GUI自動テストにおける テストスクリプト中のロケータ修正支援手法

切貫 弘之[1,a)]　丹野 治門[1,b)]　夏川 勝行[1,c)]

概要：近年，加速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，テスト対象ソフトウェアに修正が行われた場合，それに伴ってテストスクリプトにも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を行う上での大きな障害となっている．本研究では，画面から得られる多様な情報を用いることで，テストスクリプト中の誤ったロケータに対し，修正後のロケータを推測する手法および，それを利用してロケータを自動的に修正する手法を提案する．　SS 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，正しいロケータを高精度で推測できることを示した．また，被験者実験を行った結果，提案手法を用いることで，手作業の場合と比べて被験者がロケータの修正にかかる時間が約 93%削減された．

Automatic Locator Repair on GUI Test Scripts

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．

リリースサイクルを短縮する上で回帰テストにかかる工数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工数は，開発にかかる工数全体の 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工数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工数をかけることは，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工数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]．以下，このような画面操作を伴うテストを **GUI テスト**と呼ぶ．

UI テストの自動化を妨げる要因の 1 つとして，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，それに応じて既存のテストスクリプトにも修正が必要となる．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る恐れもある．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

[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)] natsukawa.katsuyuki@lab.ntt.co.jp

モチベーションの低下につながる．

eotta 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 SS 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を実行する場合に，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が修正されうるということは，テストスクリプトも今後修正される可能性が高く，テスト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，画面から得られる多様な情報を用いることで，テストスクリプト中の誤ったロケータに対し，修正後のロケータを推測する手法および，それを利用し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．テストスクリプトにおいて，このロケータの誤りが全体の約 74%を占めていることが知られており，最も重要であると考える [8]．

既存研究として，画面要素の Path のレーベンシュタイン距離を用いて，ロケータを修正する手法が存在する．Path のような位置情報を用いる手法は，画面の UI の変更が大きくなると精度が低くなる問題があった．本研究は，位置情報のみでなく，属性・テキスト・画像の情報も用いることで，既存研究の弱点を克服している．

本研究では，現在最も開発が盛んな eb アプリケーション [1] 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を削減することができる．その結果， UI テストの自動化を促進し，ソフトウェアの高速なリリースを行えるようになることが期待できる．

提案手法の有効性を確認するために， SS を用いて評価を行った．その結果，提案手法が，約 51%の確率で正しいロケータを 1 位に，約 95%の確率で 5 位以内に推薦できることを示した．また，被験者実験を行った結果，提案手法を用いることで，従来手法と比べて被験者がロケータの修正にかかる時間が約 93%削減された．

2. テストスクリプト

2.1 テストスクリプト実装手法

既存のテストスクリプト実装手法の代表的なものとして，**キャプチャ＆リプレイ**と**プログラミングによる手法**の 2 つが挙げられる．

キャプチャ＆リプレイでは，まずユーザが画面に対し操作を行うことで，ユーザが行った操作をスクリプトとして記録する．そして，次回以降そのスクリプトを実行することで，自動テストを行う．キャプチャ＆リプレイの利点として，画面操作で簡単にテストスクリプトを実装することができるため，プログラミングスキルが無くても利用可能である点が挙げられる．欠点としては，テストスクリプトが操作の羅列として出力されるため，可読性が低い点やモジュール化しにくい点が挙げられる．キャプチャ＆リプレイを行うためのツールとして，Selenium I E や Selenium uilder 等がよく用いられる．

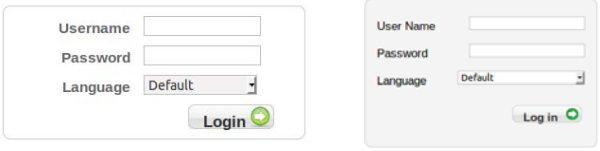
プログラミングによる手法では，Ruby や Java といったプログラミング言語を用いてテストスクリプトの実装を行う．画面を操作するための API を用いて，テストで行いたい操作をプログラムとして記述し，それを実行することで自動テストを行う．プログラミングによる手法の利点として，柔軟性が高く，実装者のスキルによっては高い保守性が期待できる点が挙げられる．欠点としては，利用するためにプログラミング言語の理解が必要であり，スクリプトの質が実装者のスキルに依存してしまう点が挙げられる． eb 画面を操作するための API として，Selenium eb river が有名である．

