Vol.2021-SE-207 No.28
2021/3/2

振動法を用いたSOFL形式仕様に基づくテストケース自動生成支援ツールの開発

佐伯 賢弥[1,a)]　劉 少英[1]

概要：振動法と呼ばれる形式仕様に基づくテストケース生成技術が近年提案されていたが，振動法を支援するツールが無いため，実際のソフトウェア開発の現場で振動法を応用することは難しい．本研究では，振動法の支援ツールの設計と実装を行った．本ツールでは，プログラムのテストケース生成だけではなく，定理証明を自動的に行うことが可能である．

Development of a Software Tool to Support the Vibration Method for Automatic Test Case Generation from SOFL Formal Specifications

1. はじめに

ソフトウェア開発におけるテスト工程はソフトウェアの動作が要求と合っているかやバグが存在しないかなどを検証するための重要な工程である．テストを行うためにはテストケースの生成が必要であるが，ここに二つの問題がある．有効なテストケースをいかにして生成するかということと，いかにして効率的にテストケース生成するかということである．有効なテストケースとは生成されたテストケースによりソフトウェアのバグを発見することができるかということである．テストケース生成の効率とは低コスト，短時間でテストケースを生成することができるかということである．この二つの問題に共通する原因としてソフトウェアの仕様が曖昧であるということが挙げられる．ソフトウェアの仕様が曖昧であれば，テストケースを自動的に生成することが困難となり，人がテストケースを生成する場合でもどのようなテストケースを生成すべきか分かりにくく，時間がかかってしまう．

形式手法はこのような課題を解決することが可能である．形式手法の利点は形式的にソフトウェア開発を行うことでソフトウェアの信頼性が確保されるということが挙げられる．この理由として，形式手法を用いたソフトウェア開発では，数学的に定義された形式仕様記述言語で仕様や設計を記述するため，自然言語とは違い，曖昧性が排除されるからである．

形式手法を用いた場合，仕様は形式仕様記述言語で書かれているためテストケースの自動生成へのハードルは下がるが，未だに有効的なテストケースの自動生成の課題は未解決のままである．本研究では，近年提唱された振動法 [1] というテストケース生成技術を用いてテストケース自動生成支援ツールの開発を行った．また本ツールでは，テストケース生成源となる形式仕様記述言語として SOFL 形式仕様言語 [2] を支援する．

2. SOFL 形式仕様

本章では本ツールで支援する形式仕様記述言語である SOFL 形式仕様について説明する．

2.1 概要

SOFL 形式仕様は Shaoying Liu によって提唱された形式手法を支援するための形式仕様記述言語の一つで Structured Object-Oriented Formal Language の略である．

言語としての特徴は，SOFL 形式仕様では構築，理解，保守を可能にするために，図と形式記法の効果的な統合を支援している．また手法としての特徴は形式仕様の記述，仕様ベースのテスト，検証の三段階のアプローチを採用している．また，ソフトウェア開発のための構造化手法とオブジェクト指向手法を組み合わせている．

[1] 広島大学 Hiroshima University, 1-3-2, Kagamiyama, Higashihiroshima-shi, Hiroshima, 739-0046 Japan
[a)] b172281@hiroshima-u.ac.jp

入力されたテストの点数が 70 点以上かつ入力されたレポートの点数が 60 点以上 70 点未満，または入力されたテストの点数が 70 点以上 80 点未満かつ入力されたレポートの点数が 70 点以上のとき出力の成績は B とする．ただし入力されるテストとレポートは百点満点とする．

図 1 自然言語で記述された仕様の例

2.2 記述法

SOFL 形式仕様では仕様の機能に対して事前条件と事後条件を記述することによって仕様の機能を定義する．事前条件はプログラムの実行前の条件となり，事後条件はプログラム実行後の条件を表している．また事後条件は入力に対する制約であるテスト条件と出力に対する制約である定義条件の論理積から構成されている．一つの定義条件に関わる一連の事前条件，事後条件の記述を総じて機能シナリオと呼ぶ．
次節で例を挙げて説明する．

2.3 記述例

本節では SOFL 形式仕様の記述例を説明する．説明する仕様の例を**図 1** に自然言語で記述して示す．図 1 のような仕様を SOFL 形式仕様で記述すると**図 2** のようになる．図 2 の 1 行目では，プログラムのメソッド名となるプロセス名と入出力に含まれる変数とそのデータ型が記述される．2 行目の pre 以降で事前条件が記述され，3 行目の post 以降で事後条件が記述される．

