Vol.2014-SE-185 No.25
2014/7/11

ソースコード生成を利用した結合テスト向けデータベース生成手法の提案

丹野 治門[1,a)]　張 暁晶[1]　生沼 守英[1]

概要：本研究では，関係データベース (DB) を用いる業務システムの結合テストを対象として，DB への参照，更新を行う各テストケースに対し，適切な DB 初期状態を自動生成する問題を扱う．既存手法では，DB への更新を伴うテストケースに対し，適切な DB 初期状態を生成することができないという問題があった．本研究ではこの問題点を解決するため，ソフトウェアの設計モデルから，テストケースごとにシミュレーション用のソースコードを生成し，そのソースコードに Concolic Testing を適用することで，DB への参照，更新アクセスを伴う各テストケースに対して，適切な DB 初期状態を生成する手法を提案する．本論文では，実際の業務システムとサンプルアプリケーションを用いた適用評価についても述べる．

キーワード：ソフトウェアテスト, テストデータ生成，データベース，モデルベーステスト，Concolic Testing

Initial Database Generation for Integration Testing by Source Code Generation

Abstract: This research focuses on how to automatically generate initial test data of relational database properly for each test case referring to and updating database tables, when testing enterprise systems. Existing approaches cannot generate appropriate initial database for test cases updating database tables. To break the limitation, we propose a method for initial database generation using source code generation. This method adopts a software design model, and from this design model, our method generates source codes which can simulate behaviors of the software. Then by applying concolic testing to the source codes, appropriate initial database for each teat case can be obtained. This paper also describes the evaluation with an industry level web application and a sample application.

Keywords: Software Testing, Test Data Generation,Database, Model Based Testing, Concolic Testing

1. はじめに

関係 DB を用いる業務システムの結合テストを行う際には，確認したいソフトウェアの特定の振る舞いを起こすために，テストケースごとに適切な DB 初期状態を用意する必要がある．本研究ではテストケースごとに適切な DB 初期状態（及び画面入力値）を自動生成する問題を扱う．

結合テスト向けの DB 初期状態を自動で生成する手法としては，ソフトウェア設計情報などから DB 初期状態と画面入力値の両方を自動生成する手法 [14, 24, 26] が存在する．これらの手法では，テストケースの事前状態として適切な DB 初期状態や，画面への入力値について，これらが満たすべき制約を集め，制約ソルバ [7] を用いて制約を解き，DB 初期状態，入力値の具体値を生成することが可能である．しかしながら，これらの既存手法は，DB 初期状態がテストケース実行中に更新されないことを前提とした手法であり，DB への参照アクセスのみを扱うテストケースには対応できるが，処理の途中に更新アクセスを伴い，テスト中に DB の状態が刻々と変わるようなテストケースに対しては，適切な DB 初期状態を生成することができないという問題がある．

このような問題が起こる具体的な例として，図 1 のようなイベント参加募集システムのテストを行う場合を考えて

[1] NTT ソフトウェアイノベーションセンタ，東京都港区港南 2-13-34 NSS-II ビル 6F
[a)] tanno.haruto@lab.ntt.co.jp

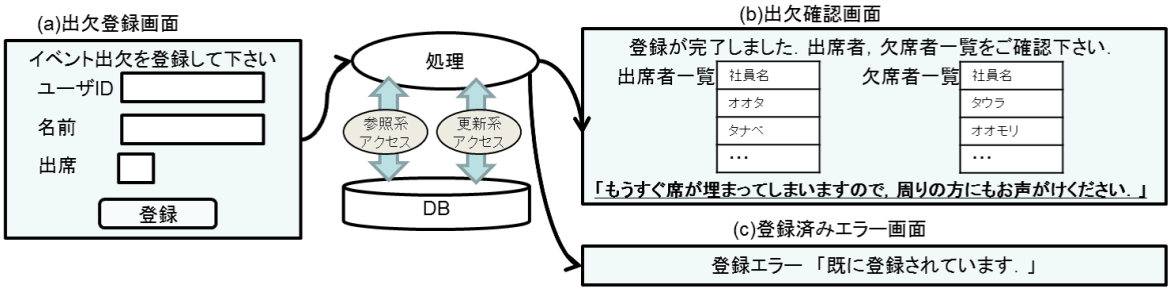

図 1 例題：イベント参加募集システム

みる．このシステムでは，イベントの席がある場合に図 1「(a) 出欠登録画面」が表示され，ユーザが自分の ID と名前を入力し，出席する場合はチェックボックスにチェックを入れてから登録ボタンを押す．もし，登録しようとしたユーザがまだ登録されていなければ，システムで登録が行われ，図 1「(b) 出欠確認画面」へ遷移し，登録が完了したというメッセージと，出席者，欠席者の一覧が表示される．また，出席者が 10 人以上になった場合は，イベントの席が埋まりつつあるため，周りの人に声がけをして欲しいというメッセージが表示される．また，ユーザが既に登録されていた場合は，エラーとなり図 1「(c) 登録済みエラー画面」へ遷移する．