ここで説明したいずれの手法を用いても，テストスクリプトの保守の問題は避けることができない．プログラミングによる手法を用いた場合，適切にモジュール化を行うことで修正が必要になった場合の修正箇所を減らすことは可能である．しかし，適切にモジュール化しても保守作業は必ず必要になる上，テスト担当者が必ずしもそのようなスキルを持っているとは限らない．したがって，誰でも簡単にテストスクリプトを保守できることが開発現場において強く求められている．

本研究が対象とするテストスクリプトとして，キャプチャ＆リプレイツールの Selenium I E で実装されたものを採用した．なぜなら，Selenium I E 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，開発現場で広く用いられているためである．

2.2 テストスクリプトの例

SS のコンテンツマネジメントシステムである Joomla!



(a) バージョン 1.5　　(b) バージョン 2.5

図 1: Joomla!のログインフォーム

表 1: Joomla!バージョン 1.5 に対応するスクリプト

#	命令	ロケータ	値
1	open	/joomla	
2	click	css=li.item17>a>span	
3	type	id=modlgn_username	user01
4	type	id=modlgn_passwd	pass01
5	click	link=Login	
6	assertTitle		Main

を例として説明する．Joomla!のバージョン 1.5 において，管理者としてログインを行うことを考える．ユーザはまず，アプリケーションのトップ画面において，「Administrator」と書かれたリンクをクリックし，ログイン画面に移行する．次に，ログイン画面において，「Username」と「Password」を入力して「 ogin」ボタンを押すことでログインする．図 1a はそのログインフォームである．この操作を Selenium I E 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**とは，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．Selenium I E が利用できるロケータとして，id, name, css, Path 等が存在し，あらかじめ定められた優先度に従って，スクリプト実装時にいずれか 1 つのロケータが決定される．例えば，2 行目の「css=li.item17>a>span」は，該当の css セレクタで表される要素（「Administrator」リンク）を指す．また，3 行目の「id=modlgn_username」は HTM 中で id が「modlgn_username」である画面要素（「Username」入力フォーム）を指す．**値**はユーザが与える入力値もしくは期待結果を表し，3 行目では「Username」への入力として「user01」を与えている．また，6 行目ではメインページに遷移するという事後条件を与えている．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．

2.3 テストスクリプトの修正

実際にテストスクリプトに修正が必要になる例を紹介する．Joomla!のバージョン 2.5 では，ログインフォームが図 1b のようになっており，バージョン 1.5 で作成した表 1 のテストスクリプトを実行するとエラーが発生する．なぜなら，バージョン 1.5 とバージョン 2.5 ではアプリケーションの UI や画面要素が持つ属性が異なっており，表 1 中のロケータが指す画面要素がバージョン 2.5 の画面には存在しないためである．表 1 のテストスクリプトは，バージョン 2.5 では，例えば表 2 のように修正すればよい．人間がテストスクリプトを修正する場合，まず，テストスクリプト中のロケータやコメント，テスト手順書を参考にしてテストスクリプトの動作を理解する必要がある．しかし，テストスクリプトの可読性が低い，もしくはドキュメントが不足している場合は，テストスクリプトの動作を理解することは難しい．例えば，表 1 の 2 行目はロケータが CSS セレクタで記述されており，どのような処理が行われているのかはテストを実行しない限り分からない．このようなテストスクリプトの理解・修正は，人間にとって大きな負担であり，テスト自動化のモチベーションを下げる原因となるため，機械的な支援を行えることが望ましい．

表 2: Joomla!バージョン 2.5 に対応するスクリプト

#	命令	ロケータ	値
1	open	/joomla	
2	click	link=Site Administrator	
3	type	id=mod_login_username	user01
4	type	id=mod_login_password	pass01
5	click	link=Log in	
6	assertTitle		Main

