SOFL 形式仕様に基づくC#プログラムのテストツール

網谷　拓海
法政大学情報科学研究科
takumi.amitani.2c@stu.hosei.ac.jp

劉　少英
法政大学情報科学研究科
sliu@hosei.ac.jp

要旨

形式仕様をプログラムの実装だけでなくテストにも利用できれば，ソフトウェアの開発コストを削減でき，形式仕様の有用性も高まる．具体的には，形式仕様に記述された入力に関する条件をテストケースの生成に，出力に関する条件はテスト結果の評価に利用することができる．先行研究ではこの特性を生かし，SOFL 形式仕様を使ってテストケースの生成とテスト結果を評価するプログラムが作成された．しかし，生成したテストケースをテスト対象プログラムに入力するプログラムの作成と，テストの実行，テスト結果を評価プログラムに入力することはすべて手動で行い，別のツールを使用する必要があった．そこで本研究では，SOFL 形式仕様に基いて実装された C#プログラムのテストケース生成，テスト用プログラムの生成と実行，テスト結果の評価までの一連の作業をすべて行えるツールを考案，実装し，実際の事例に適用可能なことを確認した．

1. はじめに

ソフトウェア開発プロジェクトにおいてテストの占める割合は新規開発の中で結合テストと総合テストを合わせると全行程の約30%を占める[1]．よってテストの割合は大きく，テスト時間の削減はプロジェクト全体のコスト削減につながる．

プログラムのテストを行う場合，通常はテストケースの生成と，テストの実行，テスト結果の評価は手動で行う必要がある．しかしプログラムに対する要求を形式仕様であらかじめ記述することにより，要求が数学的，論理的に表現され，テストケースの生成とテスト結果の評価の一部を自動化することができる．先行研究[2]ではこれを生かしてSOFL 形式仕様[3]を用いてテストケースの生成とテスト結果の評価を行うプログラムが作成された．しかしこのプログラムでは生成されたテストケースをテスト対象プログラムに入力してテストを行うことと，テスト結果を評価プログラムに入力することは手動で行い，別のツールを用意する必要がある．

そこで本研究ではテストケース生成とテスト結果評価を先行研究のプログラム[2]を利用して行い，加えて，テストケースをテスト対象プログラムに入力するプログラムであるテストスクリプトを生成して実行することと，テスト結果を評価プログラムに入力して評価を行うという，テストに必要な一連の作業を一つのツールで行えるようにした．

2 章では SOFL 形式仕様の説明，3 章から 7 章では，ツールの説明，8 章では実際の事例にツールを適用できるか確認し，9 章ではまとめと今後の課題を記す．

2. SOFL 形式仕様について

SOFL 形式仕様では一つの仕様を C#や Java のクラスと対応する，モジュールの中に記述する．モジュール内には型定義，データストア変数の定義 C#や Java のメソッドと対応するプロセス定義が記述される．プログラムに対する条件式はプロセスの事前条件と事後条件として記述される．この事前条件と事後条件にはメソッドの入力と出力が満たすべき条件が数学的，論理的に記述されるため，テストケース生成とテスト結果評価に利用できる．

SOFL は先行研究[4]で編集支援ツールが開発されており，そのツールで記述されたモジュールは fModule ファイルとして保存される．本研究で実装するツールではこの fModule ファイルを読み込むことで仕様の情報を取得する．

3. ツールの目的と概要

従来 SOFL 形式仕様を用いたテストを行う場合，テストケース生成からテスト結果評価までの流れを別々の場（ソフトウェアなど）で行う必要があった．そこでそれらの工程を同一ソフトウェア上で行えるようにし，テストの効率を上げて時間を削減することがツールの目的である．

本ツールは SOFL 形式仕様から実装された C#プログラムのテストをサポートする．主な機能は，テストケース生成，テストスクリプト生成，テストスクリプト実行，テスト結果