ユーザが出席する旨で登録を行った際に，席が埋まりつつあるというメッセージが表示されることを確認するテストケースを考えてみると，例えば「出席として登録している社員をあらかじめ 9 人 DB の登録者テーブルへ登録し，まだ登録していないユーザが出席で登録する」など，最終的に登録者が 10 人以上となるよう事前状態として適切な DB 初期状態及び画面入力値を用意する必要がある．このシステムでは，登録者テーブルに対し参照アクセス，更新アクセスが行われるため，テスト中に DB の状態が変更される．そのため，上述のようなテストケースでは，既存手法を用いても適切な DB 初期状態を生成することができない．

本研究では，これらの問題を解決し，DB への更新アクセスを伴うテストケースに対しても適切な DB 初期状態を生成できるようにし，現実的なアプリケーションにおいて，より多くのテストケースに対して DB 初期状態を生成できるようにすることを目指す．また，本研究では，図 1 に示したような業務システムの 1 回の画面遷移の動作を確認するためのテストで用いる DB 初期状態の生成をスコープとする．

本研究の貢献は以下の 2 点である．

- 著者らの既存手法 [24] を拡張し，参照アクセスに加え更新アクセスの記述もできる設計モデルを規定し，更新アクセスを伴うテストケースに対しても，適切な DB 初期状態を生成する方式を提案した．提案方式では，Concolic Testing を用いた結合テスト向けテストデータ生成 [21, 22] のコンセプトに基づき，設計モデルから抽出したそのパスごとに Concolic 実行可能なシミュレーション用ソースコードへ変換し，Concolic 実行することでテストデータ生成を行うことを特徴とする．
- 提案手法，既存手法 [24] の両方式を，筆者らが開発している結合テスト設計支援ツール TesMa [25] へ組み込み，実開発案件と JavaPetStore を用いた評価を行い，既存手法と比較して提案手法はより多くのテストケースに対して DB 初期状態を生成できることを確認した．

以降，2 章では，テストデータ生成に関する既存技術とその問題，及び Concolic Testing について述べる．3 章で，本研究が提案する手法について述べ，4 章では提案手法の評価とその結果について述べる．そして，最後に 5 章で本論文の結論を述べる．

2. 既存研究

2.1 業務システムの結合テスト向けテストデータ生成の既存手法

本研究がスコープとする結合テストの DB 初期状態生成に関する既存手法としては，DB スキーマで定められた制約を満たした DB 初期状態を生成する手法と，各テストケースの事前状態として適切な DB 初期状態を生成する手法が存在する．

前者の手法としては，乱数を用いて DB レコードの自動生成を行う DBMonster [2]，疑似個人情報ジェネレータ [5]，Visual Studio Database Edition [3] などがあるが，テストケースごとにレコードの追加や削除など，DB 初期状態の微調整を行ったり，画面入力値については各テストケースごとに別途で用意したりといったことが必要になる．

後者は，各テストケースごとに事前状態として適切な DB 初期状態を生成するため，前者よりも有用な手法である．後者の手法で初期の研究としては，DB 初期状態が満たすべき事前条件をユーザが記述し，テストを実施する前に使用する DB が事前条件を満たしているかを確認する手

法 [6] が提案されている．最近では，テストケースごとに適切な画面入力値と DB 初期状態の両方を同時に自動生成する手法も提案されている．筆者らの手法 [24] や，藤原らの手法 [14,26] では，ソフトウェアの設計情報である DB 定義，処理フロー，画面入力値定義などの情報や，テストケースの事前条件に基づき，テストデータが満たすべき条件を制約として全て抽出し，これらの制約を SMT ソルバ [7] や制約プログラミング [13] を用いて解くことでテストケースごとに適切なテストデータの具体値を生成している．これらは，テストケースごとに DB 初期状態が満たすべき制約を集めて解く方式であり，更新アクセスによって DB 状態が変化していくような場合には対応できず，適用領域が狭い．更新系アクセスを扱う場合，DB の状態が刻々と変わるため，DB 初期状態の生成にあたって，DB のライフサイクルを考慮できるよう，異なった手法や工夫が必要になる．

本研究では，筆者らの既存手法 [24] の設計モデルを拡張し，業務システムの 1 回の画面遷移の動作を確認する結合テストにおいて，参照アクセスに加えて，更新アクセスを伴うテストのためのテストケースに対しても事前状態として適切な DB 初期状態，画面入力値のペアを生成することを目指している．

2.2 Concolic Testing を用いたテストデータ生成技術

単体テストにおけるテストデータの自動生成手法として Concolic Testing [18] がある．Concolic Testing とは，主に単体テストを対象とし，効率よくパスカバレッジを向上することを目指したテストデータの自動生成技術である．Concolic Testing はソースコード上で，未到達のパスを探索しながら，繰り返しプログラムの実行を行うことを特徴とする．具体的には，具体値によるプログラムの実行と同時に記号実行 [15] を行い，通ったパスの分岐条件を記録していく．そして，記録した分岐条件のひとつを論理否定させた制約を作成し，その制約を満たすような値を次の実行の入力として用いることで，未到達のパスを通るようなテストデータを次々に生成していく．このように，Concolic Testing を用いると，ソースコード上の未到達パスへ到達するようなテストデータを自動で次々に得ることが可能である．Concolic Testing を用いると，プログラムの実行に従って変数の状態が更新されていくような場合でも問題なく扱うことができる．