2.4 テストスクリプトの誤りのパターン

2.3 節で示した例のように，ソフトウェアの修正によって，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，2.3 節の例のようにテストにおいて操作される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，アプリケーションの UI の変更や画面要素が持つ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．本論文では，ロケータの誤りとはテストスクリプト中のロケータのみを変更することで修正できる誤りと定義する．

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．本論文では，論理の誤りとはテストスクリプト中のロケータのみを変更しても修正できない誤りと定義する．

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%を占めていることが知られている [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する．

3. 既存研究

ロケータの誤りの修正の支援を目指した研究として，Choudhary らの研究がある [6]．Choudhary らは Path 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． Path は HTM における特定の画面要素の位置を表す．Choudhary らは，テスト対象の修正前後で， Path 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 UI の変更に弱いことが挙げられる．アプリケーションの UI が変更されると，画面要素の Path 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， eotta らの研究がある [7]． eotta らは，FirePath や Robula+等の Path ベースの 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，テスト対象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．

また，これらの既存研究では，修正を行う際，誤ったロケータを最も確からしいロケータに書き換えるのみで，修正結果の妥当性については言及されていない．そのため，手法による修正結果が正しいかどうかが，テストスクリプトを実行するまでは分からないという実用上の問題がある．

既存研究の改善として，Hammoudi らは，テストスクリプトを自動修正する際，リリースバージョン毎ではなく，コミット毎等のより細かい差分に対して手法を適用することで，最終的な修正の精度が高くなることを示した [5]．この手法は，他の手法と併用することが可能であるが，既存手法の問題点を根本的に解決するものではない．したがって，1 つのコミットで大きな UI の変更がある場合，この手法は有効ではない．

テスト対象の修正に合わせてテストスクリプトを修正するのではなく，最初からテスト対象の修正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．

eotta らは，従来の M 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， SS 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，UI の変更が多く行われるプロジェクトでは有効ではない．

andrapally 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が含まれる場合はこの手法のみでスクリプトを記述することができない．

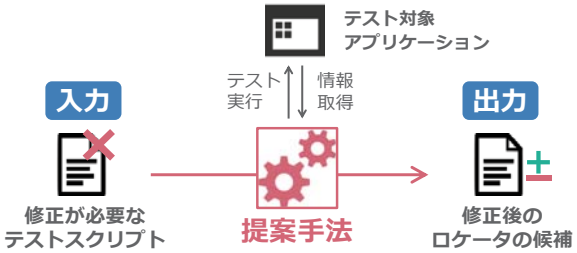
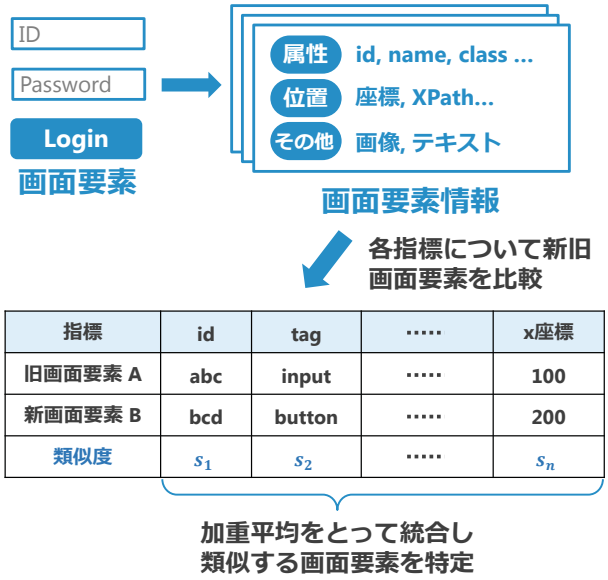


