Testing-Based Formal Verification のための SMT ソルバを用いた定理検証
SMT-Based Theorem Verification for Testing-Based Formal Verification

須貝 健太 *
Kenta Sugai
法政大学大学院 情報科学研究科 情報科学専攻
Email: kenta.sugai.8k@stu.hosei.ac.jp

***Abstract*—Testing-based formal verification with symbolic execution (TBFV-SE) checks whether programs correctly implement their formal specifications. Given a formal specification and a program, it first derives a theorem expressing the correctness property of the executed program paths and then formally verifies the validity of the theorem. If programs have bugs, it derives invalid theorems because the programs do not follow their specifications. Such incorrect programs output variable values that do not satisfy conditions in their specifications. By contrast, valid theorems guarantee the correctness of target programs based on formal specifications. However, such theorems still need to be verified manually due to the lack of a tool support for dealing with expressions involved. In this paper, we propose a method for automatically verifying theorems for TBFV-SE. This method uses an SMT solver to check whether a theorem is valid. For this purpose, the method converts the conditions in the theorems written in the SOFL specification language into appropriate constraints that are supported by the SMT solver. This method derives test programs from theorems and formal specifications and verifies the theorems by automatically executing the programs. We present a tool to support the method, and present some case studies to demonstrate how the tool works in the context of Java programs and SOFL specifications.**

1. 導入

形式仕様はプログラムを正確に記述するために用いられるが，形式仕様を用いたプログラムが必ずバグを含まないとは限らない．Testing-Based Formal Verification with Symbolic Execution (TBFV-SE) [1] はプログラムが形式仕様を満たしているか調べる手法である．この手法は与えられた形式仕様とプログラムから，記号実行 [2] されたプログラムパスの正しさを表す定理を導出し，その定理の有効性を検証する．導出された定理は形式仕様と記号実行によって得られたプログラムの変数と述語論理式からなる．この定理に含まれる式の表現は自動的に処理されないため，手動で定理の検証をしなければならない．

本論文では，TBFV-SE から導出された定理を自動的に検証する手法を提案する．この手法は Satisfiability Modulo Theories (SMT) ソルバによって定理が妥当であるか検証する．そのために定理に含まれている Structured Object-Oriented Formal Language (SOFL) [3] の形式仕様で記述された条件式を SMT ソルバで扱える制約へ変換する必要がある．与えられたプログラムの入力変数と出力変数はそれぞれ定理の前提と結論に含まれている．SMT ソルバによって前提と結論がどちらも True になることを確認する．TBFV-SE によって導出された定理は基本的に SOFL 形式仕様特有の表現を含んでいるため，SMT ソルバで扱うことのできる形に変換しなければならない．そこで，TBFV-SE を支援するツールの実装を示し，Java プログラムと SOFL 形式仕様に対してどのようにツールが動作するかを説明する．

2. 関連研究

定理証明の手法として，有名なものに Coq [4] がある．この手法では帰納法などに対応しており，正しさが証明された関数をソースコードに変換することができるが，ある程度手動での証明手順が必要になる．Testing-Based Formal Verification (TBFV) [5], [6] では検証方法としてテストが考えられている．形式仕様を用いた自動テストデータ生成手法 [7] は存在するが，部分的にしか正しさを示せないため，定理を検証することは難しい．Spec#では，拡張した C#プログラムのメソッドに事前条件と事後条件を与え，条件を満たさない反例を探すことでエラーを出力できる．反例を探す際に SMT ソルバを利用している．SMT ソルバは制約条件として表現された述語論理式の充足可能性を調べることができる．

3. 準備

3.1. SOFL

SOFL [3] は仕様記述言語の 1 つである．形式仕様にはプログラムが満たすべき条件を記述できるため，仕様に対して正しいプログラムを記述しやすいという利点がある．SOFL には様々な変数の型があるが，set (集合) 型と sequence (列) 型が多く用いられる．また，SOFL では process という部分にプログラムの関数の仕様を記述する．process は宣言部分，事前条件，事後条件からなる．宣言部分は process の名前，入力変数，出力変数が含まれている．事前条件は process に対応したプログラムの関数が実行する前に満たすべき制約である．事後条件は対応する関数が実行した後に満たすべき制約である．提案手法では，形式仕様に定義された 1 つの関数についてその動作を保証するために定理検証をする．