Concolic Testing を利用し，単体テストにおける入力値や，プログラムがアクセスする DB の初期状態を自動生成する試みとしては，未到達のパスへ到達するよう入力値と DB 初期状態の両方を調整しながらプログラムを繰り返し実行する手法 [11,17] や，既に存在する DB 初期状態を考慮しプログラムの入力値のみを次々に生成していく手法 [16] が存在する．

最近では結合テスト向けに Concolic Testing を応用する試み [21,22] も行われている．これらの手法ではソフトウェアの設計モデルからシミュレーション用のソースコードを生成し，そのソースコードに対して Concolic Testing を行うことで，結合テスト向けの DB 初期状態，画面入力値を生成する．Concolic Testing は元々はソースコードのパスカバレッジ向上を目指した手法であるため，単純に適用すると，結合テストで必要となるテストケースを効率よく得ることができなくなるため，ソースコードの生成の仕方には工夫が必要となる．これらの試み [21,22] では，設計モデル全体からシミュレーション用のソースコードを生成し，そのコードを実行することで各テストケースの DB 初期状態値，画面入力値を生成していたが，この方法では現実的な時間で解けないケースが多く，実用的ではなかった．

3. 提案手法

本研究では，モデルベーステスト [8] の考え方を用い，テスト対象システムの設計情報をモデル化した**設計モデル**から，テストケースとそのテストケースの事前状態として適切な DB 初期状態の生成を行う．提案する手法の特徴を以下に示す．

- 提案手法が入力とする設計モデルは，筆者らの既存手法 [24] における，**処理フロー**，**入力定義**，**DB スキーマ**で構成される設計モデルを拡張し，処理フローでは，DB 参照アクセスに加え，DB 更新アクセスについても記述すること可能である．そして，この設計モデルから DB 更新アクセスを伴うテストケースを抽出し，そのテストケースの事前状態として適切な DB 初期状態，入力値のペアを生成する．
- 本手法が出力するテストケースは処理フローの各**パス**に対応しており，DB 初期状態と入力値から構成される．提案手法では，過去の試み試み [21,22] と異なり，設計モデル全体ではなく，設計モデルから抽出した各テストケースごとにソースコードを生成し，それらに対して個別に Concolic Testing を適用することで，DB 初期状態，入力値のペアを得る．そのため，テストケース個別にタイムアウトを設定することが可能であり，DB 初期状態値，画面入力値の生成に時間のかかるテストケースについては個別に打ち切ることができる．

これらの特徴により，業務システムにおいて，DB 更新アクセスを伴うビジネスロジックの振る舞いを確認するための様々なテストケースに対して事前状態として適切な DB 初期状態を生成することが可能である．

図 2 に提案手法の全体像を示す．ビジネスロジックの振る舞いを網羅的にテストするため，処理フローから，Decision/Condition Coverage [20] を満たすパスを既存技術 [23] を用いて抽出し，パスごとにテストケースを生成

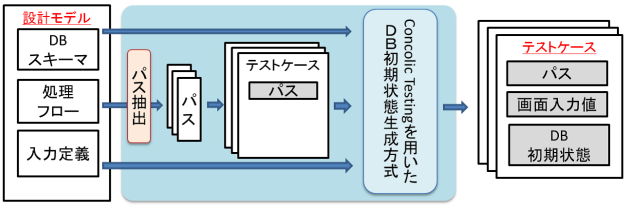

図 2　提案手法の全体像

TableId = 登録者			
ColumnId	型	定義域	カラムの種類
ID	IntegerType	Min = 0, Max = 5000	PK
名前	StringType,	MinLength = 0, MaxLength = 32	Normal
出席	IntegerType	Min=0, Max=1	Normal

図 3　設計モデル：DB スキーマ

し，テストケースに含まれる DB 初期状態と入力値を生成する．以降，本手法で扱う設計モデル，テストケース，そして Concolic Testing を用いた DB 初期状態の生成アルゴリズムについて説明する．

3.1　設計モデル

本手法が入力とする設計モデルの定義のうち，筆者らの既存手法 [24] との差分のみ表 1 に示す．表 1 の網掛けの部分 (項番 4,6,7,8) が既存手法の設計モデルを拡張した部分でであり，ノードには DB 参照アクセス (`DBSelect`) に加えて，更新アクセス (`DBInsert`,`DBUpdate`,`DBDelete`) を記述することが可能である．入力定義，DB スキーマなど他の設計情報の定義は既存手法 [24] と同じものを用いている．処理フローはビジネスロジックを記述する設計情報であり，ノードとエッジで構成される有向グラフである．ノードは処理の内容を，エッジはある処理から次の処理への遷移をそれぞれ表す．エッジには，一つ前のノードで DB への参照を用いた場合のレコード検索結果件数に対する制約や，入力値が満たすべき条件を，ガード条件として記述することができる．

