COBOLプログラムのための SMTソルバによるテストデータ生成

佐々木 裕介[1]　前田 芳晴[1]　上村 学[1]　小林 健一[1]　松尾 昭彦[1]　木村 茂樹[2]　殿岡 弘範[3]

概要：稼働中の多くの業務システムでは，COBOL 言語で書かれたプログラムが使われている．長期に渡り利用されている COBOL プログラムは，何度も修正されるために複雑になっており，修正のたびにテストに多くの工数がかかっている．COBOL プログラムをテスト実行するためのプログラム中の変数への入力値（テストデータ）を自動で生成できれば，業務システムのテストにかかるコストの削減が期待できる．そこで本研究では，SMT ソルバを用いて COBOL プログラムのテストデータを生成する手法を提案する．提案する手法により，COBOL 言語に特有の，変数に対して指定された桁数制約や比較規則などがあっても，テストデータを生成できるようになる．提案する手法を実システムの COBOL プログラムを対象としたテストへ適用することで，テスト実行に必要なテストデータを正しく生成できることを確認した．

Test Data Generation using SMT Solver for COBOL Programs

Yusuke Sasaki[1]　Yoshiharu Maeda[1]　Manabu Kamimura[1]　Kenichi Kobayashi[1]
Akihiko Matsuo[1]　Shigeki Kimura[2]　Hironori Tonooka[3]

Abstract: Programs written in COBOL language are used in many enterprise systems. If we can automatically generate test data for COBOL programs, the cost of testing for enterprise systems can be reduced. In this paper, we propose a novel technique to generate the test data for COBOL programs by using an SMT solver. The proposed technique can consider the restricted number of significant figures in each declaration of a variable. The technique can also consider a peculiar comparison rule of COBOL. With the experiment, we confirmed that our technique can correctly generate test data for COBOL programs of a real enterprise system.

1. 背景

多くの企業や自治体の業務システムにおいて，COBOL 言語で書かれたプログラムが利用されている [1]．現実の組織体制やビジネスは変化を続けるため，業務システムが古くなると，現実に合わせて改修する必要が出てくる．近年では，次のような理由から，COBOL プログラムの保守作業を効率化することが急務となりつつある．

修正コストの増大　長期に渡り使われている業務システムの多くは，何度も修正が行われ複雑になっている．COBOL プログラムはこのような業務システムに利用されていることが多く，動作を理解し，テストを行うのが困難となっている．

エンジニアの不足　COBOL プログラムの開発に携わっていた団塊の世代の退職なども相まって，COBOL のソースコードに手を加えたり，動作を確認したりすることのできる人材が急激に減っている [2]．

長期間にわたり何度も修正されているような複雑なシステムが保守対象である場合，改修を行い，再リリースするまでの工程において，特に多くの工数を要するのはテストの工程である．改修時に行うテストでは，改修箇所を通過する処理経路を実行して，その実行結果を確認する．そのため，プログラム中の変数へ入力するテストデータには，テストの対象となる処理経路を実行できるものを選ばなければならない．

[1] 株式会社富士通研究所，ソフトウェアシステム研究所
FUJITSU LABORATORIES LTD.
[2] 富士通株式会社
FUJITSU LIMITED
[3] 富士通アプリケーションズ株式会社
FUJITSU APPLICATIONS, LTD.

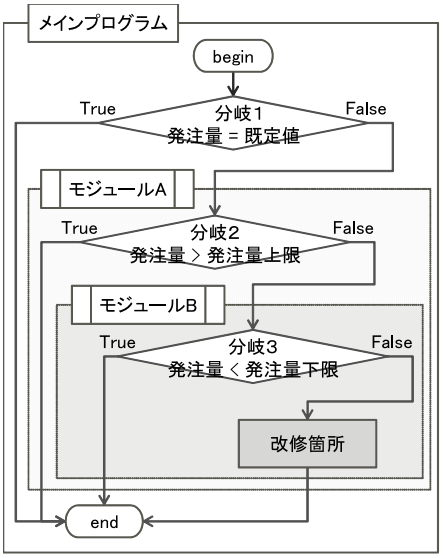

図 1 プログラムの処理経路
Fig. 1 Program execution paths

表 1 変数表
Table 1 Data type definition

項目名	型	桁数	固定値
発注量	数値	3	N/A
既定値	文字	3	“009”
発注量上限	数値	5	00099
発注量下限	文字	5	“001␣␣”

表 2 決定表
Table 2 Decision table

モジュール名	分岐条件	判定
メインプログラム	発注量 = 既定値	偽
モジュール A	発注量 > 発注量上限	偽
モジュール B	発注量 < 発注量下限	偽

テストデータの導出手順は，次の過程に分けられる．

- 手順 1.：必要なテストケースを実行するために，テストデータの満たすべき制約条件を導出する
- 手順 2.：導出した制約条件を満たすよう，テストデータを決定する

