ユースケース間の関係を考慮した網羅的な受け入れテストの支援

雁行進夢[†] 久保淳人[†]
鷲崎弘宜[†,††] 深澤良彰[†]

ユースケースから受け入れテストのテストケースを作成することができる．しかし，ユースケースの実行パスの識別を人手で行う手法では，特にユースケース間の関係が複雑な場合に，実行パスの識別に漏れが出る可能性がある．また，ユースケースを用いた受け入れテストの網羅性の定義がなかったため，テスト終了の判定に対して属人性が高くなる．本稿では，まずユースケースを用いた受け入れテストの網羅度を定義し，次に，ユースケースの実行パスを自動で識別したうえで，指定の網羅性を満たすテストシナリオ群とテストプログラムの雛形を自動生成する手法を提案する．提案手法により，ユースケースの実行パス識別の漏れがなくなり，受け入れテストの終了判定について属人性を排除できる．また，テスト環境の雛形の自動生成により受け入れテストの効率の向上が見込める．

Support for acceptance test based on the inter-use case relationships

Susumu Kariyuki,[†] Atsuto Kubo,[†] Hironori Washizaki[†]
and Yoshiaki Fukazawa [†]

Manual detection of execution paths of use cases can miss some execution paths. Moreover, lack of coverage metric of acceptance test results ambiguous judgement of acceptance test ’s completion. In this paper, we propose metrics of acceptance test ’s coverage and automated procedure of test scenarios and skeleton code in specified condition of coverage.

1. はじめに

受け入れテストとは，開発されたシステムが顧客の要求を正しく実現しているかを，顧客が主体となって確認する作業である．機能要求を表す手法の一つとして，ユースケース[3] がある．ユースケースとは，利用者や外部システム (アクタと呼ばれる) から見たシステムの外部機能を表現したものである．ユースケースによって機能要求が識別されている場合は，ユースケースの実行パスを人手で識別し，識別された実行パスを元にテストシナリオを作成する．そのため，実行パスの識別に漏れがでるおそれがある．特にユースケース間の関係が複雑な場合はその可能性は高まり，受け入れテストの信頼性を損なうおそれがある．

一方，テスト終了判定の指針としてテスト網羅率が用いられる．テスト網羅率とは，テストすべき項目数に対する実際にテストされた項目数の割合である[1]．テスト網羅率の例として，ホワイトボックス・テストにおける命令網羅率や分岐網羅率などがある．しかし，ユースケースから識別されたテストシナリオを対象とする受け入れテストの網羅性に関する明確な定義がない．そのため，受け入れテストの終了判定の属人性が高い．結果として、受け入れテストの網羅的な実施についてブレが生じ信頼性を損なうおそれがある．

本稿では，ユースケース記述を元に識別したテストシナリオを対象として，受け入れテストの網羅率を定義する．この定義のもとで，ユースケース記述とドメインモデルとテスト網羅性を入力して，特定のテスト・フレームワーク形式の受け入れテスト環境を自動生成する手法を提案する．提案手法により，ユースケースから識別されたテストシナリオを対象とする受け入れテストの終了判定に対して属人性を排除することが可能となる．その結果，受け入れテストの対象となるテストシナリオの漏れがなくなるため，受け入れテストの信頼性の向上が見込める．また，受け入れテスト環境雛形の自動生成により受け入れテストの効率の向上が見込める．

† 早稲田大学
Waseda University
†† 国立情報学研究所 GRACE センター
National Institute of Informatics GRACE Center

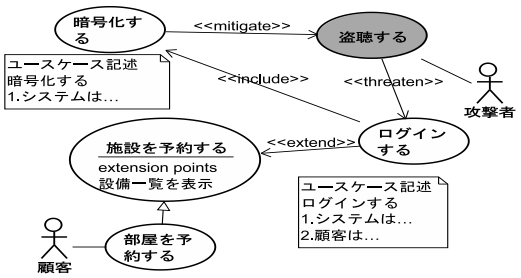

図 1 ユースケース図の例
Fig. 1 Example of use case diagram

ユースケース：ログインする
基本フロー:ログインする
1. システムは顧客に ID と…
2. 顧客は ID と…
代替フロー：認証に失敗する
「認証する」の後に分岐する
4a. システムは基底ユースケースを終了する
「ログインする」の「システムは…」ステップは
「暗号化する」ユースケースを実行する
拡張点:施設を予約する. 設備一覧を表示