具体的な例として，図 1 で紹介したイベント参加募集システムの設計モデルを図 3，4，5 に示す．図 4 の処理フローでは，ノード (2) のユーザが登録済みかの確認，ノード (7) の出席者人数の確認において DB 参照アクセス (`DBSelect`) を用い，ノード (5)，(6) の登録処理では DB 更新アクセス (`DBInsert`) を用いて設計情報を記述している．また，図 3 の DB スキーマでは登録者テーブルが定義されており，図 5 の入力定義では，ユーザ ID など画面への入力がそれぞれ定義されている．

3.2　テストケース

本手法が出力するテストケースはパス，DB 初期状態，入力値の 3 者で構成される．DB 初期状態は DB スキーマ

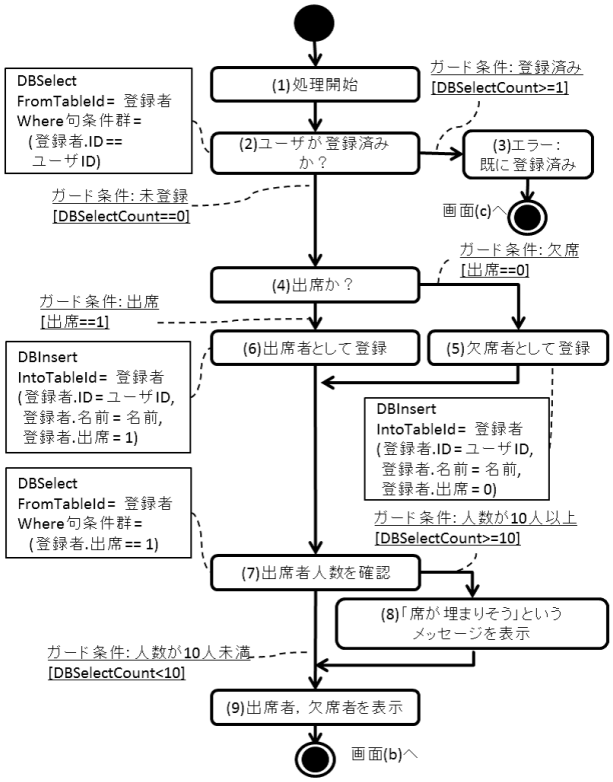

図 4　設計モデル：処理フロー

VariableId	型	定義域
ユーザID	IntegerType	MinValue = 0, MaxValue = 5000
名前	StringType	MinLenght = 1, MaxLength = 32
出席	IntegerType	MinValue = 0, MaxValue = 1

図 5　設計モデル：入力定義

を満たし，かつパス上のガード条件を全て満たすものとなる．また，全ての入力値は，パス中のガード条件と入力定義における定義域を満たすものとなる．

具体的な例として，図 3，4，5 において，ユーザが出席で登録をしたときに，席が埋まりつつあるメッセージが表示される振る舞い（図 4 におけるノード (1)，(2)，(4)，(6)，(7)，(8)，(9) のパス）を確認するテストケースを図 6 に示す．このテストケースは，テスト開始時には 9 人の出席を表明したユーザが登録された登録者テーブルを DB 初期状態としてもち，登録者テーブルには存在しない ID のユーザを出席として登録するための入力値をもっているため，テストの事前状態として適切な DB 初期状態と入力値のペアとなる．

3.3　Concolic Testing を用いた DB 初期状態生成方式

提案手法では図 2 に示したとおり，「入力定義，DB スキーマ，テストケースに対応するパス」を入力として，テストケースごとにシミュレーション用ソースコードを生成し，そのソースコードに対して Concolic Testing を適用す

表 1 設計モデルの定義 (一部抜粋)

項番	式		
	処理フロー		
1	＜処理フロー＞	::=	＜ノード＞+ ＜エッジ＞*
2	＜ノード＞	::=	NodeId:String Text:String NextEdgeId:String* ＜ DB アクセス＞?
3	＜エッジ＞	::=	EdgeId:String NextNodeId:String ＜ガード条件＞?
4	＜ DB アクセス＞	::=	＜ DBSelect ＞	＜ DBInsert ＞	＜ DBUpdate ＞	＜ DBDelete ＞
5	＜ DBSelect ＞	::=	FromTableId:String+ ＜ Where 句条件群＞
6	＜ DBInsert ＞	::=	IntoTableId:String ＜ Name:String Value ＞+
7	＜ DBUpdate ＞	::=	TableId:String ＜ Name:String Value ＞+ ＜ Where 句条件群＞
8	＜ DBDelete ＞	::=	FromTableId:String ＜ Where 句条件群＞
9	＜ガード条件＞	::=	＜入力値条件＞ — ＜結果件数条件＞
10	＜結果件数条件＞	::=	DBSelectCount ＜整数比較演算子＞ ResultCount:Integer
11	＜入力値条件＞	::=	省略
12	＜ Where 句条件群＞	::=	省略

