Vol.2017-SE-195 No.28
2017/3/13

画面操作を伴うテストにおける テストスクリプト中のロケータ自動修正手法の提案

切貫 弘之[1,a)]　丹野 治門[1,b)]　岩田 真治[1,c)]　夏川 勝行[1,d)]

概要：年々増加するビジネススピードに対応するため，ソフトウェアのリリースサイクルを短縮することが求められている．リリースサイクルを短縮するための方法として，テストの自動化は欠かせないものとなってきている．画面操作を伴うテストを自動化するためのツールとして Selenium，Appium といったツールが広く用いられている．しかし，これらのツールを用いて実装されたスクリプトは，ソフトウェアの変更に伴って修正を行うことが求められる．このテストスクリプトの修正にかかるコストはテスト自動化において大きな問題となっている．本研究では，画面の HTML ソースコード等のテスト実行時に得られる情報のみを用い，テストスクリプトを自動的に修正する手法を提案する．OSS を用いて提案手法を評価した結果，提案手法が様々なソフトウェアにおいて，テストスクリプトを高精度で修正可能であることが分かった．また，被験者実験を行い，提案手法がロケータの修正にかかる時間を約 93%削減できることを示した．

Automated Repair of Locators in GUI Test Script

Hiroyuki Kirinuki[1,a)]　Haruto Tannno[1,b)]　Sinji Iwata[1,c)]　Katsuyuki Natsukawa[1,d)]

1. はじめに

近年，ビジネスにおけるスピードの重要性がますます増加しており，ソフトウェア開発もそれに合わせて高速化することを余儀なくされている．このようなソフトウェア開発においては，1 度目のリリースを行った後も，必要に応じて素早くサービスを改良してリリースを行うことが求められる．

ソフトウェア開発では，リリースを行う前にソフトウェアが適切に動作していることを確認するためのテストを行う．テストでは，新規に追加した機能が適切に動作するかどうかを確認するだけではなく，既存の機能が今まで通りに動作するかどうかも確認する必要がある．既存の機能が今まで通りに動作するかどうかを確認するためのテストを**回帰テスト**と呼ぶ．

[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 NSS2 ビル 6F
[a)] kirinuki.hiroyuki@lab.ntt.co.jp
[b)] tanno.haruto@lab.ntt.co.jp
[c)] iwata_shinji_s5@lab.ntt.co.jp
[d)] natsukawa.katsuyuki@lab.ntt.co.jp

リリースサイクルを短縮する上で回帰テストにかかる稼働を削減することは大きな課題である．拡張・改修案件の場合，回帰テストが占める稼働は，開発にかかる稼働全体の 25%であると言われている [1], [2]．リリースサイクルの短縮を目指す場合，短期間で何度も回帰テストを行う必要があるため，開発全体において回帰テストの稼働が占める割合が大きくなる．ビジネススピードが増加している昨今，回帰テストに多くの稼働をかけることは，ビジネスチャンスを逃すことにつながる恐れがある．

回帰テストにかかる稼働を削減するための取り組みとして，テストの自動化が広く行われている．しかし，実際に画面を操作して動作を確認する結合テストやシステムテストを自動化するためのツール（Selenium[3], Appium 等）は単体テスト自動化ツールと比べると利用が進んでおらず，その利用率はわずか 12%になっている [1].

画面操作を伴うテストの自動化を妨げる大きな要因として，テストスクリプトの保守にコストがかかることが挙げられる．テスト対象のソフトウェアを修正する場合，既存のテストスクリプトにも修正が必要となることが多い．テ

ストスクリプトに対し頻繁に修正が行われる開発では，テストを自動化することで余計にコストがかかってしまう恐れもある．そのようなことが起こると，開発現場におけるテスト自動化へのモチベーションの低下につながる．

Leotta らは，実際にテストスクリプトの修正がどの程度発生するのかについて調べている [4]．彼らは，まず，6 つのドメインが異なる OSS について，アプリケーションのあるバージョンをテストするためのテストスクリプトを作成した．そして，8 ヶ月以上期間をおいた後のあるバージョンに対してテストが実行される場合に，どの程度テストスクリプトに修正が必要になるのかを調査した．その結果，91.8%(180/196) ものテストスクリプトに修正が必要となることが分かった．このことから，テストスクリプトの修正は開発において，多く発生することが分かる．

テストスクリプトの修正を支援するために，テストスクリプト修正の自動化を目指した研究が行われている [5], [6], [7]．もし，テストスクリプトを自動で修正することができれば，テストスクリプトの保守にかかるコストを大きく削減することができる．

テストスクリプトの修正の自動化は，ソフトウェアの機能全体におけるテストの自動化率の向上にもつながる．実際の開発では，今後の仕様の変更が予想される画面や機能については，テストを自動化しないことが多い．なぜなら，テスト対象のソフトウェアだけでなくテストスクリプトも今後修正される可能性が高く，自動化のコストに対してメリットが見合わないためである．テストスクリプトの保守が容易になることで，より多くの画面や機能のテストを自動化することができる．