図 2 「ログインする」ユースケース記述
Fig. 2 「Log-in」use case specification

2. ユースケースを用いた受け入れテストとその問題点

2.1 ユースケースとユースケース間の関係

ユースケースとは，アクタから見たシステムの外部機能を表現したものである．ユースケースはシステムの内部構成には触れず，アクタの視点に基づいたシステムに対する要求を整理するのに用いられる．ユースケース記述のフローに記載されている 1 アクションをユースケース・ステップと呼ぶ．以下では，ユースケース・ステップを単にステップと呼ぶ．

Unified Modeling Language(UML)2.0[4] によるユースケース図の例を**図 1** に示す.「ログインする」ユースケース記述の一部を**図 2** に示す.

UML2.0 で定義されているユースケース間の関係 (包含，汎化，拡張) を用いて既存のユースケースを変更せずに動作を追加することで，ユースケースの記述を再利用することが可能となる．

包含 (include) 包含とは，あるユースケースの中に別のユースケースの処理が含まれることを表す関係である．図 1 では,「ログインする」ユースケースは「暗号化する」ユースケースを含む.

汎化 (generalization) 汎化とは，ユースケース間の is-a 関係である．図 1 では,「部屋を予約する」ユースケースは「施設を予約する」ユースケースの一種であることを意味する．

拡張 (extend) 拡張とは，既存のユースケースに別のユースケースの内容を追加する関係である．拡張されるユースケースは，拡張点 (extention points) と呼ばれる追加的な振る舞いを挿入可能な特定の実行ポイントを持つ．拡張するユースケースは拡張点を指定することで追加する場所を指定する．図 1 では,「ログインする」ユースケースは「施設を予約する」ユースケースを拡張する．拡張点として「設備一覧を表示する」を指定する．

また，拡張関係の応用例として，ミスユースケースやセキュリティユースケースがある[5),6)]．ミスユースケースとは設計中のシステムに対する敵対的なアクタの視点から記述するユースケースである．セキュリティユースケースとはミスユースケースの影響を緩和するユースケースである．文献[6)] では，あるミスユースケースがユースケースに「脅威」を与えることを threaten という関連として定義し，あるユースケースがミスユースケースの脅威を「緩和」していることを mitigate という関連として定義する．また，脅威は拡張を特化した関連と定義できると述べている．図 1 では,「盗聴する」ユースケースが「ログインする」ユースケースに脅威を与えることを，<<threaten>> のステレオタイプ付きの矢印で表している．同様に図 1 において,「暗号化する」ユースケースが「盗聴する」ユースケースの脅威を緩和していることを<<mitigate>> のステレオタイプ付きの矢印で表している．

2.2 受け入れテスト

ユースケースは，顧客の要求を表すので，ユースケースからテストケースを作成することで，顧客の要求が実装されているかテストすることができる．顧客の要求に関するテストの一つに受け入れテストがある．受け入れテストとは，開発しているソフトウェアが，顧客の望む内容を本当に実装しているかどうか，顧客が主体となってテストすることである．

2.3 ユースケースから作成した受け入れテストの問題点

ユースケースから実行パスを識別してテストシナリオを作成する既存手法として文献[7),8)] などが挙げられる．文献[7),8)] では，ユースケースの実行パスを人手で識別し，識別された実行パスを元にテストシナリオを作成している．そのため, 以下の問題がある．

問題 1 ユースケースから識別されたテストシナリ

オを用いた受け入れテストを行なう際，テストの網羅性について統一された基準がない．そのため，テスト終了の判定に属人性がでてしまう．その結果，受け入れテストの網羅的な実施にブレが生じ，受け入れテストの信頼性を損なうおそれがある．

問題 2　ユースケース間の関係が複雑な場合，実行パスの識別に漏れが発生するおそれがある．実際の開発ではユースケースの数が数十，数百になることもあり，それに伴ってユースケース間の関係も複雑になる．また，通常のユースケースだけでなく，セキュリティに関する要求が高まっていることから，ミスユースケースやセキュリティユースケースなども考慮する必要な場合もある．その場合は，通常のユースケースのみを考える場合よりも実行パスが漏れてしまう可能性が高くなる．

