情報処理学会論文誌 Vol.58 No.4 818–832 (Apr. 2017)

テストケースの事前条件として適切なDB初期状態の状態数とデータサイズを削減する手法の提案

丹野 治門[1,a)]

受付日 2016年8月10日, 採録日 2017年1月10日

概要：本研究は，関係データベース（DB）を用いる業務システムの機能性に関する結合テストの一部を支援対象として，DB への参照を行う各テストケースに対し，事前条件として適切な DB 初期状態を自動生成する問題を扱う．既存手法では，それぞれのテストケースごとにテストケースの事前条件として適切な DB 初期状態を 1 つずつ自動生成しているため，2 つの問題点があった．1 点目は，DB 初期状態がテストケースごとに生成されているため，テスト実施時に多くのテストケースに対して，テストケースごとに DB 初期状態を入れ替える労力がかかる点である．2 点目は，多くの DB 初期状態が生成されると結果としてレコードの合計件数が多くなり，DB 初期状態全体のサイズが大きくなるため，テスト資材の版管理や，オフショア先とのデータ送受信といったデータ管理のコストが大きくなってしまう点である．本研究ではこのような問題点を解決するため，複数のテストケースで共有できる DB 初期状態を自動生成する手法を提案する．提案手法では，DB 初期状態の数と DB 初期状態全体のサイズをなるべく小さくするための技術課題として，テストケースグルーピング，レコード集合決定，DB 初期状態値制約生成の 3 つをあげ，それぞれの課題への解法を考案した．業務システム 3 件を用いて提案手法と解法の適用評価を行ったところ，既存手法に対して DB 初期状態数を 23%に削減，DB レコード件数を 64%に削減でき，提案手法の有効性を確認することができた．

キーワード：テストデータ，テストケース，データベース，モデルベーステスト

Reduce The Number And The Size of Initial Database States for Testing Applications

Haruto Tanno[1,a)]

Received: August 10, 2016, Accepted: January 10, 2017

Abstract: This research focuses on testing database applications, more concretely on how to automatically generate the initial database states which are appropriate preconditions of each test case. Existing approaches generate initial database states for each test case one-by-one; however there are 2 problems with the approaches. The first problem is that switching initial database states for each test case is too time consuming when we use the generated initial database states for testing. The second problem is that the total number of DB records tends to be large because many initial database states are generated. As a result, the total size of test data becomes large, which increases the cost of managing the test data and the time needed to switch initial database states for each test case. To solve these problems, we propose an approach for generating initial database states that are shared by multiple test cases by introducing reasonable solutions to 3 key challenges that are grouping test cases, deciding record arrangement, and solving constraints for appropriate an initial database states shared by multiple test cases. Using three industrial-level enterprise systems, we confirm that our proposal reduces the number of initial database states by 23%, and the total number of DB records by 64%.

Keywords: test data generation, test case generation, database

1. はじめに

関係 DB（以降，単に DB と呼ぶ）を用いたソフトウェアのテストを行う際には，確認したいソフトウェアの特定

[1] NTT 研究所
NTT Laboratories, Konan-ku, Tokyo 108–0075, Japan
[a)] tanno.haruto@lab.ntt.co.jp

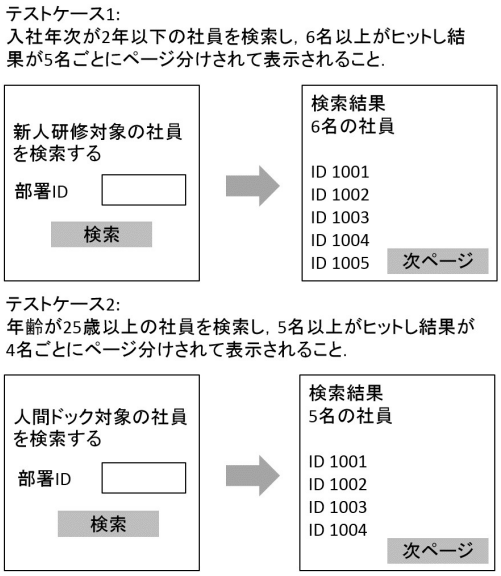

図 1 テストケースの例
Fig. 1 An example of test case.

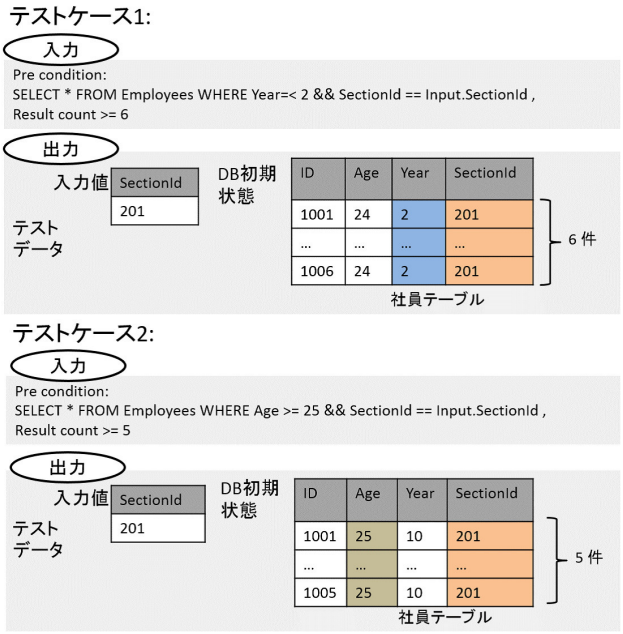

図 2 既存手法（DB 個別生成手法）で生成された DB 初期状態
Fig. 2 Initial database states generated By existing approach.

の振舞いを起こすために，テストケースごとにテストデータとして適切な入力値と DB 初期状態のペアを用意する必要がある．一般に，ソフトウェアのテストでは多くのテストケースを作成する必要があるため，それらのテストケースそれぞれに対して，1 つずつ人手で適切な DB 初期状態を用意する必要があり，これは作成者の負担が大きい．

たとえば，**図 1** に示すような社員管理システムのテストケースを作成する場合を考えてみる．図 1 では 2 つのテストケースが示されている．図 1 のテストケース 1 は，新人研修対象の社員を検索する機能のテストである．この機能はユーザが入力した部署において，入社年次が 2 年以下の社員を検索するという仕様を持つ．このテストでは，検索結果として 6 名以上がヒットし，複数のページ（1 ページにつき 5 名）に分かれて社員が表示されることを確認する．図 1 のテストケース 2 は，人間ドック対象社員を検索する機能のテストである．この機能はユーザが入力した部署において，年齢が 25 歳以上の社員を検索するという仕様を持つ．このテストでは，検索結果として 5 名以上がヒットし，複数のページ（1 ページにつき 4 名）に分かれて社員が表示されることを確認する．図 1 のテストケース 1，2 では，それぞれのテストケースごとで，確認したい振舞いを起こすための適切な DB 初期状態，入力値のペアを作成する必要がある．たとえば，図 1 のテストケース 1 では，入力値として部署 ID が必要であり，DB 初期状態としては，複数ページが表示されることを確認するために，条件（入力値の部署 ID と同じ部署 ID を持ち，入社年次が 2 年以下である）を満たす社員のレコードが 6 件以上ヒットするよう，レコードを保持している必要がある．また，ヒット件数が 5 件から 6 件になったときに複数ページ表示に切り替わる点を考慮すると，6 件のレコードがヒットし表示されるよう DB 初期状態がレコードを保持していることが望ましい．多くのテストケースに対し，このような DB 初期状態と入力値のペアをそれぞれ人手で作成するというのは大きな負担である．

人手によるテストデータ作成の負担を軽減するため，テストデータを自動で生成する手法 [4], [9] が多く提案されてきた．図 1 で示したような，業務テストの機能テスト向けの DB 初期状態，入力値のペアを自動生成する既存手法の 1 つとして，モデルベーステスト [8] に基づいた，筆者らが過去に提案した手法 [18] や藤原らの手法 [11], [20] がある．モデルベーステストとは，なんらかのモデルに基づき，そのモデルからテストケースや，テストデータを自動で生成する手法である．筆者が過去に提案した手法，藤原らの手法では，モデル化したソフトウェア設計情報，テスト設計情報などから，テストケースごとにテストデータ（DB 初期状態，入力値）がそのテストケースの事前条件として満たすべき制約を抽出し，それらの制約をもとに DB 初期状態，入力値のペアを自動で生成する．以降，この DB 初期状態生成の手法を *DB* **個別生成手法**と呼ぶことにする．**図 2** は，図 1 で示した社員管理システムのテストケース，テストデータをこれらの手法で自動生成した例である．図 2 に示すように，DB 個別生成手法では，各テストケースごとに，そのテストケースの事前条件となる制約を満たすよう，1 つのテストケースにつき，1 つの DB 初期状態を生成している．これらの手法は，人手によるテストデータ作成のコストをなくせるという点でとても有用であるが，テストデータを作成した後の，テスト実施，データ管理までを考えると以下のような問題点がある．