本研究では，テスト対象アプリケーションの画面から得られる様々な情報を用いて，ロケータを自動的に修正する手法を提案する．ロケータとは，テストスクリプトにおいて操作する画面要素を指定するための識別子のことである．本研究は，現在最も開発が盛んな Web アプリケーションにおける回帰テストを対象としている．提案手法を用いることで，テストスクリプト保守コストの削減が期待できる．

提案手法の有効性を確認するために，OSS を用いて評価を行った．その結果，提案手法が様々なソフトウェアにおいて，テストスクリプトを高精度で修正可能であることが分かった．また，提案手法により，ロケータの修正にかかる作業時間を約 93%削減することができた．

2. テストスクリプト

本研究が対象とするテストスクリプトとして，Selenium IDE で実装されたものを採用した．Selenium IDE は，ユーザが画面に対して行った操作をスクリプトとして記録し，以降その操作を再現することができる．Selenium IDE はプログラミング経験が無いユーザでも利用可能であり，開発現場でも広く用いられている．

図 1　航空券予約システムのログインフォーム

2.1 具体例

例として，図 1 のようなログインフォームを持つ Web アプリケーション（航空券予約システム）を考える．航空券予約システムにログインするためには，「お客様番号」と「パスワード」を入力して「ログイン」ボタンを押すという操作が必要である．この操作を Selenium IDE のスクリプトとして実装すると表 1 のようになる．テストスクリプトの 1 行を 1 つの**操作**と定義する．

表 1 の見方について説明する．操作は**命令・ロケータ・値**の 3 つの要素から成る．**命令**は操作の種類を表し，例えば「type」はキーボードからの入力を表す．**ロケータ**について，2 行目の「id=customerNo」は HTML 中で id が「customerNo」である画面要素（お客様番号入力フォーム）を指す．Selenium IDE が利用できるロケータとして，id, name, css, XPath 等が存在する．スクリプト実装時には指定された優先順に従って，いずれか 1 つのロケータが採用される．**値**はユーザが与える入力値もしくは期待結果を表し，2 行目ではお客様番号への入力として「user01」を与えている．また，5 行目では期待結果として，メインページに遷移することを与えている．表 1 のスクリプトを実行すると，1 行目から操作が逐次実行され，最後まで完了すればテストに合格したと判定される．

アプリケーションの次のバージョンで，「お客様番号」の id が可読性向上のために「customerNumber」に変更されたとする．HTML 中の id が変わっても，ログイン機能自体に影響を及ぼさないため，テスト時に行うべき操作はアプリケーションの修正前後で変わらない．しかしこのとき，修正されたアプリケーションに対して表 1 のスクリプトを実行すると，2 行目のロケータが見つからず実行不可能となる．このような場合，テスト担当者はテストを実行するために，テストスクリプトに修正を施す必要がある．この場合は 2 行目のロケータを「id=customerNumber」に変更すればテストスクリプトが実行可能になる．

このように，テスト対象のソフトウェアに軽微な修正を行うだけで，テストスクリプトにも修正が必要になる場合がある．

表 1　ログインを行う Selenium IDE スクリプト

#	命令	ロケータ	値
1	open	http://localhost:8080/atrs	
2	type	id=customerNo	user01
3	type	id=password	pass01
4	click	css=button.button	
5	assertTitle		Main

2.2 修正が必要なパターン

2.1 節で示した例のように，アプリケーションの修正によって，既存のテストスクリプトに修正を施す必要が生じる場合がある．この状態をテストスクリプトが誤りを含んでいる状態とする．テストスクリプトの誤りは**ロケータの誤り**と**論理の誤り**の 2 種類に分類できる．

ロケータの誤りとは，2.1 節の例のようにテストにおいて操作すべき画面要素がロケータによって指定されていない誤りを指す．これらは，アプリケーション画面のレイアウト変更や画面要素の属性の変更により起こることが多い．ロケータの誤りを修正するためには，操作すべき画面要素を参照するように，ロケータを正しく設定し直す必要がある．

論理の誤りとは，テストにおいて行うべき操作自体が異なっている，もしくは入力として与える値が異なっている誤りを指す．例えば，ユーザ登録時に確認のチェックボックスをクリックさせるような修正をアプリケーションに行った場合，テストスクリプトにもチェックボックスをクリックする操作を追加する必要がある．論理の誤りは，アプリケーションの仕様変更や機能追加により起こることが多い．論理の誤りを修正するためには，テストシナリオを理解し，テストスクリプト中の適切な位置に操作を追加および削除する，もしくは適切な値を入力する必要がある．