3. テスト基準に基づく受け入れテスト環境の作成

2.3 節で述べた問題に対し，次の解決を提案する．

問題１の解決　ユースケース記述から識別したシナリオを対象とする受け入れテスト網羅率の定義

問題 2 の解決　ユースケース記述とドメインモデルと網羅性を入力して特定のテスト・フレームワーク形式の受け入れテスト環境の雛形を生成するシステム

3.1 システム構成

ユースケースの実行パスを人手で識別すると，漏れが発生する恐れがある．ユースケース間の関係が複雑な場合はさらにその可能性が高まる．解決策として，ユースケース記述とドメインモデルと網羅性を入力として，実行パスの識別を自動で行い特定のテスト・フレームワーク形式の受け入れテスト環境の雛形を生成するシステムを提案する．本稿では，多言語対応や普及度を考慮して Framework for Integrated Test(FIT)[2] 形式にした．システムの概要を**図 3** に示す．

ユースケース記述とドメインモデルと網羅性の選択を提案手法のシステムに入力してテストシナリオ (HTML ファイルの表形式) とフィクスチャの雛形を自動生成する．テストシナリオ群は選択された網羅性を満たす必要最低限の数だけ出力される．テストシナリオ内に記述されたテスト対象項目を実行しその結果を出力する．ユースケース記述は専用のエディタで入力する．生成されるフィクスチャは Java 形式である．

顧客は生成されたテストシナリオにテスト内容を追加する．テストプログラム開発者はフィクスチャの雛形に手を加え対応する機能を呼び出す記述を追加する．以上の後，詳細化されたテストシナリオとフィクスチャを FIT に入力して受け入れテストを行なう．

3.2 ユースケースを用いた受け入れテストの網羅性

ユースケース uc から識別されたテストシナリオに基づく受け入れテストの網羅性について，次の通りに定義する．すべてのステップの集合を S, すべての分岐の集合を B とすると，uc は以下の通りになる．

$$S = \{s_1, s_2, \ldots\}$$
$$B \subseteq S \times S$$
$$uc = (S, B)$$

ユースケース記述中のフローとプログラムフローの類似性に着目しプログラムフローのテスト網羅性の定義を参考として，ステップ網羅，分岐網羅，全パス網羅の 3 種を提案する．$|X|$ を集合 X の要素数とする．

定義１　ステップ網羅 C'_0

ステップ網羅 C'_0 は全てのユースケース・ステップの中でいくつのステップが実行されたかを百分率で表す．テストケースの集合 T が含むテスト項目 t により処理されたステップの集合を S_t として，ステップ網羅率 C'_0 を次のように定義する．

$$C'_0 = \frac{|S_t|}{|S|} \times 100$$

定義 2　分岐網羅 C'_1

分岐網羅 C'_1 は全てのフローの分岐の中でいくつの分岐が実行されたかを百分率で表す．テストケースの集合 T が含むテスト項目 t により処理された分岐の集合を B_t として分岐網羅率 C'_1 を次のように定義する．

$$C'_1 = \frac{|B_t|}{|B|} \times 100$$

定義 3　全パス網羅 C'_∞

全パス網羅 C'_∞ は全てのフローのパスの中でいくつのパスが実行されたかを百分率で表す．すべてのパスの集合を P とする．テストケースの集合 T が含むテスト項目 t により処理されたパスの集合を P_t として，全パス網羅率 C'_∞ を次のように定義する．

$$C'_\infty = \frac{|P_t|}{|P|} \times 100$$

実際に，**図 4** のユースケースに対して，C'_1 の算出例を示す (文献[3] より一部変更して抜粋)．実行パスをグラフ化したものを**図 5** に示す．実際にテストされた分岐を $\{c_2\}$ として，分岐網羅率 C'_1 は以下となる．