(1) テスト実施における DB 初期状態の入れ替え回数の

問題：テストケースごとにそのテストケースの事前条件となる DB 初期状態が生成されているため，DB 初期状態をテストケースごとに入れ替える必要がある．図 1 に示したような業務システムの結合テストでは，たとえば画面の操作であれば操作手順をスクリプト化するなどそれぞれのテストケースの実施手順の一部を自動化できる場合もあるが，自動で実施することが難しい手順が含まれることがあったり，対応するテスト自動実行ツールがなかったりする場合，画面への操作がすべて手動で行われることもある．このとき，テスト実施では，それぞれのテストケースごとに「テストケースの事前条件の DB 初期状態を DB へ投入」，「手動で 1 番目のテストケースを操作して実行し結果を確認」という 2 ステップを行う必要があるため，作業ステップが多く手間がかかり，DB 初期状態を入れ替え忘れて手順誤りが生じ，手戻りが発生する可能性も高まる．たとえば，100KL のシステムならば数千件のテストケースを実施する必要があり [21]，このような毎回の DB 初期状態の交換は大きな負担となる．仮に画面操作を含めテスト実施が全自動で行える場合は，ステップ数が多くてもすべてのステップが自動で行われるため，上述したことは問題にならないが，DB 初期状態を入れ替えるときにマシン時間がかかるため，テスト実行時間が長くなってしまうという問題もある．これは個々の DB 初期状態のサイズが大きい場合には特に顕著である．

(2) データ管理における DB 初期状態のサイズの問題：テストケースごとに DB 初期状態を用意しているので，結果として全体の合計レコード件数が多くなり，データサイズが増えるので，データ管理のコストが大きくなってしまう．近年のソフトウェア開発では，頻繁に行われるリリースにともなうテスト資材の版管理や，テスト実施を委託したオフショア先との頻繁なデータ送受信などが必要になることも多い．DB 初期状態のデータサイズが大きくなることは，このようなデータ管理のコストの増大につながる．たとえば，図 2 を見ると，2 つのテストケースで合計 11 件レコードを用意しているがこれは冗長であり，DB 初期状態を複数のテストケースで共有することができれば合計 6 件で済む可能性がある．

本研究では，上述した 2 つの問題を解決するため，図 2 のように，1 つのテストケースにつき 1 つの DB 初期状態を生成するのではなく，**図 3** のように，複数のテストケースで共有することが可能な DB 初期状態の生成手法を提案する．提案手法は，既存の DB 個別生成手法に比べ，DB 初期状態の数を少なくすることで DB 初期状態を入れ替える手間を減らし，DB レコード件数を少なく抑えることで DB 初期状態のサイズを小さくすることが可能である．

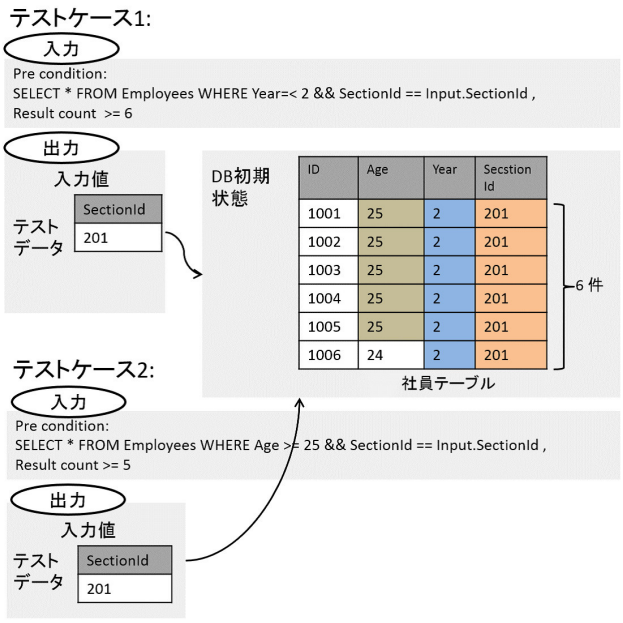

図 3 提案手法で生成可能な DB 初期状態
Fig. 3 Initial database states generated by proposed approach.

これにより，テストデータ自動生成手法を開発現場へ導入したときのテスト実施，データ管理の負担を軽減し，テスト工程全体の効率化へ寄与できると考える．本研究ではテスト実施，テストデータ管理を効率化するための先行指標としてテストケース数に対する DB 初期状態の数（***DB 初期状態数***）と全 DB 初期状態の合計レコード件数（***DB レコード件数の合計***）の 2 点を考え，既存の DB 個別生成手法に対してこの指標を改善することを目標とする．本研究では，図 1 で示したような，ユーザからの入力を画面で受け付け，その入力に基づき DB へのアクセスを行い，実行結果を画面へ表示するテスト，すなわちプレゼンテーション層，アプリケーション層，データ層を結合させたときの各機能の動作を確認するためのテストを支援対象とする．本論文では，各テストケースで用いられるテストデータは DB 初期状態，入力値のペアと考える．また，DB アクセスを以下の 2 パターンに分類して定義する．

- 参照系アクセス：SELECT のように，DB の状態を参照のみ行い，書き換えは行わないアクセスを指す．
- 更新系アクセス：INSER，DELETE，UPDATE のように，DB の状態の書き換えをともなうアクセスを指す．

本研究では様々な種類のシステムで用いられる頻度が高く [6]，自動化の効果が顕著であるため，参照系アクセスのあるテストケースを対象とし，更新系アクセスについては扱わない．

本研究の貢献点は以下の 2 点である．

(1) 複数のテストケースで共有することが可能な DB 初期状態を生成する手法を提案した．本手法では，複数のテストケースの事前条件として適切な DB 初期状態の

具体値を，なるべく少ないDB初期状態数で，かつなるべく少ないレコード件数で生成できるよう工夫を行っている．具体的には，同一のDB初期状態を共有するテストケースのグルーピング方法，複数のテストケースの事前条件として適切なDB初期状態のレコード集合の決定方法，そして，そのDB初期状態値が満たすべき制約の抽出方法を考案した．

(2) 上述した提案手法を用いて，実開発案件3件を用いた適用評価を行い提案手法の有効性を確認した．提案手法は既存のDB個別生成手法に比べて，DB初期状態数を23%に削減，DBレコード件数の合計を64%に削減できている．

提案手法は既存のDB個別生成手法を拡張したものとなっている．以降，2章で，筆者らが過去に提案した手法を例にとり，DB個別生成手法について説明する．3章で，本研究で提案する複数のテストケースで共有することが可能なDB初期状態を生成するための手法を説明する．4章で，実開発案件を用いた適用評価とその結果について述べる．5章ではDB初期状態生成を扱っている関連研究について紹介する．最後に，6章で本論文の結論を述べる．

2. DB個別生成手法

本章では，本研究で提案する手法のベースとなっている，各テストケースに対応するDB初期状態を個別に生成するDB個別生成手法について，筆者らの過去の手法[18]を例にとり説明する．

この手法では，モデルベーステストの考え方を用い，設計工程で作成される設計ドキュメント相当の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っている．**図5**にこの手法の全体像を示す．ビジネスロジックの分岐経路を網羅的にテストするため，処理フローから，Decision/Condition Coverageを満たす経路を経路抽出技術[17]を用いて抽出し，経路ごとにテストケースを生成し，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，この手法で扱っている設計モデル，テストケースについて例を用いながら簡単に説明する．

入力となる設計モデルは，**処理フロー**，**入力定義**，に加えて**DBスキーマ**を記述することができ，DB参照を行う業務システムのビジネスロジックを記述することが可能である．既存手法において，処理フロー，入力定義，DBスキーマ定義として必須となる情報の定義を**表1**に示す．また，表1におけるガード条件やDB検索条件のWHERE句条件群に記述できる制約を**表2**に示す．DB参照を行うときの検索条件には整数比較や四則演算などを用いた多様な条件が記述できる．入力定義は処理フローで用いるユーザからの入力，設定ファイルやシステムから得る入力とその定義域を記述できる．DBスキーマでは，主キー制約を記述可能である．具体的な例として，図1のテストケース1で紹介した社内新入研修対象社員を検索する機能の設計モデルを**図4**に示す．図4(a)の処理フローでは，入力値の部署IDに対して入力範囲のチェック（図4(a)-(2)）を

表1 設計モデルの定義
Table 1 Definition of the design model.

項番	式		
	DBスキーマ		
1	⟨DBスキーマ⟩	::=	⟨テーブル定義⟩*
2	⟨テーブル定義⟩	::=	TableId:String ⟨カラム⟩+
3	⟨カラム⟩	::=	ColumnId:String ⟨型と定義域⟩⟨カラムの種類⟩
4	⟨型と定義域⟩	::=	IntegerType ⟨整数定義域⟩ | StringType ⟨文字列定義域⟩
5	⟨整数定義域⟩	::=	MinValue:Integer MaxValue:Integer
6	⟨文字列定義域⟩	::=	MinLength:Integer MaxLength:Integer
7	⟨カラムの種類⟩	::=	Normal | PK
	処理フロー		
8	⟨処理フロー⟩	::=	⟨ノード⟩+⟨エッジ⟩*
9	⟨ノード⟩	::=	NodeId:String Text:String NextEdgeId:String*
10	⟨エッジ⟩	::=	EdgeId:String NextNodeId:String (⟨ガード条件⟩? | ⟨DB検索条件⟩?)
11	⟨DB検索条件⟩	::=	FromTableId:String ⟨Where句条件群⟩⟨結果件数条件⟩
12	⟨Where句条件群⟩	::=	⟨And制約群⟩
13	⟨結果件数条件⟩	::=	ResultCount ⟨整数比較演算子⟩ Count:Integer
14	⟨ガード条件⟩	::=	⟨And制約群⟩
15	⟨And制約群⟩	::=	⟨制約⟩+
	入力定義		
16	⟨入力定義⟩	::=	⟨入力値⟩+
17	⟨入力値⟩	::=	⟨InputValueId:String⟩⟨型と定義域⟩

表 2 Where 句条件とガード条件で記述できる制約の定義
Table 2 Definition of the constraints used in the WHERE clauses and the guard conditions.