評価の4つである．ツールはWindowsフォームアプリケーションを C#で実装した．ツールの実行イメージは以下の図の通りである．

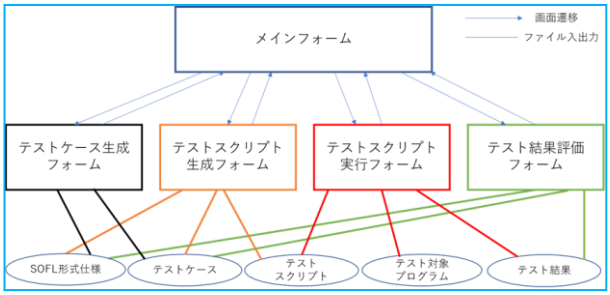

図 1：ツールの概要

各機能を持ったフォームへはメインフォームを経由して遷移することができる．ユーザーはテストケース生成フォームから順に進めていくことでテストを完了することができるようになっている．このツールの特徴として，各フォームでの実行結果を外部ファイルに保存することで，各機能ごとに作業を中断したり，再開することができる．例えばテストケース生成フォームでテストケースを生成した後にツールを閉じてしまっても，生成したテストケースはファイルに保存しているので，またツールを起動してテストスクリプト生成から始めることができる．メインフォームは以下の図のとおりである．

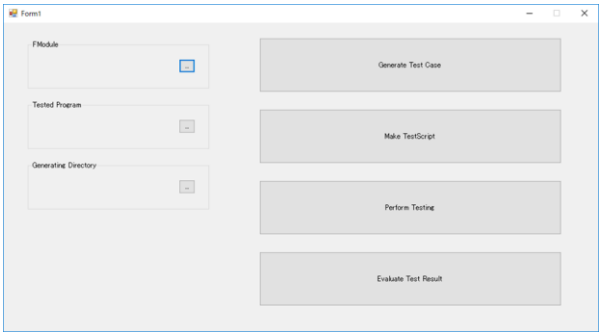

図 2：メインフォーム

まず各機能を使う前に，ファイルパスを指定する必要がある．左側の一番上の「FModule」欄のボタンを押すとフォルダブラウザが表示されるので，そこで SOFL 形式仕様のモジュールファイルである fModule ファイルを指定する．同様にして「Tested Program」欄ではテスト対象プログラムをまとめた dll ファイル（C#プログラムをライブラリ形式でまとめたファイル）を指定する．「Generating Directory」欄ではツールによって生成されるテストケース，テストスクリプト，テスト結果を保存するフォルダを指定する．

4. テストケース生成

テストケース生成フォームを使用する前に，SOFL 形式仕様のモジュールである fModule ファイルを指定する必要がある．ツールは指定された fModule ファイルを解析し，モジュールに含まれる各プロセスに対してテストケースを生成する．テストケース生成後のフォームの様子は以下の図の通りである．

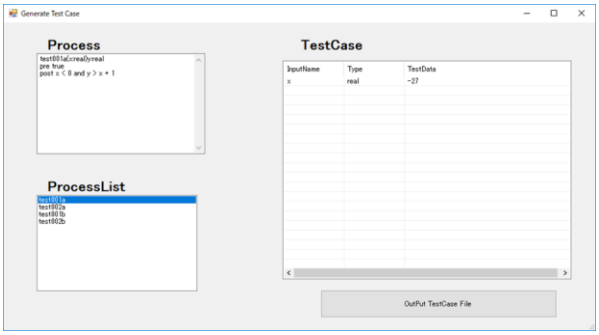

図 3：テストケース生成フォーム

テストケース生成後，まずモジュールに含まれる全プロセスが ProcessList ボックスの中にリストアップされる．ユーザーがリストからプロセス名をクリックすると Process ボックス内にプロセスの記述が，TestCase ボックスにはそのプロセスに対して生成されたテストケースが表示される．これによってユーザーは，各プロセスがどのようなプロセスであったか，さらにそのプロセスに対してどのようなテストケースが生成されたのかを確認することができる．もしテストケースを自分で指定したい場合は TestCase ボックスの TestData の項目をクリックすることで，入力することができる．そしてすべてのプロセスに対して正しいテストケースが準備できたと思ったら右下の「OutPut TestCase File」というボタンをクリックすることでテストケースをテキストファイルとして保存することができる．