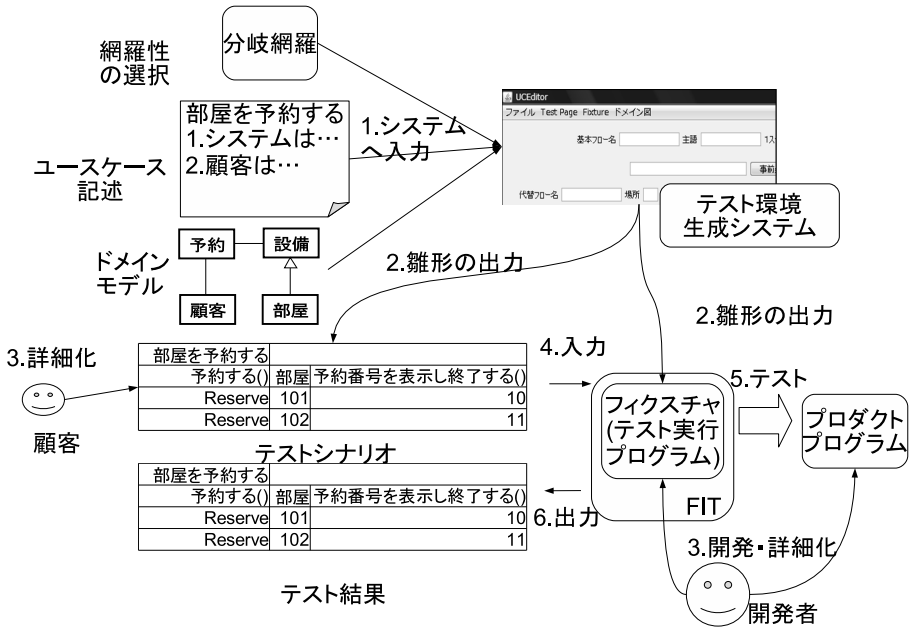

図 3 提案手法のシステムの概要
Fig. 3 Outline of our proposal system

ユースケース：リソースへのアクセスを待たせる
基本フロー：リソースへのアクセスを待たせる
1. リソースロックは，サービスクライアントのプロセスを一時中断させる
2. サービスクライアントは再開されるのを待つ
3. サービスクライアントのプロセスが再開される
代替フロー
A1. 待ち時間のタイマー切れ
基本フローのステップ 2 において，
待ち時間のタイマーの期限が切れたとき
1. 要求されたアクセスが許可できないことをサービスクライアントにシグナルで知らせる
2. ユースケースが終了する．

図 4 「リソースへのアクセスを待たせる」ユースケース記述
Fig. 4 「Wait for the access to the resource」use case specification

$$C_1' = \frac{|\{c_2\}|}{|\{c_2, c_4\}|} \times 100 = 1/2 \times 100 = 50\%$$

ステップ網羅率 C_0' は 80 %，全パス網羅率 C_∞' は 50 %となる．

3.3 実行パスの特定

実行パスの識別手順を**図 6** に示す． 図 1 のユースケース群を例として用いる．ここでは,「部屋を予約する」ユースケースのテストシナリオを出力するものとする．入力されたユースケース記述から，ユースケースの実行パスのグラフを作成する．

(1) もし，汎化しているユースケースが存在すれば，

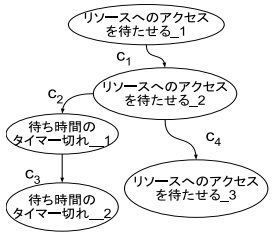

図 5 「リソースへのアクセスを待たせる」ユースケースの実行可能パス
Fig. 5 Execution path of 「Wait for the access to the resource」use case

汎化しているユースケースの実行パスに特化しているユースケースの実行パスをユースケース・ステップ単位で上書きする．この例における，汎化による変化が終わった後のグラフを**図 7** に示す．灰色の枠は汎化により変化した部分である． 結果,「施設を予約する」ユースケースで未定義であった「設備を選択する」フローが,「部屋を予約する」ユースケースに定義されている「設備を選択する」フローに置き換わる．

(2) もし，包含するユースケースが存在すれば，包含対象を参照するステップを包含されるユースケースの実行パスで上書きする．包含により変化したグラフ例を**図 8** に示す．包含により変化した部分は灰色の枠の部分である． これにより,「ログインする」ユースケースの「入力され

```
作成対象の UC のみに作成されているパスを作成;
if(汎化関係がある){
  汎化している UC の実行パスと作成対象
の UC の実行パスの差分をステップ単位で上書き;
}
if(包含関係がある){
  包含対象を参照しているステップを
    包含対象の UC の実行パスで上書き;
}
if(拡張関係がある){
  拡張する側の UC の実行パスを拡張点に追加;
}
```

図 6 実行パスの識別手順
Fig. 6 Discrimination procedure of execution pass

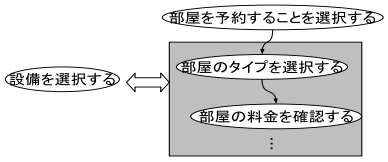

図 7 汎化によるパスの変化
Fig. 7 Execution path changed by Generalization

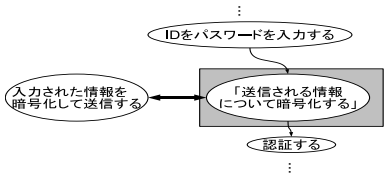

図 8 包含によるパスの変化
Fig. 8 Execution path changed by Include

た情報を暗号化して送信する」ステップが,「ログインする」ユースケースが参照していた「暗号化する」ユースケースに定義されている「暗号化する」フローに置き換わる.