3. 振動法

本章では本ツールにおいて中心的な機能であるテストケース生成に用いる技術である振動法について説明する．

3.1 動機

一般にブラックボックステストでは仕様を基にテストケースを生成し，ソフトウェアを実行した際の出力が要求と合っているか検証することを目的としている．しかしながら，プログラムを基にテストケースを生成していないためプログラムの内部構造を考慮することは出来ない．即ち，ブラックボックステストでは，プログラムの内部構造の信頼性を保証することが出来ないため，プログラムの潜在的なバグを検知できない可能性がある．

振動法はこのような問題を解決することが可能である．振動法では形式仕様を基にプログラムの網羅性の高いテストケースを生成することを目的としている．

3.2 原理

振動法は形式仕様記述言語によって記述された論理式に対して距離 *distance* を定義し，その論理式を満たしながら距離を振動させるように大きく，小さくすることによって，テストケースを生成する．

表 1 原子述語における距離 *distance* の定義

関係演算子 R	距離 *distance*	制約
$<$	$E_2 - E_1$	$distance > 0$
$<=$	$E_2 - E_1$	$distance \geq 0$
$>$	$E_1 - E_2$	$distance > 0$
$>=$	$E_1 - E_2$	$distance \geq 0$
$=$	$E_1 - E_2$	$distance = 0$

3.3 距離の定義

本節では SOFL 形式仕様で定義されている数値型の原子述語，論理積に対してどのように距離を定義するか説明する．

3.3.1 原子述語

本節では原子述語に対する距離の定義について説明する．
原子述語である論理式 $E_1(x_1, x_2, ..., x_n)\ R\ E_2(x_1, x_2, ..., x_n)$ を考える．ここで E_1, E_2 は数値型の単項式または多項式で，$x_1, x_2, ..., x_n$ は入力変数である．また R は SOFL 形式仕様で数値型に対して定義された関係演算子 $<, <=, >, >=, =$ のいずれかである．この論理式 $E_1(x_1, x_2, ..., x_n)\ R\ E_2(x_1, x_2, ..., x_n)$ の距離 *distance* を**表 1** のように定義する．

原子述語に対する距離の定義の例を挙げる．図 2 に示した SOFL 形式仕様記述に含まれる原子述語の一つである test>= 70 に対して表 1 より距離 *distance* は test−70 ($distance \geq 0$) と定義される．

3.3.2 論理積

本節では二つ以上原子述語から構成される論理積に対する距離の定義について説明する．
先行研究で [1] は原子述語にのみ距離を定義していたが，本研究では論理積に対しても距離を定義することにより，より有効なテストケースを生成することが可能となる．

n 個の原子述語 $AP_1, AP_2, ..., AP_n$ の論理積 $AP_1 \wedge AP_2 \wedge, ..., \wedge AP_n$ を考える．ここで原子述語 AP_i は論理式 $E_{i1}(x_i)\ R_i\ E_{i2}(x_i)$ で，E_{i1}, E_{i2} は数値型の単項式または論理積であり，x_i は入力変数である．また R_i は SOFL 形式仕様で数値型に対して定義された関係演算子 $<, <=, >, >=, =$ のいずれかである．
この論理積を入力変数 x に対して整理することにより，**表 2** の計算結果のような形に整理することができ，それぞれに対して非負の距離 *distance* を定義する．

論理積に対する距離の定義の例を挙げる．図 2 に示した SOFL 形式仕様記述に含まれる論理積の一つである test >= 70 and test < 80 に対して，70 ≤ test< 80 と整理することができるため，表 2 より距離 *distance* を test−70 ($0 \leq distance < 10$) と定義する．

