# Web アプリの動的部分に着目したグレーボックス統合テストとテンプレート変数カバレッジの提案

## Gray-box Integration Testing and Template Variable Coverage for Web Applications

坂本 一憲* 海津 智宏† 波村 大悟‡ 鷲崎 弘宜§ 深澤 良彰¶

**あらまし** Web アプリケーションは様々なブラウザやプラットフォーム上で動作する上，複数のシステムから構成される非常に複雑なソフトウェアである．そのため，Web アプリケーションの検証に莫大なコストが必要であり，効率の良いテスト手法の確立が大きな課題である．これまでホワイトボックス単体テストとブラックボックス結合テストを組み合わせて，Web アプリケーションがテストされてきた．しかし，前者のテストでは本番環境でテストできず，後者のテストではテストすべき箇所が不明瞭であるという問題がある．そのため，テストすべき箇所を明瞭化した上で，本番環境でテストを実施することが困難である．

本論文では，HTML テンプレートを解析することで，テストすべき箇所を明瞭化して，さらに，テストオラクルの品質指標となるテンプレート変数カバレッジを提案する．その上で，テンプレート変数カバレッジが 100%を満たすように，グレーボックス結合テストとテスト支援ツールである POGen を提案する．評価実験において提案手法が上述の問題を解決して，テストの記述・保守コストを低減することを示す．

## 1 はじめに

インターネットの普及に伴い Web アプリケーション（以降，Web アプリ）の社会的な重要性は増す一方である．Web アプリはサーバークライアントモデルに従い，サーバーサイドとクライアントサイドのプログラムが協調して動作する．サーバープログラムは様々なシステムから構成されている上，サーバーおよびクライアントプログラムの動作環境が多様化している．そのため，システム間の結合部にバグが生じたり，特定の動作環境においてのみバグが生じるケースがある [1]．このようなバグに対処するため，効率良く網羅的な検証が可能なテスト手法が求められている．

ソフトウェアテストはいくつかの観点から分類することができる．テスト対象の粒度という観点からは，モジュール単体を対象とした単体テスト，モジュールを組み合わせたソフトウェア全体を対象とした結合テストに分類される[1]．また，テストで利用する情報という観点からは，ソースコードを参照してプログラムの構造などに着目するホワイトボックス（WB）テスト，ソースコードを参照せずにソフトウェアの仕様書を元に実施するブラックボックス（BB）テスト，一部のソースコードを参照する両者の中間に当たるグレーボックス（GB）テストに分類される．

テスティングフレームワークを用いてテストコードを記述することで，機械的にテストを実施できる．テストコードとは，複数のテストケースをプログラムのコードとして表現したものである．テストケースは，製品コード（テスト対象のプログラム）を試験的に動かすテストシナリオと，テストシナリオに従って製品コードを

---

*Sakamoto Kazunori, 早稲田大学

†Tomohiro Kaizu, グーグル株式会社

‡Daigo Hamura, グーグル株式会社

§Hironori Washizaki, 早稲田大学

¶Yoshiaki Fukazawa, 早稲田大学

[1] さらに，システムテストや受け入れテストといった分類も可能だが，本論文では簡単のためにモジュールを結合するか否かで単体テストか結合テストかに分類する．

実行した結果，プログラムが期待通りの動作をしたかどうか判定するためのテストオラクルから構成される．一般に，テストオラクルとして，製品コードを実行して得られる値について期待値と等しいか検証する仕組みが用いられる．なお，本論文では，Web アプリが出力する HTML 文章において，テストの成否を判定するために検証する箇所をテストオラクルで検証すべき箇所と呼ぶ．

Web アプリ開発ではしばしばテンプレートエンジンが利用される．テンプレートエンジンとは，HTML テンプレートに対して，プログラムを実行する際に得られる値の文字列表現を埋め込むことで，HTML 文章を生成するシステムである．テンプレートエンジンはサーバーサイドで動作するソフトウェアとクライアントサイドで動作するソフトウェアに分かれており，前者の例として ejs [2] が，後者の例として Closure Templates [3] [2]が挙げられる．

Web アプリは静的に HTML 文章を出力するだけではない．上述のテンプレートエンジンを用いることで，静的に得られる部分とプログラムの実行状態に依存して動的に生成される部分を併合させて，ユーザが閲覧する HTML 文章を生成する．静的な部分は常に同じ内容であるため，バグの検出が容易である．一方，動的な部分はプログラムの実行状態に依存して変化するため，一部のケースでのみ HTML 文章が正常に生成されない可能性があり，静的な部分よりもバグの検出が困難である．