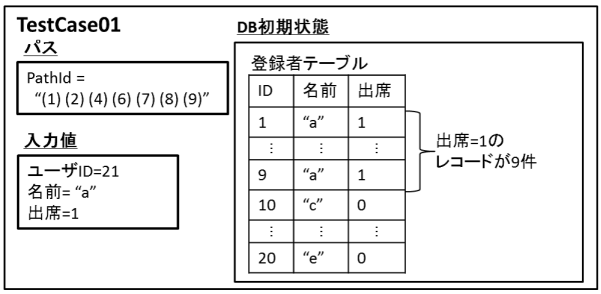

図 6 テストケース

ることで，DB 更新アクセスを伴うテストケースに対して，DB 初期状態と入力値のペアを生成する方式を用いる．生成するソースコードはあくまで DB 初期状態，入力値を生成するために最低限必要なシミュレーション用のコードであるため，エラー処理，通信処理等を含まず，ビジネスロジックの本質的な流れのみから成る．2.2 節で述べたとおり，Concolic Testing を用いると，変数の状態が刻々と更新されていく場合でも問題なく扱うことが可能となる．

本方式では，テストケースのパスに基づいて，システムの振る舞いをシミュレーションする関数 F を生成する．F は，F を構成するステートメントのうち最後尾にあるステートメントまでプログラムが実行されたとき $true$ を返し，それ以外の場合は $false$ を返す．F は，元の処理フローのうち，特定のテストケースのパスのみ基づき構成されたソースコードであるため，F に Concolic Testing を適用すると，元の処理フローの他の余計なパスへ探索が行われることなく，効率よく探索を行うことが可能である．関数 F の最後ステートメントまでプログラムが実行され，F が $true$ を返したとき，そのパスを通る DB 初期状態と入力値が得られる．

関数 F は，具体的には以下の手順に従って生成する．手順 5,6,7 はパス上のノード，エッジを順次解析しながら行っていく．

- 手順 1 : DB スキーマの情報に基づき DB スキーマが満たすべき制約 (主キー制約，外部キー制約) を確認する関数 $CheckDBConstraints$ を生成する．$CheckDBConstraints$ は 1 つでも制約違反があれば $false$ を返し，それ以外のときは $true$ を返す関数であり，関数 F 内部で使用するユーティリティ関数である．
- 手順 2 : 設計モデルにおける DB スキーマ，入力定義の情報を用いて，F の先頭に DB 初期状態，入力値を変数として宣言する．DB 初期状態の各テーブルは，レコードのリストとして宣言する．また，各テーブルにおけるレコード件数 (リストサイズ) の初期値はユーザが与えるものとする．これは，各テーブルのレコード件数を変えながら探索を行うと探索空間が大きくなってしまうので，あらかじめ絞り込んでおくためである．
- 手順 3 : 入力定義の情報に基づき入力値が満たすべき制約それぞれにつき，制約 c に対して $if(!c)returnfalse;$ というステートメントを F へ追加していき，これらのステートメントの最後まで到達したときには全ての入力値に対する制約を満たしているようにする．
- 手順 4 : DB スキーマの制約が崩れていないことを確認するため $if(!CheckDBConstraints())returnfalse;$ というステートメントを，DB 初期状態の変数宣言の後に追加する．
- 手順 5 : 入力値に対するガード条件 c があれば，$if(!c)returnfalse;$ というステートメントを F へ追加する．同様に参照結果のレコード件数に対するガード条件があれば，対応するステートメントを F へ追加する．これにより，パス上のガード条件を満たさない場合には即時関数が終了するため，対象外のパスの探索に時間をとられない．
- 手順 6 : DB に対する参照操作があった場合，レコードのリストを操作することにより DB 操作を擬似するステートメント群を F へ追加する．
- 手順 7 : DB に対する更新操作があった場合，レコードのリストを操作することにより DB 操作を擬似するステートメント群を F へ追加する．そ

して，そのコードの後に更新操作により DB スキーマの制約が崩れていないことを確認するため $if(!CheckDBConstraints())returnfalse;$ というステートメントも追加する．