項番	式		
1	〈 制約 〉	::=	〈 整数制約 〉 | 〈 文字列制約 〉
2	〈 整数制約 〉	::=	〈 整数算術式 〉 〈 整数比較演算子 〉 〈 整数算術式 〉
3	〈 整数算術式 〉	::=	〈 整数項 〉 | 〈 整数算術式 〉 "+" 〈 整数項 〉 | 〈 整数算術式 〉 "-" 〈 整数項 〉
4	〈 整数項 〉	::=	〈 整数因子 〉 | 〈 整数項 〉 * 〈 整数因子 〉 | 〈 整数項 〉 / 〈 整数因子 〉
5	〈 整数比較演算子 〉	::=	">" | ">=" | "<" | "<=" | "==" | "! ="
6	〈 整数因子 〉	::=	ConstantValue:Integer | 〈 入力変数参照 〉 | 〈 カラム参照 〉 | "(" 〈 整数算術式 〉 ")"
7	〈 文字列制約 〉	::=	〈 文字列因子 〉 〈 文字列比較演算子 〉 〈 文字列因子 〉
8	〈 文字列比較演算子 〉	::=	EqualsString | NotEqualsString
9	〈 文字列因子 〉	::=	ConstantValue:String | 〈 入力変数参照 〉 | 〈 カラム参照 〉
10	〈 入力変数参照 〉	::=	VariableId:String
11	〈 カラム参照 〉	::=	TableId:String ColumnId:String

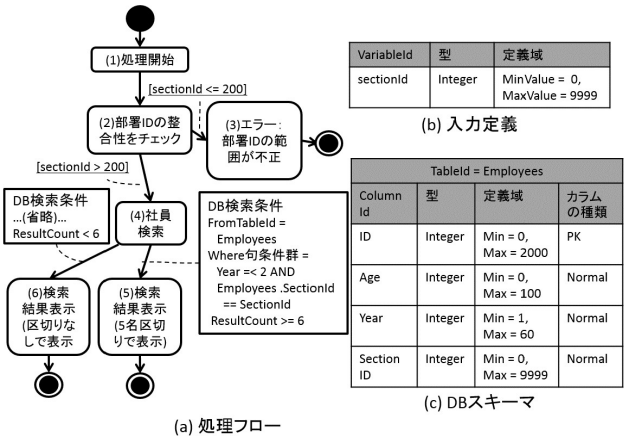

図 4 設計モデルの例
Fig. 4 An example of design model.

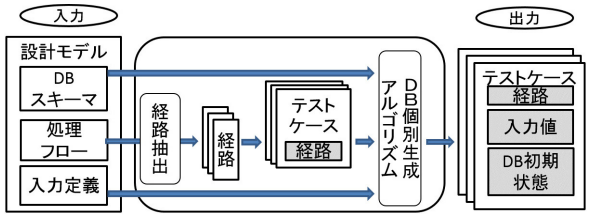

図 5 既存の DB 個別生成手法
Fig. 5 Overview of existing approach.

行った後に，入社年次が 2 年以下の社員検索（図 4 (a)-(4)）を行うようになっている．そして，この機能では 6 名以上検索にヒットした場合に 5 名区切りで検索結果の表示を行うため，`ResultCount >= 6`（図 4 (a)-(4) の DB 検索条件）という条件で，図 4 (a)-(5) へ遷移するよう記述されている．また，図 4 (c) の DB スキーマでは，社員情報が格納された Employees テーブルが定義されている．ID カラムは主キー制約がついている．

自動生成されるテストケースは経路，DB 初期状態，入力値の 3 者で構成される．DB 初期状態は，DB スキーマを満たし，**表 3** に示す方法で，DB 検索条件（条件には

表 3 結果件数条件からレコード件数を算出
Table 3 Rules for calculating the number of records.

項番	結果件数条件	レコード件数
1	$ResultCount >= C$	C
2	$ResultCount > C$	C+1
3	$ResultCount <= C$	C
4	$ResultCount < C$	C-1
5	$ResultCount == C$	C
6	$ResultCount! = C$（$C = 0$ の場合）	1
7	$ResultCount! = C$（$C \geq 1$ の場合）	C-1

複数の入力値を使用する場合もある）の結果件数条件を満たすように，存在すべきレコード件数を決定している．そして，すべての入力値は，経路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．以上により，生成された DB 初期状態と入力値の組は，テストケースの事前状態として適切なテストデータになっている．図 4 (a) において，検索結果表示画面へ遷移する経路（図 4 (a) における (1)，(2)，(4)，(5) の経路）に対応するテストケースが，図 2 のテストケース (1) である．

設計モデルの情報から，このようなテストケースの事前条件として適切な DB 初期状態，入力値のペアをどのように生成するかを説明する．この手法では，DB 初期状態，入力値の両方を変数と見なし，これらの変数がテストケースにおける経路を活性化し，かつ DB スキーマや入力値定義を満たすよう，制約を生成し，それらの制約を制約充足問題として既存制約ソルバで解くことで，これらの解，すなわちテストケースの事前条件として適切な DB 初期状態と入力値のペアを求める．DB 初期状態に関しては，DB 検索条件，DB スキーマにおける各 DB カラムの定義域，カラムの種類から抽出した制約を満たす必要がある．入力値に関しては，経路中のガード条件，そして，入力値定義における定義域から抽出した制約を満たす必要がある．

図 2 のテストケース 1 を例にとると，この例では Em-

ployees テーブルに 6 件のレコードが必要であるから，DB 初期状態，入力値に関する変数は以下のようになる（以下で E[i] は Employees テーブルの i 番目のレコードを表す）．

```
--変数群--
DB 初期状態
E[0].Id, E[0].Age, E[0].Year,E[0].SectionId,
... 省略...
E[5].Id, E[5].Age, E[5].Year,E[5].SectionId
入力値
sectionId
----
```

テストケース (1) の事前条件として適切な DB 初期状態，入力値を生成するため，設計モデルから以下のような制約を抽出する．

```
--制約群--
DB 検索条件から抽出した制約
E[0].Year <= 2, E[0].SectionId = SectionId,
... 省略...
E[5].Year <= 2, E[5].SectionId = SectionId

DB スキーマにおける各定義域から抽出した制約
distinct (E[0].Id, E[1].Id, ... 省略... , E[5].Id),
E[0].Id >= 0, E[0].Id <=2000,
E[1].Id >= 0, E[1].Id <=2000,
... 省略...
E[4].SectionId >= 0, E[4].SectionId <= 9999,
E[5].SectionId >= 0, E[5].SectionId <= 9999

経路中のガード条件から抽出した制約
sectionId >= 200

入力値定義における各定義域から抽出した制約
sectionId >= 0, sectionId <= 9999
----
```

distinct は一意性を保証するための制約であり，たいていの制約ソルバでは扱うことが可能である．上述したような変数，制約を制約ソルバに与えると，図 2 のテストケース (1) に示すような，DB 初期状態値，入力値の具体的な値を解として得ることができる．

このように，DB 個別生成手法はテストケース 1 つ 1 つに対して事前条件として適切な DB 初期状態が満たすべき制約を抽出し，その制約を解くことで DB 初期状態の具体値を生成している．そのため，1 つのテストケースにつき 1 つの DB 初期状態が必要となり，DB 初期状態数や合計のレコード件数が多くなってしまう．

3. 提案する DB 初期状態生成手法

本研究では，既存の個別 DB 生成手法の問題を解決し，1 つのテストケースにつき 1 つの DB 初期状態を生成するのではなく，複数のテストケースで共有することが可能な DB 初期状態生成手法の確立を目指す．提案手法は既存手法に比べて，DB 初期状態数の数を削減することでテスト実施時のテストケースごとの DB 初期状態の入れ替え回数を少なくし，かつ合計レコード件数を削減することで DB 初期状態のサイズを小さくすることが可能である．

本章では，まず問題点を解決するために解くべき技術課題を整理する．そして次に，提案手法とそれぞれの技術課題の解法について説明する．

3.1 技術課題の整理

本研究では，複数のテストケースで共有することが可能な DB 初期状態を生成するうえでの技術課題を以下のように整理した．本研究では，複数のテストケースで共有することが可能な DB 初期状態を生成するうえでの技術課題を 3 つに整理した．技術課題 1，2，3 で段階的に DB 初期状態を作成しているため，以降の説明では作成途中の DB 初期状態のことを**作成途中 *DB* 状態**と呼び，最終的に生成する DB 初期状態と区別して扱う．

- 技術課題 (1)：同じ DB 初期状態を共有するテストケースのグループ分けをどのように行うか．
- 技術課題 (2)：複数のテストケースから参照アクセスが行われる DB テーブルのレコード集合，レコード件数をどのように決めるか．
- 技術課題 (3)：複数の参照アクセスに対して期待した適切なレコード件数がヒットし，DB スキーマを満たす DB 初期状態をいかにつくるか．

以下，技術課題について詳しく説明する．

技術課題 (1)：まず第 1 に，同じ作成途中 DB 状態を共有するテストケースのグループをどのように決めるかを検討する必要がある．すべてのテストケースで共有する DB 初期状態を 1 つ生成するのが理想的ではあるが，テストケースの数が多い場合，このような DB 初期状態が必ず存在するとは限らないためである．このグルーピングの仕方の総数 C は，テストケースの総数を N 個とすると，$C = \sum_{i=1}^{N} S(N,i)$ で表すことができる．ここで，S は第 2 種スターリング数 $S(n,k)$ である．テストケース数 N が増えるとそのグルーピングの仕方の総数 C は指数関数的に増えていく．そのため，総当りで試すことは現実的ではなく，選び方には工夫が必要である．