多くのモジュールや分岐を通過させないと実行できない箇所をテストするときには，制約条件が非常に多くなる．制約条件が多くなると，手順 2. において，人手で条件を満たすテストデータを考えるのが困難になる．特に，組織統合などが原因で別々のシステムが連携を行っている場合には，システム間で扱っているデータ長や内部表現が異なるために，テストで入力するデータ型の調整が必要となる場合もある．このため多くの工数がかかってしまい，さらには誤ったテストデータを生成してしまう可能性も高くなる．そこで本研究では，後述する COBOL プログラムをテストするための決定表から，プログラムへ入力するテストデータを自動生成することで，保守作業の効率化を行うことを目的としている．

1.1 制約条件の導出

例えば，**図 1** および**表 1** の構成を持つプログラムにおいて，モジュール B 内の改修箇所をテストするには，メインプログラム内の分岐 1 を偽，かつモジュール A 内の分岐 2 を偽，かつモジュール B 内の分岐 3 を偽とするよう，不定値の項目 “発注量” に対するテストデータを作成する（本稿では，表中の “␣”，および図中の “□” は空白を表す）．

実際の保守現場などにおいて，テストデータを作成するときには，仕様書から，改修箇所を通る処理経路を実行するための条件を抽出し，決定表を作成する．**表 2** は，図 1 の改修箇所を通る処理経路を実行するための条件を抽出して作成した決定表の例である．“分岐条件” は図 1 に記載しているそれぞれの分岐条件に対応し，“判定” にはその分岐条件を真とするか偽とするかを記述している．表中の各行は，プログラム実行時に変数が満たすべき条件を示す．例えば，分岐条件 “発注量 = 既定値” の “判定” は偽であるので，変数の満たすべき条件は “¬(発注量 = 既定値)” となる．この決定表から，改修箇所を実行するために入力変数 “発注量” が満たすべき制約条件として，

$$\neg(\text{発注量} = \text{既定値}) \wedge \neg(\text{発注量} > \text{発注量上限}) \wedge \neg(\text{発注量} < \text{発注量下限}) \quad (1)$$

を導出できる．

制約条件の導出については，他に様々な手法が研究されている．例えば，“契約による設計”（Design by Contract）[3] の考え方に基づき，仕様書から，プログラム中の変数などが満たすべき制約を決定する手法がある [4], [5], [6]．“契約による設計” とは，プログラム開発者が想定しているプログラムの状態を，事前条件や，事後条件，不変条件という形で記述する手法である．また，記号実行 [7], [8], [9] などを用いて，プログラムから，特定の経路を通過させるためにプログラム中の変数が満たすべき制約を求め，テストデータを生成する手法もある [10], [11], [12], [13]．記号実行とは，プログラム中の変数を記号値として解釈し，プログラムを実行する手法である．

図 1 に示す例には記述されていないが，代入文により変数の値が更新される場合には，制約条件中の変数にも値を反映する．制約条件中に更新された値を反映させる手法については King の文献 [7] に詳細が載っているため，ここでは説明を省略する．

1.2 テストデータの生成

導出した制約条件を満たすよう，テストデータを作成する．例えば，導出された制約条件が（1）式に示すものなら，これを満たす “発注量” の値（例えば “001” “008” “010” など）を 1 つ作成する．

多くのモジュールや分岐を通過させないと実行できない箇所をテストするときには，分岐条件が非常に多くなる．分岐条件が多くなると，人手でテストデータを作成するのに工数がかかるといった課題が生じたり，誤ったテストデータを生成してしまう恐れがある．

そこで既存研究では，SMT ソルバを用いて制約条件を満たす具体値を生成する手法が研究されている．SMT ソルバ（以降，ソルバ）とは，述語論理式の充足可能性判定問題を解き，充足値を出力するツールである．制約条件をソルバで解ける論理式に変換し，変換した論理式をソルバで解くことで，テストデータを自動生成する．

2. 制約条件から述語論理式への変換

本研究ではソルバとして，Java Path Finder [14][*1] などでも用いられている，Yices [15] を利用する．Yices で解ける述語論理式には，整数値および整数型の配列しか利用できないので，制約条件に出現する変数やリテラルを，すべて数値もしくは数値型の配列で表現するための工夫が必要となる．

2.1 既存手法による変数の表現

既存研究では文字列型の変数が登場する制約条件を解く手法が提案されている．例えば，文字列を ASCII コード値の入っている配列として扱う手法がある [16] ．文字列中の 1 文字は配列中の 1 要素に対応する．