具体的にどのようにソースコードの生成を行うかを，図 6 のテストケース (図 4 のノード (1)，(2)，(4)，(6)，(7)，(8)，(9) に対応) の DB 初期状態，入力値を生成する場合を例にとって説明する．以下のコードは，理解しやすさのため簡略化した擬似コードにしている．実際には，DB や，DB への操作をクラス化，メソッド化し，Concolic Testing を行う外部ライブラリが提供する API に従ったコードを生成しているが，本質的なコードのロジックは以下の擬似コードと同じである．

```
//ユーザが設定する各テーブルの初期レコード件数
#degine 登録者テーブル_SIZE=20;

//DB スキーマの制約 (主キー，外部キーの整合性など) を
//確認する関数 (手順 1 により生成)
boolean CheckDBConstraints(...){
    …中略…
}

boolean F(){
    //各 DB 初期状態，入力変数の宣言 (手順 2 により生成)
    int ユーザ ID = ...;
    …中略…
    List<登録者> 登録者 List =
        new List<登録者>(登録者テーブル_SIZE);

    //入力定義の制約確認 (手順 3 により生成)
    if(!(ユーザ ID >= 0)) return false;
    …中略…

    //DB スキーマ制約の確認 (手順 4 により生成)
    if(!CheckDBConstraints(登録者 List))
        return false;

    //(1) 処理開始

    //(2) ユーザが登録済みか? (手順 6 により生成)
    resultList = {};
    foreach(登録者レコード in 登録者 List){
        if(登録者レコード.ID == ユーザ ID)
            resultList.add(登録者レコード);
    }
    int dbSelectCount = resultList.count();

    //(2) の後のガード条件：未登録
    if(!(dbSelectCount == 0))return false;

    //(4) 出席か? (手順 5 により生成)
    if(!(出席==1))return false;
    …中略…

    //(6) 出席者として登録 (手順 7 により生成)
    newRecord = new 登録者 (ユーザ Id, 名前, 1)
    登録者リスト.add(newRecord);
    //更新後には DB スキーマの確認
    if(!CheckDBConstraints(登録者 List))
        return false;

    //(7) 出席者人数を確認
    …中略…

    //(7) の後のガード条件：人数が 10 人以上
    if(!(dbSelectCount >= 10))
        return false;

    //(8) 席が埋まりつつあるメッセージを表示

    //(9) 出席者，欠席者を表示

    //最後まで到達．DB 初期状態，入力値の生成が完了
    return true;
}
```

このように，登録者テーブルは登録者レコードのリスト，そしてユーザ ID は Integer 型といったように DB 初期状態，入力値は全て変数として表され，パス中のガード条件，DB アクセスなども，それぞれプログラムにおける条件分岐や，リストへの参照，更新として表現される．このソースコードに対して Concolic Testing を適用することで，最終的には図 6 で示したような DB 初期状態と入力値のペアを得ることができる．

4. 評価

本研究の評価指標はより多くのテストケースに対して適切な DB 初期状態を生成できるかである．そのため，事前状態として DB 初期状態を必要とするテストケースのうち，どれだけのテストケースに対して DB 初期状態を生成できたかを「DB 初期状態の生成率」として定め評価基準とする．

4.1 評価方法

提案手法及び既存手法 [24] を実装し，著者らが開発している結合テスト設計支援ツール TesMa [25] へ組み込み，評価に用いた．TesMa は開発現場で使用されている設計書に近いフォーマットを読み込み，それらを内部的に設計モデルへ変換し，そこからテストケースを生成することができるツールである．評価用のシステムとしては，以下の (a),(b) を用いた．(a) は実際に使用されているシステムであり，(b) に比べ DB アクセスのパターンが複雑である．

- (a) 業務システム : ネットワークオペレーションシステムの Web フロントエンドの一部 (31 画面，42DB テーブル)
- (b)JPS : Java Pet Store の一部 (3 画面, 2DB テーブル)

(a) については，実開発案件の設計書を，TesMa の入力となる設計書へ書き換えて，TesMa が読み込めるようにした．(b) については設計書が存在しないため，アプリケーションの動的解析や，ソースコードの情報を元に TesMa

表 2　DB 初期状態の生成率

	(a) 業務システム	(b)JPS
全テストケース	743	97
要 DB のテストケース	625	24
既存手法の DB 生成率	3% (21/625)	-
提案手法の DB 生成率	13% (83/625)	79% (19/24)

表 3　「(b) 業務システム」の DB 生成不可の原因

項番	原因	占める割合
1	LIMIT,OFFSET 句	56% (306/542)
2	EXISTS 句	52% (282/542)
3	動的 SQL による検索条件の指定	32% (172/542)
4	INNER JOIN 句以外の結合	26% (144/542)
5	全検索結果 0 件	6% (33/542)
6	集計関数 (MAX 等) の使用	6%(30/542)

形式の設計書を復元した．また，提案手法と既存手法の比較については，(a) についてのみ行った．Concolic Testing の外部ライブラリとしては，CATG [1] を用い，CATG が使用する SMT ソルバとしては Yices [4] を用いた．

4.2　評価結果

評価結果を表 2 に示す．(a) については，DB 初期状態生成率が既存手法では 3% であったのに対し，提案手法では 13% のテストケースに対して事前状態として適切な DB 初期状態を生成することができ，DB 更新アクセスを伴うテストケースを含めより多くのテストケースに対して DB 初期状態を生成できた．また，(b) のような単純なアプリケーションでは，DB 初期状態生成率は 79% と (a) よりも高い割合で生成できた．

