社団法人　電子情報通信学会
THE INSTITUTE OF ELECTRONICS,
INFORMATION AND COMMUNICATION ENGINEERS

信学技報
TECHNICAL REPORT OF IEICE.

異なるスキーマ間に対応するSQL文の整合性のAlloy Analyzerを用いた一検証手法

藤田　悠矢† 　岡野　浩三† 　楠本　真二†

† 大阪大学大学院情報科学研究科，吹田市
E-mail: †{f-yuya,okano,kusumoto}@ist.osaka-u.ac.jp

あらまし　ビジネスアプリケーション等において，データベーススキーマが正しく設計・分割できていること，そのスキーマに対応するSQL文が正しく記述できていることが保証されているということは非常に重要である．データベーススキーマは慎重に設計を行った場合においても運営上の不具合が生じることや，長期にわたって使用することによりスキーマの構造が時代遅れのものとなることがある．その際， 業務に適した形へスキーマの改良 (スキーマ進化) を行う必要性があり，そのスキーマに対するSQL文も新しいスキーマに適するものに変換する必要がある．一方，ソフトウェアの仕様を形式的に記述できる言語にAlloyがあり，Alloyの制約記述に対してその制約を満たす例 (インスタンス)，満たさない例 (反例) を有界網羅的に検出するAlloy Analyzerというツールがある．本研究では，スキーマ変更に伴うSQL文の変換が正しく行われていることの確認をAlloy Analyzerによって行う手法を提案し，その手法の正しさの数学証明を行った．また，ケーススタディとして在庫管理プログラムで用いるデータベーススキーマに対して手法を適用し，その有用性の確認を行った．結果，Alloy Analyzerにおいてスキーマと SQL 文の整合性を確認することができた．

キーワード　Alloy，SQL，スキーマ変換，在庫管理問題

A Verification Method on Consistency between Different SQL Statements and Schemas using Alloy Analyzer

Yuya FUJITA†, Kozo OKANO†, and Shinji KUSUMOTO†

† Graduate School of Information Science and Technology, Osaka University, Suita-shi, 565-0871, Japan
E-mail: †{f-yuya,okano,kusumoto}@ist.osaka-u.ac.jp

Abstract　In software using database, it is very important that database schema is properly designed to assure that and also that SQL statements which correspond to the schema are correctly described. Long-term operation sometimes, however, causes operational defects becuase database shema might not fit to the current deployment despite that it was carefully designed in the past. In such a case, we have to perform improvement of the schema (Schema evolution) to a form suitable for the work, and also we have to convert the SQL statements into a form suitable for new schema. Alloy is a language which can formally describe specification of software. Alloy Analyzer is a tool to find instances that satisfy the constraint described in Alloy or counterexamples that do not satisfy. This report proposes a method that lets Alloy Analyzer verify if the conversion of SQL statements associated with the given schema modification have conformance. It also gives mathematical proof of the correctness of the method. The method is applied to the database schema used in a warehouse management program as a case study to confirm its usefulness. The author consequently can verify the conformance of the SQL statements using Alloy Analyzer.

Key words　Alloy, SQL, Schema evolution, Warehouse Management Program

1. まえがき

ビジネスアプリケーション等において，データベーススキーマが正しく設計・分割できていること，そのスキーマに対応する SQL 文が正しく記述できていることが保証されているということは非常に重要である．そのため，データベーススキーマの設計は慎重に行われるが，その場合においても実際に運用してみることで不具合を見つけることがある．また，アプリケーションの世界は常に変化し続けているため [1]，長年にわたってデータベースを運用し続けることにより，そのスキーマの構造が時代遅れのものとなることがある．その際，業務や時代に則した形へスキーマの改良 (スキーマ進化) を行う必要性があり，そのスキーマに対する SQL 文も新しいスキーマの形に適するものに変換する必要がある．正しい変更を実現するという試みは，アプリケーションソフトウェア業界において長年にわたり主要な活動とされてきた [2] [3] [4] [5] [6]．スキーマ進化に関してもその例外ではなく，スキーマの変換に関する研究 [7] [8] [9] や，XML のスキーマ進化に伴う問い合わせ言語の変更に関する研究 [10] は過去にも存在している．しかし，著者の調べた限りでは，スキーマ進化に伴い変更された SQL 文が意図通りのものかどうかを検証する方法に関する研究は見つけることができなかった．