この手法によると，例えば表 1 中の長さが 3 である文字列変数 “既定値” は，ソルバへ入力する述語論理式において，要素数 3 の配列として扱い，例えば以下のように表せる．

$$S[i](0 \leq i \leq 2) \tag{2}$$

このとき，$S[0]$ は，文字列変数 “既定値” の 1 番始めの文字に対応する．同様に，$S[1]$ は 2 つ目の文字に，$S[2]$ は 3 桁目の文字に対応する．さらにこの “既定値” には具体値 “009” が固定されているため，述語論理式には以下の条件を加える．48 は ASCII コードで ‘0’ を，57 は ‘9’ を表す．

$$(S[0] = 48) \wedge (S[1] = 48) \wedge (S[2] = 57) \tag{3}$$

しかし，COBOL プログラムから導出する制約条件には，既存研究の手法では解けない条件が登場する．既存研究で想定している分岐条件中の比較文は，次のようなものである．

- 数値は数値とのみ比較，文字列は文字列とのみ比較
- 数値間の大小判定は代数的に決まる
- 文字列間の大小判定では，辞書順で先に来るものを小さい文字列として判定する

一方，COBOL の比較文には，既存研究で想定されていない次のようなものが存在する．

- 文字列と数値の間での比較がありうる．文字列と数値の大小を判定する条件はソルバで解くべき制約条件の中にも出現する
- 文字列と数値の大小判定では，辞書順で先に来るものを小さい文字列として判定する
- 変数の宣言時に桁数を指定しなければならない．桁数が異なるオペランド同士を比較した際に，特有の比較規則が適用される（後述）

そこで，COBOL で出現する，文字列と数値を比較している条件を，述語論理式でも矛盾なく表現する手法を提案する．

2.2 提案手法

COBOL は C や Java と異なり，変数に対して桁数を指定する必要がある．そのため，述語論理式上でも同様に，桁数を表現する必要がある．例えば，表 1 中の “発注量” は桁数 3 の数値が入る変数であることを示す．“既定値” は桁数 3 の文字列値が入る変数である．

そこで，文字列型の変数，数値型の変数ともに配列へ置き換える．つまり，数値型の変数も文字列型の変数も，ASCII コード値の入っている配列として扱う．より具体的には，1 つの変数に 1 つの配列を対応させ，各桁の数値を ASCII コード値で表す．COBOL 文字列中の 1 文字は，述語論理式において配列中の 1 要素に対応する．同様に，COBOL 数値の 1 桁は，述語論理式において配列中の 1 要素に対応する．配列の長さは指定されている桁数分だけ取る．

例えば，3 桁の数値変数 “発注量” は，述語論理式において，以下の数値型の配列によって表す．末尾の要素 $N[2]$ が一の位の数値である．

$$N[i](0 \leq i \leq 2) \tag{4}$$

数値以外の値が入らぬよう，述語論理式上の配列の各要素について，とりうる ASCII コード値の上限に ‘0’ を，下限に ‘9’ を設定する．

$$48 \leq N[i] \leq 57(0 \leq i \leq 2) \tag{5}$$

さらに，数値も文字列と同様に配列として表現するので，文字列との比較条件をそのまま論理式として表現できる．大小関係は配列の先頭から辞書順で判定する．

例えば，長さが 3 である文字列 “既定値” があり，

[*1] Java のバイトコードに対するモデル検査を行うツールで，バイトコードの記号実行にも使われている．記号実行の分岐判定時に Yices などの制約ソルバを利用している．

“既定値 = 発注量” を真にするという制約条件がある場合，述語論理式は，既定値を配列 S，発注量を配列 N とすると以下のように表せる．

$$S[i] = N[i](0 \leq i \leq 2) \quad (6)$$

仮に，“既定値 > 発注量” を真にするという制約条件であった場合，述語論理式は以下のようになる．

$$\begin{aligned}&(S[0] > N[0])\\ &\quad \vee ((S[0] = N[0]) \wedge (S[1] > N[1]))\\ &\vee ((S[0] = N[0]) \wedge (S[1] = N[1]) \wedge (S[2] > N[2]))\end{aligned} \quad (7)$$

2.3 分岐条件判定時の転記ルールに起因する課題

前述の通り，数値変数を配列に変換するので，述語論理式上で数値間の大小関係を代数的に記述することはできない．そこで，数値間の大小関係も文字列と同様に，辞書順で制約したい．

しかし，COBOL 言語では次に示す特有の比較規則を採用しているために，辞書順での比較と，代数的な比較の両方が行われる数値型の変数があると，制約条件をソルバの解ける述語論理式へ正しく変換することができないという課題がある．

COBOL では，分岐条件中に，桁数の異なるオペランドに対し比較を行う条件を記述できる．COBOL プログラムの実行時，桁数や型の異なるオペランドを比較する場合には，比較するオペランドの値を転記してから比較を実行する．転記とは，分岐条件の判定時に，オペランドの値を別の型へと変換することであり，次の規則に従って実行される．

A. 数値と文字列を比較する際には，数値を文字列へ転記し，文字列比較を行う．例えば図 1 の分岐 1 実行時，比較演算子 “=” のオペランドとなる “発注量” の値を 3 桁の文字列へ転記し，“既定値” に入っている値と文字列比較を行う（**図 2** A）．

