形式仕様に基づくテストケースの自動生成と形式仕様に基づくテストケースの自動生成とテスト結果の自動評価テスト結果の自動評価

IKEDA, Hayato / 池田, 逸人

(出版者 / Publisher)
法政大学大学院情報科学研究科
(雑誌名 / Journal or Publication Title)
法政大学大学院紀要. 情報科学研究科編
(巻 / Volume)
12
(開始ページ / Start Page)
1
(終了ページ / End Page)
6
(発行年 / Year)
2017-03-31
(URL)
https://doi.org/10.15002/00014401

形式仕様に基づくテストケースの自動生成とテスト結果の自動評価

Automatic Test Case Generation and Test Result Evaluation based on Formal Specifications

池田　逸人 *

Hayato Ikeda

法政大学情報科学研究科情報科学専攻

Email: hayato.ikeda.7r@stu.hosei.ac.jp

***Abstract*—Software testing is a time-consuming activity and automatic testing is a desirable solution to this problem. In this paper, we describe a software supporting tool for automatic test case generation and automatic test result evaluation based on formal specifications written in the Structured Object-oriented Formal Language (SOFL). We discuss the algorithms for generating test cases from atomic predicates and their conjunctions that may involve various operations on data items of various data types such as set, sequence, and composite types. We describe the details of the software tool by presenting the four major functions implemented: (1) editing a SOFL specification, (2) generating test cases from the specification, (3) managing test cases in files, and (4) evaluating test results. We also present an experiment to show that our tool can significantly save time in test case generation.**

1. まえがき

ソフトウェアは，ATM，交通システム，医療機器等の様々な分野で利用されている．その中で，私たちの生活に欠かせないお金や交通システム，人命に関わるシステム等は，プログラムのエラーでシステムが停止することは合ってはならない．よって，信頼性のあるソフトウェアを開発するために十分なテストと結果の検証が必要である．信頼性を確保するために，形式手法および形式仕様に基づくテスト技術が利用できる [1]．しかしながら，テストと評価に時間がかかるという問題点が残っている．この問題を解決するために，形式仕様から自動的にテストケース生成する方法と評価するツールを開発した．ツールは形式仕様の入力変数，出力変数，事前条件と事後条件を解析して，仕様を満たすかどうかを確かめるためのテストケースを自動生成する．

以下 2 章では，形式仕様からテストケースを生成する方法について説明する．3 章では，作成したツールにつて説明する．4 章では，テストケースを生成能力を比較する実験について説明する．5 章では，考察と今後の課題について説明し，最後に 6 章で本論文をまとめる．

2. 形式仕様によるテストケース生成方法

SOFL 形式仕様 [2] からテストケースを自動生成することを試みる．SOFL 形式仕様記述では，操作の機能を Process 仕様 で定義する．SOFL 形式仕様の Process 仕様で定義された入力変数の値の生成を試みる．この入力変数の値がテストケースとなる．Process 仕様では，入力変数 S_{iv}，出力変数 S_{ov}，事前条件 S_{pre} と事後条件 S_{post} を定義する．S_{pre} と S_{post} は，述語論理で表現される．事後条件 S_{post} は，$S_{post} \equiv (G_1 \wedge D_1) \vee (G_2 \wedge D_2) \vee \cdots \vee (G_n \wedge D_n)$　と表現できる．この条件の中で，G_i は出力変数を含まない“guard-condition”と呼ばれる条件である．D_i は出力変数を含む“defining-condition”と呼ばれる．テストケースの生成や評価を行うためには，S_{pre} と S_{post} から Functional Scenario Form(FSF) に変換する [3].

定義 *1 (Functional Scenario Form).* Functional Scenario Form は，$(S_{pre} \wedge G_1 \wedge D_1) \vee (S_{pre} \wedge G_2 \wedge D_2) \vee \cdots \vee (S_{pre} \wedge G_n \wedge D_n)$ と定義する．ここで，$(S_{pre} \wedge G_i \wedge D_i)$ を Functional Scenario(FS) と呼ぶ.