3.2. TBFV-SE

TBFV-SE [1] は形式仕様とプログラムから定理を導出する．i を対象のプログラムのパスの番号とすると，TBFV-SE は i 番目のパスに対して以下のような定理を導出する．

$$T_i \wedge C_i \wedge \mathrm{states}_i \Rightarrow D_i$$

ここで，T_i は test condition と呼ばれるもので，事前条件と事後条件に含まれるガード条件の論理積である．test condition には入力変数のみが含まれる．C_i は path condition といい，記号実行によって得られたプログラ

* Supervisor: Prof. Hiroshi Hosobe

ム内の条件である．states_i は記号実行で得られた変数の状態を表している．D_i は defining condition といい，事後条件に含まれる出力変数に関する条件である．この定理は，T_i，C_i，states_i を満たす変数がある場合，出力変数は必ず D_i を満たすということを意味している．対象のプログラムパスが正しいか調べるために，この定理の有効性を検証する．しかし，TBFV-SE には定理の正しさを検証する方法やツールが提供されていない．$T_i \wedge C_i \wedge \text{states}_i \wedge \neg D_i$ が満たされる場合，プログラムがバグを含んでいるといえる．

3.3. Z3

Z3 [8] はプログラム検証などに用いられている SMT ソルバである．Z3 は制約を満たす変数の値が存在するかどうかを判定することができる．すべての制約を満たす値の組み合わせが見つかった場合，Z3 は sat を返し，モデルを 1 つ出力する．モデルは制約を満たすことのできる値の組み合わせである．制約を満たす値の組み合わせが見つからなかった場合，Z3 は unsat を返す．例えば，$a > 0$ が制約として与えられた場合，Z3 は sat を返し，モデルとして $a = 1$ を出力する．$a > 0 \wedge a < -1$ が制約として与えられた場合，Z3 は unsat を返す．このように SMT ソルバを用いることで変数に対する条件式の矛盾を示すことができる．

4. 提案手法

本論文では，TBFV-SE によって導出された定理の自動検証手法を提案する．この手法では，SMT ソルバを用いて定理が有効であるかを調べる．より具体的には，SMT ソルバを用いて形式仕様と対象のプログラムから TBFV-SE によって導出された i 番目の定理について $T_i \wedge C_i \wedge \text{states}_i \wedge \neg D_i$ が充足可能であるか調べる．3.2 節で述べたように，条件の充足可能性は対象のプログラムが形式仕様を満たしていないことを示す．

この手法では，SMT ソルバとして Z3 を用いる．以下の理由から，SOFL の変数の型と演算子への対応が重要である．

1) Z3 は SOFL の set 型と sequence 型を扱えないが，vector は扱うことができる．
2) Z3 は SOFL 演算子を直接扱うことができない．

例えば，a, b, c, z を set 型変数とすると $z = \text{union}(c, \text{union}(a, b))$ という SOFL の条件式は Z3 の制約として扱うことができない．Z3 は SOFL の set 型を扱うことができないため，z と $\text{union}(c, \text{union}(a, b))$ の等価性を評価できない．また，SOFL 演算子の union は Z3 に定義されていない．したがって，この手法では SOFL の式を Z3 の vector を用いた制約へ正確に変換する必要がある．

例として，$a = \{v_0, v_1\}, b = \{v_0, v_2\}, c = \{v_1\}, z = \{v_0, v_1, v_2\}$ に対する $z = \text{union}(c, \text{union}(a, b))$ の検証方法を示す．まず，set 型の変数を全て Z3 の vector として再定義する．これらの変数に対し，$v_0, v_1 \in a,\ v_0, v_2 \in b,\ v_1 \in c,\ v_0, v_1, v_2 \in z$ という制約を与える．変数は vector になっているため，Z3 の Distinct 関数によって要素の重複を許さないようにする．次に，a, b を探索して共通の要素の数を計算する．op_l_0 を新しい vector の変数として定義し，それが a と b の和集合と等しいという制約を与える．このとき，$\text{op\_l}_0 = \{v_0, v_1, v_2\}$ となる．また，op_l_1 を定義し，c と op_l_0 の和集合と等しいという制約を与える．この結果から $\text{op\_l}_1 = \{v_0, v_1, v_2\}$ が得られる．最後に，z と op_l_1 の要素を比較し，それぞれに含まれる要素がすべて等しいことを示す．