B. 数値比較の実行において，桁数の異なるオペランド間で比較を行う場合，桁数が小さい方のオペランドの値の左側に，桁数が大きい方のオペランドと桁数が等しくなるまで 0 を詰める．例えば図 1 の分岐 2 実行時，比較演算子 “>” のオペランドとなる “発注量” の値を 5 桁の数値へ転記し，さらに先頭 2 桁へ 0 を詰め，“発注量上限” に入っている値と数値比較を行う（図 2 B）．

C. 文字列比較の実行において，桁数の異なるオペランド間で比較を行う場合，桁数が小さい方のオペランドの値の右側に，桁数が大きい方のオペランドと桁数が等しくなるまで空白を詰める．例えば図 1 の分岐 3 実行時，比較演算子 “<” のオペランドとなる “発注量” の値を 5 桁の文字列へ転記し（上記 A. も適用される），さらに末尾 2 桁へ空白を詰め，“発注量下限” に入っている値と文字列比較を行う（図 2 C. 中の “□” は空白を示す）．

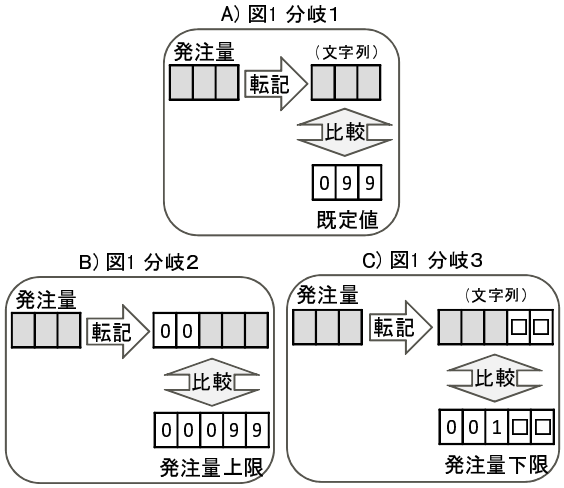

図 2 分岐条件判定時に起こる転記の例
Fig. 2 Examples of comparison of variables in COBOL

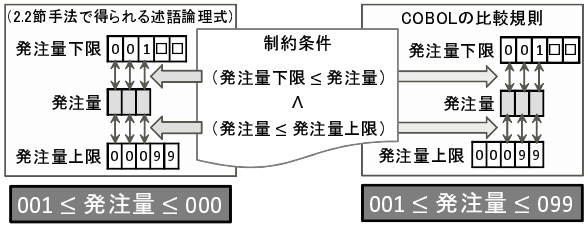

図 3 分岐条件に対する解釈の不整合
Fig. 3 Inconsistency of branch conditions

COBOL で図 2 B と図 2 C に示す転記が両方起こると，前述までの手法により得られる述語論理式との間で，**図 3** に示すような，条件に対する解釈の不整合が生じる．図 3 左は，図 3 中央の “制約条件” を 2.2 節で述べた手法により変換したとき，得られる述語論理式内で，変数の各桁がどのように制約されるかを表す．図 3 右は，COBOL において，“制約条件” 内の変数の各桁がどのように比較されるかを表す．変換後の述語論理式の解釈が，COBOL の解釈に整合しないものとなっている．

よって，2.2 節で述べた手法だけでは，COBOL で出現する，文字列と数値を比較している分岐条件を，述語論理式でも矛盾なく表現することができない．

2.4 転記ルールに起因する課題の解決策

2.3 節の課題を解決するため，事前に図 2 A, B, C に示す転記が起こりうる変数を制約条件の中から検出し，次の対処を行ってから，ソルバへ入力する述語論理式を生成する．

処理（a） 数値型変数と文字列型変数それぞれの大小判定を分けるため，ダミー変数を追加

処理（b） ダミー変数に対し，桁揃えのための制約を追加

2.4.1 処理（a）文字列比較に用いる配列変数の追加

転記規則 A の適用により文字列に転記される数値型の変数があれば，述語論理式では，文字列比較用の配列を別に宣言する．つまり，**図 4** に示すように，文字列比較用の変数を別に用意し，数値比較用の変数と文字列比較用の変数の値が等しくなるよう制約を追加する．図 4 の例では，“発注量” に対して文字列比較用の “発注量 1” と数値比較用の “発注量 2” の 2 つの変数を用意している．これにより，次の処理（b）を行った後でも，配列 “発注量” と “発注量上限” を先頭から辞書順に制約することを可能にする．