提案手法でも DB 初期状態を生成することができなかった (a) のテストケース 542 件 (625 件-83 件) について，できなかった主な原因を表 3 に列挙する．表 3 の項番 1，2，4，6 は提案手法の設計モデルでは対応していない SQL クエリの表現が必要となったテストケースである．表 3 の項番 3 は Where 句を構成する検索条件が動的に変わるものである．例えば，社員を検索するときに，年齢の条件をオプションで指定できる場合などがこれに相当する．表 3 の項番 5 は，テーブルに対する全件検索（Where 句の条件が無い，無条件の検索）が行われ，その結果件数が 0 件となることが条件として必要なテストケースである．このテストケースでは，検索対象となるテーブルにレコード件数が 1 件でも入っていると，必ず検索でヒットしてしまい結果件数が 0 件という条件を満たせなくなってしまう．今回の実験では，各テーブルのレコード件数の初期値は 10 件ずつ，といったように 1 件以上のレコードをテーブルの初期値として与えていたため，このような場合には無条件検索で 1 件もヒットしないテーブルをもつ DB 初期状態を生成することができなかった．

4.3　考察

4.3.1　生成できなかった原因の分析と解決案

表 3 に挙げた，提案手法で DB 初期状態を生成することのできなかったテストケースについて，どのようにすれば，DB 初期状態を生成することが可能となるかを考察する．

表 3 の項番 1，2，4，6 については，現状の設計モデルでは記述することができないが，設計モデル上の記法が定まり，その記法からソースコードへの変換方式が確立すれば，解決することが可能である．基本的には，全ての SQL クエリで表現される宣言的なロジックは，Concolic Testing が適用可能となる手続的なロジックへと変換することが可能と考えられるので，これらへの対応は技術的には比較的容易であると考えられる．

表 3 の項番 3 については，現状でも設計モデルの作成の仕方を工夫し，Where 句の特定の条件が存在する場合と存在しない場合についてパスを分けて記述するなどすれば，ある程度の対応は可能である．ただし，Where 句を構成する条件のうち，いくつもがオプションである場合は，組み合わせによってパス数が爆発的に増えてしまう．いずれにせよ，項番 3 のケースではオプションの有無を考慮するとテストケースのバリエーションが増えてしまうため，何らかの絞り込みが必要になると考えられる．

項番 5 のケースは，レコード件数をあらかじめ固定し，探索空間を絞り込んでしまったために起きた問題である．これを解決するには，DB の各テーブルのレコード件数の初期値を 0 を含めてランダムにふらせて実行を繰り返すなどの方法 (この場合，探索空間は大きくなってしまう) や，パス上に参照アクセスしかない場合は既存手法を用いて解く方法などが考えられる．既存手法は，パス上のレコード件数についての制約も抽出し，それらを解き各テーブルのレコード件数を決めているため，パス上に更新アクセスがなければ対応することが可能である．

4.3.2　提案手法と既存手法で生成される DB 初期状態の特性

提案手法では，あらかじめ DB の各テーブルのレコード件数を決めてから，Concolic Testing を適用してテストケースの事前状態として適切な DB 初期状態のテーブルへと各レコードの値を調整していくため，必要最低限のレコードしか生成しない既存手法と比べると，生成した DB 初期状態の各テーブルは多くのレコードをもつ特性がある．例えば，参照アクセスで条件を満たすレコードが 1 件存在していればよい場合でも，例えば 10 件以上のレコード (1 件以外は全て条件に不一致となる) をもつテーブルが生成される．これは，テストの際に条件に一致するレコードがヒットするだけではなく，条件に一致しないレコードが誤って返ってこないことを確認するためには便利な特性であり，Where 句の条件に関する実装誤りを検出するというテスト観点を含む DB 初期状態を生成できるという見方

もできる．また，多くのレコードが存在する中で数件のみヒットするというのは，よりシステムの実運用の状況に近いと考えられる．

5. 結論

設計モデルに基づき，更新系アクセスを伴うテストケースに対しても，適切な DB 初期状態を生成する方式を提案した．提案手法では，設計モデルからパスを抽出し，そのパスを Concolic 実行可能なシミュレーション用ソースコードへ変換し，Concolic 実行することでテストデータ生成を行うことを特徴とする．実開発案件を含む 2 つのアプリケーションを用いて用いて評価を行い，既存手法 [24] と比較し，より多くのテストケースに対して DB 初期状態を生成できることを確認した．今後はより多くの案件を用いて評価を行い，提案手法の有効性検証を行っていきたい．