F_i に対してそれぞれ S_{iv} の具体的な値がテストケース T_i である．T_i が満たす必要がある条件は，FSF から D_i

* Supervisor: Prof. Shaoying Liu

を除いた Test Condition Form である．

定義 *2 (Test Condition Form).* $(S_{pre} \wedge G_1) \vee (S_{pre} \wedge G_2) \vee \cdots \vee (S_{pre} \wedge G_n)$ を Test Condition Form と定義する．ここで，$(S_{pre} \wedge G_i)$ を Test Condition(TC) と呼ぶ．

3. テストケース自動生成方法

この章では，章 2 で示した方法を自動で行うプログラムのアルゴリズムについて説明する．次の図 1 の手順でテストケース自動生成を行う．

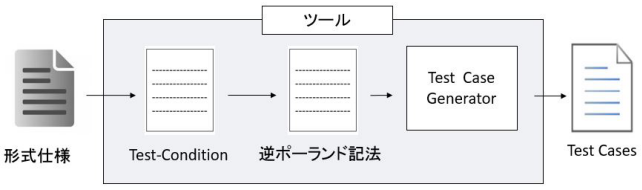

図 1: テストケース生成手順

まず Process 仕様の事前条件 S_{pre} と事後条件 S_{post} から TC を導出する．次に，TC を逆ポーランド記法に変換する．そして，Test Case Generator によって TC を満たす値を生成して，生成されたものがテストケースである．

3.1. 逆ポーランド記法に変換する

原子論理式を逆ポーランド記法に変換する．演算子を非演算子（入力変数や値）の後ろの形にする．逆ポーランド記法に変換することで演算子を適用する順番を知ることができる．例えば，“$1 + 3 * x > 0$”を逆ポーランド記法に変換すると“$1\ 3\ x\ *\ +\ 0\ >$”であり，節 3.2 で左から読み取る．

3.2. TC からテストケース生成

TC は連言標準形であるので，$TC \equiv P_1 \wedge P_2 \wedge \cdots \wedge P_n$ のように表現できる．P_i は節で，原子論理式または否定を含む原子論理式である．

3.2.1. 論理積からテストケース生成

TC からテストケースを求めるアルゴリズムを疑似コード 1 に示す．まず，P_1 に対して Generate Test Case によってテストケース生成を行う．P_i を原子論理式に変換する．節 3.2.2. で否定から原子論理式に変換する方法について，節 3.3. にて原子論理式からテストケース自動生成する方法について説明する．そして求めた S_{iv} は $P_2 \wedge \cdots \wedge P_n$ に代入する．この動作を P_n まで繰り返し，テストケースを生成したら，TC が真であるかどうかを判断する．TC が真であれば，条件を満たすテストケースである．偽であれば生成は失敗であり，求めた S_{iv} を初期化して，結合法則を利用して原子論理式 $P_1, ..., P_n$ の順番を入れ替える．テストケースを生成する順番を変えることで，成功することがあるためである．入れ替える TC の順番は n!通り存在し，TC が真になるまでこの方法を適用する．疑似コード 1 の Permutation は，BaseTC に対して原子論理式の順番を入れ替えた i 通り目の TC を取得する関数である．それでもテストケースを生成できない場合は，テストケースを“nil”にする．

Algorithm 1 Generate test case from Test Condition

```
1: (TC = P_1 ∧ P_2 ∧ ··· ∧ P_n)
2: BaseTC ⇐ TC
3: BaseS_iv ⇐ S_iv
4: for i ← 1 to n! do
5:    for j ← 1 to n do
6:       S_iv ⇐GenerateTestCase(P_j)
7:       TC に S_iv を代入する
8:    end for
9:    if TC = true then
10:      S_iv がテストケースとなる
11:      break;
12:   else
13:      S_iv ⇐ BaseS_iv
14:      TC ⇐ Permutation(BaseTC, i)
15:   end if
16: end for
```