この処理を行うと，先ほど配列 N と表現していた “発注量” は次のように $N1$ および $N2$ の 2 つの配列で表現することになる．$N1$ は数値，$N2$ は文字列を表す配列とする．

$$N1[i](0 \leq i \leq 2) \tag{8}$$

$$N2[i](0 \leq i \leq 2) \tag{9}$$

$$N1[i] = N2[i](0 \leq i \leq 2) \tag{10}$$

2.4.2 処理（b）配列長の引き伸ばし

転記規則 B, C の適用により桁数が引き延ばされる変数があるなら，述語論理式で宣言する配列の長さは，転記によって増える桁数分だけ引き延ばして宣言する（**図 5**）．つまり，図 2 B のように，3 桁から 5 桁へ転記される変数があれば，元の桁数が 3 桁であっても，述語論理式で宣言する配列長は 5 とする．配列長を引き延ばすと，引き延ばした分の要素に対する値もソルバから得られるため，ソルバから得られた充足値からは，引き延ばされた要素に対する値を除外する．また，図 2 B のように，転記により桁数の増える変数が数値なら，増える桁数分だけ先頭を 0 値に制約する．図 2 C のように，文字列なら，転記により増える桁数分だけ末尾を空白値に制約する．

先ほど配列 $N1$ および $N2$ で表現していた 発注量 は，長さ 5 の数値 発注量上限 との比較条件と，長さ 5 の文字列 発注量下限 との比較条件があることから，次のように表現することになる．$N1$ は数値，$N2$ は文字列を表す配列とする．48 は ‘0’ の ASCII コード値であり，32 は空白の ASCII コード値である．

$$N1[i](0 \leq i \leq 4) \tag{11}$$

$$N2[i](0 \leq i \leq 4) \tag{12}$$

$$N1[i+2] = N2[i](0 \leq i \leq 2) \tag{13}$$

$$N1[i] = 48(0 \leq i \leq 1) \tag{14}$$

$$N2[i] = 32(3 \leq i \leq 4) \tag{15}$$

さらに，¬(発注量 > 発注量上限) を真にするという制約条件は次のように表現できる（(17) 式は長いため後半部分を省略）．配列 M は，“発注量上限” に対応する配列である．

$$M[i](0 \leq i \leq 4) \tag{16}$$

$$\begin{aligned}&(N1[0] \leq M[0])\\ &\quad \vee ((N1[0] = M[0]) \wedge (N1[1] \leq M[1]))\\ &\vee ((N1[0] = M[0]) \wedge (N1[1] = M[1]) \wedge (N1[2] \leq M[2]))\\ &\quad \vee \ldots\end{aligned} \tag{17}$$

¬(発注量 < 発注量下限) を真にするという制約条件は次のようになる（(19) 式は長いため後半部分を省略）．配列 T は，“発注量下限” に対応する配列である．

$$T[i](0 \leq i \leq 4) \tag{18}$$

$$\begin{aligned}&(N2[0] \geq T[0])\\ &\quad \vee ((N2[0] = T[0]) \wedge (N2[1] \geq T[1]))\\ &\vee ((N2[0] = T[0]) \wedge (N2[1] = T[1]) \wedge (N2[2] \geq T[2]))\\ &\quad \vee \ldots\end{aligned} \tag{19}$$

以上により，COBOL で出現する，文字列と数値を比較している分岐条件を，述語論理式でも矛盾なく表現することができるようになる．

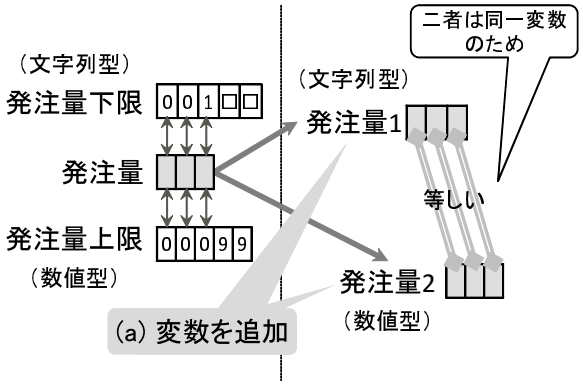

図 4 処理 (a) 文字列比較に用いる配列変数の追加
Fig. 4 Variable generation for string comparison

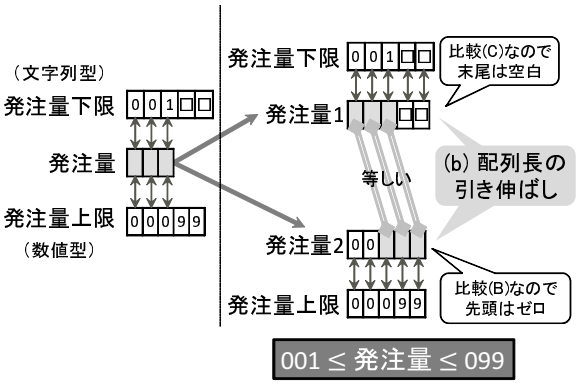

図 5 処理 (b) 配列長の引き伸ばし
Fig. 5 Comparison tuning for each variable

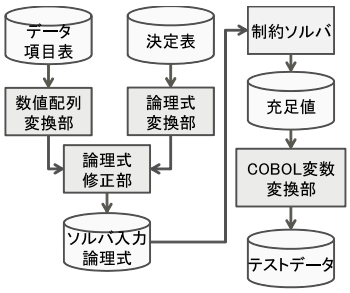

図 6 システム構成
Fig. 6 System configuration

3. テストデータ生成システム

提案した変換方式をシステムとして実装した（**図 6**）．入力は，COBOL における変数の定義情報と変数の満たすべき条件であり，出力は，入力された条件を満たす COBOL 変数へのテストデータである．システムの実装は Java で行った．