一方，ソフトウェアの仕様を形式的に記述できる言語に Alloy [11] [12] があり，Alloy の制約記述に対してその制約を満たす例 (インスタンス)，満たさない例 (反例) をユーザの指定したスコープ内において網羅的に探索する Alloy Analyzer というツールがある．小さなスコープであっても非常に大きな空間内を探索するので，微妙なバグの発見するのにも十分な威力を発揮する．抽象度の高い仕様の段階で Alloy を使って検査しておくことで，その後の設計や実装段階で根本的な仕様バグが見つかることによる手戻りの発生を減らすことができる．

本研究では，広く使用されている関係データベースにおけるスキーマの変更に関して考え，それに伴う SQL 文の変換が正しく行われていることの確認を Alloy Analyzer によって行う手法を提案し，その手法の正しさの数学証明を行った．また，在庫管理プログラムで用いるデータベーススキーマに対して手法を適用することで，その有用性の確認を行った．結果，Alloy Analyzer においてスキーマと SQL 文の整合性の確認をすることができた．

本稿の構成は以下のとおりである．2 章では本稿に関する諸定義について述べ，3 章で提案手法，4 章でケーススタディへの適用結果について説明する．5 章では関連研究について言及し，最後に 6 章でまとめる．

2. 準　　備

2.1 スキーマ

スキーマはデータベースの構造を表しており，テーブル名，属性などの情報を持つ．また，テーブル間の関係やテーブルに関連する制約なども規定されている (図 1)．図 1 では，attribute_1はそれぞれのテーブルの主キーを表している．また

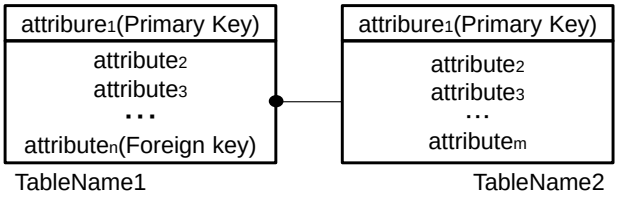

図 1　スキーマ例

```
sig Address {}
sig Name {
   add : one Address
}
fact {
   Name <: add
      in Name lone -> one Address
}
run {}
```

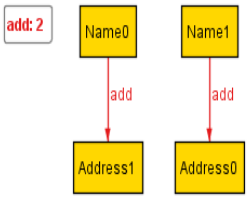

図 2　Alloy 記述例・Alloy Analyzer 出力例

, $TableName1$ における $attribute_n$は $TableName2$ の属性を参照している外部キーを表している．

このように簡単なスキーマでも複数のデータベースを組み合わせて構成することは多く，このような構成が意図通りであることを確認する需要があると考えることができる．

2.2 Alloy・Alloy Analyzer

Alloy は集合と関係からなる一回述語論理に基づいたオブジェクト指向モデリングのための形式仕様記述言語である [11] [12]．Alloy Analyzer は，Alloy の自動解析ツールであり，記述したモデルに対して表明の検証を行うことができる．一方モデルの探索範囲は有界であり，記述クラスも関係が表現できる，一回述語論理のサブクラスに限定されている．Alloy では，オブジェクトに相当するシグネチャやシグネチャが持つフィールド，そしてそれらの間に成り立つ論理式などにより，モデルを記述する．Alloy Analyzer はモデルに対してその反例，あるいは正例を有限探索した結果として出力することができる (図 2)．

図 2 では，Address シグネチャと一つの Address をフィールドとして持つ Name シグネチャの間に，一つの Address に対してその Address を保持する Name は高々一つであるという記述に対して，正例として Name0，Name1 がそれぞれ Address0，Address1 を保持している例があることを表すグラフを表しており，これがこのツールの出力としていることができる．