図 2: 提案手法の全体像



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

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

表 3: 用いた指標と重み付け

指標	id	class	name	value	type	tag_name	text	xpath	css	image	neighbor_text
重み	3.0	1.5	2.0	2.0	1.0	1.0	3.0	4.0	2.0	4.0	2.0
指標	x_axis	y_axis	height	width	alt	src	href	target	size	onclick	
重み	1.0	1.0	1.0	1.0	2.0	2.0	2.0	1.0	1.0	2.0	

テスト対象の修正に強い手法を実現する．提案手法の適用に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，修正後のロケータを推測するための手がかりとする．

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，誤っている各ロケータについて，修正後のロケータの候補を出力する．ロケータの候補は確からしさの順に複数提示される．

加えて，本章では，この出力結果を用いて，修正結果の妥当性を担保しつつテストスクリプトの自動修正を行う手法を説明する．

提案手法の利用法として，まず自動修正を試み，修正できなかった部分は手作業で修正するという使い方を想定している．

4.1 ロケータ推測アルゴリズム

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 と役割が等しい画面要素 $e' \in E$ 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．役割が等しい画面要素とは，修正前後で同一の役割を持っていることが人間から見て明らかなものを意味する．アプリケーションの修正後においても，修正前と同じ役割の要素に対して同じ操作を行うことができれば，正しく修正できたとみなす．

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 と役割が等しい画面要素 $e' \in E$ を推定する．類似度が高いもの程，修正後のロケータとして確からしいとみなす．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 (Path・座標)
- 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 e_i とする．ここで，$Max(i)$ は $|e_i - e'_i|$ がとりうる最大値 (例えば，i が　座標ならば画面の横幅)，$Lev(e_i, e'_i)$ は e_i と e'_i のレーベンシュタイン距離，$MaxLength(e_i, e'_i)$ は e_i と e'_i の文字列の長い方の長さとすると，e_i と e'_i の類似度 $s(e_i, e'_i)(0 \leq s(e_i, e'_i) \leq 1)$ は以下のように算出される．

e_i が数値の場合:

$$s(e_i, e'_i) \quad 1 - \frac{|e_i - e'_i|}{Max(i)} \tag{1}$$

e_i が画像または特定の文字列 (タグ名など) の場合:

$$s(e_i, e'_i) \quad \begin{cases} 1 & (e_i \quad e'_i) \\ 0 & (otherwise) \end{cases} \tag{2}$$

e_i が任意の文字列の場合:

$$s(e_i, e'_i) \quad 1 - \frac{Lev(e_i, e'_i)}{MaxLength(e_i, e'_i)} \tag{3}$$

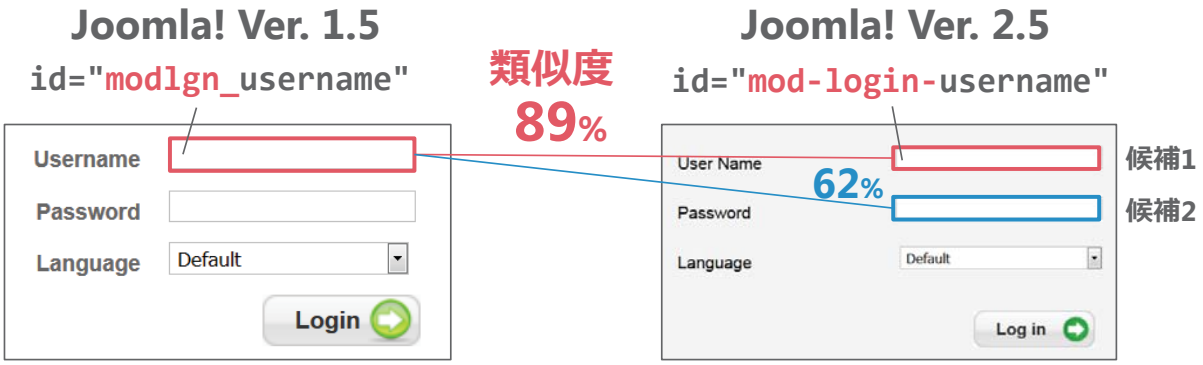
類似度は，e_i と e'_i が同一である場合は 1，e_i と e'_i が離れるほど 0 に近づくような値となっている．各指標について算出された $s(e_i, e'_i)$ を統合するにあたって，それぞれ画面要素の類似度に寄与する度合いが異なると考えられる．例えば，異なる画面要素が同じ id を持つことはないが，異なる画面要素が同じ class を持つことは起こりうるため，class の一致は id の一致に比べて類似度に寄与する度合いが小さいと考えられる．これを考慮し，各指標について，$s(e_i, e'_i)$ を算出した後，重み w_i で加重平均を取ることで e と e' の類似度 $S(e, e')(0 \leq S(e, e') \leq 1)$ を算出する．