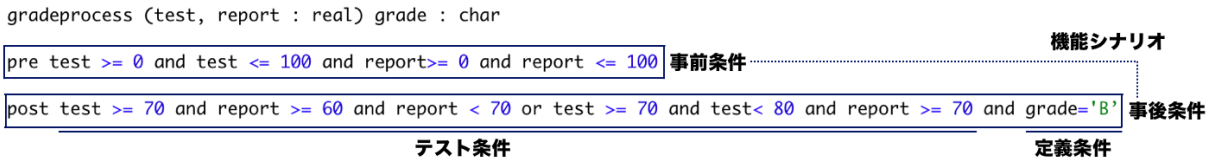

図 2 図 1 の SOFL 形式仕様記述例

表 2 論理積の計算結果と距離 *distance* の定義

	計算結果	距離 *distance*	制約
1	$a < x_i < b$	$x_i - a$	$0 < distance < b - a$
2	$a \leq x_i \leq b$	$x_i - a$	$0 \leq distance \leq b - a$
3	$a \leq x_i < b$	$x_i - a$	$0 \leq distance < b - a$
4	$a < x_i \leq b$	$x_i - a$	$0 < distance \leq b - a$
5	$a < x_i$	$x_i - a$	$distance > 0$
6	$a \leq x_i$	$x_i - a$	$distance \geq 0$
7	$x_i < b$	$b - x_i$	$distance > 0$
8	$x_i \leq b$	$b - x_i$	$distance \geq 0$
9	$x_i = a$	$x_i - a$	$distance = 0$
10	解なし	定義不可能	

a, b は実数で $a \leq b$

3.4 テストケース生成

本節では 振動法を用いてテストケースを生成する方法について説明する．

前節で記述したように原子述語，論理積に対して距離を定義することが可能である．定義した距離には入力変数が含まれており，この距離に対して距離の制約を満たしている任意の実数 d を与え，方程式 $distance = d$ を入力変数に対して解くことでテストケースを得ることができる．

本ツールでは方程式を解く手法には二分法を用いる．先行研究 [1] では代数的に計算をしていたが，本ツールでは二分法により数値的に計算を行うことで高次の変数を含む計算を可能にする．

4. ツール機能

本章では本研究で開発したツールの機能を説明する．

4.1 テストケース自動生成

本節ではテストケース生成機能について説明する．

本ツールでは SOFL 形式仕様で記述された機能シナリオ，論理積または原子述語より振動法に基づいてテストケースを自動生成することが可能である．

4.1.1 条件と制約

はじめに，本ツールにおいてテストケース生成可能な条件と制約を説明する．条件と制約を明確にするため前述した条件や制約も再度記述する．

- テストケースは SOFL 形式仕様に基づく．
  - SOFL 形式仕様で定義されている実数値型に対応する．
  - SOFL 形式仕様で数値型に対して定義されている関係演算子 $<, <=, >, >=, =$ に対応する．
  - SOFL 形式仕様で数値型に対して定義されている演算子 $+, -, *, /$ に対応する．
  - 原子述語，変数の従属がない論理積，またはそれらを含む機能シナリオよりテストケースを生成する．
- テストケース生成は振動法に基づく．
  - 距離は手動またはランダムで与える．
  - 振動回数は手動で与える．
  - 出力は実数値である．
  - 計算は二分法を用いる．

本ツールでは上記の条件，制約の下でテストケース生成を行うことができる．

4.2 自動定理証明

本節では自動定理証明機能について説明する．

ソフトウェア開発のテスト工程において，最も重要な作業の一つとしてテスト結果の検証が挙げられる．テストケースを生成し，テストプログラムを実行しただけではテスト工程は不十分であり，テストを行った結果の出力と要求が一致しているか検証する必要がある．本節で説明する機能はテスト結果の検証を支援する機能である．

4.2.1 定理証明

本ツールでは SOFL 形式仕様に基づいて記述された定理に対して振動法を用いて定理の仮定を満たすデータセットを生成することが可能で，その生成されたデータセットに対して結論が正しいか，そうでないか検証することが可能である．データセットを生成する際の制約は 4.1.1 と同様である．

4.2.2 仕様と定理

4.2.1 で記述したように本ツールでは定理の証明が行える．この定理の考え方を仕様に対して応用することが可能である．

SOFL 形式仕様で記述された一つの機能シナリオは定理として置き換えることが可能である．機能シナリオのテスト条件が定理の仮定部分となり，定義条件が定理の結論部分となる．

このように仕様を定理として扱うことにより，テスト結果を検証することが可能となる．

$a >= 0$ and $b >= 0$ and $c >= 0$ |- $a*a+b*b+c*c >= a+b+c$

図 3 SOFL 形式仕様に基づいて記述された定理の例

表 3 振動法により生成されたデータセットと検証結果

	a	b	c	距離	検証結果
1	0	0	0	0	true
2	0.5	0.5	0.5	0.5	false
3	1	1	1	1	true
4	1.5	1.5	1.5	1.5	true
5	2	2	2	2	true
6	2.5	2.5	2.5	2.5	true
7	3	3	3	3	true
8	3.5	3.5	3.5	3.5	true