これらの 2 種類の誤りのうち，ロケータの誤りが全体の約 74%を占めている [8]．したがって，本研究においては，より自動修正した場合の効果が高いロケータの誤りに着目した．

3. 既存研究

ロケータの誤りの自動修正を目指した研究として，Choudhary らの研究がある [6]．Choudhary らは XPath のレーベンシュタイン距離を用いて，ロケータを自動的に修正する手法を提案している．XPath は HTML における特定の画面要素の位置を表す．Choudhary らは，アプリケーションの修正前後で，XPath のレーベンシュタイン距離が近いものが同一の画面要素であるとみなしてロケータの修正を行っている．この手法の問題点として，テスト対象のソフトウェアの画面のレイアウト変更に弱いことが挙げられる．画面のレイアウトが変更されると，画面要素の XPath が大きく変わってしまう場合がある．そのような場合，この手法を用いても正しくロケータを修正することができない恐れがある．

同様の研究として，Leotta らの研究がある [7]．Leotta らは，XPath を含む類似した 5 種類の位置情報を指標として用いて，ロケータを自動的に修正する手法を提案している．この手法の問題点として，アプリケーションの修正前後でどの指標が一致したかのみを評価しており，一致していないが近いといったことは評価できない点がある．また，5 つの指標全てが位置情報であるため，Choudhary らの手法と同様の問題を抱えている．

アプリケーションの変更に対し，テストスクリプトを自動修正するのではなく，最初から変更に強いテストスクリプトを作成することを目的とする研究も存在する．

Leotta らは，従来の DOM ベースのテストスクリプトを画像ベースに変換する手法を提案している [9]．画像ベースのテストとは，OSS の Sikuli[10] 等，画像処理を用いて，特定の画像と一致，もしくは類似する対象を操作する手法のことである．画像ベースのテストの利点として，ロケータが変更されても画像さえ変わらなければテストスクリプトに修正が不要である点，操作対象が画像で表示できるため，テストスクリプトの理解性が高い点が挙げられる．しかし，これについても，画像が変わってしまうとテストスクリプトに修正が必要となってしまうため，デザインの変更が多く行われるプロジェクトでは有効ではない．

Yandrapally らは，ラベルを用いて操作対象を指定することで，自然言語に近い形でテストスクリプトを記述する方法を提案している [11]．画面要素のラベルは属性や画像より変更されにくいという仮定のもと，その耐変更性の高さを主張している．この手法の問題点として，必ずしも操作対象の画面要素にラベルがあるわけではないということが挙げられる．例えば，画像で作成されたボタンや JavaScript によるポップアップ等はこの手法のみで対応することが難しい．

4. 提案手法

提案手法は，位置情報だけでなく属性・テキスト・画像も手がかりとして用い，同一の画面要素であるかどうかを多面的に判断することで，既存研究の弱点を克服し，よりアプリケーションの変更に強い自動修正技術を実現する．

提案手法の全体像を図 2 に示す．提案手法は，修正が必要なテストスクリプトを入力とし，修正を試みる．修正の結果，テストに通過すれば修正に成功したとみなし，修正後のスクリプトを出力する．修正したテストスクリプトがテストに通過しない場合，テストに通過するまで，テストスクリプトを修正しつつ試行を繰り返す．また，テストスクリプトの途中にアサーションが存在する場合は，そのアサーションの条件を満たしていれば，その時点までのテストスクリプトは修正に成功したとみなす．設定した時間内にテストに通過しなければ，可能な範囲で修正を行ったテストスクリプトを出力する．出力されたテストスクリプトについては，ユーザが適宜レビューを行い，必要があればユーザが修正を行う．

提案手法は新旧アプリケーションの実行環境を必要とする．新旧アプリケーションに対してテストを実行することで，レンダリングされた画面から情報を取得し，自動修正の手がかりとする．具体的な手順を以下で説明する．