4.1. 型の変換

SOFL 形式仕様の process 部分を読み込み，SOFL の変数の型を Z3 の変数の型に変換する．process 部分の 1 行目は入力変数と出力変数の情報が含まれている．表 1 に SOFL の型から Python 版 Z3 (Z3Py) の型への変換を示す．SOFL の set 型を Z3 では vector として扱う．

表 1. SOFL から Z3Py への型変換

SOFL 型名	z3Py 型名
nat0, nat, int	Int
real	Real
seq of nat0, seq of nat, seq of int	IntVector
set of nat0, set of nat, set of int	IntVector
seq of real	RealVector
set of real	RealVector

4.2. 演算子の変換

この手法では，TBFV-SE から導出された定理を Z3 の制約に変換する．導出された定理には SOFL 特有の演算子が含まれていて，それらを Z3 のプログラムで扱える形にする必要がある．基本的に SOFL 演算子には bool 値を返すものと変数を返すものがある．bool 値を返す SOFL 演算子は引数に対する制約に変換し，引数に対して演算子の処理を満たすように制約を生成する．変数を返す SOFL 演算子は，引数を用いて返値となる変数を新しい変数として定義し，それが演算子の処理を満たすように制約を生成する．SOFL 演算子が入れ子になった場合，生成される制約や変数を引数として扱うために，スタックで管理する．スタック S には変数，スタック operates と predicates には制約が格納される．変数を返す SOFL 演算子を処理する場合，S から引数を取り出して返値の変数を S に積み，返値の変数に関する制約を operates に積む．bool 値を返す SOFL 演算子を処理する場合，S から引数を取り出して生成した制約を predicates に積む．

例えば，SOFL 演算子の tl は sequence 型の引数をとり，引数の先頭以外の要素からなる sequence を返す．この演算子を処理する場合，S から引数を 1 つ取り出して新しい変数 op_l_0 を生成し，S に追加する．op_l_0 が引数の sequence の先頭以外の要素と等しいという制約を生成し，operates に追加する．また，SOFL 演算子の inset は引数から bool 値を返す．この演算子を処理する場合，2 つの引数が S から取り出される．v が数値，a が set または sequence として，v inset a という形で用いられる．$v \in a$ という制約を生成し，predicates に追加する．このように，生成された変数は他の演算子の引数として，制約は 1 つの論理式として扱われる．

4.3. bool 値を返す演算子

bool 値を返す演算子を取り扱う場合，S から引数を取り出し，制約を predicates に追加する．条件式の否定には Z3 の Not 演算子を用いる．この節では，演算子の変換手法とそれぞれの実装アルゴリズムについて記述する．

Algorithm 1 equal_apply

Require: S, predicates, negation
1: i is the length of a
2: $b \leftarrow S$.pop()
3: $a \leftarrow S$.pop()
4: Declare p as a constraint
5: **if** (a and b are numerical) **then**
6: $p \leftarrow (a = b)$
7: **end if**
8: **if** (length of $a \neq$ length of b) **then**
9: $p \leftarrow$ False
10: **else**
11: **if** (a and b are Set) **then**
12: $p \leftarrow (a[1] \in b \wedge a[2] \in b \wedge \cdots \wedge a[i] \in b)$
13: **else if** (a and b are Sequence) **then**
14: $p \leftarrow (a[1] = b[1] \wedge a[2] = b[2] \wedge \cdots \wedge a[i] = b[i])$
15: **end if**
16: **end if**
17: **if** (negation = True) **then**
18: $p \leftarrow \neg p$
19: **end if**
20: predicates.push(p)

4.3.1. 等号 ($a = b$). 等号では，引数が数値，sequence 型変数，set 型変数のそれぞれについて別の処理をする．数値であれば，制約 "$a == b$" を生成して predicates に保存する．sequence 型であれば，すべての i について a の i 番目の要素が b の i 番目の要素と等しいという制約を predicates に保存する．set 型であれば，a のすべての要素が b に含まれるという制約を predicates に保存する．set は vector として実装しているため，vector 同士を要素ごとに評価することで含まれるという制約を記述する．a が b に含まれるという制約は，すべての i について $a[i] = b[1] \vee a[i] = b[2] \vee \cdots \vee a[i] = b[i]$ であるという制約になる．また，sequence 型や set 型の引数の a と b の要素数が等しくない場合，predicates に False を保存する．Algorithm 1 に等号の処理を示す．