振動数 : 7，距離 : 0，振幅 : 0.5

4.2.3 実行例

本節では自動定理証明の例を挙げる．
ここでは，**図 3** のような SOFL 形式仕様に基づいて記述された定理を証明する．この定理の仮定部分は「 |-」より以前の式であり，結論部分は「 |-」以降の式である．この定理の仮定に対して振動法を用いてデータセットを生成し，自動検証を行うと**表 3** のような結果が得られた．表 3 の 2 行目より，検証結果が「false」であるデータが存在するため，この定理は偽であることが証明できた．

4.3 GUI

本節では本ツールの GUI について説明する．
本ツールの全体の UI を**図 4** に示す．図 4 のように本ツールの GUI は 4 つのフレームから構成されている．本節では 4 つのフレームの各機能について説明する．

4.3.1 ファイル管理

図 4 の左上のフレームではテストケース生成や，自動定理証明に関わるファイルの入力が行える．
このフレームは上下二つのフレームに分割されているが，上部のフレームではプログラムのテストケース生成に関わるファイルを，下部のフレームでは自動定理証明に関わるファイルを表示している．
また，入力されたプログラムのテストケース生成に関わるファイルは階層構造で表示され，その階層は上位から，Java ファイル，Java ファイル中に含まれるメソッド，SOFL 形式仕様である．SOFL 形式仕様は txt ファイルで与えられるものとする．
また，自動定理証明に関わるファイルも階層構造で表示され，階層は上位から，定理ファイル，定理である．定理ファイルとは，SOFL 形式仕様により記述された定理が複数含まれているテキストファイルである．

ファイルの入力はこのフレーム右上部に配置されたボタン，または階層構造内で右クリックを行いコンテクストメニューを開くことで行える．また，このコンテクストメニューには，ファイルの削除機能も含まれている．

4.3.2 テストケース・データセット生成

図 4 の中央上部のフレームでは入力された SOFL 形式仕様を基にテストケースの生成や，自動定理証明のデータセット生成が行える．図 4 の中央上部のフレームを拡大したものを**図 5** に示す．
このフレームでは入力された SOFL 形式仕様に含まれる機能シナリオなどの情報が表示され，それらを基にテストケースを生成できる．このフレームは左右二つのフレームに分割されている．
左側のフレームでは，SOFL 形式仕様に含まれる機能シナリオ，その機能シナリオに含まれる論理積が階層構造で表示される．また左側のフレーム下部にはその機能シナリオの定義条件が表示される．
右側のフレームでは，左側のフレームで選択されている論理積を構成する原子述語が表示されている．

本ツールでは，機能シナリオ，論理積，原子述語のいずれからでもテストケースを生成することが可能で，テストケース生成源となる条件を選択したまま，フレーム上部に配置された「Generate」ボタンを選択することにより，テストケースを生成できる．
定理証明を行う場合は図 5 の表記と多少異なるが，機能の仕様は同じである．

また，テストケースを生成する際に，振動法を用いるが，この振動法に対するオプションをフレーム右上部に配置された「Setting」ボタンを選択ことにより指定することができる．
このボタンを選択すると，**図 6** のようなダイアログが出現する．このダイアログで，生成するテストケースのデータ型，距離，振動回数，振動のパターンとその振幅を以下のような条件で指定することができる．

- データ型
  - SOFL 形式仕様で定義されている数値型のみ選択可能である．
- 距離
  - 非負の実数のみ指定可能である．
  - 指定しない場合はランダムに距離が与えられる．
- 振動回数
  - 1 から 20 の範囲で指定可能である．
- 振幅
  - 正の実数で指定可能である．
  - 指定しない場合はランダムで振幅が与えられる．
- 振動パターン
  - 下記 3 パターンより選択可能である．
    - 「Random」ランダムに距離を指定する．
    - 「Insreasing」距離が振幅を基に増加する．
    - 「Decreasing」距離が振幅を基に減少する．