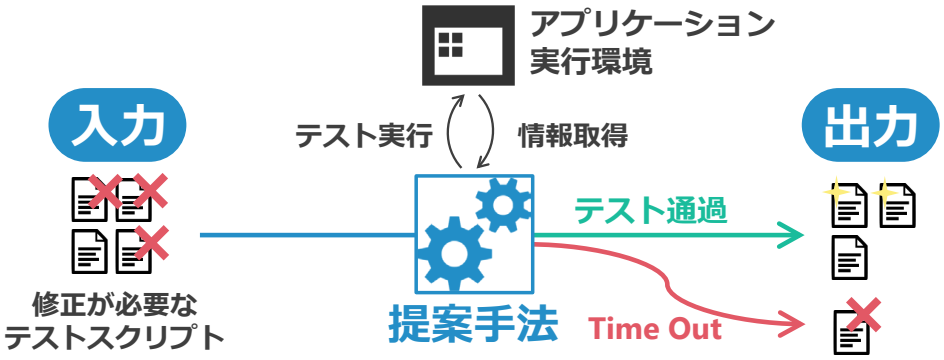
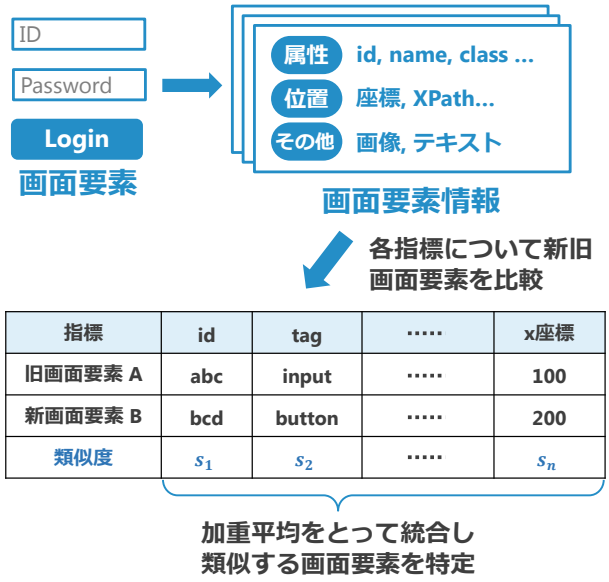


図 2　提案手法の全体像



指標	id	tag	·····	x座標
旧画面要素 A	abc	input	·····	100
新画面要素 B	bcd	button	·····	200
類似度	s_1	s_2	·····	s_n

図 3　類似度算出アルゴリズム

ロケータを修正するためには，そのテストで本来操作されるべき画面要素を特定する必要がある．ここで，テスト中に操作される画面要素 e を指すようなロケータを l_e と定義する．テスト実行時，l_e が修正後のアプリケーションにおいて画面要素を一意に特定できない場合，操作は実行できない．修正後のアプリケーション画面に存在する画面要素の集合を E とする．ここで，e と同一の画面要素 $e' \in E$ が存在する場合，それが本来操作されるべき画面要素であると考えられる．e' を特定し，ロケータ l_e を $l_{e'}$ に変更することで，修正が完了する．

本研究では，修正前後の画面要素について，画面要素が持つ複数の指標を用いて類似度を定量的に算出することで，e と同一の画面要素 $e' \in E$ を推定する．より類似度が高いものを優先的に修正の候補とすることで，少ない試行回数でロケータの修正を行う．用いる指標は以下の 4 つに分類される．

- 属性 (id・class・ name 等)
- 位置 (XPath・座標・サイズ)
- テキスト (リンクテキスト・近傍のテキスト)
- 画像 (レンダリングされた画面要素の画像)

これらの指標の情報は，テスト実行と同時に取得する．修正前のアプリケーションに対して，あらかじめテストを実行し，情報を取得しておく．類似度算出アルゴリズムの概略を図 3 に示す．

e と e' について，まず，各指標における類似度を算出し，次にそれらを統合して 1 つの類似度とする．e と e' が共通して持つ指標の集合を I，e における $i \in I$ の値を i_e とする．ここで，$Max(i)$ は $|i_e - i_{e'}|$ がとりうる最大値，$Lev(i_e, i_{e'})$ は i_e と $i_{e'}$ のレーベンシュタイン距離，$MaxLength(i_e, i_{e'})$ は i_e と $i_{e'}$ の文字列の長い方の長さとすると，i_e と $i_{e'}$ の類似度 $s_i (0 \leq s_i \leq 1)$ は以下のように算出される．

i_e が数値の場合:

$$s_i = 1 - \frac{|i_e - i_{e'}|}{Max(i)}$$

i_e が画像または特定の文字列 (タグ名など) の場合:

$$s_i = \begin{cases} 1 & (i_e = i_{e'}) \\ 0 & (otherwise) \end{cases}$$

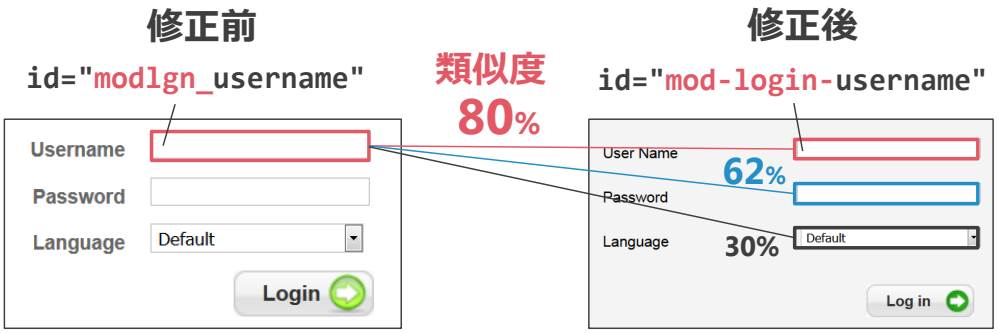
i_e が任意の文字列の場合:

$$s_i = 1 - \frac{Lev(i_e, i_{e'})}{MaxLength(i_e, i_{e'})}$$

各指標について算出された s_i を統合するにあたって，それぞれ画面要素の類似度に寄与する度合いが異なると考えられる．例えば，異なる画面要素が同じ id を持つことはないが，異なる画面要素が同じ class を持つことは可能であるため，class の一致は id の一致に比べて類似度に寄与する度合いが小さいと考えられる．これを考慮し，各指標について，s_i を算出した後，重み w_i で加重平均を取ることで e と e' の類似度 $S (0 \leq S \leq 1)$ を算出する．

$$S = \frac{\sum_{i \in I} s_i w_i}{\sum_{i \in I} w_i}$$

より S が高い e' から順に修正候補とし，l_e を $l_{e'}$ に置き換えてテストを実行する．設定した時間内で，テストに通るまで試行を繰り返し，テストに通れば修正が成功したとみなす．



命令	ロケータ	値
type	**id=modlgn_username**	user01
type	id=password	pass01
click	id=login	

修正

命令	ロケータ	値
type	**id=mod-login-username**	user01
type	id=password	pass01
click	id=login	

図 4　修正のイメージ

修正のイメージを図 4 に示す．図 4 の例では，修正前後で Username 入力フォームの id が変更されており，修正前のテストスクリプトは修正後のアプリケーションに対して実行することができない．なぜなら，テストスクリプトの 1 行目のロケータが指す画面要素が修正後の画面には存在しないためである．

提案手法では，まず，修正前の Username 入力フォームと修正後の各画面要素の類似度を計算する．その結果，修正後において，id が「mod-login-username」である画面要素が最も類似度が高くなった．したがって，id が「mod-login-username」となる画面要素が，修正後における Username 入力フォームであると推測できる．そこで，テストスクリプトの 1 行目のロケータを「mod-login-username」に変更する．ここで修正したテストスクリプトがテストに通過すれば，修正に成功したことになる．

5. 評価実験

提案手法について，精度と稼働の 2 つの観点で評価を行った．評価には提案手法を実装したプロトタイプを用いた．提案手法で用いる指標とその重み付けについて，今回は著者の手で任意の 20 種類の指標を選択し，任意の重みを設定した．

実験対象として，表 2 に示す 4 つのソフトウェアを採用した．これらはオープンソースの Web アプリケーションであり，PHP で実装されている．各ソフトウェアごとに 2 つのバージョンを用意し，1st Release のバージョンを修正前，2nd Release のバージョンを修正後とみなす．実験対象のソフトウェアは，2.2 節で述べた Leotta らの研究 [4] で用いられているものの一部である．これらを採用した理由は，アプリケーションの画面に多くのレイアウト変更が行われており，かつそれぞれドメインが異なっているためである．

5.1 評価方法

5.1.1 精度評価

提案手法により，ロケータの誤りをどの程度自動的に修正できるかを調べる．これにより，提案手法が様々なドメインのアプリケーションに有効かどうか，提案手法が画面のレイアウト変更を含むアプリケーションに有効かどうかを評価する．

精度評価の手順について述べる．まず，評価対象ソフトウェアの 1st Release のバージョンにおいて，主要な画面に存在する全ての操作可能な画面要素 (ボタン・入力フォーム・リンク) 計 375 個についてロケータを取得し，2nd Release のバージョンにおいてもそれが同一の画面要素を参照するかを調べた．結果の偏りを減らすため，表形式等，同じ役割の画面要素が複数存在する場合はこれらを 1 つと数えた．

ここで，取得したロケータが 2nd Release のバージョンにおいても同一の画面要素を参照しなかった場合，2nd Release の時点ではそのロケータは誤っていると言える．誤ったロケータに対し提案手法を適用すると，2nd Release のバージョン中の画面要素が類似度の高いものから順に修正候補として提示される．この際，正解の画面要素が何位に提示されるかを調べた．

5.1.2 稼働評価

提案手法を用いることで，テストスクリプト修正の稼働を実際にどの程度削減することができるのかを評価した．6 名の被験者に誤りを含むテストスクリプトを修正させ，**「Case1: 従来手法を用いる場合」**と**「Case2: 提案手法を用いる場合」**の稼働を比較した．従来手法として，開発現場で広く用いられている Selenium IDE を用いて手作業で

表 2　実験対象のソフトウェア

Apps	Description	1st Release				2nd Release			
		Release	Date	File	kLOC	Release	Date	File	kLOC
MantisBT	バグトラッキングシステム	1.1.8	Jun-09	492	90	1.2.0	Feb-10	733	115
PPMA	パスワード管理システム	0.2	Mar-11	93	4	0.3.5.1	Jan-13	108	5
MRBS	会議室予約システム	1.10.7	Dec-11	840	277	1.11.5	Feb-13	835	285
Collabtive	プロジェクト管理システム	0.65	Aug-10	148	68	1.0	Mar-13	151	73