(3) もし，拡張するユースケースが存在すれば拡張するユースケースの実行パスを拡張ポイントカットに指定されているステップに追加する．拡張による変化が終わった後のグラフを**図 9** に示す．拡張により変化した部分は灰色の枠の部分である． これにより「盗聴する」の拡張点に指定されていた「ログインする」ユースケースの基本フローの「入力された情報を暗号化して送信する」の後に「盗聴する」フローが追加される．同様に「ログインする」の拡張点に指定されていた「施設を予約する」ユースケースの基本フローの「利用可能な設備を表示する」の前に「ログインする」フローが追加される．

なお，脅威の関係は UML2.0 で定義されている拡張

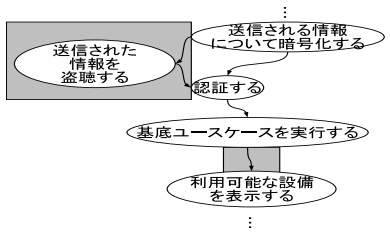

図 9 拡張によるパスの変化
Fig. 9 Execution path changed by Extend

部屋を予約する		
部屋を予約する ()	**部屋**	予約番号を表示し終了する ()
Reserve	*101*	943789
Reserve	*201*	984547

表 1 生成された「部屋を予約する」ユースケースに対応するテストシナリオファイル (その 1)
Table 1 Generated test scenario 1 about 「Reserve Room」use case

の特化版として解釈できるので[6]，実行パスの特定時には脅威の関係は拡張の関係として扱う．また，緩和は脅威の影響が緩和されたことを表し，実行パスに影響を与えないので，実行パスの特定時には緩和の関係は存在しないものとして扱う．

3.4 テストシナリオの生成

3.3 節により作成された実行パスからテストシナリオを出力する．テストシナリオの出力は次の様にする．

- テスト項目の行に，ステップを実行順に記述する．
- その際，ステップの中にドメインモデルに含まれるクラスの名前が記述されていれば，含まれるクラス名のうちシステムの利用者が指定したクラス名をテスト項目の列内の該当するステップの前に記述する．例えば，ステップが「顧客は部屋を選択する」の場合，ドメインモデルのクラス名の中に「顧客」,「部屋」の二つがあれば,「顧客」と「部屋」の二つがテスト項目の候補になる．このうち，システムの利用者が「部屋」のみを選択すると,「部屋」と「顧客は部屋を選択する」の二つがテスト項目の行に記述される．
- 選択された網羅性を満たす必要最低限の数のテストシナリオ群を生成する．例えば，提案手法のシステムへの入力時にステップ網羅を選択すれば，出力されるテストシナリオ群は各ステップが少なくとも一回はテストされる最低限の数だけテストシナリオ群を出力する．

実際に全パス網羅を指定して得られたテストシナリオの例を**表 1**, **表 2** に示す. 入力したドメインモデル中のクラスは,「部屋」,「顧客」,「設備」,「予約」である.

部屋を予約する		
部屋を予約する ()	**部屋**	予約を作成せずに終了する ()
Reserve	*101*	error
Reserve	*201*	error

表 2 生成された「部屋を予約する」ユースケースに対応するテストシナリオファイル (その 2)
Table 2 Generated test scenario 1 about 「Reserve Room」use case

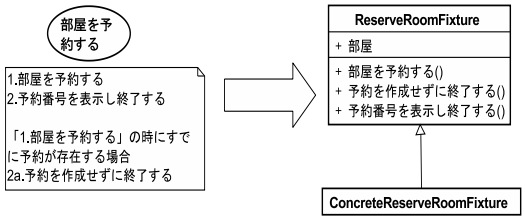

図 10 フィクスチャの生成例
Fig. 10 Example of generated fixture

3.5 フィクスチャの生成

フィクスチャの生成例を**図 10** に示す．ユースケース A に対し出力するとき,

- AFixture という名前のフィクスチャのみ
- AFixture という名前のフィクスチャとそれを継承する ConcreteAFixture という名前のフィクスチャ