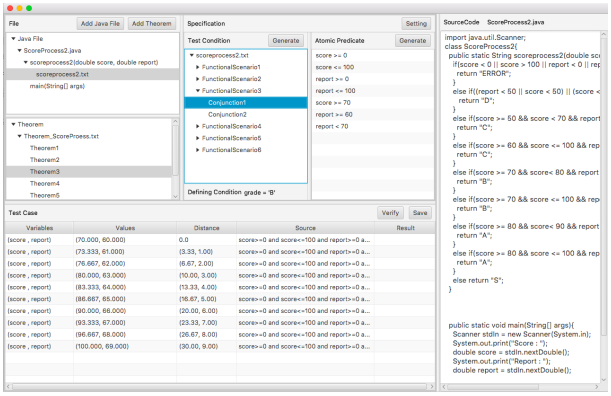

図 4　本ツールの GUI

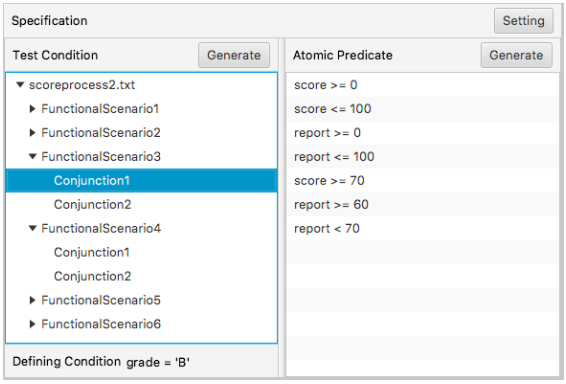

図 5　テストケース生成の GUI

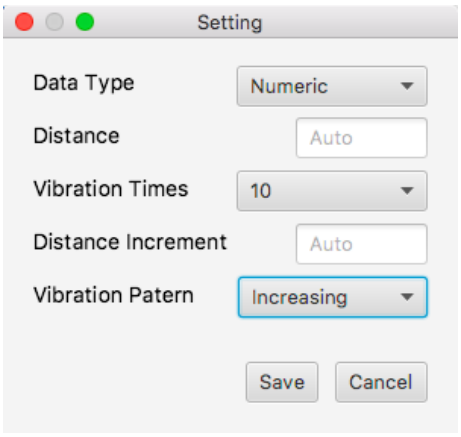

図 6　振動法に対するオプション指定の GUI

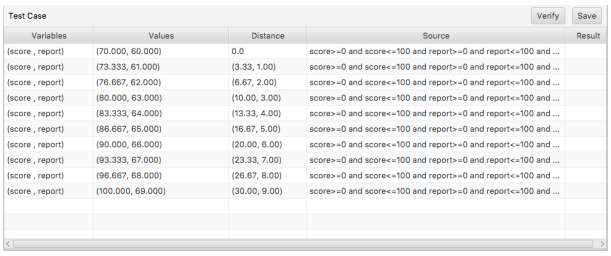

図 7　テストケース・データセットの管理と検証の GUI

```
gradeprocess(test,report:real)grade:char
pre test>=0 and test<=100 and report>=0 and report<=100
post
    test>=90 and report>=80 and grade='S'

    test>=80 and report>=70 and report<80
    or test>=80 and test<90 and report>=80 and grade='A'

    test>=70 and report>=60 and report<70
    or test>=70 and test<80 and report>=70 and grade='B'

    test>=60 and report>=50 and report<60
    or test>=50 and test<70 and report>=60 and grade='C'

    test<50 or report<50
    or test>=50 and test<60 and report>=50 and report<60 and grade='D'
```

図 8　成績処理システムの SOFL 形式仕様

4.3.3　テストケース・データセットの管理と検証

図 4 の左下のフレームでは生成したテストケース，データセットの管理と検証が行える．このフレームを拡大したものを図 7 に示す．

生成されたテストケース，データセットは図 7 のようなテーブルに保持される．このテーブルには，変数名，その値，振動法を用いた際の距離，生成源の条件式が表示される．生成されたテストケース，データセットはフレーム右上部に配置された「Save」ボタンを選択することで CSV 形式で保存することが可能である．

また「Save」ボタンの左側に配置された「Verify」ボタンを選択することで定理の証明を行うことができる．検証の結果は真であれば「true」，偽であれば「false」，変数不足などで判定不可能であれば「nil」となる．

検証する際の制約として，定理の結論部分は SOFL 形式仕様に基づいて記述された原子述語であり，数値型，または文字型である必要がある．

5.　ツール評価

本ツールにより生成されたテストケースがプログラムに存在するバグを発見するために有効であるかミューテーション解析を行うことにより評価する．対象となるシステムに対してプログラムを二つ用意し，それぞれのプログラムに対してミューテーション解析を行う．