4.3.2. 等号否定 ($a \neq b$). 等号否定では，等号と逆の処理をする．数値であれば，制約 "$a \neq b$" を生成して predicates に保存する．sequence 型であれば，a の i 番目の要素が b の i 番目の要素と異なるという制約を predicates に保存する．set 型であれば，a のいずれかの要素が b に含まれないという制約を predicates に保存する．また，sequence 型や set 型の引数の a と b の要素数が等しくない場合，predicates に True を保存する．

4.3.3. set 型，sequence 型の引数をとる演算子. inset，notin，subset，psubset は bool 値を返す SOFL 演算子である．inset と notin は sequence または set 型の変数を第 2 引数に取る．v inset a は，v が a に含まれることを表す式である．v notin a は，v が a に含まれないことを表す式である．subset と psubset は set 型の変数を引数に取る演算子である．subset(a, b) は a が b の部分集合であることを表す演算子である．これを制約とした場合，a のすべての要素は b に含まれている必要がある．psubset(a, b) は a が b の真部分集合であることを表す演算子である．これを制約とした場合，a の要素数は b の要素数より少なく，a のすべての要素は b に含まれている必要がある．このようにそれぞれの演算子の処理を満たすように引数に制約を与える．Algorithm 2 に subset の処理を示す．

Algorithm 2 subset_apply

Require: S, predicates, negation
1: i is the length of a
2: $b \leftarrow S$.pop()
3: $a \leftarrow S$.pop()
4: Declare p as a constraint
5: **if** (a and b are Set) **then**
6: $p \leftarrow (a[1] \in b \wedge a[2] \in b \wedge \cdots \wedge a[i] \in b)$
7: **else**
8: $p \leftarrow$ False
9: **end if**
10: **if** (negation = True) **then**
11: $p \leftarrow \neg p$
12: **end if**
13: predicates.push(p)

4.4. 変数を返す演算子

引数を使って演算結果を返す演算子では，演算結果を新しい変数として S に保存し，制約を operates に保存する．そこで，定理の中の SOFL 演算子よる i 番目の新しい変数を op_v_i または op_l_i とする．op_v_i は数値型の変数で，op_l_i は sequence または set 型の変数とする．

4.4.1. sequence 型の演算子. conc と tl は sequence 型の変数を返す SOFL 演算子である．conc(a, b) は sequence a の末尾に sequence b の要素が並んだ sequence を返す．tl(a) は sequence a の先頭以外の sequence を返す．a の要素数が 1 以下の場合，新しく生成される変数の要素数が 0 になり，制約が記述できない．これは，Z3 で sequence を比較する際に各要素の比較をするためである．

4.4.2. set 型の演算子. set 型の演算子では，引数の a と b の要素数をそれぞれ s，t とし，n を引数同士が共有する要素の数とする．関数 common_check を実装し，実行時の引数同士が持つ共通の要素の数を計算する．union，diff，inter は set 型変数を返す SOFL 演算子である．union(a, b) は set 型の演算子で，集合 a と b の和集合を返す．この演算子によって新しく生成される集合 op_l_i の要素数は $s + t - n$ である．よって，op_l_i に a と b の要素がすべて含まれるように制約を記述する．diff(a, b) は set 型の演算子で，集合 a と b の差集合を返す．この演算子によって新しく生成される集合 op_l_i の要素数は $s - n$ である．op_l_i の要素が a に全て含まれて，b には 1 つも含まれないように制約を記述する．inter(a, b) は set 型の演算子で，集合 a と b の共通部分を返す．op_l_i の要素は n となり，その全ての要素が a と b のどちらにも含まれるように制約を記述する．Algorithm 3∼Algorithm 5 にそれぞれの処理を示す．

5. ツール

提案手法にしたがって，形式仕様とプログラムから TBFV-SE によって導出された定理を検証するツールを開発した．条件式を検証プログラムに変換し，SMT ソルバが unsat を返すことを確認する．このツールは定理と形式仕様の構文解析をすることで，検証プログラムを生成する．検証プログラムは Python で記述され，Z3Py を用いる．検証プログラムはこのツールによって自動的に生成され，実行される．Z3 が検証プログラムに対して unsat を出力した場合，その定理は正しいといえる．このツールは形式仕様の process 部分と定理の条件式を