技術課題 (2)：同じ作成途中 DB 状態を共有するテストケースのグルーピングが終わった後は，第 2 の課題として，同じグループの複数のテストケースで共有する作成途中 DB 状態における該当テーブルのレコード件数，レコード集合を検討する必要がある．たとえば，図 3 では，2 つのテストケースで共有することが可能な DB 初期状態を生成しているが，この DB 初期状態において，レコード件数，レコード集合のパターンはほかにも存在する．たとえば，計 11 件のレコードを生成し，最初の 6 件をテストケース 1 の DB 検索条件にヒットするレコードとして利用し，残り 5 件をテストケース 2 の DB 検索条件にヒットするレコードとして利用するという配置の方法もある．テストケースにおける DB 検索条件の組合せ次第で，解が存在しないレ

コード集合も存在するため，レコード集合の選び方にも工夫が必要である．

技術課題 (3)：レコード集合まで決まったら，第 3 の課題として，同じ作成途中 DB 状態を共有するそれぞれのテストケースで用いられている DB 検索条件それぞれに対して，期待する件数のレコードがヒットするようにかに DB 初期状態として満たすべき制約を生成するかの検討が必要となる．単純に，DB 個別生成手法でテストケースごとに個別に生成した DB 初期状態におけるレコードを組み合わせて，それらのテストケースの間で共有する DB 初期状態を作ろうとしても，DB スキーマの制約を満たさなくなってしまったり，それぞれの DB 検索に対して，適切な件数のレコードがヒットしなくなってしまったりといった問題が発生する．たとえば，**図 6** のようにテストケース 1，2 それぞれの事前条件として生成した DB レコードを単純に直列につなぎ合わせると，主キーが重複してしまったり (図 6 (a))，テストケース 1 の DB 検索条件（年次が 2 年以下の社員を検索）で 6 件ヒットさせるはずが，11 件ヒットするようになってしまったりといった問題が発生する．そのため，それぞれのテストケースの DB 検索条件に対して適切な件数のレコードがヒットし，かつ DB スキーマ制約も満たすようにするためには，DB 初期状態が満たすべき制約を生成するときに工夫が必要である．

3.2 提案手法

前節で述べた 3 つの技術課題を解決し，複数のテストケースで共有する DB 初期状態を生成するため，**図 7** のような手順で DB 初期状態を生成する手法を提案する．提案手法は表 1，表 2 で定義される設計モデルを必須の入力とし，テストケースグループの集合を出力する．テストケースグループは入力値のみを保持する複数のテストケースと，それらのテストケースで共有する DB 初期状態 1 つから構成される．以下，図 7 の各機能部が何をしているか簡単に説明する．

(1) テストケース抽出部：設計モデル（処理フロー，入力値定義，DB スキーマ）からテストケースを抽出する．

(2) テストケースグルーピング部：抽出したテストケースを，同じ作成途中 DB 状態を共有するテストケースごとにグルーピングし，いくつかのテストケースグループに分ける．

(3) 入力値制約および DB スキーマ制約生成部：入力値定義，経路上のガード条件からテストケースの入力値が満たすべき制約を抽出する．また，DB スキーマから DB 初期状態が満たすべき DB スキーマ制約も抽出する．

(4) レコード集合決定部：作成途中 DB 状態のレコード件数，レコード集合を決定する．

(5) DB 初期状態値制約生成部：DB スキーマ，各テストケースにおける DB 検索条件から，複数のテストケース間で共有される作成途中 DB 状態が，どのテストケースに対しても適切なテストの事前条件となるような DB 初期状態として満たすべき制約を生成する．

(6) 制約解法部：(5) DB 初期状態値制約生成部で生成した制約を，制約ソルバを用いて解き，DB 初期状態，入力値の具体値を得る．

(7) テストケース，テストデータ出力部：生成したテストデータ（DB 初期状態，入力値）をそれぞれテストケースグループ，テストケースへ紐付け，出力する．

機能部 (1)，(3)，(6)，(7) については，2 章で述べた DB 個別生成手法をそのまま使って解くことが可能である．機能部 (2)，(4)，(5) についてはそれぞれ前節で述べた技術課題 (1)，(2)，(3) にそれぞれ対応しており，本研究

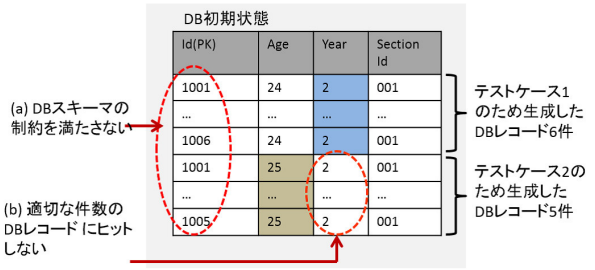

図 6 DB 個別生成手法の生成結果を直列につなげたときの問題点
Fig. 6 Problems when naively combining records.

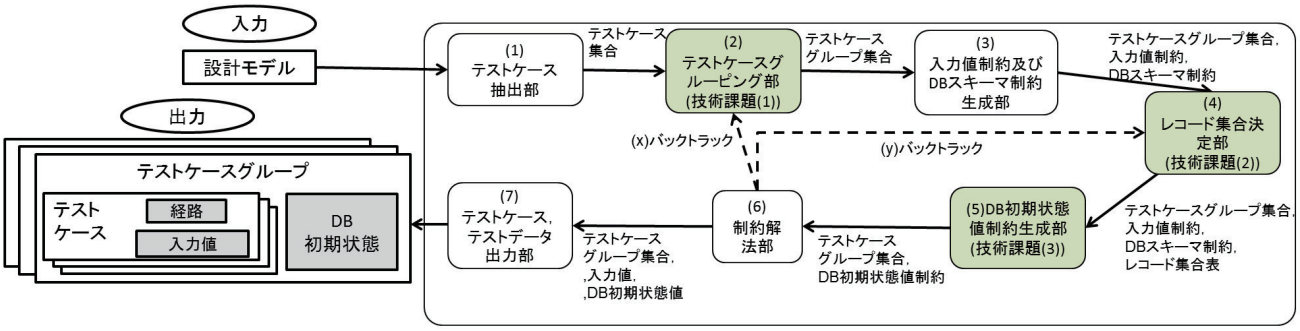

図 7 提案手法の全体像
Fig. 7 Overview of proposed approach.

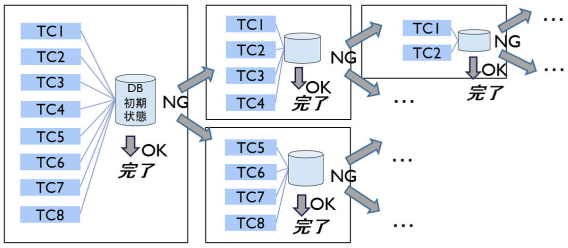

図 8 テストケースグルーピングの方式
Fig. 8 Grouping test cases.

ではこれらに解法を考案した．以降の節では，この 3 つの機能部 (2)，(4)，(5) について，技術課題の解決方法を詳細に述べていく．

3.3 各技術課題の解法

3.3.1 (2) テストケースグルーピング部（技術課題 (1)）

3.1 節で述べたとおり，テストケースグルーピングの組合せは無数に存在する．なるべく DB 初期状態数を減らしかつ必ずすべてのテストケースについてそれぞれの事前条件として適切な DB 初期状態を生成したいが，これらの組合せをすべてを試行すると，テストケースの数が多い場合に時間がかかってしまう．

そこで，コストのかかる理想解の探索ではなく，局所解であっても実用上有効なグルーピングを得られるよう，最悪の場合でも試行回数がそれほど多くならず，かつ確実にすべてのテストケースの事前条件として適切な DB 初期状態を得ることを方針とした．

この方針に基づき，本論文では**図 8** に示すように，まず最初にすべてのテストケースで共有することが可能な DB 初期状態の生成を試み，(6) 制約解法部において解を得られなかった場合にはバックトラック（図 7 の (x)）を行い，解となる DB 初期状態値が存在しないテストケースグループを半分ずつのグループに分け，再度，それぞれのテストケースのグループで共有することが可能な DB 初期状態の生成を試み，この手続きを再帰的に繰り返していくという方式を採用した．解が得られないのは，同じグループに存在するテストケースの事前条件となる制約どうしが互いに矛盾してしまうことが原因であり，テストケースグループを半分に分割していくことで，互いに矛盾する制約を持つテストケースどうしが別々のグループになり，分割が進むにつれて矛盾が解消され解が得られやすくなっていく．また，この方式の計算量は，テストケース数 N に対して，あるテストケースグループに対して，テストケースグループに属するテストケース間で共有できる DB 初期状態を得ようと試行する回数は，最悪の場合でも $\mathcal{O}(N \log N)$ 回となるため爆発的に増えることはない．また，複数のテストケースで共有する DB 初期状態が生成できなかった場合でも，最終的には DB 個別生成手法と同様に 1 つのテストケースの事前条件として適切な DB 初期状態を 1 つ必ず生成するので，確実にテストで必要となる DB 初期状態をすべて得ることができる．

この方式では，与えられたテストケースの並び順に従い，前半と後半で半分ずつのグループに分ける．そのため，少ないバックトラックで解が得られるかはテストケースの並び順に依存する．この点については，4 章で考察する．