我々は，動的な部分についてテストすべきであるという考察に基づいて，HTML テンプレート中のテンプレート変数に着目したテンプレート変数カバレッジを提案する．さらに，HTML テンプレートを解析することで，テンプレート変数カバレッジが 100%を満たせるような GB 結合テストを提案する．その上で，ユーザのテストコード記述を支援するために，PageObjects デザインパターンに基づいて，テンプレート変数へのアクセサメソッドを備えたスケルトンテストコードを自動生成するツール POGen を開発した．現在，POGen はオープンソースソフトウェアとして公開中である（http://code.google.com/p/pageobjectgenerator/）．なお，本論文ではテンプレートエンジンの利用を前提に論じる．

## 2　従来のテスト手法

Web アプリ開発では，開発したコンポーネントの一部を対象に単体テストを行った上で，実際に Web アプリを動かす本番環境で結合テストを行うプロセスが取られる．単体テストは Web アプリの開発者自らが行う場合が多く，ソースコードを理解した上で実施する WB 単体テストが利用される．一方，結合テストは Web アプリの開発者以外が行う場合が多い．Web アプリを開発した組織とは異なる組織が実施するケースもあり，ソースコードを参照しない BB 結合テストが利用される．

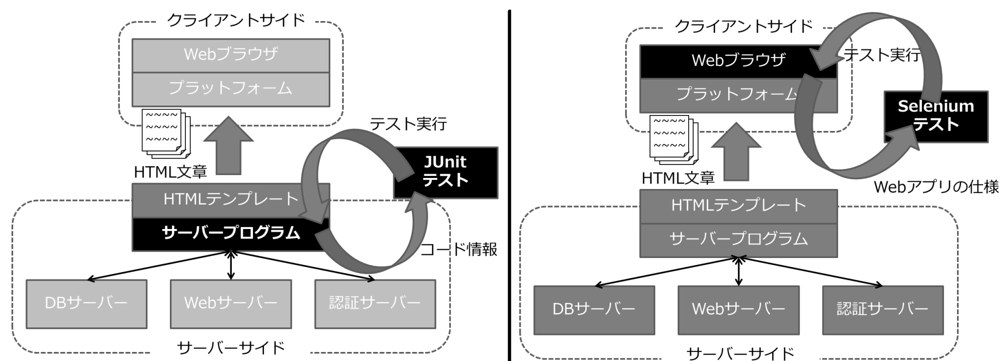


**図 1　Web アプリの構成に対する WB 単体テスト（左）と BB 結合テスト（右）の関係**

[2]サーバーサイドとクライアントサイドの両方で動作する．

**WB 単体テスト：** 図 1 の左部に Web アプリの構成と WB 単体テストの関係を示す．サーバープログラムと HTML テンプレートが存在しており，これらを動作させることで HTML 文章が生成され，あるプラットフォーム上のブラウザを通して閲覧することで，クライントプログラムが動作するという構成を考える．

WB 単体テストでは Web アプリのソースコードを参照して，テストコードを記述することで自動テストを実施する．その際，xUnit と呼ばれる単体テストのテスティングフレームワークを利用することが多い．例えば，サーバープログラムが Java で記述されている場合は，JUnit を用いてテストが行われる．

WB 単体テストはモジュール単体をテストするため，モジュール間の結合に潜むバグを発見できない．また，Web アプリが複数のシステムと相互に作用する上，様々な実行環境が存在する．システム間の結合においてバグがあったり，開発環境では再現されないバグがあったりする．そのため，実際に Web アプリを稼働させる本番環境でテストする必要がある．しかし，WB 単体テストでは，モックなどを利用してモジュールを単体でテストするため，本番環境でテストすることが難しい．

WB 単体テストはソースコードを参照するため，テストを実施した際に製品コードが実行された割合を示すテストカバレッジを算出できる．テストカバレッジはソースコードと実行履歴から定量的に算出可能なテスト十分性の指標である．一般に，テストカバレッジの目標値を掲げることで，テストの品質を保証できる．そのため，テストカバレッジを利用することで，テストにおける属人性を排除できる．

なお，WB 結合テストは次の理由から実施が困難である．

- Web アプリを構成するシステムが多岐にわたり，全てのシステムのソースコードを入手することは難しく，入手できても網羅的なテストが規模的に困難．
- 各システムが異なるプログラミング言語でばらばらに開発されており，全てに対応するテスティングフレームワークやカバレッジ測定ツールが存在しない．