Algorithm 3 union_apply

Require: S, operates
1: $b \leftarrow S$.pop()
2: $a \leftarrow S$.pop()
3: Declare op_l$_i$as a new variable
4: Declare p as a constraint of SMT
5: s is the length of a
6: t is the length of b
7: $n \leftarrow \mathrm{common\_check}(a, b)$
8: length of op_l$_i \leftarrow s + t - n$
9: $p \leftarrow (a[1] \in \text{op\_l}_i \wedge a[2] \in \text{op\_l}_i \wedge \cdots \wedge a[s] \in \text{op\_l}_i) \wedge (b[1] \in \text{op\_l}_i \wedge b[2] \in \text{op\_l}_i \wedge \cdots \wedge b[t] \in \text{op\_l}_i)$
10: S.push(op_l$_i$)
11: operates.push(p)

Algorithm 4 diff_apply

Require: S, operates
1: $b \leftarrow S$.pop()
2: $a \leftarrow S$.pop()
3: Declare op_l$_i$ as a new variable
4: Declare p as a constraint of SMT
5: s is the length of a
6: t is the length of b
7: $n \leftarrow \mathrm{common\_check}(a, b)$
8: **if** $(0 < s \wedge 0 < t)$ **then**
9: $p \leftarrow (\text{op\_l}_i[1] \in a \wedge \text{op\_l}_i[2] \in a \wedge \cdots \wedge \text{op\_l}_i[s-n] \in a) \wedge (\text{op\_l}_i[1] \notin b \wedge \text{op\_l}_i[2] \notin b \wedge \cdots \wedge \text{op\_l}_i[s-n] \notin b)$
10: **end if**
11: S.push(op_l$_i$)
12: operates.push(p)

構文解析することで，Z3 に変換するための制約表現を取得する．形式仕様からは変数とその型の情報が得られる．入出力変数に関する条件式は定理の T_i, C_i, D_i から求められる．states$_i$ は変数の情報を含んでいる．このツールでは，SOFL の構文解析をするために Lark という Python のパッケージを用いた．Lark は文法を定義することによって，それに従って自動的に構文を解析する．この機能を利用するために SOFL の文法を Lark で扱えるように書き直した．

ここで，$a = \{v_0, v_1\}, b = \{v_0, v_2\}$ として $z = \mathrm{union}(a, b)$ をツールで処理する場合の例を説明する．まず，文法にしたがって式を解析し，構文木を取得する．構文木を探索し，Z3 で扱う制約を生成する．これらの制約は Z3Py で使うことのできるコードとして表される．ツールのパーサは文字列の a と b を発見し，それらを set 型変数と解釈する．これは，union が SOFL の文法上で set 型変数を引数とする演算子として定義されているからである．a と b を union の引数として使うためにスタックに積む．次に，union を処理する．union_apply を呼び出し，変数 op_l0 を定義してそれに関する制約を生成する．op_l0 が a と b の和集合であるという制約は以下のようになる．

$$((a[0] = \text{op\_l0}[0] \vee a[0] = \text{op\_l0}[1] \vee a[0] = \text{op\_l0}[2]) \wedge$$
$$(a[1] = \text{op\_l0}[0] \vee a[1] = \text{op\_l0}[1] \vee a[1] = \text{op\_l0}[2])) \wedge$$
$$((b[0] = \text{op\_l0}[0] \vee b[0] = \text{op\_l0}[1] \vee b[0] = \text{op\_l0}[2]) \wedge$$
$$(b[1] = \text{op\_l0}[0] \vee b[1] = \text{op\_l0}[1] \vee b[1] = \text{op\_l0}[2]))$$

SOFL の set を Z3 で vector として扱うために，$a \in$ op_l0 のような式は論理和を用いた形で要素ごとの制約を生成する．op_l0 についての制約が生成されると，op_l0 はスタックに積まれて equal_apply の引数として使われる．最後にパーサは z を発見し，equal_apply を処理する．この関数によって z と op_l0 が等しいという制約が生成される．

Algorithm 5 inter_apply

Require: S, operates
1: $b \leftarrow S$.pop()
2: $a \leftarrow S$.pop()
3: Declare op_l$_i$ as a new variable
4: Declare p as a constraint of SMT
5: s is the length of a
6: t is the length of b
7: $n \leftarrow \mathrm{common\_check}(a, b)$
8: **if** $(0 < n)$ **then**
9: $p \leftarrow (\text{op\_l}_i[1] \in a \wedge \text{op\_l}_i[2] \in a \wedge \cdots \wedge \text{op\_l}_i[n] \in a) \wedge (\text{op\_l}_i[1] \in b \wedge \text{op\_l}_i[2] \in b \wedge \cdots \wedge \text{op\_l}_i[n] \in b)$
10: **end if**
11: S.push(op_l$_i$)
12: operates.push(p)