3.3.2 (4) レコード集合決定部（技術課題 (2)）

レコード集合決定部では，作成途中 DB 状態のレコード件数，およびレコード集合を決定し，レコード集合表を作成する．

レコード集合を決める前に，作成途中 DB 状態へ参照アクセスを行う N 個のテストケース $TestCase_i$ $(1 \leq i \leq N)$ の各 DB 検索条件の結果件数条件を満たしかつ境界値となるようレコード件数 R_i $(1 \leq i \leq N)$ を，それぞれ表 3 の定義に従って算出する．たとえば，図 1 のテストケース (1)（図 4 (a) の (1)，(2)，(4)，(5) の経路に相当し，ノード (4) と (5) をつなぐエッジにおける DB 検索条件から結果件数条件の情報を取得できる）の結果件数条件は $ResultCount >= 6$ であるから，これは表 3 の項番 1 に相当し，レコード件数は 6 件となる．レコード集合の決定については，テストケースグルーピング部における課題と同じく，局所解であっても実用上有効なレコード集合を得られることを方針とした．本論文では生成する作成途中 DB 状態のレコード集合について，**理想パターン**，**妥協パターン**という 2 つのパターンを順に試す．

- 理想パターン：レコード件数が最小になるようなレコード集合である．最終的に生成される DB 初期状態のレコード件数が最小になるというメリットがあるが，複数の DB 検索条件から抽出した制約が重なりあいやすいため，最終的に制約が解けなくなる可能性もある．N 個のテストケース $TestCase_i$ $(1 \leq i \leq N)$ の各 DB 検索条件が，それぞれ R_i $(1 \leq i \leq N)$ 件のレコードを取得することを期待する場合，理想パターンのレコード件数は $Max(R_1, R_2, .., R_{N-1}, R_N)$ 件となる．レコード集合表では，レコードの j 件目に $TestCase_i$ $(1 \leq i \leq N)$ それぞれの j 件目のレコードが対応するように配置を行う．たとえば，図 1 におけるテストケース 1，2 で共有する DB 初期状態に最終的になるような作成途中 DB 状態における社員テーブルのレコード集合を考えると，理想パターンでは，**図 9** (a) のようなレコード集合表を作成する．このとき，最終的に生成する DB 初期状態は図 3 のようになる．
- 妥協パターン：レコード件数が最大になるようなレコード集合である．複数の DB 検索条件から抽出した制約が重ならないため，理想パターンで失敗し (6) 制約解法部で解が得られなかった場合でも，バックトラック（図 7 の (y)）して妥協パターンで試せば制約

(a)理想パターン

社員テーブル

Index	参照テストケース
0	テストケース1,2
...	...
4	テストケース1,2
5	テストケース1

(b)妥協パターン

社員テーブル

Index	参照テストケース
0	テストケース1
...	...
5	テストケース1
6	テストケース2
...	...
10	テストケース2

図 9 理想パターン，妥協パターンにおけるレコード集合表の例
Fig. 9 Examples of record alignment tables.

が解ける可能性があり，最終的に生成される DB 初期状態のレコード件数を減らすことはできないが，多くのテストケースで共有可能な DB 初期状態をつくれる可能性がある．妥協パターンのレコード件数は $R_1 + R_2 + ... + R_{n-1} + R_n$ 件となる．たとえば，図 1 における，テストケース 1，2 で共有する DB 初期状態に最終的になるような作成途中 DB 状態における社員テーブルのレコード集合を考えると，妥協パターンでは，図 9 (b) のようなレコード集合表を作成する．

妥協パターンは，理想パターンで解けない場合に制約どうしが重なりあわないようにすることで制約解法を行いやすくすることが狙いであるが，各 DB 検索条件における制約や DB レコード件数の組合せによっては，妥協パターンで解けないが理想パターンで解けるというケースもありうる点を注記しておく．たとえば，「A を満たすレコードが 5 件出てくる」という制約と「A かつ B を満たすレコードが 2 件出てくる」というように，2 つの検索条件が包含関係を満たす場合，妥協パターンのレコード集合では条件を充足させることができず，理想パターンでのみ解が存在する．

3.3.3 (5) DB 初期状態値制約生成部（技術課題 (3)）

個別のテストケース用に生成した DB レコードを単純に直列につなぐと，前述したように図 6 のような問題が発生する．そこで，同一テーブルにアクセスするすべての DB 検索条件に対して過不足なく適切な件数のレコードがヒットし，かつ DB スキーマを満たすような制約を生成し，それら DB 初期状態に関する制約を (6) 制約解法部で同時に解けるようにすることで，作成途中 DB 状態のレコード集合における各値を求め，複数のテストケースの事前状態として適切な DB 初期状態の値を得る．以下の説明では，フィールド変数とは 2 章で説明した `E[1].Id` のような作成途中 DB 状態のレコードにおける各フィールドに対応する変数を指すものとする．

DB 初期状態値制約生成部では，DB の各テーブルごとに複数のテストケースの事前条件として適切な DB 初期状態が満たすべき制約の集合である DB 初期状態値制約を生成する．あるテーブル $TableA$ に対する DB 初期状態値制約生成部のアルゴリズムの入力，出力はそれぞれ以下のものである．

- 入力：$TableA$ を参照する N 個のテストケース $T_1, ..., T_N$，図 7 (3) 入力値制約および DB スキーマ制約生成部で生成した入力値制約の集合 $\mathbb{C}_{Input}$ および DB スキーマ制約の集合 $\mathbb{C}_{DBScheme}$，図 7 (4) レコード集合決定部で作成した $TableA$ に関するレコード集合表 $RATable_A$
- 出力：$TableA$ へアクセスするテストケースそれぞれの事前条件として適切な DB 初期状態が満たすべき制約である DB 初期状態値制約の集合 $\mathbb{C}$

提案するアルゴリズムの疑似コードを以下に示す．

```
ℂ_DBTable ← φ
for i ← 1 to N do
  w_i ← T_i の DB 検索条件の WHERE 句条件群
  for j ← 1 to w_i の WHERE 句条件の数 do
    w_ij ← w_i の j 番目の WHERE 句条件
    c ← w_ij が参照する TableA のカラム
    L ← RATable_A より TableA のレコード件数を取得
    f_1, ..., f_r ← c に対応する L 個のフィールド変数
    for k ← 1 to L do
      Rec ← TableA の k 番目のレコード
      w ← w_ij を複製
      w におけるカラム c を f_k で置き換える
      if RATable_A で w_ij が Rec を参照する then
        ℂ_DBTable へ w を追加する //(a)
      else if RATable_A で w_ij が Rec を参照しない then
        ℂ_DBTable へ w の論理否定を追加する //(b)
      end if
    end for
  end for
end for
ℂ ← ℂ_Input ∪ ℂ_DBScheme ∪ ℂ_DBTable
```

このアルゴリズムでは，同一テーブルにアクセスするすべての DB 検索条件すべてに対して過不足なく適切な件数のレコードがヒットするよう，必要な件数のレコードにヒットさせるための制約（疑似コードにおける (a) の個所）と，不必要なレコードにヒットしないようにするための制約（疑似コードにおける (b) の個所）の両方の制約をすべて抽出し，それらの制約の集合（疑似コードにおける $\mathbb{C}_{DBTable}$）を作成している．たとえば，図 1 におけるテストケース 1，2 で共有する DB 初期状態を，理想パターンのレコード集合を用いて生成する場合，制約集合 $\mathbb{C}_{DBTable}$ は以下のようになる．

```
DB 検索条件から抽出した制約
E[0].Year <= 2, E[0].Age >= 25,
・・・省略・・・
E[4].Year <= 2, E[5].Age >= 25,
E[5].Year <= 2, not(E[5].Age >= 25),
E[0].SectionId = SectionId,
・・・省略・・・
```

テストケース 2 では，年齢が 25 歳以上の社員が 6 名以

上ヒットすると，適切なDB初期状態ではなくなってしまうため，not(E[6].Age >= 25) という制約を入れることで，6名以上はヒットしないようにしている．

アルゴリズムの出力であるDB初期状態値制約の集合$\mathbb{C}$は，$\mathbb{C}_{DBTable}$と入力値制約の集合$\mathbb{C}_{Input}$，DBスキーマ制約の集合$\mathbb{C}_{DBScheme}$の和集合である．そのため，このような制約集合を満たすDB初期状態は，同一テーブルにアクセスするすべてのDB検索条件すべてに対して過不足なく適切な件数ヒットし，DBスキーマを満たし，かつ入力値との整合性もとれており，複数のテストケースの事前条件として適切なものとなる．(6) 制約解法部で$\mathbb{C}$を同時に解くことにより1つのテストケースグループにおいて，図3で示した例のような，すべてのテストケースの事前状態として適切なDB初期状態と各テストケースの入力値を生成することが可能となる．

4. 評価

提案手法に対する評価観点および評価結果を示し，考察を述べる．

4.1 観点

本研究では，複数のテストケースで同じDB初期状態を共有することでDB初期状態の数とデータサイズの削減を行い，テスト実施，データ管理を効率化することを目指している．1章で述べたとおり，本研究では効率化の先行指標として，以下の2点を既存のDB個別生成手法よりも改善することを目標としている．

- DB初期状態数：テストケース数に対するDB初期状態の数
- DBレコード件数の合計：全DB初期状態のDBレコード件数合計

今回の評価ではこの2つの指標で，提案手法と既存のDB個別生成手法の比較を行う．

テストケースグルーピングがうまく行われ，最終的にDB初期状態数の削減効果があるかどうかは，提案手法の入力となるテストケースの初期の並び順に強く依存すると予想される．そのため，テストケースの初期の並び順によりDB初期状態数，DBレコード件数の削減効果が低くなってしまうことがないかどうかの確認のため，この初期のテストケースの並び順を変えることで，DB初期状態数およびDBレコード件数合計の最大値（最悪の場合の削減効果）および平均値（平均的に期待できる削減効果）を求めた．また，テストケースの並び方次第でどれだけ削減効果を大きくする余地があるかどうかの確認を行うため，最小値についての計測も行った．