ロケータを修正する方法を採用した．Selenium IDE には画面要素をクリックすることでロケータを取得する機能があり，スクリプト実装および修正の負担を減らしている．

6 名の被験者は，いずれも実務での Selenium IDE の使用経験は無かった．実験を円滑に行うため，Selenium IDE の使用方法や実験の手順については事前にレクチャーを行った．

準備として，まず，各ソフトウェアの 1st Release のバージョンにおいて，アプリケーションの主要機能をテストするための Selenium IDE スクリプトを作成した．次に，作成したテストスクリプトを 2nd Release のバージョンおいて実行したところ，それぞれのソフトウェアでテストに通らないスクリプトが複数存在した．テストに通らないスクリプトを各ソフトウェアから無作為に 2 つずつ選出し，テストに通らないスクリプト A〜H を用意した．これらについて，ロケータの誤りの自動修正の効果を測るため，論理の誤りについては事前に著者の手で取り除いた．次に，稼働の測定方法について説明する．

Case1: 従来手法を用いる場合　被験者は，事前に自然言語で書かれたテストケースの説明を読んで内容を理解しておく．テストケースを理解した時点で作業開始とする．従来手法でロケータを修正し，2nd Release のバージョンでテストに通れば作業完了とする．このとき，被験者が作業を行うのにかかった時間を測定した．ここで，Case1 と Case2 で 1 人の被験者が同じテストスクリプトを修正することがないよう，Case1 では 6 名の被験者のうち 3 名がスクリプト A〜D を修正し，残り 3 名の被験者がテストスクリプト E〜H を修正することとした．

Case2: 提案手法を用いる場合　まず，事前に提案手法を適用し，テストスクリプトの自動修正を試みた．このとき，提案手法で 3 分以内に修正できない場合は試行を打ち切った．提案手法で取り除けなかった誤りを 3 名の被験者が Case1 と同様の方法で取り除いた．この 3 名は Case1 で該当テストスクリプトを修正しなかった 3 名とした．このとき，提案手法の適用にかかったマシンタイムおよび，被験者が手作業で修正を行うのにかかった時間を測定した．

本実験で用いるテストスクリプトは，テスト結果を正しく判定するための適切なアサーションを設定している．そのため，テストに通過する場合は，正しい手順でテスト実行が行われており，すなわち正しいロケータが設定されていると仮定する．したがって，本実験ではテストに通過した時点で，正しい修正が行われたとみなす．

5.2　評価結果

5.2.1　精度評価

表 3 は各ソフトウェアの 2nd Release のバージョンにおいて，何個のロケータの誤りを含んでいたかを示している．また，表 4 は提案手法を用いて正解の画面要素を何位に提示することができたかを示している．表 4 より，Collabtive，MantisBT については，全ての誤りについて正解を 1 位に提示することができていることが分かる．また，PPMA も 2 位に提示した 1 つを除いて正解を 1 位に提示することができた．

5.2.2　稼働評価

図 5 は被験者がテストスクリプト A〜H に対して修正を完了するのにかかった時間の合計を表している．縦軸が各スクリプトを修正するのにかかった時間（秒），横軸がテストスクリプトの名前を表す．各テストスクリプトについて，左が従来手法，右が提案手法でかかった時間を表しており，手作業とマシンタイムを区別して表記している

図 5 より，テストスクリプト C 以外は，提案手法によって全ての誤りを修正することができたため，かかった時間はマシンタイムのみであった．また，全てのテストスクリ

表 3　調査対象

OSS	調査対象ロケータ数	修正が必要なロケータ数
MantisBT	103	7
PPMA	45	9
MRBS	102	24
Collabtive	125	1
合計	375	41

表 4　正解の修正を提示した順位

OSS	1 位	2 位	3 位	4 位	5 位	6 位以下
MantisBT	7	0	0	0	0	0
PPMA	8	1	0	0	0	0
MRBS	10	7	2	2	1	2
Collabtive	1	0	0	0	0	0
合計	26	8	2	2	1	2

