should contain such a variable renaming mechanism. To add such a mechanism to the generation procedure is a future work.

7. Another Example

Our intension is to give procedures which do not depend on target software. To show this, we give another example. Consider an adder for two single figure decimals. The input \((a, b)\) and the output is also a pair of integers \((c, d)\), where \(0 \leq a \leq 9, 0 \leq b \leq 9, c \geq 0, 0 \leq d \leq 9\) and \(a + b = 10c + d\).

First, we show test cases by equivalence partitioning.

\[
\begin{align*}
\{& \{a, b, c, d\} = a + b == 10 c + d \\
& \{a, b, c, d\} = 0 <= a <= 9 && \\
& 0 <= b <= 9 && c >= 0 && 0 <= d <= 9 \\
& s = \{\text{x1[a, b, c, d]}, \text{x2[a, b, c, d]}\} \\
& v = \{a, b, c, d\}
\end{align*}
\]

We need more test cases for the case that both \(x1\) and \(x2\) are true. To obtain more, we only add the optional fourth argument to \texttt{FindInstance}.

Next, we show test cases by white-box testing. (In the model description, we give “\(c == -1\)” for an exceptional case.)

\[
\begin{align*}
\text{cond1} &= \text{HoldForm}[a >= 0 \&\& a <= 9 \&\& b >= 0 \&\& b <= 9 \&\& c >= 0 \&\& d >= 0 \&\& d <= 9] \\
\text{cond2} &= \text{HoldForm}[a + b >= 10] \\
\text{stat1} &= \text{HoldForm}[d == a + b - 10] \\
\text{stat2} &= \text{HoldForm}[c == 1] \\
\text{stat3} &= \text{HoldForm}[d == a + b] \\
\text{stat4} &= \text{HoldForm}[c == 0] \\
\text{stat5} &= \text{HoldForm}[c == -1] \\
\text{prog} &= \{\text{cond1}, \{\text{cond2}, \{\text{stat1}, \text{stat2}\}, \{\text{stat3}, \text{stat4}\}\}\}\{\text{stat5}\} \\
\text{v} &= \{a, b, c, d\}
\end{align*}
\]
invar = \{a, b\}
outvar = \{c, d\}

\{\{8, 2\}, \{1, \theta\\}\}, \{\{\theta, \theta\}, \{\theta, \theta\}\},
\{\{-7\theta, 75\}, \{-1, 36\}\}\}

8. Related Works

8.1 Comparison with HOL-TestGen

In the field of test case generation with formal approach, HOL-TestGen [5] by Brucker and Wolff is an important work. HOL-TestGen is based on a theorem prover Isabelle/HOL [10]. Using several transformation procedures in HOL-TestGen, a test specification is reduced to the testing normal form [5]. In [5], they also deal with the triangle problem. A piece of the testing normal form of the test specification has the form \( x \neq z \), \( 0 < x \), \( 0 < z \), \( z < x + z \), \( x < z + z \) \( \Rightarrow \) prog\((x, z, z) = \) isosceles. Finally, a ground instance of each variable in the testing normal form is computed and then test cases are obtained.

Here, we compare HOL-TestGen and our approach. An advantage of HOL-TestGen is that it is an open source system. The engine Isabelle/HOL and its engine, Poly-ML [11] are also open source systems. Hence, in obtaining test cases by HOL-TestGen, all are transparent. In our approach, some part of generation, for example finding an instance of variables in an expression, relies on built-in commands of Mathematica.

Another advantage of HOL-TestGen is that it has a complexity reduction mechanism. At the phase of computing the testing normal form, a test specification is transformed into the Horn-clause normal form (HCNF). Computation of a HCNF in [5] is less complex than computation of a DNF in [7]. We cannot improve the complexity of a part which relies on Mathematica built-in commands.

HOL-TestGen allows recursive function and abstract data type in a test specification. We expect that these can also be allowed in a specification in Mathematica. A case study for a specification containing these is a future work.

We use Mathematica not only to solve expressions but also as a functional programming language. One can describe any procedure to obtain expressions for generating test cases. Because each procedure given in this paper is only an example, one can freely customize a procedure into a desirable one. We think that this is an advantage of our approach. Because HOL-TestGen is an open source system, one may also customize HOL-TestGen, but this means a change of the HOL-TestGen system.

Note that there is a work which uses a functional programming language to generate test cases: QuickCheck [6] uses Haskell for this aim.

Another good point of our approach is that Mathematica is easy to install in a developing environment. To use HOL-TestGen, we are required to install (compile the source codes) the three systems, Poly-ML, Isabelle/HOL and HOL-TestGen.

8.2 Test Generation by Other Mathematical Systems

We introduce other works which deal with test case generation by a mathematical approach. Our work uses Mathematica throughout a testing process. However, we do not know another work which uses a computer algebra system in such a way. There exist some works which use a computer algebra system or another mathematical system at the final phase of test case generation. At the final phase, in general, equations and inequalities must be solved by using some mathematical system.

Müller et al. proposed a white-box test case generation method for a Java program [8]. (They use the terminology "glass-box testing".) They developed symbolic Java virtual machine (SJVM) to analyze a Java byte code. SJVM analyzes a Java method in the byte code by symbolic execution. SJVM has a backtracking mechanism, so that it can detect all candidates for an execution path in the method. Checking feasibility of a path is out of scope of SJVM. So, the test case generator needs a constraint solver. They use some solvers including Mathematica. In our work, we use a Mathematica program analzyeblock to analyze a Java method. It corresponds to SJVM in their work.