テストケース生成に使用しているプログラムは先行研究の[2]のものである．このプログラムではプロセスに記述された，プロセスの入力に関する条件式をシナリオごとに分けて，さらに各シナリオの条件式を逆ポーランド記法に変換してテストケースを生成する．例えば以下のようなプロセスを考える．

```
Process Test (x : int) y : int
pre x > 0
post x > 10 and y = 5 or
x <= 10 and y = 20
end_process;
```

このプロセスの入力は int 型の x である．x に対する条件は，事前条件の x > 0，そして事後条件の x > 10 and y = 5 or x <= 10 and y = 20 である．事後条件は x の値によって二つのシナリオに分けられる．一つは x > 10 and y = 5，もう一つは x < = 10 and y = 20 である．ここから出力の条件を省き，事前条件はどちらのシナリオも満たさなければならないので加えると，x > 0 and x > 10 と x > 0 and x <= 10 という入力に関する二つの条件を得ることができ，テストデータは各条件に対して一つずつ生成されて，それらをまとめてテストケースとする．なおここで評価することは，事前条件を満たすテストケースを入力したときの出力の値であるため，事前条件を満たさない場合のテストケースは生成しない．

5. テストスクリプト生成

テストスクリプトはテスト対象プログラムをテストするためのプログラムで，テスト対象プログラムと同じプログラミング言語で記述される．本ツールは C#のプログラムをテストすることを想定しているので，テストスクリプトも C#プログラムとして生成する．

テストスクリプトはテストメソッド，クラススクリプト，テスト詳細メソッド，テスト結果書き出しメソッド， Main メソッドで構成される．各メソッドの説明を以下に記す．

5.1. テストメソッド

テストメソッドは各プロセスに対して生成され，各プロセスに対応したテスト対象メソッドをテストする．テストメソッド内には，各テストシナリオに対してテストケースを変数に格納する文，テスト詳細メソッド実行文が記述される．あるプロセスに二つのテストシナリオがあった場合，そのプロセスのテストメソッドには，一つ目のテストシナリオのテストケースを格納するための変数宣言文，その変数にテストケースを格納する文，その変数をテスト詳細メソッドに入力して実行する文があり，二つ目のテストシナリオに対しても同じように生成される．このとき変数に格納するテストケースは，テストケース生成フォームで保存したテストケースファイルから取得する．ちなみにテストメソッド内ではテストケースを格納する変数の型は，プロセスの入力の SOFL 型と一意に対応している．各 SOFL 型と C#の型の対応は以下の表のとおりである．

表 1：SOFL 型と C#型の対応

SOFL 型	C#型
nat, nat0, int	int
real	double
char	char
bool	bool
string, enum	string
Map	Dictionary
Set	HashSet
Sequence	List
Composite, Product	SOFL 型と同名のクラス

上の表のうち，Composite と Product 型は複数の要素を持つ混合型で，モジュールの型定義内で新たな型として定義される．そのため，C#には対応する型が存在しないので，テストケースを格納するために対応するクラスをテストスクリプト内に生成する．それをここではクラススクリプトと呼ぶ．

5.2. クラススクリプト

SOFL モジュール内に Composite 型か Product 型の新たな型が定義されていた場合，新たなクラスを用意する．クラス名は定義された型の名前である．クラスのフィールドには Composite もしくは Product の要素に対応したものが作られる．さらにクラススクリプトはコンストラクタを持ち，各フィールドにテストケースを代入できる．

5.3. テスト詳細メソッド

テストメソッドではテストケースを変数に格納する際に，SOFL 型と C#の型の対応を表 1 のように決めたが，実際にはテスト対象プログラムの入力が別の型で実装されている可能性がある．そのため，テスト詳細メソッドはテストケースが収められた表 1 通りの C#型の変数を入力として受け取り，それをテスト対象プログラムの入力の実際の型に変換するプログラムをユーザーに記述してもらう．さらにテスト対象プログラムの実行文とテスト結果の取得もユーザーが記述する．これによって，どのような実装方法であっても，テストを行うことができる．