2.3 スキーマと SQL 文の整合性

変更前後のスキーマと SQL 文の満たすべき制約として，以下のようなスキーマと SQL 文の整合性を定義する．

［定義 1］（整合性） 変更前のリレーションスキーマ R，変更後のリレーションスキーマ R'，それぞれに対応する SQL 文の系列を s， s' とするとき，下の 2 つの写像があるとする．

$$f : R \to R'$$

$$g : s \to s'$$

R を満たす任意のリレーション r に任意の s を適用して得られる $h(r, s)$ と

R' を満たす任意のリレーション r' に $g(s)$ を適用して得られる $h(r', g(s))$ に対して次の式が成り立つとき，整合性が成り

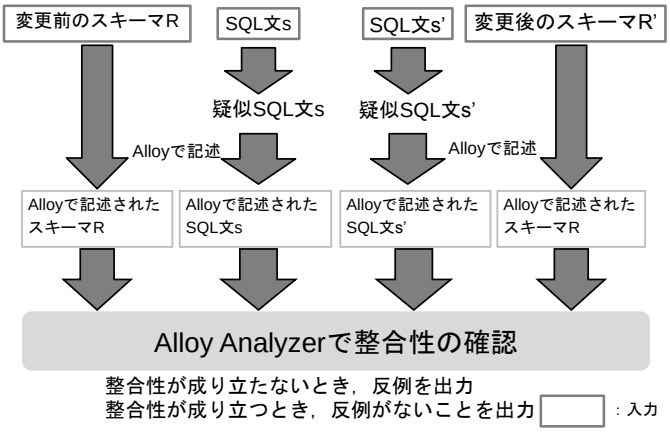

図 3　提案手法の流れ

入力として与えるSQL文 : DELETE FROM r WHERE $A_k = a_k$

疑似SQL文 : DELETE FROM r WHERE $A_K = x$ ($x \in A_k$)

図 4　疑似 SQL 文

立つという．

$$f(r) = r' \Rightarrow f(h(r,s)) = h(r', g(s))$$

3. 提 案 手 法

本節では，提案手法について説明する．提案手法における整合性確認の流れは次のようになる (図 3)．

（1） 入力として変更前後のスキーマ，SQL 文を与える
（2） SQL 文を疑似 SQL 文に変換する
（3） 各スキーマ，疑似 SQL 文を Alloy で記述する
（4） Alloy Analyzer を用いて整合性の確認を行う

疑似 SQL 文とは，変数を用いており，その変数に対して取りうる値に制約を付加した SQL 文である．この SQL 文に変換することで，ある特定の値のみでなく，その属性の取りうる値すべてに対して検証できる (図 4)．

従来の実データに依存する検証方法では，実データの状態によっては検証結果に不備が生じることがあったが，Alloy Analyzer を用いることにより，データベースの実データによらない有界網羅的かつ論理的な検証が可能となる． 以下では，スキーマおよび各 SQL 文，そして定義 1 で与えたスキーマと SQL 文の整合性を Alloy においてどのように記述したかを示す．記述において，A_i は属性名，そのドメインを $D_{A,i}$ で表している．なお，記述の正しさの証明に関してはページの都合上割愛する．定義 1 では，SQL 文は系列として与えていたが，証明では各 SQL 文に関してのみ定義している．これは，各 SQL 文において整合性を保つことができているならば，SQL 文の系列においても整合性を保つことができているからである．各 SQL 文の Alloy 記述は直観的に行うことができる．基本的には Alloy 記述に対する整合性の定義の妥当性に帰着することができる．

ス キ ー マ

スキーマの定義にはシグネチャを使用する．シグネチャはフィールドとして属性間の関係を持ち，また，主キーや外部

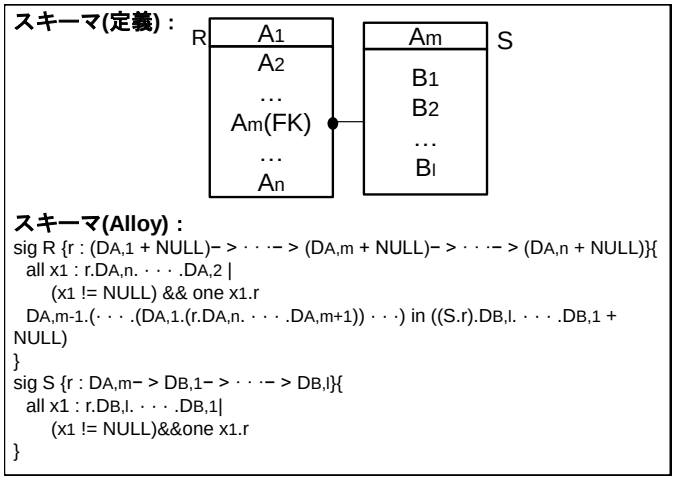

図 5　スキーマの Alloy への変換

SELECT A'1, …, A'm FROM r1, …, rn WHERE F;
↓
$\pi_{A'1,...,A'm}(\sigma_F(r1 \times \cdots \times rn))$