3.2.2. 否定から原子論理式に変換

節 3.3. にて原子論理式からテストケース自動生成する際に，not があるとテストケース自動生成に支障がある．節 3.2.1 で述べた，TC の任意の P_i が否定演算子 not を含んでいた場合は，次の式のように not を含まない形に変換してテストケースを試みる．原子論理式に含まれる関係演算子 Θ ($\Theta \in \{<, <=, >, >=, =, <>\}$) を反対の意味をもつ演算子Θ’ にする．Θに対応するΘ’ を表 1 に記載した．

$$not\ E_1 \Theta\ E_2 \Leftrightarrow\ E_1 \Theta' E_2 \tag{1}$$

例えば,「$not\ x > 0$」は「$x <= 0$」に変換して，テストケース自動生成を試みる．

表 1: 演算子Θと反対の意味を持つ演算子Θ’

Θ	Θ’
>	<=
<	>=
>=	<
<=	>
<>	==
==	<>

3.3. 原子論理式からテストケース生成方法

次の 2 つの手順によって，テストケースを自動生成する．1. 1 つの演算子を含む式に変換する．2. オペランドの値を求める．

3.3.1. 1 つの演算子を含む式に変換

テストケース自動生成を行うために，1 つの演算子を含む式に変換する．優先度の低い演算子に対して，その演算子とオペランドの結果を仮のオペランドと置くことで，演算子が 1 つの式に変換する．優先度を知るために式を逆ポーランド記法に変形し，オペランドと演算子の結果を仮のオペランドと置いて，最終的に式が E_1 Θ E_2 ($\Theta \in \{<, <=, >, >=, =, <>\}$) の形になるまでこの動作を行う．

逆ポーランド記法に変形された原子論理式を演算子が 1 つの式に変換の流れを疑似コード 2 に示す．トークンを左から順に取り出し，演算子だった場合はそのオペランドを仮のオペランドとおいて，OperandTable に (仮のオペランド名, 元の式，“ nil ”) を加える．“ nil ”は値を格納するための初期値で，後で節 3.2.2. にて求めた値を保存する．トークンを最後まで読み取るまで繰り替えす．例えば，$x+2*y>0$ は, 逆ポーランド記法に変換すると「$x\ 2\ y\ *\ +\ 0\ >$」となる．初めに，“x 2 y”はオペランドであるのでスタックにプッシュする．“$*$”は演算子であるので，そのオペランド’2 y’ を取り出し，これらの結果を仮のオペランド AS_1 と置いて，OperandTable に (AS_1，“ 1 x + ”，“ nil ”) を加える．“ nil ”の部分は節 3.3. によって生成した AS_1 の値として値を格納する．最終的に，$AS_2 > 0$ ($AS_1 = 2 * y,\ AS_2 = x + AS_1$) と変形できる．

Algorithm 2 Dump token to Stack

```
for i ← 0 to トークンの数 −1 do
    if tokens(i) ∈ Operators then
        Operator ⇐ tokens(i)
        Counter ⇐ Counter + 1
        j ⇐ i - Operator のオペランドの数
        OperandTable に (AS_Counter, Tokens(j), ... ,Tokens(i), nil) を加える
    else
        pushStack(token(i))
    end if
end for
```

3.3.2. データ生成

まずは， $E_1\ \Theta\ E_2$ を $E\ \Theta\ V$ の形に変換することを考える．V は具体的な値である．E_1，E_2 が両方とも具体的な値でない場合は次のような式に変形する．

$$E_1 \Theta\ E_2 \Leftrightarrow\ E_1 \Theta\ V, E_2 = V \tag{2}$$

E_2 に対してランダムな値を与え，$E_1\ \Theta\ E_2 \Leftrightarrow\ E_1\ \Theta\ V, E_2 = V$ のように変形して，それぞれの条件に対してテストケースを生成する．E_2 が具体的な値の場合は，演算子と値によってテストケースを生成する．E_1 が具体的な値の場合は，$V\ \Theta\ E_2 \Leftrightarrow\ E_2\ \Theta$ ’ V と変形する．Θ ’ はΘの反対の意味の演算子である（表 1）．