**BB 結合テスト：** 図 1 の右部に Web アプリの構成と BB 結合テストの関係を示す．BB 結合テストでは Web アプリの仕様書を参照して，ブラウザを介して Web アプリを操作することでテストを実施する．そのため，Web アプリが動作さえしていれば，どのような形で Web アプリが構成されていようとテストを実施できる．BB 結合テストは Web アプリのソースコードを必要としないため，Web アプリを開発した組織とは異なる組織にテストの業務を外注することが多い，

人手によるテストの実施はソフトウェアによる自動テストよりも高いコストが必要になり，特に，繰り返しテストを行うようなリグレッションテストではコストの差が顕著である．そのため，BB 結合テストでは Selenium などの自動ブラウジングツールが利用され，BB 結合テストの実施コストを削減する工夫がなされている．

BB 結合テストはブラウザを介してテストを実施するため，WB 単体テストとは異なり，本番環境で発生する全てのバグを検出することが可能である．しかし，BB 結合テストは仕様書を元にテストの設計を行うため，テストの質がテストの設計者に強く依存してしまう問題がある．BB 結合テストにおけるテスト十分性として，全ての Web ページをテストしたかどうか，ページ間の遷移モデルに基づいて全ての遷移をテストしたかどうか，ページ上に存在する全ての GUI コンポーネントをテストしたかどうかなど，いくつかの指標が存在する [5] [6]．しかし，これらの指標だけではバグを完全に検出できない上，定量的に測定するツールが存在しない．

## 3　従来のテスト手法における問題点

WB 単体テストはソースコードを参照してテストを行うため属人性をある程度排除できる．しかし，WB 結合テストは既に述べたように実施することが困難である．したがって，WB 単体テストと BB 結合テストを組み合わせてテストが行われているが，BB 結合テストにおいては以下で述べるような問題が存在する．

**問題 1) テストオラクルで検証すべき箇所が不明瞭：** Web アプリによって生成さ

れるHTML文章において，静的に得られる部分はテストシナリオに依存しないため，テストオラクルで検証すべき箇所ではない．一方，動的に生成される部分はテストシナリオによって変化するため，テストの成否を判定するためにチェックすべき箇所である．しかし，テスターが得られる情報は最終的に生成されるHTML文章であるため，どこが動的な部分か正確に判断する術がない．テスターは自分の経験や直感からテストを実行した際にテストが成功したかどうか判定するテストオラクルを決定するため，テストにおいて検証すべき項目に抜けや漏れが発生する可能性が生じる．このように，テストオラクルで検証すべき箇所が不明瞭であるため，BB結合テストの品質，つまり，バグを検出する能力の属人性が高いという問題がある．

**問題2) テストオラクルの評価指標がない：**もしも，テストオラクルが不十分，すなわち，検証すべき項目に抜けや漏れがあると，バグの検出能力が落ちる．例えば，Googleの検索エンジンにおいて，検索結果の総数を出力する処理に誤りがあったとしても，HTML文章中から検索結果の総数の文字列を取得して検証しなければ，テストにおいてそのような誤りを検出できない．しかし，現状のBB結合テストでは，テストオラクルの作成における属人性が高く，その上，作成したテストオラクルに抜けや漏れがないかどうか判断する指標がない．このように，テストオラクルの評価指標がないため，テストオラクルの品質を保証する手段がないという問題がある．

**問題3) テストコードの記述コストが高い：**Seleniumなどの自動ブラウジングツールを利用した際に，テストオラクルを記述するためにHTML文章におけるDOMツリー上の要素にアクセスするコードの記述が難しい．なぜなら，要素にアクセスするためには，ブラウザでHTML文章をレンダリングした結果を見て，1) テストオラクルとして検証したい箇所を発見して，2) 該当する要素を特定した上で，3) 要素へアクセスするためのコードを記述しなければならないためである．

まず，テストオラクルとして検証したい箇所を発見する際，問題1,2) で述べたように検証すべき箇所を属人的に探索しているため，テスターの経験や能力に応じてコストがかかってしまう．次に，該当する要素を特定するために，レンダリング結果とHTML文章の対応をテスターが考えなければならない．ただし，FirefoxやGoogle Chromeなどいくつかのブラウザでは，レンダリング結果とHTML文章の対応関係を視覚化するツールの支援を受けられる．最後に，多くの自動ブラウジングフレームワークでは，CSSセレクタやXPath，IDによる検索機能などを備えているものの，当該する要素にアクセスするコードの記述は一定の労力を要する．特に，JavaScriptを用いて動的にDOMツリーを変更すると，要素にアクセスするコードの記述が困難になるケースがある．このように，テストコードの記述にコストがかかるため，短絡的な工数削減によってテストがおざなりになる問題がある．