のどちらかをシステムの利用者が指定して生成する．開発者は，ConcreteAFixture に対して実装する．選択式にしたのは，開発者が加えたフィクスチャの詳細をユースケースが変更となりフィクスチャを再生成する時に上書きされてしまうのを防ぐためである．また，ユースケース A に定義されている一つのユースケース・ステップにつき，ユースケース A に対応する親クラスとなるフィクスチャAFixture は該当のユースケース・ステップと同名の操作を所有する．同様に，ユースケース A のユースケース記述の中にドメインモデル中のクラス名を含むならば，ユースケース A に対応する親クラスとなるフィクスチャAFixture は含まれるクラスを属性として所有する． 図 10 内の ReserveRoomFixture のコード断片を**図 11** に示す.

図 10 では「部屋を予約する」ユースケースにステップ「部屋を予約する」,「予約番号を表示し終了する」,「予約を作成せずに終了する」があるので ReserveRoomFixture は「部屋を予約する」,「予約番号を表示し終了する」,「予約を作成せずに終了する」という名前の操作を所有し，入力されたドメインモデル内に「部屋」があれば AFixture は「部屋」を属性として所有する．

図 11 では，表 1 内の「部屋を予約する ()」の列が，図 10 の「部屋を予約する ()」メソッドに対応し，表 1 内の「部屋」の列が，図 10 の「部屋」の変数に対応している．表 1 の 3 行目では，順番に,「部屋を予約する ()」メソッドを実行してその結果が「Reserve」と一致するか,「部屋」に 101 を代入し,「予約番号を表示し終了する ()」を実行しその結果が「943789」と一致するかテストすることを表す.

```
public class ReserveRoomFixture
                    extends ColumnFixture{
    public 部屋 部屋;
    public String 部屋を予約する () {
        return ”pass”;
    }
    …
```

図 11 「部屋を予約する」に対応するフィクスチャの一部
Fig. 11 Code of ReserveRoomFixture

```
if(包含関係がある) {
  包含対象の参照と包含先の操作と
    同名の操作を追加するアスペクトを作成；
}
if(汎化関係がある) {
  該当するフィクスチャ間を
    継承関係にするアスペクトを作成;
}
if(拡張関係がある) {
  拡張する側への参照と拡張する側の操作と
    同名の操作を追加するアスペクトを作成;
}
```

図 12 アスペクト作成の手順
Fig. 12 Procedure of making aspect

ユースケース間の関係については，該当するフィクスチャ間の関係を，アスペクト指向プログラミング (横断的関心事を一つのモジュールとして分割する技術[9]) によるインタータイプ宣言 (他のクラスが持つべきメソッド・フィールドを記述する機構[9]) で定義する．ユースケース間の関係をアスペクトで定義することにより，ユースケース間の関係が変化した場合の修正が，アスペクトの変更のみで対応でき，フィクスチャ自体は変更をしなくてすむため，フィクスチャの再利用性が高まる．本稿では AspectJ 形式のアスペクトを生成する．アスペクト作成の手順を**図 12** に示す.

包含 ユースケース間が包含関係であれば，包含する側のユースケースに対応するフィクスチャに対し，包含される側のユースケースに対応するフィクスチャを参照し，包含される側のユースケースに対応するフィクスチャの操作と同名の操作を持つように追加するアスペクトを生成する．つまり，ユースケース A がユースケース B を包含し

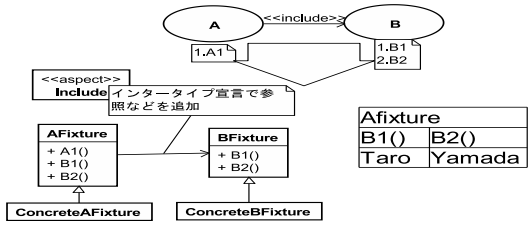

図 13 包含におけるフィクスチャの出力例
Fig. 13 Example of fixture about include

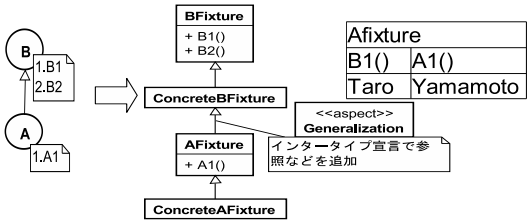

図 14 汎化におけるフィクスチャの出力例
Fig. 14 Example of fixture about generalization

ている時，出力されるフィクスチャのクラス構造は**図 13** のようになる．AFixture に，B1，B2 の操作と，BFixture への参照がアスペクトによるインタータイプ宣言で追加されている．