$E\ \Theta\ V$ の形に変換出来たら，演算子と V によって $E\ \Theta\ V$ を満たす値 V_E を求める．関係演算子Θに対して，$E\ \Theta\ V$ から E の具体的な値 x を求める方法を表 1 に記載した．例えば，$E > 2$ であれば，$E = 3$ となる．E が仮のオペランドであれば，$V_E = \Psi\ (a_1, a_2, ..., a_n)$ に対しても,$a_1, a_2, ..., a_n$ の値を求める．Ψは演算子，$a_i (i \in \{1, ..., n\})$ はΨのオペランドである．a_i が仮のオペランドであれば同様の動作を行い，TC に含まれる入力変数 S_{iv} の値が求まるまで繰り返す．本研究でテストケース生成の対象とした演算子Ψを表 5 に記載した．

表 2: 関係演算子Θによるテストケース生成

演算子Θ	テストケース生成方法
$E > V$	$E = V + \alpha\ (\alpha > 0)$
$E < V$	$E = V - \alpha\ (\alpha > 0)$
$E >= V$	$E = V + \alpha\ (\alpha >= 0)$
$E <= V$	$E = V - \alpha\ (\alpha >= 0)$
$E <> V$	$E \neq V$
$E = V$	$E = V$

3.3.3. 事例

次のような原子論理式からテストケース自動生成を行う例について述べる．

$$card(inter(a, b)) > 2 \tag{3}$$

(a,b は set of int 型の入力変数)

card は集合の要素数を取得する関数で，inter は共通集合を取得する演算子である．集合の演算子ごとにテストケースを生成する方法を表 4 に記載した．上の TC を逆ポーランド記法に変換して，節 3.3.1 の方法を用いて演算子を 1 つした場合，仮のオペランド AS_1, AS_2 を使って最終的に $AS_2 > 2$ に変換することが出来る．OperandTable の中身は次の表 3 のようになる．

次に元の式である以下の論理式について考える．

$$AS_2 \quad > \quad 2 \tag{4}$$

表 3: OperandTable

仮のオペランド	式	値
AS_1	a b inter	nil
AS_2	AS_1 card	nil

演算子は「>」であり結果が「2」であることから，表 2 の方法を用いて AS_2 は 2 よりも大きい値をランダムで生成する．この方法によって論理式を満たす値「AS_2 = 3」を求めることができる.「AS_2」は仮のオペランドであるので，元の式についても考える．

$$AS_2 \quad = \quad card(AS_1) \ = \ 2 \tag{5}$$

$$AS_1 \quad = \quad \{-1, 1, 3\} \tag{6}$$

演算子は「card」であり結果が「3」であることから，表 4 の方法を用いて集合の要素を 3 つ含む AS_1 を求める．要素の値は int 型であるので，int 型のランダムによって生成する．この方法によって論理式を満たす値「AS_1 = {-1,1,3}」を求めることができる．同様に「AS_1」は仮のオペランドであるので,「$inter(a, b) = \{-1, 1, 3\}$ に対しても，a,b の値を求める．

$$AS_1 \ = \ inter(a, b) \quad = \quad \{-1, 1, 3\} \tag{7}$$

$$a = \{-1, 1, 3\} \quad , \quad b = \{-1, 0, 1, 3\} \tag{8}$$

演算子は「inter」であり結果が「AS_1 = {-1,1,3}」であることから，表 4 の方法を用いて a,b の共通集合が {-1,1,3} となるような a,b を求める．よって，上の論理式を満たす値「a = {-1,1,3}, b = {-1,0,1,3}」を求めることができる．最終的に，論理式 (1) を満たす a,b を求めることができ，これらの値がテストケースとなる．他の方の演算子を含む場合でも，同様に論理式を満たす値を求めることができる．

表 4: 集合型演算子によるテストケース生成

演算子	意味	テストケース生成方法
$union(x_1, x_2) = V$	$x_1 \cup x_2 = V$	(1)V の要素をいくつか含む x_1 を求める (2)x2 = $V \setminus x_1$
$inter(x_1, x_2) = V$	$x_1 \cap x_2 = V$	(1)$x_1 = V, x_2 = V$ (2)x_2 にランダムな要素 r を加える
$diff(x_1, x_2) = V$	$x_1 \setminus x_2 = V$	(1) ランダムな要素 r を生成 (2)$x_1 = V \setminus r$ (3)$x_1 = r$
$get(x_1) = V$	$x_1 \in V$	要素に V を含む x_1 を求める
$card(x_1) = V$	$\|x_1\| = V$	要素を V つ含む x_1 を求める
$subset(x_1, x_2) = V$	$x_1 \subseteq x_2 == V$	・V が真の場合 $x_1 = x_2$ ・V が偽の場合 $x_1 <> x_2$