6. 事例

ここでは，想定するツールの利用方法について 2 つの事例を用いて説明する．ツールでは形式仕様と定理をそれぞれ読み込み，検証のためのコードを出力してそれを実行する．

6.1. Sequence

SOFL の sequence 型を用いた形式仕様の事例を示す．sequence は順番の決まったデータの集合である．形式仕様に対してプログラムが正しく実装されている場合の検証を示す．図 1(a) の形式仕様は 0 より大きい整数のみを sequence に追加する関数を表している．図 1(b) はこれを反映した Java プログラムであり，入力が 0 より大きいか判定して配列に追加している．この形式仕様とプログラムから TBFV-SE は以下の定理を出力する．$\sim$ g_list は関数の処理をする前の g_list である．

$$g > 0 \wedge g > 0 \wedge \text{g\_list} = [g0, g], \sim \text{g\_list} = [g0]$$
$$\Rightarrow \text{g\_list} = \mathrm{conc}(\sim \text{g\_list}, [g])$$

これの正しさを検証するために，本論文のツールは図 1(c) のような検証プログラムを出力する．ここでは，初めに定理内の変数を定義している．g_list は g0 と g を要素に持つ sequence であるため，要素数 2 の IntVector という vector として定義する．g と g0 は整数として，~g_list は g_list_pre としてそれぞれ定義される．演算子 conc の第 2 引数 [g] は要素が g のみの sequence であるため，検証プログラムでは var0 として宣言する．

op_l0 は conc(~g_list, [g]) の演算結果からなる変数であり，g_list_pre に var0 を追加した sequence になる．Z3 では制約に現れる変数はすべて静的に宣言される必要がある．そのため，変数が返される演算子を扱う場合にはその返値を新しく変数として用意し，それが演算子の意味に従った制約を持つようにする．パーサにより，conc(~g_list, [g]) が構文解析され，関数 conc_apply が処理される．conc_apply によって op_l0 に g_list_pre と var0 の要素をそれぞれ順番に格納される．また，equal_apply によって op_l0 の i 番目の要素が g_list の i 番目の要素と等しいという制約を記述している．op_l0 と g_list が sequence であるため，要素ごとの比較によって制約を

```
process func(g: nat0) g_list: seq of nat0
ext wr g_list
pre g > 0
post g_list = conc(~g_list,[g])
end_process;
```

(a)

```
void func(int g){
    if (g > 0){
    g_list.add(g);
    }
}
```

(b)

```
s = Solver()

g_list = IntVector("g_list",2)
g = Int("g")
g0 = Int("g0")
g_list_pre = IntVector("g_list_pre",1)
var0 = IntVector("var0",1)

op_l0 = []
for i in range(1):
    op_l0.append(g_list_pre[i])
for i in range(1):
    op_l0.append(var0[i])

pred0 = []
for i in range(len(g_list)):
    p = g_list[i] == op_l0[i]
    pred0.append(p)

s.add(g_list[0]==g0)
s.add(g_list[1]==g)
s.add(g_list_pre[0]==g0)
s.add(g>0)
s.add(g>0)
s.add(Not(And(pred0[0],pred0[1])))
s.add(var0[0]==g)
r = s.check()
print(r)
```

(c)

図 1. sequcence を用いた事例: (a) 形式仕様, (b) 正しく実装された Java プログラム, (c) ツールによって生成された検証プログラム

記述している．この制約を配列にまとめて取り扱うために pred0 に式を格納している．

最後にそれぞれの変数に対する制約を記述する．g_list の要素は g0 と g であり，g_list_pre の要素は g0，var0 の要素は g である．形式仕様の事前条件とプログラムの条件から g は 0 より大きい．Not(And(pred0[0], pred0[1]) では，g_list の要素と op_l0 の要素が等しいことを否定している．これにより defining condition が満たされない値の組み合わせを Z3 に探索させる．この例でプログラムは仕様に対して正しいので，test condition と path condition を満たした states の変数は defining condition を満たす．そのため，defining condition が満たされない

```
process func(a: set of nat0, b: set of nat0,
c: set of nat0, d:set of nat0) z: set of nat0
ext wr z
pre card(a) >= 2
post z = inter(diff(c, d), union(a, b))
end_process;
```