DELETE FROM r WHERE F;
↓
$r - \sigma_F(r)$

UPDATE r SET $A_i = a_i$ WHERE $A_j = a_j$;
↓
$(r - \sigma_{Aj=aj}(r)) \cup s$
$(s = \{t \mid t' \in \sigma_{Aj=aj}(r) \wedge t[Ai] = ai \wedge t[Ak] = t'[Ak](1 \le k \le n, \neq i)\})$

INSERT r INTO (a1, …, an);
↓
r ∪ (a1, …, an)

図 6　各 SQL 文の関係演算式での記述

キーといった整合性制約は不変条件として定義している．

SQL　文

各 SQL 文は関係演算を用いて記述することができる (図 6)．関係演算の和演算，差演算は Alloy において既に存在おり，その記述はそれぞれ $r + s$，$r - s(r,\ s$ はリレーション) である．関係演算の選択演算，射影演算，直積演算の Alloy における記述は図 7 に与える．

記述において，$A'_i(1 \le i \le m)$ は r_1 から r_n のいずれかに含まれる属性である．また，$t[A]$ はタプル t に含まれる属性 A の要素を，a_i は A_i の要素，F は条件式を表している．e_i は $D_{A,1} \sim D_{A,n}$ のいずれかである．図 7 の射影演算の記述において $D_{A',i} \in \{((R.r).D_{A,n}.\cdots.D_{A,2}), \ldots, (D_{A,n-1}.(\cdots.(D_{A,1}.(R.r))\cdots))\}$ かつ $D_{A',i} \neq D_{A',j}(1 \le i \neq j \le m)$ を満たしており，また，$D_{A',i}[x_1, ..., x_{i-1}](2 \le i \le m)$ はアトム $x_j \in D_{A',j}(1 \le j \le i-1)$ によって制限された範囲式である．

これらから各 SQL 文の Alloy における記述は図 8 となる．

スキーマと SQL 文の整合性

変更前後のスキーマ間，SQL 文の実行結果間で成り立つべき式を定め，それを Alloy において記述した．なお，SQL 文の整合性に関しては SELECT 文とそれ以外の SQL 文で式が異なっている (図 9)．なお， $s[r]$ はリレーション r に SQL 文 s

選択演算：
$\sigma_F(r) = \{t \mid t \in r \wedge P_F(t)\}$

```
fun Selection[r : (DA,1 + NULL)->…-> (DA,n + NULL),
                  x1 : e1, ..., xm : em]
: (DA,1 + NULL)->…-> (DA,n + NULL){
  {x1 : r.DA,n.….DA,2,
  ...,
  xn : xn-1.(….(x1.(r))…)
  |F}
}
```

射影演算：
$\pi_{A'1,...,A'm}(r) = \{t[A'1, \ldots, A'm] \mid t \in r\}$

```
fun Projection[r : (DA,1 + NULL)->…-> (DA,n + NULL)]
: DA',1−>…->DA',m {
  {x1 : DA',1,
  ...,
  xm : DA',m[x1, ..., xm-1]}
}
```

直積演算：
$r \times s = \{t * u \mid t \in r \wedge u \in s\}$

```
fun Cartesian[r : (DA,1 + NULL)->…-> (DA,n + NULL),
                  s : (DB,1 + NULL)->…-> (DB,m + NULL)]
  : (DA,1 + NULL)->…-> (DA,n + NULL)->
                  (DB,1 + NULL)->…->(DB,m + NULL){
  {x1 : r.DA,n.….DA,2,
  ...,
  xn : DA,n-1.(….(x1.(r))…),
  xn+1 : s.DA,n.….DB,2,
  ...,
  xn+m : DA,n+m-1.(….(xn+1.(s))…)}
}
```