4.2 方法

前章で解説した提案手法に基づいてプロトタイプツールを作成した．制約ソルバとしては，制約プログラミングツールである，Choco Solver [1] を用いた．

題材としては，3つの社内開発された案件を用いた．

- システムA（12画面）：スケジュール管理を行うシステム．日時の範囲を指定した検索などを行う．
- システムB（11画面）：ネットワーク装置管理システムのWebフロントエンド．IPアドレスをキーとした検討などを行う．
- システムC（29画面）：データマイニングシステム．ユーザは様々な条件でデータの検索を行う．

システムに含まれるSQL文はそれぞれシステムAが27個，システムBが11個，システムBが17件である．これらシステムA，B，Cから合計8つのテーブルと，そのテーブルへの参照アクセスを行うテストケース合計87件を選び，評価に用いることとした．

本手法のプロトタイプツールではXML形式の設計モデルを入力とする．そのため，システムA，B，Cの設計書から既存技術 [17] を用いて処理フローの経路を抽出した後，手作業でXML形式の設計モデルを記述し，さらにシステムA，B，Cの設計書からDBスキーマと処理フローの経路中の検索条件に相当する設計情報を抽出して設計モデルに書き加えた．この設計モデルを入力としてプロトタイプツールにDB初期状態を生成させ，それが正確に経路を通るもの，すなわちテストケースの事前状態として適切なものになっているかを確認した．なお，設計モデルの作成作業はすべて筆者と異なる業務システム開発経験者1名が行った．設計モデルの作成者へは，作成したモデルから制約解法を用いてなるべく多くのテストケースで共有するためのDB初期状態を自動生成するという本手法の特徴は伝えておらず，業務ロジックを適切に表現できるよう設計モデルを作成してもらうようにした．

プロトタイプツールを動作させた環境はCPUがIntel Core i7 3.0 GHz，メモリは6 GB，OSはWindows Vista Ultimate SP2である．プロトタイプツールではデータ型として，Webアプリケーションで多く用いられる文字列型（使用可能な制約は等価，非等価），整数型を扱っている．そのため，今回の評価で必要となった列挙型，日付型については，整数型とのマッピングをとることで値生成を行った．

提案手法へ与えるテストケースの並び順は，設計モデルの各処理フローから自動抽出したテストケースの並び順をそのまま用いた．8つのテーブルのうち，提案手法を適用してDB初期状態数が1とならなかった場合については，テストケースの並び方をランダムに変えて提案手法を適用することを各テーブルにつき100回ずつ行い，DB初期状態数やDBレコード件数などの結果かどのように変わるかを計測した．DB初期状態数が1となったものについてはテストケースの並び方に結果が依存することはないため，

表 4 評価結果
Table 4 Evaluation results.

		既存手法			提案手法		
システム	DB テーブル	DB 初期状態の数	レコード件数	実行時間 (s)	DB 初期状態の数	レコード件数	実行時間 (s)
A	A1	3	3	0.249	1	1	0.203
	A2	12	12	0.501	4	10	0.530
	A3	13	49	0.547	5	45	4.236
	A4	15	53	0.624	6	25	0.764
B	B1	14	14	0.516	1	1	0.234
C	C1	19	10	0.438	1	10	0.218
	C2	9	4	3.122	1	1	2.684
	C3	2	1	0.249	1	1	0.187
合計		87	146	6.246	20	94	9.056

表 5 バックトラックに関する回数
Table 5 The number related to backtrack.

テーブル	分割数	理想回数	妥協回数
A1	0	1	0
A2	3	1	3
A3	4	4	1
A4	5	6	0
B1	0	1	0
C1	0	0	1
C2	0	1	0
C3	0	1	0
合計	12	15	5

この計測は行っていない．

4.3 結果

評価結果は**表 4** のようになり，既存の DB 個別生成手法に比べ DB 初期状態は約 23.0%まで削減でき，レコード数の合計は約 64.4%にまで削減できた．また**表 5** に，バックトラックに関連する数値として，提案手法を適用したときの分割回数（あるテストケースグループにおいて，理想，妥協どちらのパターンを試しても共有可能な DB 初期状態が存在しなかったため，そのテストケースグループを分割した回数）と，理想回数（理想パターンで解けた回数），妥協回数（理想パターンでは解けなかったが妥協パターンで解けた回数）をそれぞれ示す．

表 4 の A2，A3，A4 の 3 つのテーブルではすべてのテストケースで共有可能な DB 初期状態は生成できなかった．そのため，この 3 つのテーブルについては，テストケースの並び方をランダム変えてそれぞれ 100 回ずつ提案手法を適用した．その結果を**表 6** に示す．

4.4 考察

4.4.1 提案手法の完全性

今回の評価では，すべてのテストケースにおいて，その

表 6 テストケースの並び方をランダムに変えた結果
Table 6 Evaluation results with shuffled test cases.

	DB 初期状態数	レコード件数	分割数	理想回数	妥協回数
DB テーブル A2					
平均	3.59	8.42	2.59	1.81	1.78
最大	5	12	4	5	3
最小	2	4	1	1	0
DB テーブル A3					
平均	5.33	43.17	4.33	5.12	0.21
最大	8	48	7	8	1
最小	2	22	1	2	0
DB テーブル A4					
平均	7.24	44.29	6.24	7.12	0.12
最大	10	50	9	10	1
最小	5	24	4	5	0

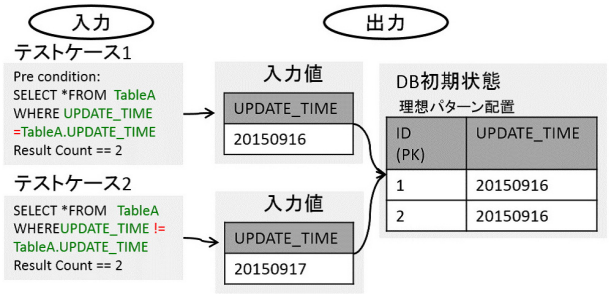

図 10 理想パターンで解けた例
Fig. 10 An example of test cases could be solved by ideal pattern.

テストケースに対応する DB 初期状態を得ることができた．提案手法のテストケースグルーピングにおいて，複数のテストケースで共有できる DB 初期状態がつくれなかった場合でも，最終的には 1 テストケースにつき 1 つの DB 初期状態を確実につくるような仕組みになっているため，すべてのテストケースについて，テストで事前条件として適切な DB 初期状態を作成することができた．

4.4.2 理想パターンと妥協パターンの効果

表 5，表 6 に示すように，理想パターンで解が得られているケースが多かった．DB 検索では，ユーザが画面から入力した入力値を用いることが多いためであり，提案手法で作成したテストケースグループにおいて，各テストケースが入力値を DB 検索条件で使用していたためである．これにより，**図 10** のように，それぞれのテストケースの入力値が異なる値をとることで解を得ていたケースが多かった．また，理想パターンでは解けず，妥協パターンでしか解けなかった場合も存在した．たとえば，**図 11** のようなケースにおいては，「テストケース 1 の入力値と `TableA.UPDATE_TIME` が等しいレコードが 2 件」，「テストケース 2 の入力値と `TableA.UPDATE_TIME` が等しくないレコードが 1 件」という条件は理想パターン（テーブルの

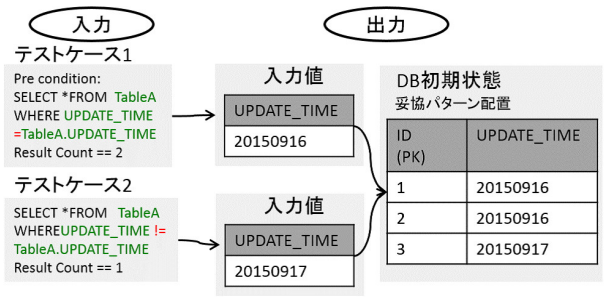

図 11 妥協パターンでなければ解けない例

Fig. 11 An example of test cases could be solved by not ideal pattern but compromise pattern.

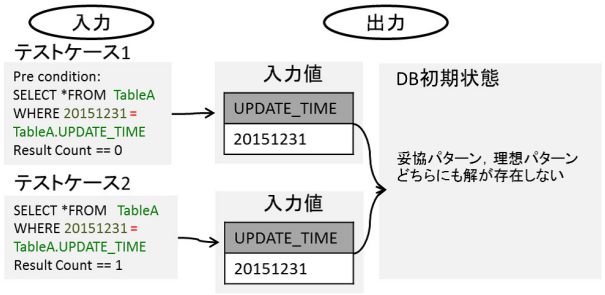

図 12 理想パターン，妥協パターンのどちらにも解が存在しない例

Fig. 12 An example of test cases could not be solved.