継承 ユースケース間が汎化関係であれば，該当するフィクスチャ間を継承関係にするアスペクトを生成する．つまり，ユースケース A がユースケース B を特化するとき，**図 14** のようなフィクスチャが出力される．ConcreteBFixture と AFixture との間に，継承関係がアスペクトによるインタータイプ宣言で追加されている．

拡張 ユースケース間が拡張関係であれば，拡張される側のユースケースに対応するフィクスチャに対し，拡張する側のユースケースに対応するフィクスチャを参照し，拡張する側のユースケースに対応するフィクスチャの操作と同名の操作を持つように追加するアスペクトを生成する．つまり，ユースケース A がユースケース B に拡張されている時，出力されるフィクスチャのクラス構造は**図 15** のようになる．AFixture に，B1，B2 の操作と，BFixture への参照がアスペクトによるインタータイプ宣言を追加されている．

以上のルールにより，図 1 のユースケース群に対し生成されたフィクスチャ群を**図 16** に示す．

3.6 提案手法の限界

提案手法だけでは，受け入れテストは十分ではない．

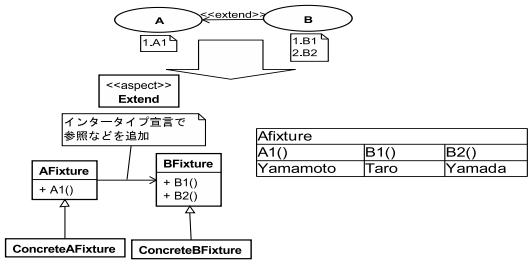

図 15 拡張におけるフィクスチャの出力例
Fig. 15 Example of fixture about extend

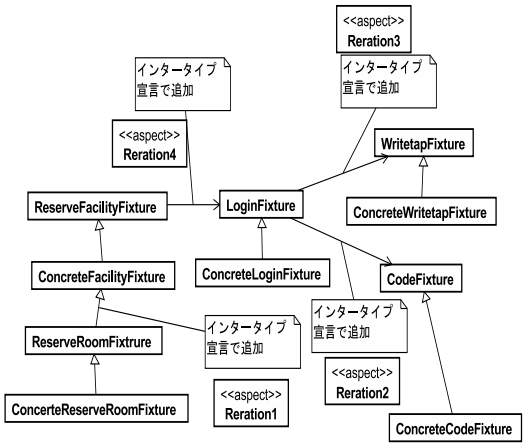

図 16 生成されたフィクスチャのクラス図
Fig. 16 Class diagram of Fixture generated by our system

提案手法はステップの網羅に着目するためフローに記述されない特性についてはテストできない．例えば，ユーザビリティに関しては開発するプロダクトを実際に利用者が操作しないと十分にテストできない．

また，提案手法は出力されたテストシナリオ群とテストフィクスチャ群に対し，人手で詳細化する必要があるため，この工程において誤りが入る可能性がある．

4. 実　　験

従来手法[7] と提案手法の実験を行った．被験者は UML の経験が約 2 年の学生三名である．システムの利用環境は一般のノート PC である．図 1 のユースケース図とユースケース記述を被験者に与えて,「部屋を予約する」ユースケースの実行されうる全ての実行パスについて識別を行う．その際に実行パスの誤りや漏れの数を比較した．最初に従来手法による実行パスの識別および実行パスの列挙を行う．その後に提案手法のシステムへ入力しテストシナリオ群を出力する．

誤りや漏れの数であるが，従来手法の場合，被験者

C のみが, 誤った実行パスを 6 個あげた．他の被験者, および，被験者 C において提案手法を用いた場合には誤りはなかった．従来手法において，被験者 C は「ログインする」ユースケースのどのステップが「暗号化する」ユースケースを参照するステップなのかを誤って挙げた.「ログインする」ユースケースのどのステップが「暗号化する」ユースケースを参照するのかを読み間違えたのが原因と考えられる．提案手法ではユースケース間の関係による実行パスの変化を自動で識別するため，提案手法では誤りや漏れがなくなる．

この実験から，人手で識別する際に実行パスの識別に誤りが入る可能性があることを確認できた．提案手法を用いることで実行パスの漏れや誤りがなくなり, 受け入れテストの網羅性が向上した.