参考文献

[1] CATG. https://github.com/ksen007/janala2.
[2] DBMonster. http://dbmonster.sourceforge.net/.
[3] Visual Studio Database Edition. http://www.microsoft.com/japan/msdn/vstudio/.
[4] The Yices Smt Solver. http://yices.csl.sri.com/.
[5] 擬似個人情報ジェネレータ. http://www.start-ppd.jp/.
[6] David Chays, Saikat Dan, Phyllis G. Frankl, Filippos I. Vokolos, and Elaine J. Weber. A framework for testing database applications. In *Proceedings of the 2000 ACM SIGSOFT international symposium on Software testing and analysis*, ISSTA '00, pp. 147–157, New York, NY, USA, 2000. ACM.
[7] Leonardo de Moura, Bruno Dutertre, and Natarajan Shankar. A tutorial on satisfiability modulo theories. In Werner Damm and Holger Hermanns, editors, *Computer Aided Verification*, Vol. 4590 of *Lecture Notes in Computer Science*, pp. 20–36. Springer Berlin / Heidelberg, 2007. 10.1007/978-3-540-73368-3_5.
[8] Arilo C. Dias Neto, Rajesh Subramanyan, Marlon Vieira, and Guilherme H. Travassos. A survey on model-based testing approaches: A systematic review. pp. 31–36, 2007.
[9] Arilo C. Dias Neto, Rajesh Subramanyan, Marlon Vieira, and Guilherme H. Travassos. A survey on model-based testing approaches: a systematic review. In *Proceedings of the 1st ACM international workshop on Empirical assessment of software engineering languages and technologies: held in conjunction with the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE) 2007*, WEASELTech '07, pp. 31–36, New York, NY, USA, 2007. ACM.
[10] Jon Edvardsson. A survey on automatic test data generation. In *Proceedings of the Second Conference on Computer Science and Engineering in Linköping*, pp. 21–28. ECSEL, October 1999.
[11] Michael Emmi, Rupak Majumdar, and Koushik Sen. Dynamic test input generation for database applications. In *Proceedings of the 2007 international symposium on Software testing and analysis*, ISSTA '07, pp. 151–162, New York, NY, USA, 2007. ACM.
[12] Andre Takeshi Endo and Adenilso Simao. Model-based testing of service-oriented applications via state models. In *Proceedings of the 2011 IEEE International Conference on Services Computing*, SCC '11, pp. 432–439, Washington, DC, USA, 2011. IEEE Computer Society.
[13] Barry O'Sullivan Frederic Benhamou, Narendra Jussien. *Trends in Constraint Programming*. ISTE, 2010.
[14] Shoichiro Fujiwara, Kazuki Munakata, Yoshiharu Maeda, Asako Katayama, and Tadahiro Uehara. Test data generation for web application using a uml class diagram with ocl constraints. *Innovations in Systems and Software Engineering*, Vol. 7, pp. 275–282, 2011. 10.1007/s11334-011-0162-3.
[15] James C. King. Symbolic execution and program testing. *Commun. ACM*, Vol. 19, No. 7, pp. 385–394, July 1976.
[16] Kai Pan, Xintao Wu, and Tao Xie. Generating program inputs for database application testing. In *Automated Software Engineering (ASE), 2011 26th IEEE/ACM International Conference on*, pp. 73–82, Nov 2011.
[17] Kai Pan, Xintao Wu, and Tao Xie. Guided test generation for database applications via synthesized database interactions. *ACM Trans. Softw. Eng. Methodol.*, Vol. 23, No. 2, pp. 12:1–12:27, April 2014.
[18] Koushik Sen, Darko Marinov, and Gul Agha. Cute: a concolic unit testing engine for c. In *Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, ESEC/FSE-13, pp. 263–272, New York, NY, USA, 2005. ACM.
[19] David Willmor and Suzanne M. Embury. An intensional approach to the specification of test cases for database applications. In *Proceedings of the 28th international conference on Software engineering*, ICSE '06, pp. 102–111, New York, NY, USA, 2006. ACM.
[20] 情報処理推進機構ソフトウェア・エンジニアリング・センター. ソフトウェアテスト見積りガイドブック：品質要件に応じた見積りとは. SEC books. オーム社, 2008.
[21] 石川太一郎, 高田眞吾, 丹野治門, 生沼守英. Concolic testing を用いた web アプリケーションに対するテストデータ生成に関する研究. 情報処理学会研究報告ソフトウェア工学, Vol. 2014-SE-183, No. 1, pp. 1–8, mar 2014.
[22] 丹野治門, 星野隆, Koushik Sen, 高橋健司. Concolic testing を用いた結合テスト向けテストデータ生成手法の提案. 情報処理学会研究報告ソフトウェア工学, Vol. 2013-SE-182, No. 6, pp. 1–8, oct 2013.
[23] 丹野治門, 張暁晶, 星野隆. 結合テストにおけるテスト項目自動生成手法の提案と評価. 電子情報通信学会技術研究報告. SS, ソフトウェアサイエンス, Vol. 110, No. 227, pp. 37–42, oct 2010.
[24] 丹野治門, 張暁晶, 星野隆. 設計モデルを利用したテスト用データベース自動生成手法. 情報処理学会論文誌, Vol. 53, No. 2, pp. 566–577, feb 2012.
[25] 丹野治門, 張暁晶, 田端啓一, 生沼守英, 村主一仁. ソフトウェアの品質確保と開発コスト削減を目指したテスト自動化技術. NTT 技術ジャーナル, Vol. 25, No. 10, pp. 19–22, oct 2013.
[26] 藤原翔一朗, 宗像一樹, 片山朝子, 前田芳晴, 大木憲二, 上原忠弘, 山本里枝子. Smt solver を利用した web アプリケーション用テストデータの生成 (テスト・検証, 一般セッション, ソフトウェア科学・工学, 情報処理学会創立 50 周年記念). 全国大会講演論文集, Vol. 72, No. 1, pp. 1–281–282, mar 2010.