**問題4) テストコードの保守コストが高い：**HTML文章の構造に依存するようなテストコードを記述してしまうケースが多い．特に，前述したテストコードの記述コストを抑えるために，ユーザーのブラジング履歴からテストコードを自動生成する技法が存在するが，HTML文章の構造に依存した形でノードへアクセスするようなコードを生成してしまう．しかし，そのようなコードは仕様変更によってHTML文章の構造が変わると正常に動作しなくなる．加えて，Webアプリは一般のアプリケーションとは異なりアップデートが容易であるため，仕様変更の頻度が高い．

例えば，二つの値を加算した結果を表示するWebアプリを考える．図2はClosure Templatesを利用して，二つの値とその加算結果を表示するHTMLテンプレートである．Webアプリの実行時に{`$left`}と{`$right`}に加算する値，{`$answer`}に加算結果の値の文字列表現が埋め込まれる．図3はSeleniumを利用して，加算結果の値をブラウザから取得して検証するJavaのテストコードである．図3のテストコードはHTML文章の構造に強く依存しており，例えば，{`$answer`}を囲むタグを`div`から`span`に変更すると，加算結果の値を取得できなくなる．このように，テストコードの記述方法によっては，テストコードの保守コストが高くなる問題がある．

```
1 {template .addition}
2 <p>{$left}+{$right}=<div>{$answer}</div></p>
3 {/template}
```

**図 2　Closure Templates を利用して二つの値を加算した結果を表示する HTML テンプレート**

```
1 WebElement element = new ChromeDriver().findElement(By.cssSelector(“ p > div ”));
2 assertEquals("3", element.getText());
```

**図 3　Selenium を利用して加算結果の値を検証する Java テストコード**

## 4　テンプレート変数カバレッジ

図 2 のように，テンプレートエンジンではテンプレート変数の値を文字列に変換して，HTML 文章に埋め込んでいる．一般に，テンプレート変数を埋め込む部分を特殊な記法で記述するため，HTML テンプレートを解析することで動的な部分を容易に検出できる．なお，テンプレート変数を含む式を埋め込むケースも存在するが，ここでは単にテンプレート変数と呼び，両者を区別しない．

我々は，解決 1) HTML テンプレートを解析することでテストオラクルで検証すべき動的な部分を明瞭化して，また，解決 2) HTML テンプレートの動的な部分に着目したテンプレート変数カバレッジを提案することで，定量的にテストオラクルの品質を評価するを指標を提供する．$C_{tmp}$ をテンプレート変数カバレッジ，$V_{all}$ を HTML テンプレート中のテンプレート変数カバレッジの集合，$V_{test}$ をテストで参照するテンプレート変数カバレッジの集合とおいて，以下の (1) 式のように $C_{tmp}$ を定義する．なお，$V_{all} \supseteq V_{test}$ と $0 \leq C_{tmp} \leq 1$ が常に成り立つ．

$$C_{tmp} = \frac{V_{test}}{V_{all}} \tag{1}$$

図 2 と図 3 のテストコードでは，`left`，`right`，`answer` の 3 種類のテンプレート変数のうち，`answer` のみを参照しているため，テンプレート変数カバレッジは 33%（小数点以下切り捨て）である．もしも，`right` と `answer` を参照するようなテストコードであれば 66%，全てを参照すれば 100%になる．

テンプレート変数カバレッジは HTML 文章中の動的な部分について，テストオラクルが検証している割合を示す．したがって，テンプレート変数カバレッジが高くなればなるほど，テストによるバグの検出能力が向上する．なお，全てのテンプレート変数について検証を行うためには，最低限のテストシナリオが必要になるため，従来の実行ステップに対するテスト網羅率もある程度満たすことになる．

## 5　グレーボックス結合テストとテストコード自動生成ツール POGen

我々は，テンプレート変数カバレッジが 100%を満たせるように，仕様書のみではなくソースコードの一部である HTML テンプレートを利用する GB 結合テストを提案する．図 4 の左部で Web アプリの構成と提案する GB 結合テストの関係を示す．我々は GB 結合テストの実施を支援するために，HTML テンプレートを解析した情報に基づいて，Selenium を利用するスケルトンテストコードを自動生成するツール POGen を開発した．POGen は HTML テンプレートを解析することで動的な部分を抽出して，PageObjects デザインパターンに基づく保守性の高いクラスとその部分に対するアクセサメソッドを持つテストコードを生成する．以上から，解決 3,4) テストコードの記述コストおよび保守コストを低減する．