$$S(e, e') \quad \frac{\sum_{i \in I} s(e_i, e'_i) w_i}{\sum_{i \in I} w_i} \tag{4}$$

提案手法で用いる指標とその重み付けについて，著者の手で有効と思われるものを選定した．今回用いた指標と重みの一覧を表 3 に示す．

4.2 適用例

2.3 節の例において手法を適用した場合のイメージを図 4 に示す．ここでは，修正前後で「Username」入力フォー



命令	ロケータ	値
type	**id=modlgn_username**	user01
type	id=modlgn_passwd	pass01
click	link=Login	

修正

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

図 4: 手法の適用例

表 4: 図 4 における類似度計算

指標	id	name	type	y_axis	neighbor_text	統合された類似度
Username(Ver.1.5)	modlgn_username	username	text	186	Username	-
候補 1	mod-login-username	username	text	216	User Name	-
類似度	0.78	1.00	1.00	0.94	0.78	**0 9**
候補 2	mod-login-password	passwd	password	257	Password	-
類似度	0.33	0.00	0.00	0.86	0.00	**0**

ムの id が変更されており，バージョン 1.5 用のテストスクリプトはバージョン 2.5 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 3 行目のロケータが指す画面要素がバージョン 2.5 の画面には存在しないためである．例として，1 行目のロケータを修正する場合の処理を説明する．

提案手法では，まず，バージョン 1.5 の「Username」入力フォームとバージョン 2.5 の各画面要素の類似度を計算する．ここでは，バージョン 2.5 において図 4 で示された**候補 1** と**候補 2** に対して類似度を計算することを考える．ここで計算された類似度の一部を表 4 に示す．この表では id, name, type, y_a is, neighbor_te t の 5 つの指標を抜粋して示している．これら 5 つの指標を含めた表 3 の 21 個の指標について加重平均を取ることで統合された類似度を算出する．

その結果，バージョン 2.5 において，候補 1 の画面要素が最も類似度が高くなった．したがって，候補 1 の画面要素が，バージョン 2.5 における「Username」入力フォームとして最も確からしいと推測できる．つまり，テストスクリプトの 1 行目のロケータは「id=mod-login-username」に修正すれば良いことが分かる．

4.3 スクリプトの自動修正

本節では，提案手法の出力を利用することで，テストスクリプトの自動修正を行う手法を説明する．既存研究は，誤ったロケータを候補のうちで最も確からしいロケータに書き換えるという手法を採用していた．しかし，この手法は，修正結果の妥当性を考慮しないという問題がある．本研究では，以下の仮定を設定することで，修正結果の妥当性を担保しつつ，自動修正を行う方法を提案する．

仮定 1 テストスクリプト中にアプリケーションの動作を確認するための適切なアサーションが存在する．

仮定 2 全てのアサーションの条件を満たし，テストに通過する場合，そのテストスクリプトは正しい．

仮定 3 いずれかのアサーションの条件を満たさない，またはエラーが発生し，テストに通過しない場合，そのテストスクリプトが誤っているかアプリケーションにバグが存在する．

仮定 1 における適切なアサーションとは，ある操作単体，もしくは操作の組み合わせを行うことによってのみ起こる事象を確認できるようなアサーションを意味する．例えば，ユーザ A としてログインを行うテストでは，ログイン画面に移行することを確認するだけでは十分とは言えな