よってテスト詳細メソッドの構成要素は，テストケースをテスト対象プログラムと正しく対応した型に変換する文，テスト対象プログラムを実行する文，テスト結果を回収す

る文である．なおテスト結果を回収する際には，SOFL 形式仕様と比較しやすくするために，プロセスの出力に対してテスト結果を格納する変数の型は表 1 通りにする．テスト結果を変数に格納したら，テスト詳細メソッドの最後にテスト結果書き出しメソッドを呼び出す．

5.4. テスト結果書き出しメソッド

テスト結果を入力として受け取り，それを外部ファイルに書き出すための文字列変数に追加する

5.5. Main メソッド

Main メソッドではまずテスト結果を保存するための文字列変数を作る．そして各プロセスのテストメソッドを呼び出してテストを行う．呼び出すごとにテスト結果用文字列にはテスト結果が追加されていく為，すべてのテストメソッドを呼び出した後，その文字列を外部テキストファイルに書きだす．

以上の要素で構成されたテストスクリプトの動作を以下の図にまとめた．

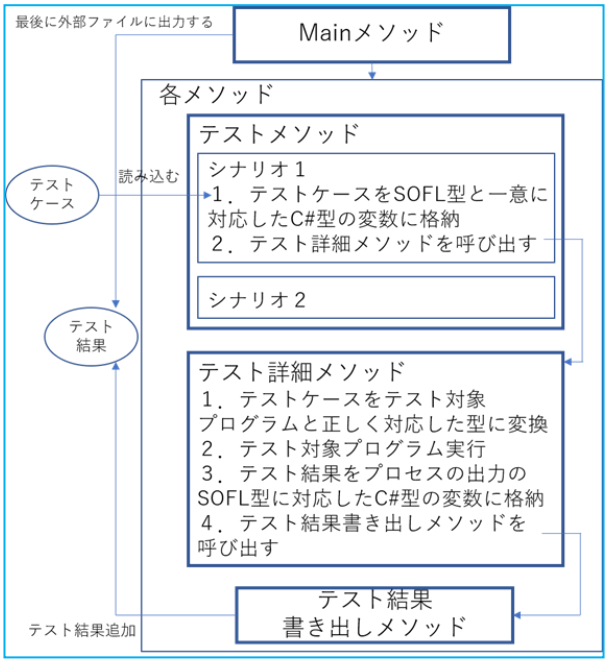

図 4：テストスクリプトの動作

テストスクリプト生成フォームでは，フォーム表示前にテストケースと SOFL 形式仕様を読み込んで，テスト詳細メソッド以外を自動生成する．フォーム出現後，フォーム内では各プロセスのテスト詳細メソッドを編集できる．その様子が以下の図である．

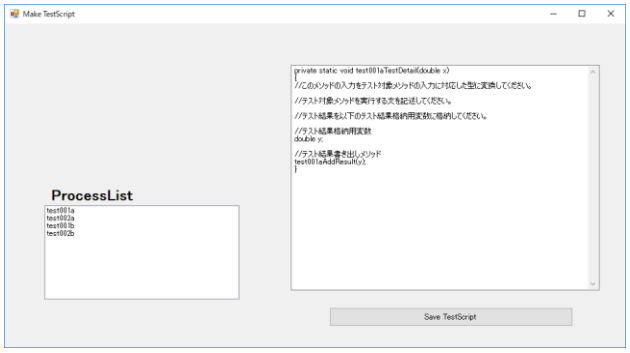

図 5：テストスクリプト生成フォーム

ProcessList ボックスにはモジュール内のプロセスがリストアップされる．そこからプロセスを選択すると，そのプロセスのテスト詳細メソッドが表示される．図のテスト詳細メソッドは自動生成直後のもので一切編集をしていない．テスト詳細メソッドにはユーザーに対する要望のコメントが記述されている．ユーザーはこれに従ってテスト詳細メソッドを完成させる．すべてのメソッドを完成させたら，右下の「Save TestScript」ボタンをクリックするとテストスクリプトを保存することができる．なおテスト詳細メソッドはテストスクリプトとは別のファイルにも保存しているので一度ツールを閉じてしまっても再度編集することができる．

6. テストスクリプト実行

このフォームを起動すると，テストスクリプト生成フォームで保存したテストスクリプトが読み込まれる．このとき左のテキストボックスにテストスクリプトの内容が表示されるが，実行前に編集可能である．以下の図がテストスクリプトを読み込んだ直後のフォームである．