表 5: 本手法で対象の演算子一覧

演算子の型	演算子
数値型	+, -, *, /
集合型	card, get, union, inter, diff, duion, dinter, power
列型	len, hd, tl, elems, inds, conc, dconc, sub
複合型	mk, modify, select
写像型	dom, rng, override, domrt, domrb, rngrt, rngrb, comp
関係演算子	=, <>, >, <, >=, <=
論理演算子	and, not

4. 支援ツール

章 3 で述べた方法を用いて，テストケース自動生成と自動評価を行うツールを開発した．SOFL 形式仕様を FSF で記述して，FS 毎にテストケースを自動生成して評価を行う．ツールの画面が，2 のなっている．今回開発したツールの機能は「エディタ」,「テストケース生成」,「テストケース表示/保存」,「評価」の 4 つに分けられる．次の節 4.1 でエディタ機能，節 4.2 でテストケース生成機能，節 4.3 でテストケース表示/保存機能について説明する．

4.1. エディタ機能

エディタには「New」,「編集」,「保存」の 3 つの機能がある.「New」では，Module，Process 仕様，FS を新規作成することが出来る．Module を新規作成すると SOFL 形式仕様の Module に関する fMoulde ファイルが作成される．Module の中では，typeDeclaration(型定義)

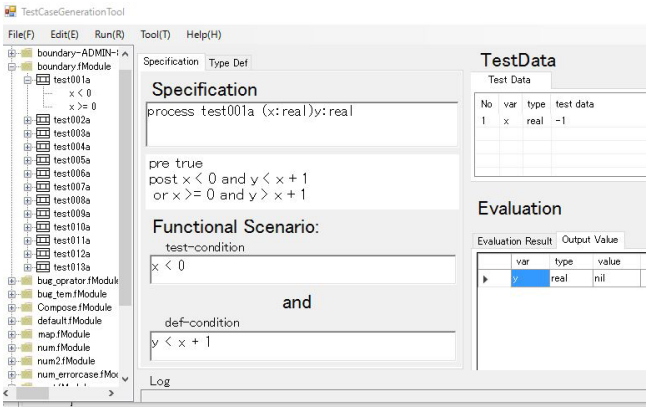

図 2: ツールの画面

と Process 仕様を編集できる．Process 仕様では，FS 毎に test-condition と def-condition(defining-condition) を編集することが出来る．編集した Module，Process 仕様，FS は fMoulde ファイルに保存することが出来る．テキストエリアでのテキスト編集は，コピー，貼り付け，Undo 機能，Redo 機能などの機能がある．エディタ画面は，次の図 3 のようになっている．