まず，数値配列変換部により，COBOL の変数情報（データ項目表）を，ソルバで扱える数値配列へと変換する．論理式変換部により，決定表から導出した制約条件をソルバで扱える述語論理式へと変換する．さらに，2.4 節で述べた転記への対処を論理式修正部で行う．生成した論理式をソルバへ入力し，充足値を得る．最後に，COBOL 変数変換部により，充足値を COBOL の変数へ入力できる値へと変換する．

例えば，実装したシステムへ表 2 の決定表を入力すると，テストしたい経路に対するテストデータとして，“発注量 = 010” が出力される．“010” を “発注量” へ入力し，図 1 のプログラムを実行すると，分岐 1 の分岐条件を偽，分岐 2 の分岐条件を偽，分岐 3 の分岐条件を偽とする処理経路を通過させることができる．

4. 評価実験

本節では，実プロジェクトの決定表やソースコードを使い，実装したシステムおよび提案した手法の評価を行う．具体的には，次の点を測定し，その結果から提案手法の有効性と，実用性を評価する．

RQ1 既存手法では解けなかった COBOL 特有の制約条件は，実プロジェクトのソースファイル中にどれくらいあるのか．つまり，文字列型の変数と数値型の変数を比較する条件や，桁数が異なるオペランドを比較する条件は，実プロジェクトのソースファイル中にどれくらいあるのか．

RQ2 実装したシステムは，入力された制約条件中に，文字列型の変数と数値型の変数を比較する条件や，桁数が異なるオペランドを比較する条件があるとき，テストデータを生成できるか

表 3 COBOL ソースファイルの調査結果
Table 3 Result of the COBOL source file research

ファイル	LOC	比較条件の総数	数値と文字の比較の数	両辺で桁数が異なる比較の数
file1	135	5	0	0
file2	355	26	0	0
file3	346	12	2	0
file4	393	45	3	0
file5	1900	667	0	0
file6	5473	8280	3179	33
file7	2551	4749	21	10
file8	625	54	3	10

RQ3 実装したシステムは期待されているパフォーマンスを満たしているか

4.1 実プロジェクトの調査

実プロジェクト中の 8 つのソースファイルに対して，数値と文字の比較条件や，桁数の異なるオペランドを比較する条件がいくつあるか調査を行った結果，**表 3** のようになった．表の各行はそれぞれのファイルの統計情報を示す．“LOC” はそのソースファイルの行数を示す．“比較条件の総数” はそのソースファイルに含まれる比較条件の総数を示す．分岐条件中に論理和や論理積があれば，区切ってからそれぞれを比較条件としてカウントしている．“数値と文字の比較の数” は数値と文字の比較条件数を示す．“両辺で桁数が異なる比較の数” は異なる桁数のオペランドを比較している条件の数を示す．

数値と文字の比較条件は，8 つのファイル中 5 つで登場しており，異なる桁数のオペランドを比較している条件は 3 つのファイルで登場している．数値と文字の比較が 3179 回と最も多く行われている file6 はオンライン業務システムのソースファイルである．入力された文字に対して数値を比較するために，数値と文字の比較する条件が多いのではないかと考えられる．

表 3 において，比較条件の総数と，数値と文字を比較している条件の数の間では正の相関がみられる．Spearman の順位相関係数 [17] を求めたところ，相関係数 ρ は 0.7075276 となった．ρ の有意性を確認するため p-値を計算したところ 0.04963 となり，有意水準 5% で ρ は有意であることを確認した．また，比較条件の総数と，異なる桁数のオペランドを比較している条件の数の間でも正の相関がみられる．Spearman の順位相関係数 ρ は 0.7698004 となった．p-値は 0.02547 となり，有意水準 5% で ρ は有意であることを確認した．

以上から，次のことが確認できた．

- 実プロジェクトにおいて，文字列型の変数と数値型の変数を比較する条件や，異なる桁数のオペランドを比

較する条件を記述することはある
- 特に調査したソースファイル群については，ソースファイル中の条件数が多いほど，文字列型の変数と数値型の変数を比較する条件や，桁数が異なるオペランドを比較する条件が多く出現する