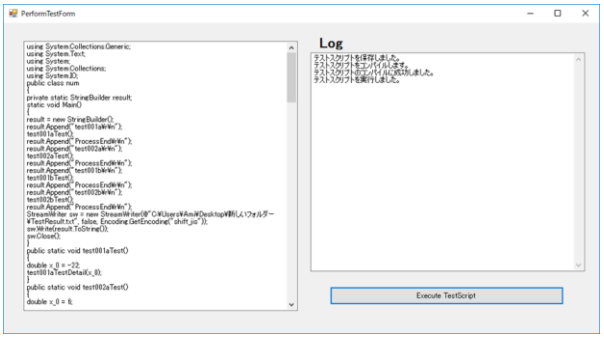

図 6：テストスクリプト実行フォーム

右下の「Execute TestScript」ボタンを押すとテストスクリプトが実行される．まずテストスクリプトが保存され，コンパ

イルが始まる．文法が正しければ「テストスクリプトのコンパイルに成功しました．」というメッセージが出るが，誤りがあれば，エラーメッセージと失敗メッセージが表示されて実行は中断される．この場合，エラーメッセージを参考にしてテストスクリプトを修正し，再度実行するとよい．コンパイルが成功すると実行可能ファイルが実行される．それが成功すると「テストスクリプトを実行しました．」という，メッセージとともにテストの実行が終了する．

テストスクリプト内にはテスト結果を外部ファイルに書き出す記述が含まれている．そのためこのフォームでテストスクリプトの実行が終了した際にテスト結果が記述されたテキストファイルが生成される．このファイルは次の章でSOFL 形式仕様の出力に関する条件と比較して評価する．

7. テスト結果評価

このフォームではテスト結果を形式仕様と比較して評価する．フォームは以下の図のとおりである．

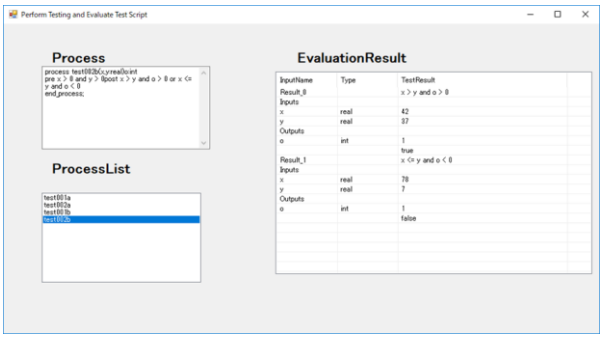

図 7：テスト結果評価フォーム

ProcessList にはモジュール内のプロセスがリストアップされ，プロセスを選択することで，評価結果を確認することができる．図のプロセスは二つのシナリオがあるので評価結果が二つある．一つ目のシナリオ Result_0 の条件は x > y and o > 0 だが，これを評価するためにまずはテストケースを条件に代入する．このテストケースはテストケース生成フォームで保存したファイルから取得する．この時条件は 42 > 37 and o > 0 となり，さらに出力 o が 1 なのでテスト結果は true となる．同様に Result_1 を確認すると 78 <= 7 and 1 < 0 となるので結果は false となる．

テスト結果の評価には[2]の先行研究のプログラムを利用している．このプログラムはまず SOFL 形式仕様を解析して，各プロセスの条件をテストシナリオごとに分ける．そして，条件とプロセスの入力（テストケース）と出力（テスト結果）を評価用メソッドに入力することでその条件の真偽値を得ることができる．よってこのフォームを使用する際は，各プロセスの条件を SOFL 形式仕様から，プロセスの入力をテストケースファイルから，出力をテスト結果ファイルから読み込む．