レコード件数は 2 件）という条件のもとでは，テストケース 1，2 の入力値がどのような値をとろうと絶対に成立せず，妥協パターンならば解が存在する．加えて，テストケースの入力値に自由度がなく固定されており定数値となるようなケースも理想パターンにおいて解が存在せず，妥協パターンで解く必要があった．たとえば，図 10 の例で，どちらのテストケースの入力値も 20150101 といった定数値が必要となる場合，理想パターンにおいて解が存在しない．このように，理想パターンを試すだけでは解が存在せず，多くのテストケース間で共有して使用できる DB 初期状態の具体値を生成できないケースも存在したため，まず理想パターンで試し，解が存在しなければ妥協パターンで試すという 2 段構えの方式は有効であった．

A2，A3，A4 のテーブルにおいては，すべてのテストケースの事前条件を満たす 1 つの DB 初期状態が生成されなかった．これは理想パターン，妥協パターンという 2 つのレコード集合において，すべての事前条件を満たす DB 初期状態が存在しなかったことを示している．実際に生成できなかった要因は，あるテストケースの事前条件として「条件 X を満たすレコードが存在する」という制約があり，別のテストケースの事前条件では「条件 X を満たすレコードが存在しない」というような互いに矛盾する制約が存在することであった．たとえば，図 12 に示すようなテストケースがあった場合には，理想パターン，妥協パターンのいずれの場合においても解を得ることができず 2 つのテストケースで共有可能な DB 初期状態を得ることができない．すなわち，提案手法が効果的に働く条件としては，図 12 のような互いに矛盾する制約を持つテストケースどうしがなるべく一緒にならないようにテストケースグルーピングが行われることであることが分かる．

4.4.3 実行時間

提案手法ではテストケースグルーピングやレコード集合のパターンについて総当りで試すことはしていないため，テストデータ生成の実行時間は既存手法に比べて 1.5 倍程度のオーバヘッドであった．また，システム C のように DB 初期状態の数を非常に少なくできたケースでは，既存の DB 個別生成手法よりも提案手法の方が高速にテストデータを生成できていた．理由としては，システム C のテストケースでは，DB 初期状態のレコード件数が 0 件でよいもの（検索結果が 0 件となった場合の結果画面を確認するテスト）が多かったため，すべてのテストケースで共有することが可能で，かつ理想パターンのレコード集合である DB 初期状態が生成でき，バックトラックが不要だったからだと考えられる．

4.4.4 テストケースの並び方の影響

表 6 に示すとおり，本評価で用いた 8 つの DB テーブルのうち 3 つは，提案手法により得られる DB 初期状態数，DB レコード件数がテストケースの並び方による影響を受ける．今回の 100 回の試行の範囲では，最悪の場合でも DB 初期状態数は必ず削減することができていたが，DB レコード件数は削減効果が出ていないこともあった．A2，A3，A4 それぞれについて，平均的には DB 初期状態数を 29.9%，41.0%，48.3%，また DB レコード件数については 60.1%，88%，83.4%に削減できることが分かった．このため，提案手法はテストケースの並び方によって影響を受けるが，どのような並び順であっても，多くの場合に DB 初期状態数の削減効果はあり，平均的には DB レコード件数の削減効果も見込めると考えられる．

提案手法を実際に用いる場合，データ生成に時間をかけることが許される状況ならば，テストケースの並びをランダムに変えて試行して最も DB 初期状態数や DB レコード件数が削減できているケースを選ぶことで，より良いものを選ぶといった方法も考えられる．

4.5 本手法の妥当性に関する考察

本節では提案手法の妥当性について考察する．

4.5.1 設計モデル作成のコスト

提案手法では DB 初期状態を含むテストケースを自動生成するために設計モデルを作成する必要があり，設計モデルを作成するコストが，DB 初期状態を含むそれぞれのテストケースをすべて手作業で作成するコストを上回ってしまう可能性がある．しかし，提案手法における設計モデルである処理フロー，入力定義，DB スキーマは，ウォータフォール開発などの開発プロセスの設計工程において通

常作成される成果物と位置づけられる．そして，DB 検索条件の制約記述についても，テストケース間で共有できる DB 初期状態が生成できるよう意識して記述する必要はなく，設計工程において詳細設計書に記述される DB 検索の詳細な情報（SQL 文相当の情報）とほぼ同等である．そのため，これまで使っていた設計ドキュメントの代わりに提案手法の設計モデルを設計工程で作成することにより，開発工程全体としては大きなオーバヘッドにならないようにできるものと考えられる．このような，設計工程の成果物である設計ドキュメントをある程度形式化した設計モデルを策定し，そこからテストケースを自動生成するモデルベーステストの考え方は，業務システムの結合テストをスコープとして実際に活用を進めている開発現場もある [19]．そのような開発現場においては，特に提案手法は導入しやすいはずである．

4.5.2 自動化範囲の妥当性

本手法では前節で述べたように設計工程の成果物に位置づけられる設計モデルから，複数のテストケースで共有できる DB 初期状態をすべて自動生成することを目指しているが，このような DB 初期状態の作成をすべて機械処理に任せるのではなく，テストケースのグルーピングやレコード集合などについては，人がレコード間の包含関係などを考慮してより賢く行い，制約生成や制約解法は機械処理で行うといった役割分担も考えられる．しかし，複雑なビジネスロジックを持つソフトウェアのテストでは，テストケースの数がとても多く，DB 検索条件も複雑になることが多いため，手動で多数のテストケースの DB 検索条件をそれぞれ確認し，DB 初期状態数や DB レコード件数が少なくなるようにそれぞれのテストケースの事前状態として適切な DB 初期状態を正確に作成していくのは大きな手間がかかる．そのため，これらの作業をすべて自動化することがテスト効率化において有用だと考えているが，複数のテストケースで共有可能な DB 初期状態を手動作成で作成することに実際どれだけの労力がかかり，かつどれだけ DB 初期状態数やレコード件数の少ないものにできるかという観点の評価も，より多角的に提案手法の妥当性を検証するうえで重要と考える．

4.5.3 本手法が想定する結合テストの支援範囲

提案手法は結合テストの一部を支援対象としており，処理フローで表現されるビジネスロジックの振舞いのパターン（処理フロー上の各経路）を網羅するようなテストケースの事前条件として適切な DB 初期状態を，複数のテストケースで共有できるように生成している．そのため，ソフトウェアの機能処理が設計どおりに動作するかについて網羅的に確認が行えるテストケースとその事前条件の DB 初期状態をつくることが可能となっている．しかし，業務システムの機能性に関する結合テストでは，テスト観点によっては，実運用の環境を意識した様々なバリエーションのデータが入った DB 初期状態や，同値分割，境界値解析や異常値を加味したパターンが含まれる DB 初期状態など，より多くのデータパターンが必要になる．このような様々なバリエーションのテストについては，基本的な機能処理がすべて正しく動作することを前提に行われることも多い．そのため，基本的な機能処理の動作を確認するテストケースが正しく実施されることは重要であり，そのテストケースの事前条件である DB 初期状態を自動生成する提案手法の重要性は高いと考える．また，今後は提案手法を発展させることで，複数のテストケースの各事前条件の制約だけではなく，DB レコードのフィールドの一部の値が境界値をとったり，異常値をとったりするなどの制約も加えて同時に解くことで，複数のテストケースの事前条件を満たしつつ，他の様々なテスト観点で必要なデータパターンも備える DB 初期状態をつくる，というアプローチも考えられる．

4.6 今後の課題

今回提案した各技術課題に対する解法を用いることで，DB 初期状態数，DB レコード件数の削減に効果があることが分かった．しかし，表 4 の結果は最適解ではないため，改良の余地はある．表 6 では，ランダムに並べ替えたとき，テストケースグルーピングの分割数や理想パターンで解を得られた回数の最大値，最小値に大きな幅があることが分かる．これは，テストケースグルーピングや理想パターンの成功率がテストケースの初期の並び順に依存していることを示しており，これは最終的な DB 初期状態数，DB レコード合計件数にも影響を与えている．たとえば，A2 の DB 初期状態数は表 4 では 4 つであるが，表 6 の実験により少なくとも 2 つまで削減可能であることが分かった．テストケースのグルーピングやレコード集合のパターンを，総当りに試したり表 6 の実験のように何度もランダムに並べ替えて試行したりするのは時間がかかるため，より賢く行っていくことが望まれる．たとえば，なるべく同一テーブルの違うカラムを参照するテストケースを同じテストケースグループにするなどの工夫を行うと，1 つのフィールド変数に矛盾する 2 つの制約がかかってしまい解がなくなるケースを減らせる可能性がある．同様に，レコード集合部においても，理想パターン，妥協パターンだけではなく，制約の重なり合いを考慮しつつ，理想パターンになるべく近いレコード集合もいくつか試してみることで，より少ないレコード件数の DB 初期状態を生成できる可能性がある．

5. 関連研究

各テストケースの事前条件として適切な DB 初期状態と入力値を自動で用意する手法は，おおまかに 3 系統に分けられる．

第 1 の手法は，個別 DB 生成手法であり，これはテストケースごとにそのテストケースの事前条件として適切な DB 初期状態を自動生成する手法である．著者らの過去の手法 [18] では，ソフトウェア設計情報である処理フロー図からパスを抽出し，そのパスごとにテストケースを生成し，パスに記述されている DB 初期状態の事前条件をもとに，テストケースの事前条件として適切な DB 初期状態と入力値のペアを生成する．藤原らの手法 [11], [20] では，ユーザがテストケースごとに DB 初期状態の事前条件を記述し，その事前条件を満たすような DB 初期状態，入力値のペアをテストケースごとに生成する．これらの手法では結合テストを対象としており，DB 初期状態，入力値の生成は自動で行われるため，制約解法さえできれば，各テストケースの事前条件として適切な DB 初期状態をそれぞれ確実に用意可能である．このほかに DBMS のテストを目的とした手法として ADUSA [12] がある．ADUSA は与えられた DB スキーマと SQL クエリの情報をもとに Alloy を活用することで，DB 初期状態を事前状態として持つ様々なバリエーションのテストケースを自動生成し，DBMS の網羅的なテストを自動で行うことが可能である．しかしながら，これら 3 つの手法ではテストケース 1 つにつき，1 つの DB 初期状態を生成しているため，テスト実施時に DB 交換の手間が増えてしまったり，DB レコード件数が増えることで DB 初期状態のサイズが大きくなったりするという問題点がある．