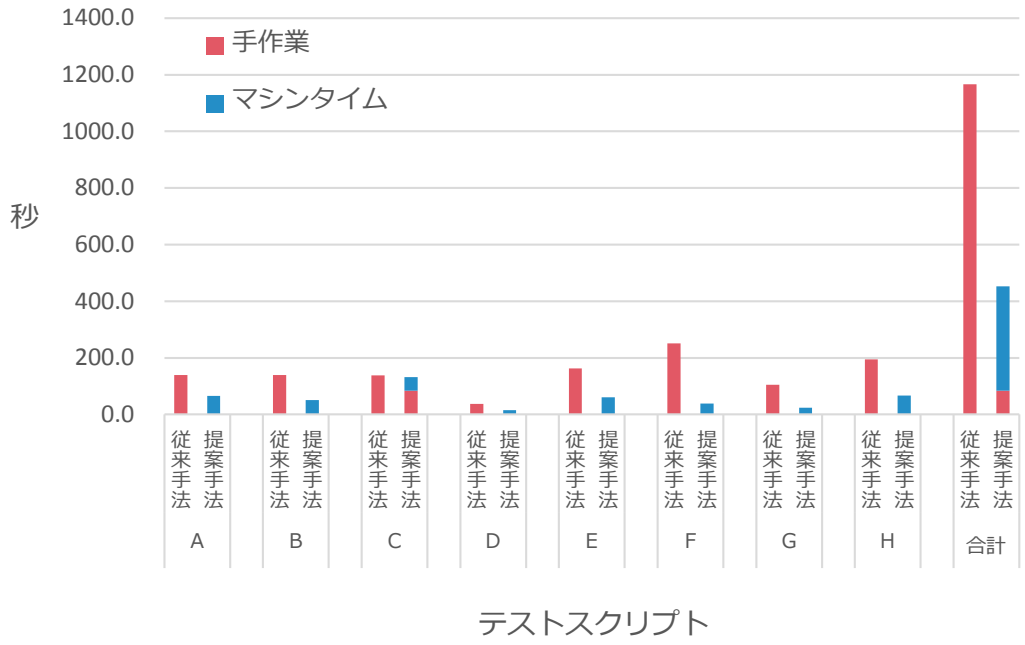


図 5　作業にかかった時間

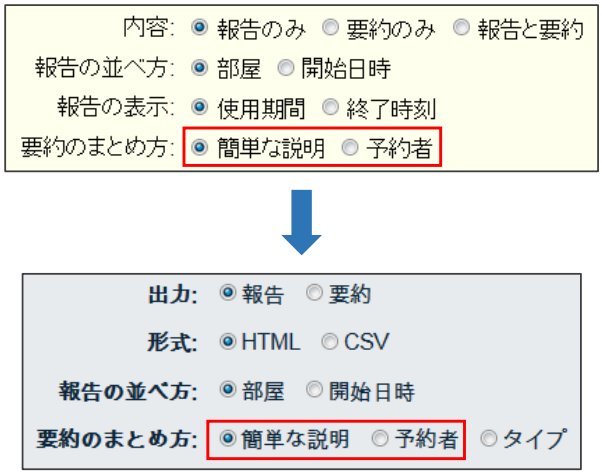


図 6　変更例: ラジオボタン

図 7　変更例: リンクからボタンへの変更

プトにおいて，提案手法を利用した場合の方が作業時間を短縮できていることが分かる．テストスクリプト A〜H の結果を合わせると，マシンタイムを含めない手作業での稼働を 92.8%削減することができた．

6. 議論

精度評価および稼働評価の結果より，提案手法は，画面のレイアウト変更が頻繁に起こる開発においても多くの場合で有効であると言える．

図 5 において，マシンタイムも計算に含める場合，61.2%の稼働削減となるが，提案手法では，マシンタイムを含めない手作業での稼働を削減することがより重要であると考える．なぜなら，ユーザはツールを動かしている間に，別の作業を行うことが可能であるためである．さらに，並列処理を用いて複数のテストスクリプトを同時に修正するといった工夫を行うことで，マシンタイムを削減する余地がある．

表 4 より，MRBS は他のアプリケーションと比べると修正の精度が低かった．特に，図 6 のような場合に，1 位に正解を提示できない場合が多かった．図 6 において，アプリケーションの変更前後で，「簡単な説明」のラジオボタンのロケータが変更されたとする．このとき，変更後のテストスクリプトにおいても「簡単な説明」を選択したいにもかかわらず，「予約者」を選択するような修正をしてしまうといった誤りが多く見られた．その原因として，2 つのラジオボタンが構造的に並列であり，持っている属性とその値が類似しているということが挙げられる．現在のプロトタイプの実装では，ラジオボタンとそれを指すテキストを結び付けられていないため，2 つを明確に区別することが難しい．アルゴリズムを改善し，左のラジオボタンが「簡単な説明」で右のラジオボタンが「予約者」であるという

情報を得ることができれば，このような問題は解決されると考える．

また，提案手法で対応することが難しい修正のパターンとして，タグの変更があることが分かった．例えば，MRBSにおいて，図 7 の修正が行われた場合に適切に画面要素の類似度を算出できなかった．図 7 では a タグで実装された削除リンクが input タグの削除ボタンに変更されている．提案手法では，修正前後の画面要素が共通して持つ指標から類似度を算出している．この例では，修正前後で共通する属性が存在せず，比較的変化しやすい位置の情報の影響が大きくなったため，類似度が低くなってしまったと考えられる．このような場合でも正しく類似性を評価するためには，属性同士を比較するのではなく，各属性に用いられている単語等から，画面要素の意味を抽出するといった工夫を行う必要がある．