8. 実際の事例

今回は SOFL 形式仕様のモジュールを用意し，それを基に実装した C#プログラムをツールによってテストした．仕様とプログラムは以下の通りである．

```
module TestModule;
type
PayingCard = Composed of
name : string
buffer : nat0
end;
process Pay(c: PayingCard, price: int) payout: int
pre c.buffer > 100000 and price > 0
post payout = c.buffer - price
end_process;
end_module
```

```
namespace TestLibrary{
    public class TestModule{
        public static int Pay(Card card, int price){
            return card.GetBuffer() - price + 1;
        }
    }
    public class Card{
        private string name;
        private int buffer;
        public Card(string name, int buffer){
            this.name = name;
            this.buffer = buffer;
        }
        public string GetName(){
            return name;
        }
        public int GetBuffer(){
            return buffer;
        }
    }
}
```

モジュールには複数の要素を持つ複合型である PayingCard 型と，PayingCard 型の c と int 型の price を入力として受け取り，int 型の payout に c の buffer から price を引いた値を出力するという Pay プロセスが定義されている．

プログラムにはTestModuleクラスとCardクラスが実装されている．Cardクラスは PayingCard 型を実装したもので，string 型の name と int 型の buffer を持つ．また TestModule クラスは仕様通りに Pay メソッドを持つが，このメソッドが出力する値は意図的に入力 card の buffer から price を引いた値に 1 を足したものにした．つまりこのプログラムは誤りが含まれる．

次にツールに仕様とテスト対象プログラムを指定してテストを行った．まずは生成されたテストケースは以下の図の通りである．

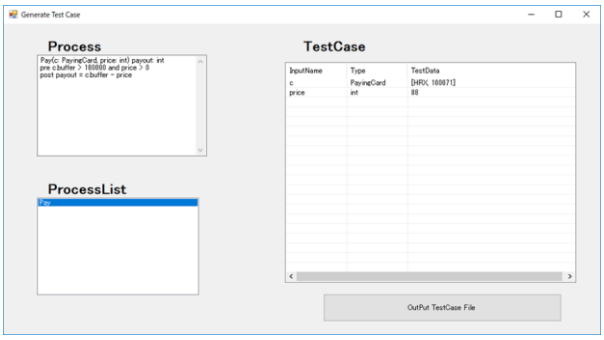

図 8：テストケース生成後

Pay プロセスの入力 c には[HRX, 100071]が，price には 88 がテストデータとして生成された．Pay プロセス内の条件で入力に関する条件は

c.buffer > 100000 and price > 0 となっているが，テストケースはこれを満たしている．

次に生成したテストケースと仕様を用いてテストスクリプトを生成した．テスト詳細メソッドを編集した後のテストスクリプトは以下の通りである．

```
using System.Collections.Generic;  using System.Text;
using System;  using System.Collections;  using System.IO;
public class TestScript{
    private static StringBuilder result;
    static void Main(){
        result = new StringBuilder();
        result.Append("Pay¥r¥n");
        PayTest();
        result.Append("ProcessEnd¥r¥n");
        StreamWriter sw =
 new StreamWriter(@"C:¥Users¥Desktop¥TestResult.txt",
            false, Encoding.GetEncoding("shift_jis"));
        sw.Write(result.ToString());
        sw.Close();
    }
    public class PayingCard
    {
        public string name;
        public int buffer;
        public PayingCard(string name, int buffer)
        {
            this.name = name;
            this.buffer = buffer;
        }
        public override string ToString()
        {
            StringBuilder sb = new StringBuilder();
            sb.Append("[");
            sb.Append(ConvertResult(name));
            sb.Append("," + ConvertResult(buffer));
            sb.Append("]");
            return sb.ToString();
        }
    }
    public static void PayTest()
    {
        string c_0_name = "HRX";
        int c_0_buffer = 100071;
        PayingCard c_0 =
            new PayingCard(c_0_name, c_0_buffer);
        int price_0 = 88;
        PayTestDetail(c_0, price_0);
    }
    private static void PayTestDetail(PayingCard c, int price)
    {
        TestLibrary.Card card =
            new TestLibrary.Card(c.name, c.buffer);
        int payout = TestLibrary.TestModule.Pay(card, price);
        PayAddResult(payout);
    }
    private static void PayAddResult(int payout)
    {
        result.Append("payout¥r¥n");
        result.Append(ConvertResult(payout) + "¥r¥n");
        result.Append("TestResultEnd¥r¥n");
    }
}
```