Bensalem et al. proposed a test case generator [3]. It generates test cases to detect an infinite loop in a flowchart. The generator is implemented in Java. However, it invokes Mathematica when it checks feasibility of the condition to fall into an infinite loop.

Henzinger et al. developed a famous model checker BLAST for a C program. As an application of BLAST, Beyer et al. proposed a white-box test case generator for a C program [4]. The generator uses an ILP (Integer Linear Programming) solver at the final phase of test case generation.

An SMT-solver [12] can be an alternative to a computer algebra system. To use a constraint programming system is also an alternative. In case that we implement a generation procedure with Java, Cream [13] can be used for this aim.

9. Conclusion

In this paper, we have introduced a computer algebra system Mathematica for the aim of test case generation. We have given three concrete procedures described in Mathematica to generate test cases.

Here, we discuss applicability of test case generation by Mathematica to real software. We think that after some problems will be solved, this approach will be applied to real applications.

9.1 Solvable Classes of Expressions

In this paper, we consider only one class of solvable expressions. It is a future work to summarize mathematically solv-
able classes and to check actual solvability by Mathematica for each class.

On the other hand, to consider unsolvable classes is also important. It is well-known that to solve an arbitrary Diophantine system is undecidable. Actually, we have found an example which Mathematica cannot find a solution. Take an example of Fig. 4.1 in Myers’ book [9] which is used for an explanation of white-box testing. There is an expression \( x/a \) to compute a quotient on integers. This is expressed by \( \text{Quotient}[x, a] \) in Mathematica. \text{FindInstance} cannot find an instance of the variables satisfying \( a > 1 \) \&\& \( \text{Quotient}[x, a] > 1 \), while these inequalities have a solution for example \((x, a) = (4, 2)\).

\begin{verbatim}
FindInstance[a > 1 \&\& \text{Quotient}[x, a] > 1, {x, a}, Integers]
\end{verbatim}

\begin{verbatim}
FindInstance::nsmet: ---------------------------------------------
FindInstance[a > 1 \&\& \text{Quotient}[x, a] > 1, {x, a}, Integers]
---------------------------------------------
\end{verbatim}

To overcome this situation, for example, we should do the following. First, we solve equations and inequalities on reals. Next, we find the nearest integer values for the obtained solution. Then, we check whether the integer values satisfy the equations and the inequalities.

We have to reveal such unsolvable classes which often appear in software. Then, we give a modified procedure to generate test cases for each class.

9.2 Implementation of a Procedure Based on Cause-Effect Graph

Next, we consider black-box test case generation. Myers’ textbook [9] deals with a real application, a memory dump command of an operating system. This problem is hard to deal only with equivalence partitioning, because the output value of the command is decided by the very complex Boolean operations for a Boolean abstraction of the input values. Then, Myers uses another method “cause-effect graph”. We are now trying to improve the generation procedure by adding the cause-effect graph method, but it is not completed. We will generate test cases for such a real application, by using the completed procedure.

9.3 Extension of the Model for Source Programs

Here, we consider the white-box test case generation procedure. The procedure shown in this paper is very simple, because the model for source programs we have introduced is simple. The procedure should be modified to deal with a more complicated model for source programs. For example, a while-loop or for-loop should be described in a model. A test case generation method which deals with loops is found in [2]. We should also allow method invocation in the right-hand side of Substitution-statement in the model for source programs. To deal with method invocation, we should prepare a test stub for an invoked method.

9.4 Complexity of Test Case Generation

Another problem is the complexity of test case generation. (Here we assume that given expressions are solvable.) We have to generate test cases within realistic time using a realistic computation environment. We have to reveal the scale of software which test cases are generated under such assumptions. For the case of the triangle problem, both black-box and white-box test cases are generated within a moment. The used environment is one PC (Pentium4 3.8 GHz, 3.5 GB RAM, WindowsXP 32 bit, Mathematica 5.2).

We also have to give a method to divide large software. It is easy to find that the method \text{type()} in the triangle problem can be divided into the sorting part and the part for deciding the type of a triangle. We explained that there are 56 candidates of execution paths. In the sorting part and the type deciding part, there are 8 and 7 candidates of paths, respectively. All the 7 candidates of the latter part are feasible, but 2 candidates of the former part are not feasible. (We know that the number of all permutation for three numbers is 6.) In such a case, we should exclude infeasible candidates and make efficient the succeeding test case generation.

References

Satoshi Hattori received the B.E., M.E. and Ph.D. degrees from Osaka University, Osaka, Japan, in 1992, 1994 and 2000 respectively. He was with Graduate School of Information Science, Nara Institute of Science and Technology, as an assistant professor, from 1999 to 2000. He was with School of Information Science, Japan Advanced Institute of Science and Technology, also as an assistant professor, from 2000 to 2008. Since 2008, he has been an associate professor (the title in Japanese is specially appointed lecturer) of Graduate School of Information Science and Engineering, Tokyo Institute of Technology. His current research interests include mathematical foundations for software engineering.