(a)

```
static void func(Set<Integer> a,
Set<Integer> b,Set<Integer> c,
Set<Integer> d, Set<Integer> z){
  c.removeAll(d);
  for(Integer v : b) {a.add(v);}
  for(Integer v : c) {
    if (a.contains(v) == false){
      z.add(v);
    }
  }
  System.out.println("z");
  for(Integer v : z) {
    System.out.println(v);
  }
}
```

(b)

図 2. set を用いた事例: (a) 形式仕様, (b) 誤りを含んだ Java プログラム

値の組み合わせは存在しないため，unsat が出力される．このようにしてプログラムの正しさを保証する．

6.2. Set

SOFL の set 型を用いた形式仕様の事例を説明する．ここでは集合 a, b, c, d, z に対し，a と b の和集合と c と d の差集合の共通部分を求める．プログラムにバグが含まれている場合にどのようにして誤りを検知するかを示す．SOFL の集合型の演算子，union，diff，inter を用いた仕様は図 2(a) のようになる．これを反映した Java プログラムは図 2(b) のようになる．形式仕様の inter に当たる部分の処理にバグを含ませた．このプログラムでは，共通しない部分を z に格納している．c と d の差集合と a と b の和集合との共通部分は $v0$ であるが，z には $y2$ が格納されている．このようにプログラムのバグによって states 部分の変数が変化するため，正しくない定理が導出されることがある．TBFV-SE によって導出された誤りを含む定理は以下のようになる．

$$\begin{aligned}&\mathrm{card}(a) \geq 2 \wedge (y0 = y0 \wedge x0 = x0 \wedge v0 = v0) \wedge \\ &(z = \{y2\}, a = \{v0, x0\}, b = \{x0\}, \\ &c = \{y0, v0, y2\}, d = \{d0, y0, y1\}) \\ &\Rightarrow z = \mathrm{inter}(\mathrm{diff}(c, d), \mathrm{union}(a, b))\end{aligned}$$

この定理を検証するためのプログラムは図 3 のようになる．初期化，集合の変数の要素の制約，出力の処理は省略している．ここでは，diff(c, d)，union(a, b)，inter(diff(c, d), union(a, b)) をそれぞれ順番に処理して演算結果を新しい変数として定義する．集合の演算子を処理する時は引数となる集合から共通の要素の数を調べる．また，集合を表す配列には Distinct 関数を適用し，重複を許さないようにする．card(a) は a の要素数を返す演算子であるため，Python の len 関数を用いて実装している．まず，diff(c, d) の演算結果を要素数 2 の配列 op_l0 として定義する．op_l0 の制約は c の要素がすべて含まれ

```
op_l0 = IntVector("op_l0", 2)
s.add(Distinct(op_l0))
pred0 = []
for i in range(2):
  p = And(Or(op_l0[i]==c[0],
      op_l0[i]==c[1],op_l0[i]==c[2]),
      And(op_l0[i]!=d[0],
      op_l0[i]!=d[1],op_l0[i]!=d[2]))
  pred0.append(p)
s.add(And(pred0[0],pred0[1]))
op_l1 = IntVector("op_l1", 2)
s.add(Distinct(op_l1))
pred1 = []
for i in range(len(a)):
  p = Or(a[i]==op_l1[0], a[i]==op_l1[1])
  pred1.append(p)
pred2 = []
for i in range(len(b)):
  p = Or(b[i]==op_l1[0], b[i]==op_l1[1])
  pred2.append(p)
s.add(And(And(pred1[0],pred1[1]),
      Or(pred2[0])))
op_l2 = IntVector("op_l2", 1)
s.add(Distinct(op_l2))
pred3 = []
for i in range(1):
  p = And(Or(op_l2[i]==op_l0[0],
      op_l2[i]==op_l0[1]),
      Or(op_l2[i]==op_l1[0],
      op_l2[i]==op_l1[1]))
  pred3.append(p)
s.add(And(pred3[0]))
pred4 = []
for i in range(1):
  p = Or(z[i]==op_l2[0])
  pred4.append(p)
s.add(len(a)>=2)
s.add(And(And(y0==y0, x0==x0), v0==v0))
s.add(Not(And(pred4[0])))
```