まず Main メソッドではテスト結果を格納する変数 result を生成している．result.Append はテスト結果についての情報を result に追加する文である．

次に Main メソッドではテストメソッドの PayTest メソッドが呼ばれる．このメソッドは生成されたテストケースを表1の対応に従った型の変数に格納し，それをテスト詳細メソッドである PayTestDetail メソッドに渡す．PayTestDetail メソッドは本ツール上で筆者が編集した．このメソッドでは，受け取った PayingCard 型の入力 c が，テスト対象メソッドである Pay メソッドの Card 型の入力 card に直接入力できないために，新たに Card 型の変数を用意して Pay メソッド

に渡すという処理と，Pay メソッドの出力を受け取りテスト結果書き出しメソッドである PayAddResult メソッドに入力するという処理が記述されている．

最後に Main メソッドでは result に貯まったテスト結果を外部ファイルに書き出してプログラムの処理は終了する．

テストスクリプトの実行は成功し，その後にテスト結果を評価した．その様子が以下の図である．

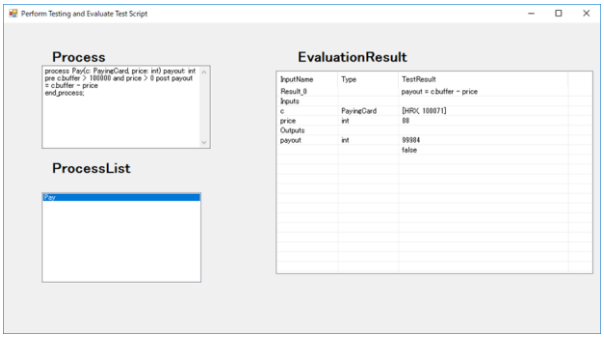

図 9：テスト結果評価後

入力 c の要素 buffer のテストデータは 100071 で入力 price のテストデータは 88 なので，出力 payout が満たすべき条件は payout = 99983 である．しかし，テストした Pay メソッドが出力した値は 99984 なので評価結果は false と出ている．これは期待していた通りの結果である．

以上のように SOFL 形式仕様に基づくプログラムに対して，テストケース生成，テストスクリプト生成と実行，テスト結果評価というテストの一連の流れをすべて本ツール上で行うことができた．

9. まとめと今後の課題

本研究では先行研究[2]のテストケース生成とテスト結果評価のプログラムに加えて，テストスクリプト生成と実行，テスト結果の入力を同じツール上で行えるようにした．これによって SOFL 形式仕様に基づくプログラムのテストをスムーズに行うことができ，テスト時間を削減することが期待できる．

今後の課題の一つ目は対応できるテスト対象プログラムの言語を増やすことである．現在は C#のプログラムのテストのみ行うことができるが，SOFL 形式仕様からは Java や C++言語のようなほかのプログラミング言語で実装が行われる可能性があるため，対応言語を増やす必要がある．この場合言語によってツールの変更すべき点はテストスクリプトの生成方法である．各言語ごとにテストスクリプトを生成するプログラムをパッケージ化し，それをスクリプト生成時に選択するような形を取れば拡張性が高まる．

二つ目はツールの自動化範囲を広げることである．現在はテストスクリプトの一部をユーザーが手動で記述する必要がある．それは主に，テストケースをテスト対象プログラムに変換する部分と，テスト結果を SOFL 型と一意に対応した型に変換させる部分である．これらを自動化するためにはテスト対象プログラムの入力と出力の型を調べて柔軟に対応できるようにしなければならない．

10. 謝辞

本研究は JSPS 科研費 26240008 の助成を受けたものです．

参考文献

[1] 独立行政法人情報処理促進機構 技術本部 ソフトウェア高信頼化センター,ソフトウェア開発データ白書 2016-2017, 2016

[2] 池田逸人，劉少英，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，第 79 回全国大会講演論文集，情報処理学会，2017

[3] Shaoying Liu , Formal Engineering for Industrial Software Development Using the SOFL Method, Springer-Verlag, 2004

[4] 劉少英，実用性が高い形式工学手法と支援ツールの研究開発，https://www.ipa.go.jp/files/000027372.pdf，2012