第 2 の手法は，テストケースごとに DB を調整し，そのテストケースの事前条件として適切な DB 初期状態にする手法である．Willmor らの手法 [16] では，DB アクセスをともなう単体テストのテストケースごとに，ユーザが事前条件として DB 初期状態が満たすべき制約を記述する．そして，各テストケースをする実施する前に，DB がテスト対象のテストケースの事前条件に合うようにするため，自動で DB レコードを追加，削除することで DB を調整し，事前条件として適切な DB 初期状態にする．Emmi らの手法 [10] では，DB アクセスをともなうメソッドを対象とし，Concolic Testing [15] の技術を用いて，メソッド内のまだ 1 度も通っていないパスを通るよう，DB 初期状態と入力値を調整し，調整した DB 初期状態と入力値を用いて何度もプログラムの実行を繰り返すことでパスカバレッジを上げることを目指している．これらの手法は，どちらも単体テストを対象としており，テスト実施（DB 初期状態の自動調整を含む）の自動化は，JUnit などを用いれば比較的容易に行うことが可能である．しかしながら，仮に，これらの技術で結合テスト向けの DB 初期状態を生成しても，前述したとおり，結合テストの場合では完全に自動実施する環境をつくることが難しく，テスト実施時の負担は大きくなってしまうという問題が残る．また，テストケースの実行のたびに DB 初期状態を調整しているため，生成される各テストケースの DB 初期状態が，テストケースの実施順序に依存してしまうという問題もある．

第 3 の手法は，DB 初期状態は調整せずすでに存在する（もしくは文献 [2], [3] などで自動生成した）ものを使用し，各テストケースの入力値のみをうまく調整し，各テストケースの事前条件として適切な DB 初期状態と入力値のペアを生成するという手法である．Pan らの手法 [13], [14] では，Concolic Testing [15] の技術を用いて，単体テストを対象として，入力値を調整しながら DB アクセスをともなうプログラムを繰り返し実行することにより，パスカバレッジを上げることができる．この手法の利点は多くのテストケースで共有できる DB 初期状態を得られる可能性がある点である．しかしながら，この手法では入力値のみの調整を行い，DB 初期状態はいっさい調整できないため，テストケースによっては適切な入力値，DB 初期状態のペアを生成できない可能性がある．

本手法は，結合テストを対象として各テストケースの事前条件として適切な DB 初期状態を確実につくれる個別 DB 生成手法をベースし，なるべく必要最小限の数で，データサイズの小さい，複数のテストケースで共有することが可能な DB 初期状態を生成できるような拡張を行った．これにより，DB 個別生成手法の利点を活かしつつ，テスト実施時に DB 初期状態入れ替えの労力を小さくし，テストデータ管理のコストを下げることが期待できる．

6. 結論

本研究では複数のテストケースで共有できる DB 初期状態を生成するための手法を提案し，技術課題を整理したうえで，それぞれの技術課題に対して解法を与えた．

テストケースグルーピング部では，効率良く，多くのテストケースで 1 つの DB 初期状態を共有できるよう最初はすべてのテストケースで共有できる DB 初期状態を探し，その後は再帰的にグループを分割しながら解となる DB 初期状態を探せるようにした．レコード集合決定部においては，理想パターン，妥協パターンという 2 段構えの方式でレコード集合を決定しており，これにより DB 初期状態数，DB レコード件数をより少なくすることが可能となる．DB 初期状態値制約生成部では，各テストケースの DB 検索条件に対して適切な件数のレコードを返すような制約を生成する工夫点により，複数のテストケースの事前条件として適切な DB 初期状態を生成することができる．

現実の業務システムを用いて，提案手法の適用評価を行ったところ，既存手法である個別 DB 生成手法に対して，DB 初期状態数を 23%に削減，DB レコード件数を 64%に削減でき，提案手法の有効性を確認することができた．

今後は，すでに開発現場でも適用が進んでいるモデルベーステストの考え方に基づき，設計モデルからテストケースを自動生成するツール [19] へ本手法を組み込むこと

で，より多くの実開発案件で適用評価を行い，開発現場のフィードバックを得ながら有効性評価を行っていきたい．加えて，各技術課題に対する解法の改良を進め，より多くのテストケースで共有できる，より小さなサイズの DB 初期状態を生成可能にし，テスト実施の負担軽減，データ管理コストの削減を行い，テストデータ自動生成技術の現場導入，テスト効率化を推進していきたい．

参考文献

[1] Choco solver, available from ⟨http://www.emn.fr/z-info/choco-solver/⟩.

[2] Dbmonster, available from ⟨http://dbmonster.kernelpanic.pl/⟩.

[3] Visual studio database edition, available from ⟨http://www.microsoft.com/japan/msdn/vstudio/⟩.

[4] Anand, S., Burke, E.K., Chen, T.Y., Clark, J., Cohen, M.B., Grieskamp, W., Harman, M., Harrold, M.J. and Mcminn, P.: An orchestrated survey of methodologies for automated software test case generation, *J. Syst. Softw.*, Vol.86, No.8, pp.1978–2001 (2013).

[5] Chays, D., Dan, S., Frankl, P.G., Vokolos, F.I. and Weber, E.J.: A framework for testing database applications, *Proc. 2000 ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA '00*, New York, NY, USA, pp.147–157, ACM (2000).

[6] Chen, S., Ailamaki, A., Athanassoulis, M., Gibbons, P.B., Johnson, R., Pandis, I. and Stoica, R.: Tpc-e vs. tpc-c: Characterizing the new tpc-e benchmark via an i/o comparison study, *SIGMOD Rec.*, Vol.39, No.3, pp.5–10 (2011).

[7] de Moura, L., Dutertre, B. and Shankar, N.: A tutorial on satisfiability modulo theories, Werner Damm and Holger Hermanns, editors, *Computer Aided Verification*, Vol.4590 of Lecture Notes in Computer Science, pp.20–36, Springer Berlin/Heidelberg, DOI: 10.1007/978-3-540-73368-3_5 (2007).

[8] Neto, A.C.D., Subramanyan, R., Vieira, M. and Travassos, G.H.: A survey on model-based testing approaches: A systematic review, *Proc. 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*, New York, NY, USA, pp.31–36, ACM (2007).

[9] Edvardsson, J.: A survey on automatic test data generation, *Proc. 2nd Conference on Computer Science and Engineering in Linköping, ECSEL*, pp.21–28 (Oct. 1999).

[10] Emmi, M., Majumdar, R. and Sen, K.: Dynamic test input generation for database applications, *Proc. 2007 International Symposium on Software Testing and Analysis, ISSTA '07*, New York, NY, USA, pp.151–162, ACM (2007).

[11] Fujiwara, S., Munakata, K., Maeda, Y., Katayama, A. and Uehara, T.: 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, DOI: 10.1007/s11334-011-0162-3 (2011).

[12] Khalek, S.A., Elkarablieh, B., Laleye, Y.O. and Khurshid, S.: Query-aware test generation using a relational constraint solver, *Proc. 2008 23rd IEEE/ACM International Conference on Automated Software Engineering, ASE '08*, Washington, DC, USA, pp.238–247, IEEE Computer Society (2008).

[13] Pan, K., Wu, X. and Xie, T.: Generating program inputs for database application testing, *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp.73–82 (Nov. 2011).

[14] Pan, K., Wu, X. and Xie, T.: Program-input generation for testing database applications using existing database states, *Automated Software Engineering*, Vol.22, No.4, pp.439–473 (2015).

[15] Sen, K., Marinov, D. and Agha, G.: Cute: A concolic unit testing engine for C, *Proc. 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering, ESEC/FSE-13*, New York, NY, USA, pp.263–272, ACM (2005).

[16] Willmor, D. and Embury, S.M.: An intensional approach to the specification of test cases for database applications, *Proc. 28th International Conference on Software engineering, ICSE '06*, New York, NY, USA, pp.102–111, ACM (2006).

[17] Zhang, X., Tanno, H. and Hoshino, T.: Introducing test case derivation techniques into traditional software development: Obstacles and potentialities, *Proc. 2011 IEEE 4th International Conference on Software Testing, Verification and Validation Workshops, ICSTW '11*, Washington, DC, USA, pp.559–560, IEEE Computer Society (2011).

[18] 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 (2012).

[19] 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 (2013).

[20] 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，全国大会講演論文集，Vol.72, pp.281–282 (2010).

[21] 独立行政法人情報処理推進機構：ソフトウェア開発データ白書 2014-2015，独立行政法人情報処理推進機構 (2014).

丹野 治門 （正会員）

2007 年電気通信大学電気通信学部情報工学科卒業．2009 年同大学大学院電気通信学研究科情報工学専攻博士前期課程修了．同年日本電信電話（株）入社．2008 年未踏ユース・スーパークリエータ認定（情報処理推進機構），2009 年山内奨励賞（情報処理学会），2013 年山下記念研究賞（情報処理学会），2016 年企業賞（情報処理学会），2016 年社長表彰（日本電信電話）．プログラミング言語，デバッガ，ソフトウェアテスト自動化に関する研究開発に従事．