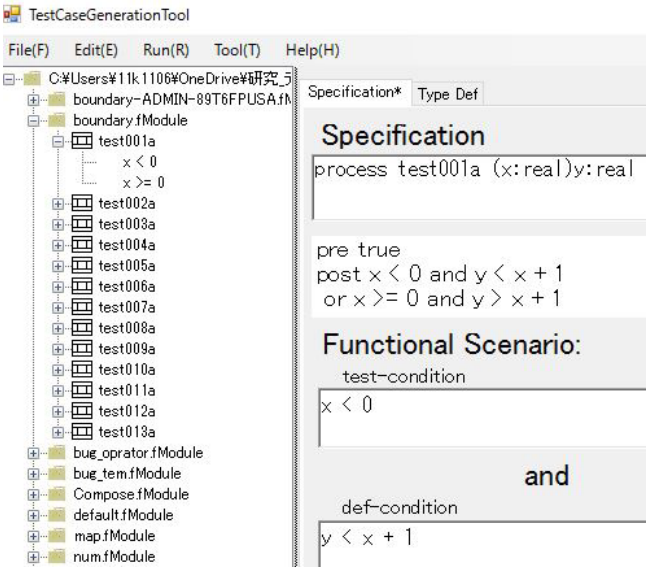

図 3: エディタ画面

左のツリー状の画面は，Module, Process 仕様, FS を表示，選択する画面である．＋を押すと中身が展開され，Module なら含まれる Process 仕様が，Process 仕様なら含まれる FS が展開される．図 1 では test001a が Process 仕様で，その下に表示されている $x < 0$ と $x >= 0$ が 2 つの test-condition である．

4.2. テストケース生成機能

ツールによって記述された test-condition を読み取ってテストケース生成を行う．メニューバーの「Run」から「Generate」を押すと，節 3.2. の方法を用いてテストケースを生成する．テストケース生成をする際に，表 2 の関係演算子 Θ からテストケース生成するとき α の値を最小にする.「Random Generate」を押すと，表 2 の関係演算子 Θ からテストケース生成するとき α の値をランダムに設定してテストケースを生成する．そうすることで，生成されるテストケースにばらつかせることができる．

4.3. テストケース表示/保存

次の図 4 がテストケース表示画面である．生成されたテストケースはエディタの右の「TestData」に表示される．No はテストケースが生成された順番，var が入力変数名，type が入力変数の型，test data はそのテストケースである．生成された結果を保存するときは，メニューバーの「File」から「Export」を押すと生成されたテストケースが Excel ファイルとして保存される．

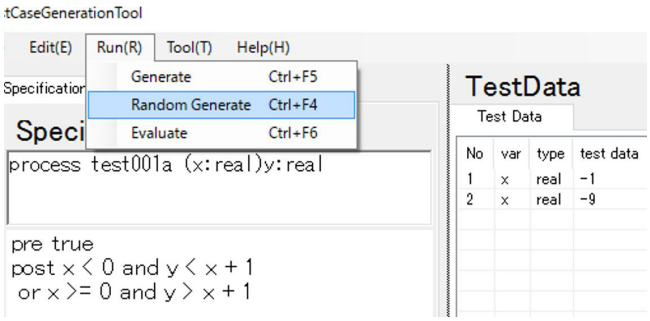

図 4: テストケース表示画面

4.4. 評価

形式仕様によって実装されたプログラムが正しいかどうかを自動で評価する．図 5 のように，2 つタブ画面があり,「Evaluate Result」と「Output Value」である.「Output Value」は Process 仕様に定義された出力変数の値を設定する画面がある．var が出力変数，type が出力変数の型，value は出力変数の値である.「Generate」をした際に，Process 仕様の出力変数を読み取り，var，type を設定する．value は手動で入力する．ここで入力された値を def-condition に代入して評価を行う.「Evaluate Result」は Def-Condition と評価結果を表示するテーブルである．No はテストケースが生成された順番であり，def-condition に入力変数が含まれる場合は，その No に対応するテストケースを代入する．Result には，入力変数や出力変数を代入して，test-condition→def-condition の結果を表示する．

Result が 1 つでも true であれば,「Bug not found」というメッセージを表示する．Result がすべて false であ

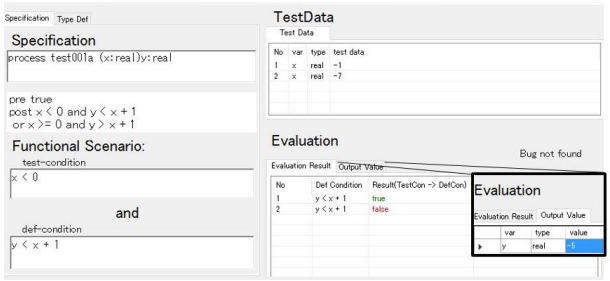

図 5: 評価結果画面

れば仕様を満たしていないとして「Bug found」というメッセージを表示する．

5. 評価実験

テストケースの生成能力を比較するために，テストケース自動生成によく利用される [4]SMT-Solver と本ツールを比較する．比較対象は，SMT-Soloverの中でも最も代表的な Z3 と比較を行う．Microsoft 社の rise4fun[5] を利用して，Z3 の制約を記述してテストケース生成を試みる．