表 5: 実験対象のソフトウェア

OSS	説明	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

い．なぜなら，他のユーザとしてログインしても，ログイン画面には移行するためである．ここでは，例えばログイン後の画面に「ようこそユーザ A さん」といったことが表示されることを確認するアサーションが適切である．

これらの仮定のもと，誤ったロケータを自動的に書き換えることで，自動修正を行う．ロケータの書き換えは，提案手法の出力において最も確からしいと推測されるロケータから優先的に行う．修正後のテストスクリプトを実行し，テストに通過した場合，仮定により修正に成功したとみなす．テストスクリプトの途中にアサーションがある場合は，そのアサーションを通過した時点でそれ以前の行は正しいとみなす．テストに通過しない場合，前回試したものとは別のロケータに書き換え，再度テストを実行する．テストに通過するまで再帰的にロケータの修正とテスト実行を繰り返すことで，最終的に正しいテストスクリプトを出力する．

本研究においては，論理の誤りは修正の対象外であるため，論理の誤りが含まれるテストスクリプトは完全に修正することができない制限がある．自動修正のみで対象テストスクリプト中の全ての誤りを修正できない場合，手作業による修正を併用する必要がある．

5. 評価実験

提案手法について，精度と時間の 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．

実験対象として，表 5 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　eb 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，2.4 節で述べた　eotta 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 UI の変更や画面要素の属性の変更が様々なパターンで行われており，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

精度評価では，提案手法が正しいロケータをどの程度正確に提示できるかを調べる．これにより，提案手法が様々なソフトウェアの修正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，類似する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，修正後のロケータの候補が確からしいものから順に提示される．この際，正解のロケータが何位に提示されるかを調べた．

5.1.2 時間評価

提案手法を利用したテストスクリプトの自動修正によって，テストスクリプト修正にかかる時間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」**の時間を比較した．従来手法として，開発現場で広く用いられている Selenium I　E を用いて手作業でロケータを修正する方法を採用した．Selenium I　E 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 I　E の使用経験は無かった．実験を円滑に行うため，Selenium I　E 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 I　E 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．テストスクリプト A〜H は全てロケータの誤りを含んでいた．これら

のテストスクリプトのうち 2 つは論理の誤りも含んでいたが，論理の誤りについては事前に著者の手で取り除いた．なぜなら，4.3 節で述べた制限に加え，現在のプロトタイプの実装では，従来手法と提案手法との併用が難しく，被験者にとって作業が困難になると判断したためである．次に，時間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～　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や，類似度が 0.5 より高い候補が存在し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，4.3 節で述べた基準でテストの合否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，4.3 節の仮定より，テストに通過する場合は，正しいロケータが設定されているとみなせる．したがって，本実験ではテストに通過した時点で，正しく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 6 は各ソフトウェアの 2nd Release のバージョンにおいて，何個のロケータが誤っていたかを示している．また，表 7 は提案手法を用いて正解のロケータを何位に提示することができたかを示している．表 7 より，Collabtive，Mantis　T 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．また，PPMA について，2 位に提示した 1 つを除いて正解を 1 位に提示している．

5.2.2　時間評価

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合計時間（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，手作業の時間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自動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．テストスクリプト C は，途中までアサーションを通過したため，それ以前のロケータは自動修正することができた．したがって，それ以降の誤ったロケータのみを被験者が修正した．また，全てのテストスクリプトにおいて，提案手法を利用した場合の方が作業時間を短縮できた．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時間を 92.8%削減することができた．

6.　議論

精度評価および時間評価の結果より，提案手法は，アプリケーションの UI や画面要素の属性の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．

提案手法の有効性を示すため，今回，提案手法で正しいロケータを 1 位に推薦することができたが，既存手法では難しい例を紹介する．ここで，MR　S バージョン 1.2.1 における「　ame」入力フォームを a とする．また，MR　S バージョン 1.5.1 における「　ame」入力フォームを a'，「Password」入力フォームを b'，「Search」入力フォームを c' とする．このとき，

$$
\begin{aligned}
a_{xpath} &\quad //form/table/tbody/tr[1]/td[2]/input \\
a'_{xpath} &\quad //div[2]/form/fieldset/div[1]/input \\
b'_{xpath} &\quad //div[2]/form/fieldset/div[2]/input \\
c'_{xpath} &\quad //div[1]/table/tbody/tr/td[6]/form/\\
&\quad div/input[1]
\end{aligned}
$$