5. 関 連 研 究

Zielczynski は，ユースケースからテストケースへの追跡可能性について述べた[7]．Zielczynski の手法では, ユースケースからすべてのシナリオを人手で確認し, その中から妥当なテストシナリオを選択している．人手で識別するため，実行パスの識別に漏れがでるおそれがある．また，ユースケース間の関係には考慮していない．一方，提案手法では実行パスの識別を自動で行うため漏れがなくなる．ユースケース間の関係を考慮した網羅的なテストシナリオを作成する.

Amyot らは，ユースケースマップ駆動の WEB アプリケーションのテストを提案した[10]．Amyot らの手法は，ユースケースマップ (シナリオをモデリングするための記法) を用いて受け入れテストの対象となる実行パスを識別しテストシナリオの識別の正確性を向上する手法である．しかし，複数のユースケースマップ間の関係を考慮しておらず網羅的なテストシナリオを作成しにくい．一方，提案手法ではユースケース間の関係を考慮した網羅的なテストシナリオを作成できる.

Nebut らは，ユースケース駆動の自動テスト生成について述べた[11]．Nebut らの手法は，ユースケース内の事前・事後条件に形式的な制約を追加してテストシナリオを自動生成する．形式記述を利用してテストシナリオを自動生成できるが，一般に形式記述はコストが高い．また，この手法は事前・事後条件を用いた網羅的なテストシナリオを生成する．一方，提案手法は非形式的記述を適用し，また，複数のユースケース間の関係を考慮するが，事前・事後条件は考慮していない.

6. お わ り に

本稿では，ユースケースから識別されたテストシナリオに基づく受け入れテストに対する 3 種類の網羅性を定義した．また，ユースケース記述とドメインモデルを入力として，指定された網羅性を満たす FIT 形式のテストシナリオとテストプログラムの雛形を自動生成する手法を提案した．提案手法と従来手法について, 小規模な実験を行い，提案手法により，実行パスの漏れがなくなったことを確認した．提案手法を用いることで，受け入れテストの網羅性の定義を用いて，属人性を排してテスト終了を判定できる．FIT 形式のテストシナリオとテストプログラムの雛形を自動生成することにより，受け入れテストの効率の向上が見込める.

今後の課題として，サンプル数を増やした追加実験や Java 以外の言語対応，実行パスだけでなくテストケースの生成などがあげられる.

参 考 文 献

1) Boris Beizer 著, 小野間彰ほか訳，”ソフトウェアテスト技法，” 日経 BP 出版センター, 1994.
2) Mugridge, R., Cunningham, W.，”Fit for Developing Software: Framework for Integrated Tests，” Prentice Hall, 2005.
3) Alister Cockburn 著, ウルシステムズ株式会社訳，”ユースケース実践ガイド―効果的なユースケースの書き方，” 翔泳社, 2001.
4) Object Management Group,Unifed Modeling Language, http://www.uml.org/.
5) Alexander, I., ”Misuse Cases: Use Cases with Hostile Intent,” IEEE Software, Vol. 20, No. 1, pp.p 58-66, January/February 2003
6) Sindre, G., Opdahl, A. L., ”Eliciting security requirements with misuse cases. ” Requirements Engineering, Vol. 10, No. 1, pp. 34 - 44, Jan. 2005
7) Zielczynski, P., ”Traceability from Use Cases to Test Cases,” developerWorks, http://www.ibm.com/developerworks/jp/opensource/rational/library/04/r-3217/, 2006
8) Jim Heumann, ”Generating Test Cases From Use Cases,” Rational edge, 2001, http://www.ibm.com/developerworks/rational/library/content/RationalEdge/jun01/GeneratingTestCasesFromUseCasesJune01.pdf.
9) 長瀬嘉秀, 天野まさひろ, 鷲崎弘宜, 立堀道昭: ”AspectJ によるアスペクト指向プログラミング入門 ”, ソフトバンクパブリッシング, 2004.
10) Amyot, D. et al., Proceeding of UCM-Driven Testing of Web Applications, 12th SDL Forum (SDL 2005), LNCS 3530, Springer, 247-264, 2005.
11) Nebut, C. et al., ”Automatic Test Generation: A Use Case Driven Approach,” IEEE Transactions on Software Engineering, Vol.32, No.3, 2006.