4.2 テストデータの生成

実プロジェクトの決定表から導出した制約条件を実装したシステムへ入力し，テストデータを生成できるかどうか確認する．

評価に用いる決定表は，実際の業務システムのテストにおいて，人手でテストデータを求めるために作成されたものである．表 2 の例に示すような条件が記載されており，導出できる制約条件は全部で 44 通りある．1 つの制約条件の中に，最小で 7，最大で 31 の比較条件が含まれている．この中には，文字列型の変数と数値型の変数を比較する条件や，桁数が異なるオペランドを比較する条件も存在する．

まず，2.4 節の処理（a）(b）を適用せずに，テストデータの生成を行った．**図 7** の “（a）(b）非適用” に結果を示す．文字列型の変数と数値型の変数を比較する条件や，桁数が異なるオペランドを比較する条件を含んでいる制約については，論理式への変換を正しく行えなかったためにいずれも充足値なしとなり，テストデータの生成に失敗した（図 7“生成に失敗”）．

次に，2.4 節の処理（a）(b）を適用し，テストデータの生成を行った．図 7 の “（a）(b）適用” に結果を示す．39 通り，つまり新たに 10 通りの制約について，テストデータを生成することに成功した．残りの 5 通りの制約についてはソルバにより充足値なしと判断されたため，決定表の作成者に確認したところ，これら 5 通りの制約導出に用いた制約条件には誤りがあり，制約を充足するテストデータが存在しないことが判明した（図 7“制約条件に誤り”）．

以上のことから，決定表から導出できる 44 通りの制約のうち 10 通りについて，処理 （a） (b） 適用前はテストデータを生成できなかったが，適用後は，テストデータを生成できるようになることがわかった．また，実際の業務システムのテストにおいて，人手でテストデータを求めるために作成した決定表から，制約条件の誤りを発見できることも確認した．

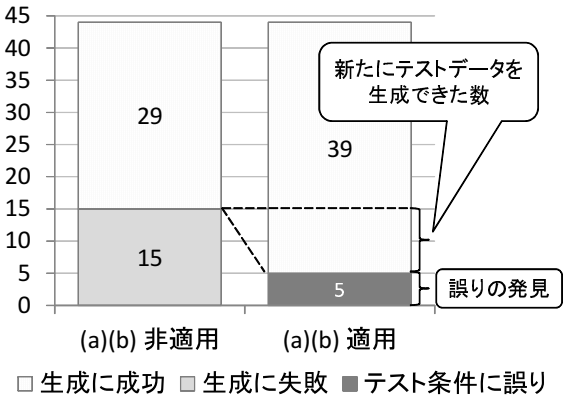

図 7 テストデータ生成の結果
Fig. 7 Result of test data generation

4.3 パフォーマンス

前述の評価実験で，パフォーマンスに関して想定していた要件は次の通りである．
- 実行時間については，人手でテストデータを考えるよりも時間がかからないこと（制約条件の導出などは終わっているものとする）
- 消費メモリについては 32 bit のマシンで動作させたとき，OutOfMemoryException が起こらない程度であること．つまり，最大でおよそ 1.3 GB 以下であればよいとする

実装したシステムを実行するコンピュータのプロセッサは Dual-Core AMD Opteron（tm） Processor 2218 2.60 GHz, OS は Windows （64 bit） である．

実行時間は 1 テストケースあたり，おおよそ 830 msec かかった．Yices の実行には平均 504 msec を費やした．処理 （a）(b） に費やした時間は 5 msec 程度であった．一方，人手でテストデータを作成する場合，例えば (1) 式に示すような条件やより複雑な条件について，1 テストケース 1 秒以下でテストデータを考えることは難しい．

また，プログラム実行時に JVM 上で消費した最大メモリは，22 MB から 35 MB であった．

以上から，今回の実験では，実行時間と消費メモリ共に，想定している要件を満たしていたといえる．

4.4 実験結果のまとめ

この結果により，RQ1 および RQ2, RQ3 それぞれについて以下のことが確認できた．

RQ1 実プロジェクトのソースファイルにおいて，文字列型の変数と数値型の変数を比較する条件や，桁数が異なるオペランドを比較する条件は実際に生じる．

RQ2 評価に用いた決定表から導出できる 44 通りの制約のうち 10 通り（約 23%）について，処理 （a） (b） 適用前はテストデータを生成できなかったが，適用後は，テストデータを生成できるようになった．この他に，実際の業務システムのテストにおいて，人手でテストデータを求めるために作成した決定表から，制約条件の誤りを発見できることも確認した．

RQ3 テストデータを自動生成するのに，1 テストケースあたり約 1 秒かかり，利用したメモリは最大時で約 30MB であった．今回の実験では，実行時間と消費メモリ共に，期待している要件を満たしていたといえる．