表 6: 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 7: 正解のロケータを提示した順位

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

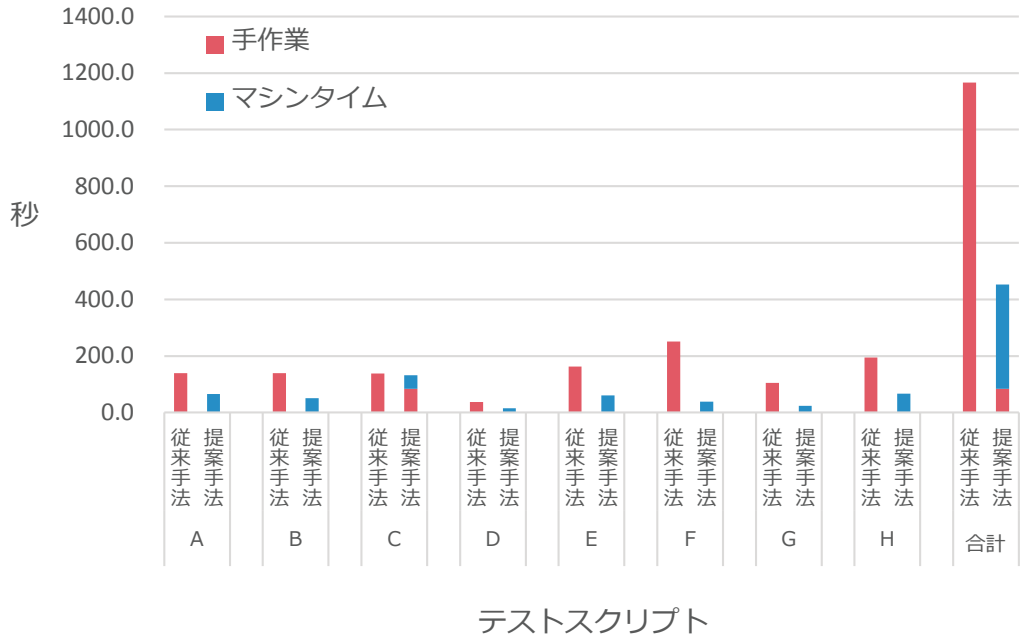


図 5: 作業にかかった時間

であり，式 (3) より， Path の類似度は以下のようになる．

$$s(a_{xpath}, a'_{xpath}) \quad 0.31$$
$$s(a_{xpath}, b'_{xpath}) \quad 0.33$$
$$s(a_{xpath}, c'_{xpath}) \quad 0.55$$

ここで，本来推薦すべき a' の類似度が他の要素よりも低くなっていることが分かる．このように， Path のような位置情報に頼る手法は，UI の変更があると精度が低くなりがちである．提案手法は位置情報以外にも多くの指標を用いるため，既存手法と比較すると，UI や一部の属性が変化してもその影響を受けにくいという強みがある．

表 7 より，MR S 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，この 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，MR S 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．

図 5 において，マシンタイムも計算に含める場合，61.2%の時間削減となるが，提案手法では，マシンタイムを含めない手作業での時間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ため，マシンタイムの長さは本実験において重要ではないと考える．

また，今回は実験の都合上， 被験者が修正するテストスクリプトは論理の誤りがあらかじめ取り除かれていた．論理の誤りを含むテストスクリプトを修正する場合，実際は提案手法と従来手法を併用する必要がある．そのため，提案手法と従来手法の併用のために発生するオーバーヘッドが実験結果に現れていない問題がある．手法の併用を簡単に行うため，現在，提案手法を Selenium I E のプラグイ