5.1　ミューテーション解析

プログラムに含まれている比較演算子，条件演算子，変数に対して表 4 に示すミュータント生成規則 [3] に基づきミューテーション操作を行う．ミュータントが生成されたプログラムに対して，テストを行い，バグを発見すること (キル) ができるか検証する．

5.2　対象システム

入力されるテストの点数とレポートの点数に応じて成績が出力されるシステムを扱う．このシステムの SOFL 形式仕様を図 8 に示す．

表 4　ミュータント生成規則

	名前	生成規則
LCR	Logical Connector Replacement	$a\&\&b \rightarrow \{a, b, a \mid\mid b, true, false\}$
ROR	Relational Operator Replacement	$a > b \rightarrow \{a < b, a <= b, a >= b, true, false\}$
UOI	Unary Operator Insertion	$a \rightarrow \{a++, a--\}$

5.3　比較対象

本ツールで振動法により生成されたテストケースと精度比較を行うため，ブラックボックステストで用いられる境界値分析法により生成されたテストケースと，テストケース生成のコストが低いランダムテストを比較対象とする．以下に評価の対象となるテストケースについて説明する．

- 振動法
  本研究で実装したツールを用いてテストケースを 2 種類生成する．
  SOFL 形式仕様に含まれる機能シナリオより，振動法において振動回数を 4 回，9 回と設定し，それぞれテストケースを生成する．本実験で扱うシステムの SOFL 形式仕様には 10 の論理積が含まれており，振動回数を 4 回とした場合，9 回とした場合，それぞれ 40 件，90 件のテストケースが生成される．
- 境界値分析法
  SOFL 形式仕様を基にして境界値分析法よりテストケースを 2 種類生成する．2 種類の内訳は一つの範囲に対して上限値と下限値を算出する場合と，上限値と下限値に加えて代表値を算出する場合である．
  本実験で扱うシステムの SOFL 形式仕様には変数が二つあり，それぞれの変数に対して範囲が 10 個存在しているため，それぞれの範囲に対して上限値，下限値を算出し，各変数においてのテストケースの組合せを考慮し，40 件，代表値も含めた場合，90 件のテストケースを生成する．
- ランダムテスト
  0 から 100 までの実数をランダムに生成し，これをテストケースとする．テストケースは 40 件と 90 件の 2 種類生成する．

5.4　実験 1

5.4.1　対象プログラム

節 5.2 で示した SOFL 形式仕様の記述に対して，プログラムの記述がほとんど同じであるプログラムを扱う．このプログラムを**図 9** に示す．

5.4.2　結果と考察

結果を**表 5** に示す．振動法と，境界値分析法を比較すると，ROR によって生成されたミュータントを含むプログラム以外では振動回数，分析方法に関わらず同等の精度を示した．これは，境界値分析法においての下限値，上限値がそれぞれ，振動法においての距離の最小値，最大値に相当するためであると考えられる．

ROR によって生成されたミュータントを含むプログラムに対して解析を行った結果，振動法でのキル数が境界値分析法に 1 件劣った理由としては，プログラムには二つの入力変数があり，境界値分析法ではそれぞれの変数に対して，上限値，下限値の組み合わせることによりテストケースを生成したが，振動法では，距離の最小値同士の組み合わせ，距離の最大値の組み合わせのみでテストケースを生成したため，振動法では生成できないテストケースが存在し，このような結果となったと考えられる．

一方で，この点以外では振動法と境界値分析法はほとんど同等の結果を示しており，これは仕様に対する記述とプログラムの記述がほとんど同じであったため振動法と境界値分析法の精度の差が大きく出なかったと考えられる．

また，ランダムテストでは，テストケース生成に関わるコストはランダムに値を生成するため非常に低く，90 件生成した場合は全てのミュータントに対して約 60%と一定の精度を示したが，振動法の精度が勝る結果となった．

5.5　実験 2

実験 1 より，境界値分析法と振動法の結果がほとんど同等であるという結果が得られた．しかしながら，仕様とプログラムの記述がほとんど同じであるという場合は，実験 1 で扱ったような基礎的なプログラムを除いては稀な場合であると考えられる．そこで実験 2 では，プログラムの記述が詳細化されたような場合においてミューテーション解析を行い，本ツールの評価を行う．