図 7　関係演算の Alloy 記述

を実行した結果を表している．この記述はデータベースに与えられている具体的な値に関わらず，SQL 文とデータベースの整合性の関係を表している．

4. ケーススタディ

本章では，ケーススタディに用いたスキーマの概要と，手法を適用した結果を示す．

4.1 スキーマ概要

変更前後のスキーマとして図 10 のものを用意した．これは [13] の在庫管理プログラムを参考にして作成している．このスキーマは変更前における Request テーブルを変更後における Request，Customer，Item の 3 つのテーブルに，変更前における Stock テーブルを変更後における Stock，Container，Item の 3 つのテーブルに分割したものとなっている．

このスキーマに対して行われうる業務処理を 3 つ用意し，それぞれの業務処理に対して 2 つの適用例の観点で検証を行った．今回検証を行った業務処理，適用例は以下のとおりである．

業 務 処 理

- 注文個数以上の在庫を持つコンテナをすべて出力
- 新規コンテナの入荷
- 新規注文の受付

適　用　例

（1） 整合性の成り立たない SQL 文を与え，反例が表示されることを確認

```
SELECT(SQL) : SELECT A'1, …, A'm FROM r1, …, rn
                                  WHERE F;
SELECT(Alloy) :
fun Select[r1rows : DA1,1->…-> DA1,k1,
           …,
           rnrows : DAn,1->…-> DAn,kn, x1 : e1, ..., xl : el]
: DA',1->…-> DA',m{
  Projection[Selection[Cartesian[r1rows, …, rnrows], x1, ..., xl]]
}

DELETE(SQL) : DELETE FROM r WHERE F;
DELETE(Alloy) :
fun Delete[rrows : DA,1->…-> DA,n, x1 : e1, ..., xl : el]
 : DA,1->…-> DA,n{
  rrows − Selection[rrows, x1, ..., xl]
}

UPDATE(SQL) : UPDATE r SET Ai = ai WHERE Aj = aj;
UPDATE(Alloy) :
fun Update[rrows : DA,1->…->DA,n, ai : DA,i, aj : DA,j]
  : DA,1->…-> DA,n{
  rrows − Selection[rrows, aj] + UpdatedRows[rrows, ai, aj]
}
fun UpdatedRows[rrows : DA,1->…->DA,n, ai : DA,i, aj : DA,j]
  : DA,1->…->DA,n{
   {x1 : (rrows).DA,n.… .DA,2,
    ...,
    xi : ai,
    xi+1 : DA,i.(xi-1.(….(x1.((rrows).DA,n.….DA,i+2))…)
    ...,
    xn : xA,n-1.(….(DA,i.(….(xn+1.(rrows))…)
     | xj = aj
   }
}

INSERT(SQL) : INSERT r INTO (a1, …, an);
INSERT(Alloy) :
fun Insert[rrows : DA,1->…-> DA,n, a1 : DA,1, ..., an : DA,n]
: DA,1->…->DA,n{
  rrows + a1->…->an
}
```

図 8　SQL 文の Alloy への変換

（2） 整合性の成り立つ SQL 文を与え，反例がないと表示されることを確認

今回はページの都合により，業務処理 1 におけるそれぞれの適用例の結果のみを示す．

4.2 業務処理 1 における実行結果

業務処理 1 は，「顧客の注文を同一コンテナから発送できないかどうかを確認し，できる場合そのコンテナの ID，入庫日，在庫数を出力する」といった処理である．適用例 1 に該当する SQL 文として図 11 を用いた．図 10 のスキーマと図 11 の SQL 文を入力として手法を適用した結果，Alloy Analyzer において，反例があると出力された (図 12)．その反例の一つとして，Alloy Analyzer では図 13 が示された．この図からわかるように，変更前と変更後において，出力結果が異なっていることがわかる．この反例は，図 11 の変更後の SQL 文において Stock テーブルの ContainerID と Container テーブルの ContainerID が一致するかどうかを確認する一文を記述し忘れたために生じたミスである．

続いて，適用例 2 に該当する SQL 文として図 14 を用いた．

スキーマ分割における整合性(定義)：

$r = \{t'[A_1, \ldots, A_m] * t''[A_{m+1}, \ldots, A_n] \mid t' \in r' \wedge t'' \in r'' \wedge t'[B_1] = t''[B_1]\}$