実験方法は，様々な演算子による共通の制約を用意して，本ツールと SMT-Solver で条件を満たす値を生成出来るかどうかを試みる．今回は，5 つの演算子，合計 31 つの制約を実験対象として比較を行った．例えば，数値型の制約であれば，“$x > 5,\ x > 5\ and\ x < 7,\ x * x > 5, ...$”などを 6 つ，集合型の制約であれば，“$|x| = 5,\ x_1 \cup x_2 = result, ...$ (x, result は集合型)”などの制約を，本ツールと SMT-Solver に与えて変数の値を生成出来るかどうかを確かめる．詳しくは本論で記載する．実験を行った結果，次の表 6 のようになった．数値型以外の演算子に対しては，Z3 よりも本ツールの方が生成能力が高かった．Z3 が生成出来たのは列型の演算子 head を使った制約に対してのみであった．数値型の制約に対しては，Z3 の方が優れていた．本ツールでは，制約に同じ変数があった場合は（$x * x < 5$ 等）失敗する場合が見られた．

表 6: 制約によるテストケース生成結果

演算子の型	用意した制約の数	Z3	My tool
数値型	6	6 (6/6)	4 (4/6)
列型	7	1 (1/6)	6 (6/7)
集合型	6	0 (0/6)	6 (6/6)
複合型	6	0 (0/6)	6 (6/6)
写像型	6	0 (0/6)	6 (6/6)
合計	31	7(7/31)	28 (28/31)

6. 考察と今後の課題

ソフトウェアテストに多くのコストと時間がかかるという問題を解決するために，SOFL 形式仕様からテストケースを自動生成するツールを開発した．実験結果から，写像や集合などの数値型以外の制約に対しては，本ツールは SMT-Solover よりもテストケース生成に成功した数が多いことがわかった．よって，本ツールは様々な演算子を含む形式仕様からテストケース自動生成が可能であると考えられる．しかしながら，自動生成は仕様に少しでも誤りがある場合に，正しいテストケースを生成することができないため，自動生成は万能ではないと考えられる．また，本研究では FSF からテストケースを生成を行ったが，SOFL 形式仕様からは自動的に FSF に変換できていない．SOFL 形式仕様に対して，ストア変数，不変数定義，関数記述など，いくつかの記述に対応できなかったため，これらを考慮してする必要がある．これらを今後の課題をする．

7. むすび

形式仕様から自動的にテストケース生成する方法と評価するツールを開発した．ツールは形式仕様の入力変数，出力変数，事前条件と事後条件を解析して，条件から仕様を満たすかどうかを確かめるためのテストケース生成を自動生成することを試みた．評価実験として，テストケースの生成能力を比較するために，SMT-Solver と本ツールのテストケース生成能力を比較した．実験結果から，写像や集合などの数値型以外の制約に対しては，本ツールは SMT-Solover よりも生成に成功した数が多かった．よって，様々な演算子を含む形式仕様からテストケース自動生成が可能であることを示した．

参考文献

[1] Phill Stock and David Carrington, A Framework for Specification-Based Testing, IEEE TRANSACTION ON SOFTWARE ENGINEERING, VOL 22, NO.11, 1996.

[2] Saoying Liu, Formal Engineering for Industrial Software Development Using the SOFL Method Springer Germany. 2004.

[3] Saoying Liu, A Decompositional Approach to Automatic Test Case Generation Based on Formal Specifications, Secure Software Integration and Reliability Improvement, p147-155, 2010.

[4] Khalek, S.A. , Guowei Yang ,Lingming Zhang, Marinov, D. ,Khurshid, S. , TestEra: A Tool for Testing Java Programs using Alloy Specifications”, IEEE Automated Software Engineering, pp. 608 - 614, 2011 .

[5] Z3@rise4fun, http://rise4fun.com/Z3 (最終閲覧日：2016 年 2 月 2 日)