5. おわりに

本研究では，COBOL プログラムへソルバによるテストデータ生成技術を適用したときに生じる，COBOL の言語仕様に起因する課題を提示した．そして，COBOL の言語仕様を考慮して，COBOL のテストデータが満たすべき制約条件を，ソルバが解ける述語論理式へ変換する手法を提案し，テストデータ生成システムとして実装した．評価実験では，実プロジェクトのソースファイル中に，提案手法により新たに解くことができるようになる制約条件，つまり，文字と数値を比較する条件や，桁数の異なる変数を比較する条件が出現することを確認した．今回は調査したソースファイル数が少なかったため，今後は調査対象を増やし，この結果が信用できるものか確認する予定である．さらに，実装したテストデータ生成システムへ，実際のテストのために人手で作成した決定表を入力し，充足値のあるケースについてはすべてテストデータを生成できることを確認した．決定表に充足値のない制約条件を記述した場合，その誤りを検出可能であることも確認した．これにより，今まで手動で作成せざるを得なかった COBOL プログラムのテストデータを自動生成できるようになり，保守現場の工数を大幅に削減できると想定している．現在，どれほどの工数削減効果があるか確認するため，実際のテストプロセスへ今回開発したツールを適用し，実践評価を進めているところである．

今後は，記号実行を用いたテストデータ生成手法技術を，COBOL プログラムへ適用できないか検討する．また近年は，限定的にではあるが非線形演算を含む制約を解けるソルバも作られているため [18]，その技術を応用することも検討したい．

参考文献

[1] Mitchell, R. L.: Cobol: Not Dead Yet, computerworld (online), available from ⟨http://www.computerworld.com/s/article/266156/Cobol_Not_Dead_Yet⟩ (accessed 2012/08/01).

[2] McAllister, N.: California's Cobol conundrum, InfoWorld (online), available from ⟨http://www.infoworld.com/d/developer-world/californias-cobol-conundrum-067⟩ (accessed 2012/08/01).

[3] Meyer, B.: Applying Design by Contract, *Computer*, Vol. 25, pp. 40–51 (1992).

[4] Offutt, J. and Abdurazik, A.: Generating Tests from UML Specifications, *Proc.of the 2nd International Conference on the Unified Modeling Language (UML '99)*, pp. 416–429 (1999).

[5] Klein, C., Flatt, M. and Findler, R. B.: Random Testing for Higher-order, Stateful Programs, *Proc.of the ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA 2010)*, pp. 555–566 (2010).

[6] Khurshid, S. and Marinov, D.: TestEra: Specification-based Testing of Java Programs Using SAT, *Automated Software Engineering*, Vol. 11, No. 4, pp. 403–434 (2004).

[7] C.King, J.: Symbolic Execution and Program Testing, *Comm.ACM*, Vol. 19, No. 7, pp. 385–394 (1976).

[8] S.Pasareanu, C. and Rungta, N.: Symbolic PathFinder: Symbolic Execution of Java Bytecode, *Proc.of the IEEE/ACM International Conference on Automated Software Engineering (ASE 2010)*, pp. 179–180 (2010).

[9] S.Pasareanu, C., Rungta, N. and Visser, W.: Symbolic Execution with Mixed Concrete-Symbolic Solving, *Proc.of the 2011 International Symposium on Software Testing and Analysis (ISSTA 2011)*, pp. 34–44 (2011).

[10] Xie, T., Marinov, D., Schulte, W. and Notkin, D.: Symstra: A Framework for Generating Object-Oriented Unit Tests using Symbolic Execution, *Proc.of the 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2005)*, pp. 365–381 (2005).

[11] Sen, K., Marinov, D. and Agha, G.: CUTE: A Concolic Unit Testing Engine for C, *Proc.of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE 2005)*, pp. 263–272 (2005).

[12] Tillmann, N. and de Halleux, J.: Pex - White Box Test Generation for .NET, *Proc.of the 2nd International Conference on Tests And Proofs (TAP 2008)*, pp. 134–153 (2008).

[13] Godefroid, P., Klarlund, N. and Sen, K.: DART: Directed Automated Random Testing, *Proc.of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2005)*, pp. 213–223 (2005).

[14] NASA: JPF .. the swiss army knife of Java verification, NASA (online), available from ⟨http://babelfish.arc.nasa.gov/trac/jpf/wiki/intro/start⟩ (accessed 2012/08/01).

[15] Dutertre, B. and Moura, L.: A Fast Linear-Arithmetic Solver for DPLL(T), *Proc.of the 18th International Conference on Computer Aided Verification (CAV 2006)*, pp. 81–94 (2006).

[16] Bjorner, N., Tillmann, N. and Voronkov, A.: Path Feasibility Analysis for String-Manipulating Programs, *Proc.of the 15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2009)*, pp. 307–321 (2009).

[17] 池田　央：統計ガイドブック，新曜社 (1989).

[18] Borges, M., d'Amorim, M., Anand, S., Bushnell, D. and Pasareanu, C.: Symbolic Execution with Interval Constraint Solving and Meta-heuristic Search, *Proc. of the 5th International Conference on Software Testing (ICST 2012, Verification and Validation)* (2012).