スキーマ分割における整合性(Alloy)：

```
pred SchemaConsistency(r : R, r' : R', r'' : R''){
  r.rows = {x1 : (s.rows).DB,1.DA,m. · · · .DA,2,
  . . . ,
  xm : xm-1.(xm-2.(· · · .(x1.(r'.rows.DB,1)) · · ·)),
  xm+1 : (DB,1.(r''.rows)).DA,n. · · · .DA,m+2,
  . . . ,
  xn : xn-1.(· · · .(xm+1.(DB,1.(r''.rows))) · · ·)
    | xm.(· · · .(x1.(r'.rows)) · · ·) = (r''.rows).xn. · · · .xm+1}
}
```

スキーマ分割におけるSQL文の整合性(定義)：

SELECT文：

$s[r] = s'[r', r'']$

DELETE文，UPDATE文，INSERT文：

$s[r] = \{t'[A_1, \ldots, A_m] * t''[A_{m+1}, \ldots, A_n] \mid t' \in s'[r'] \wedge t'' \in s'[r''] \wedge t'[B_1] = t''[B_1]\}$

スキーマ分割におけるSQL文の整合性(Alloy)：

SELECT文：

```
assert Consistency{
all x1 : e1, . . . , xn : en,
  r : R, r' : R', r'' : R'' |
   (SchemaConsistency[r, r', r''])
     => (SELECT_Before[r, x1, . . . , xn]
           = SELECT_After[r', r'', x1, . . . , xn])
}
```

DELETE文，UPDATE文，INSERT文：

```
assert Consistency{
  all x1 : e1, . . . , xn : en,
  r_pre, r_post : R, r'_pre, r'_Post : R', r''_pre, r''_post : R''|
    (SchemaConsistency[r_pre, r'_pre, r''_pre]&&
     r_post = SQL Before[r_pre, x1, . . . , xn] &&
     r'_post = SQL_After[r'_pre, x1, . . . , xn] &&
     r''_pre = SQL_After[r''_post, x1, . . . , xn])
        => (SchemaConsistency[r_post, r'_post, r''_post])
}
```

図 9　スキーマと SQL 文の整合性

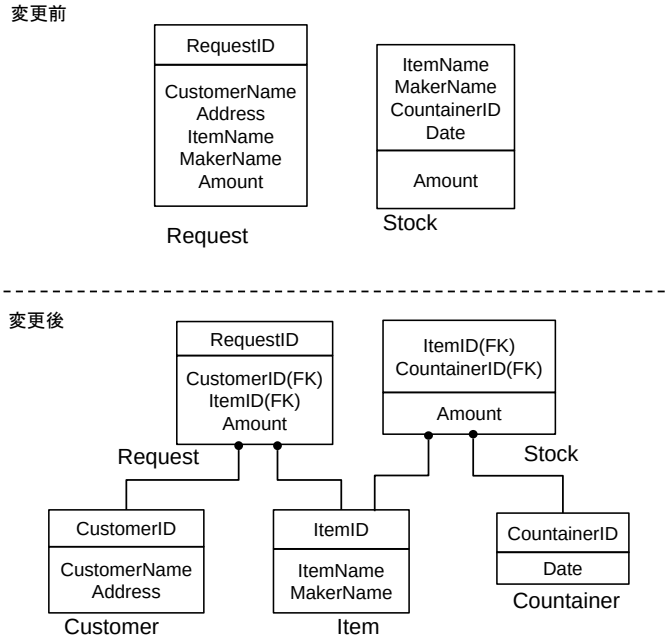

図 10　検証対象のスキーマ

変更前SQL文：

```
SELECT ContainerID, Date, StockAmount FROM Stock
 WHERE ItemName
       = (SELECT ItemName FROM Request WHERE ReqID='ReqID')
 AND MakerName
       = (SELECT MakerName FROM Request WHERE ReqID='ReqID')
 AND StockAmount
       >= (SELECT Amount FROM Request WHERE ReqID='ReqID');
```

変更後SQL文：

```
SELECT Stock.ContainerID, Date, StockAmount FROM Stock, Container
 WHERE Stock.ItemID
       = (SELECT ItemID FROM Request WHERE ReqID='ReqID')
 AND Stock.StockAmount
       >= (SELECT Amount FROM Request WHERE ReqID='ReqID');
```