5.5.1　対象プログラム

節 5.2 で示した SOFL 形式仕様の記述に対して，プログラムの記述が詳細化されたプログラムを扱う．このプログラムを**図 10** に示す．

5.5.2　結果と考察

結果を**表 6** に示す．振動法により生成されたテストケースの精度が最も高い数値を示した．また，振動法においてテストケース 40 件で解析した結果と，境界値分析法においてテストケース 90 件で解析した結果が僅かではあるが振動法の方が高いキル割合を示しており，より少ないテストケースで高い精度を示した．

この結果より，振動法はプログラムが仕様に対してより詳細化された場合により有効的であることが確認できた．

しかしながら，実験 1 でのミューテーション解析の結果と比べ，実験 2 では精度が下がった．これは使用するプログ

```
public static char gradeprocess1(double test, double report){
    if((report < 50 || test < 50) || (test < 60 && report < 60)){
      return 'D';
    } else if((test >= 50 && test < 70 && report >= 60 && report <= 100)||(test >= 60 && test <= 100 && report >= 50 && report < 60)){
      return 'C';
    } else if((test >= 70 && test< 80 && report >= 70 && report <= 100)||(test >= 70 && test <= 100 && report >= 60 && report < 70)){
      return 'B';
    } else if((test >= 80 && test< 90 && report >= 80 && report <= 100)||(test >= 80 && test <= 100 && report >= 70 && report < 80)){
      return 'A';
    } else if(test >= 90 && test<= 100 && report >= 80 && report <= 100){
      return 'S';
    } else{
      return 'E';
    }
  }
```

図 9　実験 1 で扱うシステムを実装したプログラム

```
public static char pradeprocess2(double test, double report){
    if(test >= 90 && test< 93 && report >= 80 && report <= 100) return 'S';
    else if(test >= 93 && test< 97 && report >= 80 && report <= 100) return 'S';
    else if(test >= 97 && test<= 100 && report >= 80 && report <= 100) return 'S';

    else if((test >= 87 && test< 90 && report >= 80 && report <= 100)||(test >= 87 && test <= 100 && report >= 77 && report < 80)) return 'A';
    else if((test >= 83 && test< 87 && report >= 77 && report <= 100)||(test >= 83 && test <= 100 && report >= 73 && report < 77)) return 'A';
    else if((test >= 80 && test< 83 && report >= 73 && report <= 100)||(test >= 80 && test <= 100 && report >= 70 && report < 73)) return 'A';

    else if((test >= 77 && test< 80 && report >= 70 && report <= 100)||(test >= 77 && test <= 100 && report >= 67 && report < 70)) return 'B';
    else if((test >= 73 && test< 77 && report >= 67 && report <= 100)||(test >= 73 && test <= 100 && report >= 63 && report < 67)) return 'B';
    else if((test >= 70 && test< 73 && report >= 63 && report <= 100)||(test >= 70 && test <= 100 && report >= 60 && report < 63)) return 'B';

    else if((test >= 64 && test < 70 && report >= 60 && report <= 100)||(test >= 60 && test <= 100 && report >= 57 && report < 60)) return 'C';
    else if((test >= 56 && test < 64 && report >= 60 && report <= 100)||(test >= 60 && test <= 100 && report >= 53 && report < 57)) return 'C';
    else if((test >= 50 && test < 56 && report >= 60 && report <= 100)||(test >= 60 && test <= 100 && report >= 50 && report < 53)) return 'C';

    else if(test < 60 && report < 60) return 'D';
    else if((test < 50 && report >= 60 && report < 73) || (test < 50 && report >= 73 && report < 87) || (test < 50 && report >= 87 && report <= 100)) return 'D';
    else if((report < 60 && test >= 60 && test < 73) || (report < 50 && test >= 73 && test < 87) || (report < 50 && test >= 87 && test <= 100)) return 'D';
    else return 'E';
}
```

図 10　実験 2 で扱うシステムを実装したプログラム

ラムに精度が依存するということ示しており，課題が残った．
一方で，ランダムテストでは実験 1 と比べて精度が大きく変化しておらず，対象プログラムに依存することなく一定の精度を示していた．

6. 終わりに

本研究では振動法を用いた SOFL 形式仕様に基づくテストケース自動生成支援ツールの開発とその検証を行った．その結果，振動法で生成したテストケースがプログラムに潜むバグの検知に有効であることが分かった．特に，仕様の記述からプログラムの記述が詳細化された場合に，他のテストケース生成法に比べ，より有効であることを示した．