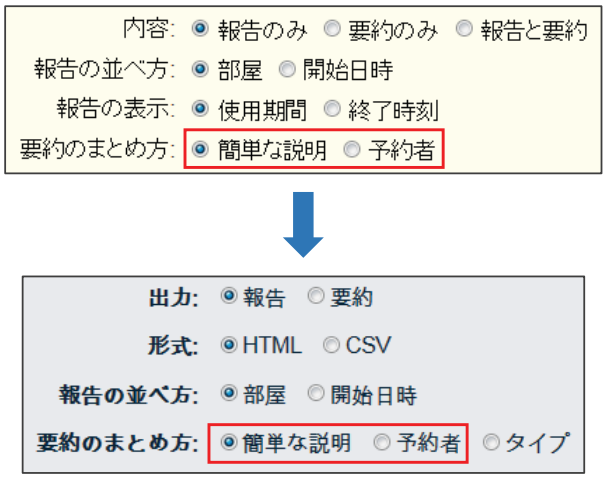


図 6: 変更例: ラジオボタン

図 7: 変更例: リンクからボタンへの変更

ンとして実装している．これにより，ツールがユーザに対し，操作すべき画面要素をブラウザ上に分かりやすく示し，ユーザが選択した画面要素のロケータが即座にスクリプトに反映されるといったインタラクティブな修正を行うことができる．これを用いることで，提案手法と従来手法の併用が容易にできるようになり，より現実のスクリプト修正作業に近い実験を行えるようになると考える．

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 SS 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 SS 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 SS 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．

また，今回提案手法で用いた指標と重みは著者の手でチューニングを行った．今回の設定を他のアプリケーションに対して適用した場合，悪い結果になる恐れがある．しかし，本研究では，重み付けが原因で著しく悪い結果が出たアプリケーションは無かったため，今回の設定にはある程度汎用性があると思われる．これについて，統計的手法を用いて重み付けを行う等の工夫を行うことで，より良い結果を得られる可能性がある．

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の修正を支援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，より高精度なロケータ推薦技術を実現した． SS を用いて提案手法を評価した結果，様々なソフトウェアにおいて，提案手法が有効であることを示した．

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，まず，追加の実験と結果の分析を行う予定である．各指標がどの程度修正の精度に寄与するのかを調べ，その結果をもとに指標の重みを決定するといったことを検討している．また，今回判明した提案手法の弱点の改善，実装中のツールを用いたより現実的な実験，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*, PR FES 10, erlin, Heidelberg, Springer- erlag, pp. 3 16 (2010).

[3] : Selenium, http://www.seleniumhq.org/.

[4] eotta, M., Clerissi, ., 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, . and Stocco, A.: ATER-FA : An Incremental Approach for Repairing Record-replay Tests of eb Applications, *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, ew ork, , USA, ACM, pp. 751 762 (2016).

[6] Choudhary, S. R., Zhao, ., ersee, H. and rso, A.: ATER: eb 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, . and Tonella, P.: hy do Record/Replay Tests of eb Applications reak , *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pp. 180 190 (2016).

[8] Hammoudi, M., Rothermel, . and Tonella, P.: hy do Record/Replay Tests of eb Applications reak , *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pp. 180 190 (2016).

[9] eotta, M., Stocco, A., Ricca, F. and Tonella, P.: Automated eneration of isual eb Tests from M-based eb Tests, *Proceedings of the 30th Annual ACM Symposium on Applied Computing*, SAC 15, ew ork, , USA, ACM, pp. 775 782 (2015).

[10] eh, T., Chang, T.-H. and Miller, R. C.: Sikuli: Using UI Screenshots for Search and Automation, *Proceedings of the 22Nd Annual ACM Symposium on User Interface Software and Technology*, UIST 09, ew ork, , USA, ACM, pp. 183 192 (2009).

[11] andrapally, R., Thummalapenta, S., Sinha, S. and Chandra, S.: Robust Test Automation Using Conte - tual Clues, *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ISSTA 2014, ew ork, , USA, ACM, pp. 304 314 (2014).