図 11　変更前後の SQL 文 (適用例 1)

```
Executing "Check SelectContainer for 3 int, 3 seq, 3 Domain"
  Solver=sat4j Bitwidth=3 MaxSeq=3 SkolemDepth=1 Symmetry=20
  22514 vars. 1149 primary vars. 61101 clauses. 573ms.
  Counterexample found. Assertion is invalid. 459ms.
```

図 12　Alloy Analyzer 出力結果 (適用例 1)

適用例 1 における変更後の SQL 文と異なる点は，太字で示されている箇所である．変更前の SQL 文は図 11 と同様のものである．これらの SQL 文と図 10 のスキーマを入力として手法を適用すると，Alloy Analyzer では反例がないことが示された (図 15)．

これにより，与えた SQL 文は期待通りの変換が行えていたことがわかる．

4.3　考　　察

検証した 3 つの業務処理，2 つの適用例に対して，期待通りの結果を得ることができた．しかし，今回採用した記述方法では，大規模なスキーマの検証は不きであることが分かった．また，各業務処理，各適用例における記述行数は表 1 のとおりである．表 1 からわかるように，記述においてかかるコストは決して低いとは言えない．しかし，証明を通して SQL 文から Alloy 式への変換を機械的に行えることは確認している．

5.　関 連 研 究

スキーマや SQL 文を Alloy に変換することで，DBMS のテストを行うツールとして ADUSA [14] がある．このツールは，Alloy Analyzer の有界網羅的に正例を探索する特性を活かして，スキーマと SQL 文を Alloy で記述し，DBMS のテストデータを大量に作成するといったものである．このツールを用いることにより，人力で作成するよりも大量のテストデータを短時間で作成することができる．また，テスト用の SQL 文を生成するアプローチ [15] と組み合わせることにより，DBMS のテストの自動化も行っている [16]．しかし，この研究において SQL 文は

表 1　記 述 行 数

	業務処理 1	業務処理 2	業務処理 3
適用例 1	203	231	242
適用例 2	207	232	240

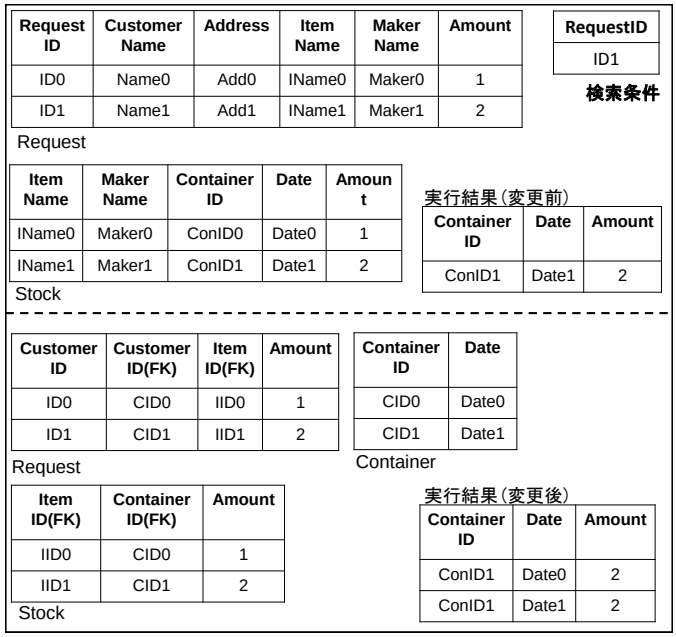

図 13　得られた反例 (上：変更前，下：変更後)

```
SELECT Stock.ContainerID, Date, StockAmount FROM Stock, Container
WHERE Stock.ItemID
        = (SELECT ItemID FROM Request WHERE ReqID='ReqID')
AND Stock.StockAmount
        >= (SELECT Amount FROM Request WHERE ReqID='ReqID')
AND Stock.ContainerID=Container.ContainerID;
```

図 14　変更前後の SQL 文 (適用例 2)

```
Executing "Check SelectContainer for 3 int, 3 seq, 3 Domain"
  Solver=sat4j Bitwidth=3 MaxSeq=3 SkolemDepth=1 Symmetry=20
  22505 vars. 1149 primary vars. 61071 clauses. 554ms.
  No counterexample found. Assertion may be valid. 24808ms.
```

図 15　Alloy Analyzer 出力結果 (適用例 2)