7. 妥当性への脅威

7.1 外的妥当性

本研究における外的妥当性への脅威として，まず，評価実験で用いた OSS の数が少なく，結果に偏りが発生しうることが挙げられる．これについて本研究では，既存研究で用いられており，さらにドメインが異なる OSS を用いることで結果の偏りを減らすようにしている．今後，実験対象を増やすことによって結果に影響が出るかどうかを確認したい．また，本研究で選択した 2 つのバージョンについても同様である．1st Release と 2nd Release の間で行われた変更の種類により，ロケータの自動修正の可否が変化する恐れがある．今後，1 つの OSS について 3 つ以上のバージョンを用いて実験を行うことで，より一般的な結果を求めたい．

7.2 内的妥当性

本研究における内的妥当性への脅威として，被験者実験における被験者の熟練度が実験結果に影響を与えうる点があげられる．今回の被験者は全員 Selenium IDE の実務での使用経験がなく，熟練者と比べて修正に多くの時間がかかる傾向があると言える．したがって，熟練者が実験に参加した場合，提案手法と既存手法で，今回ほどの差が出ない恐れがある．これについて本研究では，あらかじめツールの利用方法についてのレクチャーを行い，熟練者との差が出にくい工夫をしている．今後は，様々な熟練度の被験者を用いて実験を行い，結果に与える影響を調べたい．

8. まとめ

本研究では，テストスクリプトの修正にコストがかかる問題に対し，テストスクリプト中のロケータを自動修正する手法を提案した．提案手法では，画面要素の属性・位置・画像・テキストといった情報を用い，より高精度な自動修正技術を実現した．OSS を用いて提案手法を評価した結果，提案手法が様々なソフトウェアにおいて，テストスクリプトを高精度で修正可能であることが分かった．

また，被験者実験を行い，提案手法がロケータの修正にかかる時間を約 93%削減できることを示した．提案手法を用いることで，テストスクリプトの保守コストが削減され，アプリケーションの修正からリリースまでをより短い間隔で実施することができる．

今後は，提案手法の弱点の改善，評価実験の強化および，論理の誤りの自動修正に取り組む予定である．

参考文献

[1] 日本情報システム・ユーザー協会：ユーザー企業 ソフトウェアメトリックス調査 2012.
[2] Engström, E. and Runeson, P.: A Qualitative Survey of Regression Testing Practices, *Proceedings of the 11th International Conference on Product-Focused Software Process Improvement*, PROFES'10, Berlin, Heidelberg, Springer-Verlag, pp. 3–16 (2010).
[3] : Selenium, http://www.seleniumhq.org/.
[4] Leotta, M., Clerissi, D., Ricca, F. and Tonella, P.: Capture-replay vs. programmable web testing: An empirical assessment during test case evolution, *2013 20th Working Conference on Reverse Engineering (WCRE)*, pp. 272–281 (2013).
[5] Hammoudi, M., Rothermel, G. and Stocco, A.: WATERFALL: An Incremental Approach for Repairing Record-replay Tests of Web Applications, *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, New York, NY, USA, ACM, pp. 751–762 (2016).
[6] Choudhary, S. R., Zhao, D., Versee, H. and Orso, A.: WATER: Web Application TEst Repair, *Proceedings of the First International Workshop on End-to-End Test Script Engineering*, ACM, pp. 24–29 (2011).
[7] Hammoudi, M., Rothermel, G. and Tonella, P.: Why do Record/Replay Tests of Web Applications Break?, *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pp. 180–190 (2016).
[8] Hammoudi, M., Rothermel, G. and Tonella, P.: Why do Record/Replay Tests of Web Applications Break?, *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pp. 180–190 (2016).
[9] Leotta, M., Stocco, A., Ricca, F. and Tonella, P.: Automated Generation of Visual Web Tests from DOM-based Web Tests, *Proceedings of the 30th Annual ACM Symposium on Applied Computing*, SAC '15, New York, NY, USA, ACM, pp. 775–782 (2015).
[10] Yeh, T., Chang, T.-H. and Miller, R. C.: Sikuli: Using GUI Screenshots for Search and Automation, *Proceedings of the 22Nd Annual ACM Symposium on User Interface Software and Technology*, UIST '09, New York, NY, USA, ACM, pp. 183–192 (2009).
[11] Yandrapally, R., Thummalapenta, S., Sinha, S. and Chandra, S.: Robust Test Automation Using Contextual Clues, *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ISSTA 2014, New York, NY, USA, ACM, pp. 304–314 (2014).