図 3. set の事例の検証プログラム

て，d の要素が含まれないことである．パーサから呼び出された関数 diff_apply によって制約が生成される．次に union(a, b) の演算結果を要素数 2 の配列 op_l1 として定義する．制約は op_l1 に a と b の要素がすべて含まれることである．union の第 2 引数が第 1 引数の部分集合の場合，演算結果は第 1 引数と同じになる．inter(diff(c, d), union(a, b)) の演算結果を要素数 1 の配列 op_l2 として定義する．op_l2 は op_l0 と op_l1 の共通部分になる．制約は op_l2 の要素が op_l0 と op_l1 のどちらにも含まれることである．以上の演算からなる op_l2 と z が等しいことを検証する．Not(And(pred4[0])) で z の要素が op_l2 に含まれないことを表している．inter(diff(c, d), union(a, b)) の結果は {y2} とならないため，sat が出力される．このようにして定理の矛盾を検知できる．

7. 議論

提案手法では，プログラムが形式仕様の条件を満たしていない場合に，エラーを表示できる．しかし，形式仕様そのものの条件が矛盾していたり，文法が誤っていたりする場合，表示されるエラーを区別できない．また，形式仕様が正しく書かれていても，sequence 型や set 型の出力変数の要素数が 0 であるとき検証手法が効果的でない．これらの型の変数に対する制約はそれらが持つ要素に対する制約として記述するため，要素数が 0 の場合に制約が記述できなくなる．よって，要素数が 0 になるような特殊なパターンについて検証する場合は，テストデータを用いた検証が比較的有効であると考えられる．

実装したツールで対応する SOFL の型は sequence 型と set 型である．set 型は vector に対して要素が重複せず，要素の順番による違いがないように制約を与えて実装した．Z3 では Arrays や Sequence などの型が存在するが，データ構造が複雑であり，SOFL 演算子の意味との整合性を確認することが難しかったため利用しなかった．TBFV-SE と本ツールでは SOFL の複雑な型への対応がないため一般的なシステム全体の検証は難しく，単純な関数ごとの検証しかできない．対応する型を増やしたり構文解析機能を拡張したりすることで，仕様作成者が定義した関数や module で定義されたシステム全体の検証が可能になると考えられる．

8. 結論

TBFV-SE から導出された定理を自動で SMT ソルバの制約へ変換する手法を提案し，SMT ソルバで定理を検証するツールを開発した．定理に含まれる形式仕様の文法が使われた述語論理式を SMT ソルバで扱える形へ変換し，自動での検証を行った．プログラムが形式仕様に対して正しく実装されていない場合に誤りを検知できる．SOFL には様々な変数の型があるが，形式仕様では主に set 型と sequence 型が用いられるため，それらへの対応を実装した．今後の課題は，より多くの型への対応を実装することで，より複雑な SOFL 形式仕様の条件式を扱えるようにすることである．

参考文献

[1] R. Wang and S. Liu, “TBFV-SE: Testing-Based Formal Verification with Symbolic Execution,” in *Proceedings of the 2018 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, 2018, pp. 59–66.

[2] C. Cadar and K. Sen, “Symbolic Execution for Software Testing: Three Decades Later,” *Communications of the ACM*, vol. 56, no. 2, pp. 82–90, 2013.

[3] S. Liu, *Formal Engineering for Industrial Software Development: Using the SOFL Method*. Springer Science & Business Media, 2013.

[4] P. Letouzey, “Certified Functional Programming: Program Extration within Coq Proof Assistant,” Ph.D. dissertation, Université Paris-Sud, 2004.

[5] S. Liu, “Testing-Based Formal Verification for Theorems and Its Application in Software Specification Verification,” in *Proceedings of the 10th International Conference on Tests and Proofs (TAP)*. Springer International Publishing, 2016, pp. 112–129.

[6] S. Liu, “Testing-Based Formal Verification for Algorithmic Function Theorems and Its Application to Software Verification and Validation,” in *Proceedings of the 2016 International Symposium on System and Software Reliability (ISSSR)*, 2016, pp. 1–6.

[7] R. Wang, Y. Sato, and S. Liu, “Specification-based Test Case Generation with Genetic Algorithm,” in *Proceedings of the 2019 IEEE Congress on Evolutionary Computation (CEC)*, 2019, pp. 1382–1389.

[8] L. de Moura and N. Bjørner, “Z3: An Efficient SMT Solver,” in *Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Springer Berlin Heidelberg, 2008, pp. 337–340.