しかしながら，本ツールには一定の制約があり，特に二変数以上が依存した論理式に対してテストケースを生成することができないことは大きな制約の一つである．一般に多くのシステムで複数の変数が従属した条件式を扱うため，この多変数が従属した論理式に対して，振動法においてどのように距離を定義すべきかが課題となっている．また本ツールの制約として数値型以外のデータ型は扱えないため，他のデータ型への拡張も必要である．
またミューテーション解析の結果，振動法は一定の精度を示したものの，精度はプログラムに依存するという課題が残った．しかしながらランダムテストでは精度はプログラムに大きく依存しないという結果が得られたため，振動法において距離をランダムに与えるといったことに対しても今後の課題として検討していきたい．

参考文献

[1] Shaoying Liu, Shin Nakajima (2011) ”A ”Vibration” Method for Automatically Generating Test Cases Based on Formal Specifications,” 18th Asia-Pacific Software Engineering Conference, pp. 73-80
[2] Shaoying Liu (2004) ”Formal Engineering for Industrial Software Development,” Springer-Verlag
[3] Goran Petrović, Marko Ivanković (2018) ”State of Mutation Testing at Google,” 40th IEEE/ACM International Conference on Software Engineering, pp. 163-171
[4] A. Jefferson Offut, Shaoying Liu (1999) ”Generationg test data from SOFL specifications,” The Journal of Systems and Software 49, pp. 49-62
[5] Shaoying Liu, A. Jeff Offut, Chris Ho-Stuart, Yong Sun, Mitsuru Ohba (1998) ”SOFL: A Foraml Engineering Methodology for Industrial Applications,” IEEE Transactions on Software Engineering, 24(1), pp. 24-45
[6] Jonathan Peter Bowen, Kirill Bogdanov, John A. Clark, Mark Harman, Robert M. Hierons, Paul John Krause (2002) ”FORTEST: Formal Methods and Testing,” 26th Annual International Computer Software and Applications Conference, pp. 91-101
[7] 中島 震 (2007) ”ソフトウェア工学の道具としての形式手法,” NII

表 5 実験 1 の解析結果

テストケース生成手法	備考	テストケース数	ミュータント種類	ミュータント数	キル数	キル割合
振動法	振動回数 4 回	40	LCR	135	87	64.444%
			ROR	160	138	86.250%
			UOI	64	51	79.688%
			計	359	276	76.880%
	振動回数 9 回	90	LCR	135	87	64.444%
			ROR	160	138	86.250%
			UOI	64	51	79.688%
			計	359	276	76.880%
境界値分析法	境界値	40	LCR	135	87	64.444%
			ROR	160	138	86.250%
			UOI	64	52	81.250%
			計	359	277	77.159%
	境界値，代表値		LCR	135	87	64.444%
			ROR	160	138	86.250%
			UOI	64	52	81.250%
			計	359	277	77.159%
ランダムテスト		40	LCR	135	76	56.296%
			ROR	160	81	50.625%
			UOI	64	21	32.813%
			計	359	178	49.582%
		90	LCR	135	87	64.444%
			ROR	160	106	66.250%
			UOI	64	21	32.813%
			計	359	214	59.610%

キル割合はミュータント数に対するキル数の割合により算出した．

表 6 実験 2 のミューテーション解析結果

テストケース生成手法	備考	テストケース数	ミュータント種類	ミュータント数	キル数	キル割合
振動法	振動回数 4 回	40	LCR	395	287	72.658%
			ROR	496	348	70.161%
			UOI	208	105	50.481%
			計	1099	740	67.334%
	振動回数 9 回	90	LCR	395	297	75.190%
			ROR	496	373	75.202%
			UOI	208	108	51.923%
			計	1099	778	70.792%
境界値分析法	境界値	40	LCR	395	226	57.215%
			ROR	496	265	53.427%
			UOI	208	115	55.288%
			計	1099	606	55.141%
	境界値, 代表値	90	LCR	395	267	67.595%
			ROR	496	349	70.363%
			UOI	208	122	58.654%
			計	1099	738	67.152%
ランダムテスト		40	LCR	395	243	61.519%
			ROR	496	243	48.992%
			UOI	208	29	13.942%
			計	1099	515	46.861%
		90	LCR	395	268	67.848%
			ROR	496	282	56.855%
			UOI	208	73	35.096%
			計	1099	623	56.688%

キル割合はミュータント数に対するキル数の割合により算出した．