SELECT 文の変換のみしか考慮されておらず，データベースの更新が生じる操作 (DELETE，UPDATE，INSERT) の Alloy への変換は考慮していない．

本研究では，スキーマや SELECT 文の Alloy 記述は [14] を参考にして変換を行ったが，データベースの更新を伴う操作に関しても変換を行い，検証を行っている．

6. あ と が き

本稿では，スキーマ変換が生じた際の SQL 文の変更が正しく行われているかどうかの確認を Alloy Analyzer を用いて行う手法を提案した．また，ケーススタディとして在庫管理プログラムにおける 3 つの業務処理に対して手法を適用し，いずれの業務処理においても整合性が成り立たない SQL 文に対しては反例を，整合性の成り立つ SQL 文に対しては反例がないことを示すことが確認された．

今後の課題として，より大規模なスキーマに手法を適用することによる検証，スキーマや SQL 文を Alloy に自動変換するツールの作成を行うことを考えている．

謝辞　本研究は，科学研究費補助金基盤研究 (S)(課題番号：25220003)，基盤研究 (C)(課題番号：21500036) の助成を得た．

文　　献

[1] D. Sjoberg, “Quantifying schema evolution,” Information and Software Technology, vol.35, no.1, pp.35–44, 1993.
[2] M.V. Zelkowitz, “Perspectives on Software Engimeering,” ACM Computing Surveys, vol.10, no.2, pp.197–216, June 1978.
[3] L.H. Putnam, Software Cost Estimating and LifeCycle Control, IEEE Computer Society, 1982.
[4] G. Parikh and N. Zvegintzov, “The World of Software Maintenance,” Tutorial on Software Maintenance, pp.1–3, IEEE Computer Society, 1983.
[5] T.A. Corbi, “Program Understanding: Challenge for the 1990s,” IBM Systems Journal, vol.7, no.2, pp.294–306, 1989.
[6] E.J. Chikofsky and J.H.C. II, “Reverse Engineering and Design Recovery: A taxonomy,” IEEE Software, vol.7, no.1, pp.13–17, Jan. 1990.
[7] P. McBrien and A. Poulovassilis, “Schema Evolution in Heterogeneous Database Architectures, A Schema Transformation Approach,” Lecture Notes in Computer Science, vol.2348, pp.484–499, 2002.
[8] S. Monk and I. Sommerville, “Schema evolution in OOODBs using class versioning,” ACM SIGMOD Record, vol.22, no.3, pp.16–22, 1993.
[9] M. Blaschkaand, C. Sapia, and G. Hofling, “On Schema Evolution in Multidimensional Databases,” Lecture Notes in Computer Science, vol.1676, pp.153–164, 1999.
[10] 長谷川数馬，池田光雪，鈴木伸崇，“スキーマ進化に伴う XPath 式修正アルゴリズム，” 情報処理学会第 74 回全国大会，pp.4–6, 2012.
[11] D. Jackson, Software Abstractions, MIT Press, Nov. 2011.
[12] 中島震，鵜林靖，“Alloy:自動解析可能なモデル規範形式仕様言語，” コンピュータソフトウェア，vol.26，no.3，pp.78–83，July 2009.
[13] 尾鷲方志，岡野浩三，楠本真二，“在庫管理プログラムに対する JML 記述と ESC/Java2 を用いた検証の事例報告，” 電子情報通信学会論文誌，vol.J91D，pp.2719–2720，Nov. 2008.
[14] S.A. Khalek, B. Elkarablieh, Y.O. Laleye, and S. Khurshid, “Query-aware Test Generation Using a Relational Constraint Solver,” Proc. of 23rd ASE ’08, IEEE/ACM Int. Conf. on Automated Software Engineering, pp.238–247, 2008.
[15] S.A. Khalek and S. Khurshid, “Automated SQL Query Generation for Systematic Testing of Database Engines,” Proc. of 25th ASE ’10, IEEE/ACM Int. Conf. on Automated Software Engineering, pp.329–332, 2010.
[16] S.A. Khalek and S. Khurshid, “Systematic Testing of Database Engines Using a Relational Constraint Solver,” ICST’11 Proc. of the 2011 Fourth IEEE Int. Conf. on Software Testing, Verification and Validation, pp.50–59, March 2011.