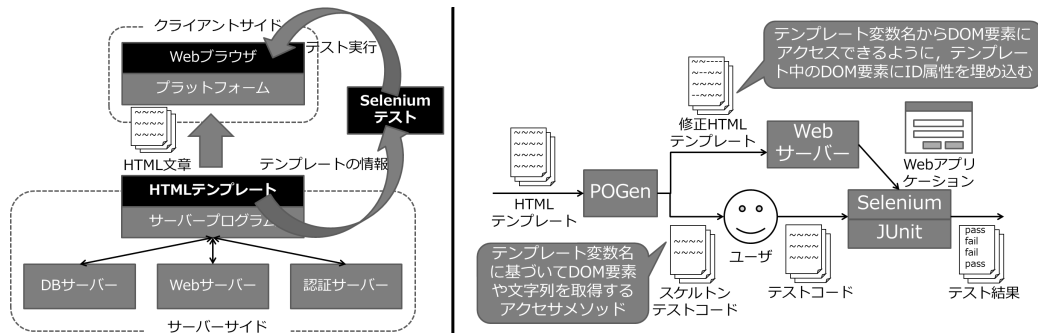


**図 4　Web アプリの構成と GB 結合テストの関係（左），POGen のアーキテクチャ（右）**

図 4 の右部で POGen のアーキテクチャを示す．POGen は HTML テンプレート解析部，HTML テンプレート変形部，スケルトンテストコード生成部から成る．

- HTML テンプレート解析部は，HTML テンプレートを解析して，動的な部分，すなわち，テンプレート変数の文字列表現を出力する部分を検出する．例えば，図 2 の例では，{`$left`}，{`$right`}，{`$answer`}が該当する．その上で，HTML テンプレート変形部に動的な部分の位置情報を伝える．現状では Closure Templates と ejs に対応しているが，テンプレートエンジン毎にパーサーを実装することで，解析可能な HTML テンプレートの種類を拡張可能である．
- HTML テンプレート変形部は，位置情報を元に動的な部分をテキストとして持つ HTML タグを特定して，該当する DOM ツリーの要素に Selenium から一意にアクセスできるような属性値を追加する．現状では ID 属性が付与されていない場合，一意な ID の属性値を生成して追加している．さらに，該当するテキストだけを抽出できるような特殊な HTML コメントを埋め込む．例えば，図 2 の HTML タグに POGen が ID 属性を追加すると図 5 のようになる．POGen は GB 結合テストを実施する際に元のファイルと置き換えるために，ID 属性を追加した修正 HTML テンプレートを出力する．

```
1 {template .addition}
2 <p id="_pogen_1">{$left}+{$right}=<div id="_pogen_2">{$answer}</div></p>
3 <!-- POGen:left:{$left} --><!-- POGen:right:{$right} --><!-- POGen:answer:{$answer} -->
4 {/template}
```

**図 5　HTML テンプレート変形部が ID 属性を追加した HTML テンプレート**

- スケルトンテストコード生成部は，追加した ID 属性を利用して DOM ツリーの要素へのアクセサメソッドを持つ，PageObjects デザインパターンに基づいた Selenium を利用する Java のテストコードを生成する．アクセサメソッドの名前をテンプレート変数名から決定することで，テストコードからテンプレート変数を持つ要素もしくはテキストにアクセスする際に，HTML 文章の構造に依存せずにメソッドを呼び出すだけでアクセスできる．
  POGen でテストコードを生成した後に仕様変更が起こると，修正 HTML テンプレートにおいては全体を再生成して，テストコードにおいては自動生成した部分のみ再生成して差し替える．その際，HTML 文章の構造が変化してもテンプレート変数の名前が変化しなければ，テストケースでは依然として同じ名前のアクセサメソッドを呼び出せる．

PageObjects デザインパターンは 1 ページを 1 クラスで表現することで，保守性

の高いテストコードを設計するためのデザインパターンである [4]．ページクラスは，ページ中の GUI コンポーネントや特定の HTML タグに該当する DOM ツリーの要素を参照するフィールド変数と，ページ中の特定のテキストを取得するメソッド，別のページに遷移するためのメソッドなどを持っており，ページクラスが持つメソッドを適切に呼び出すことでテストケースを記述できる．一般に，ページの HTML 構造が変化することで DOM ツリーへの参照方法が変化することは多いが，ページが持つ情報や遷移先が変化することは少ない．そのため，PageObjects デザインパターンでは，仕様変更の影響を受けやすい DOM ツリーの参照処理をページクラスに記述して，それを利用して仕様変更の影響を受けにくいテストケースを記述することで，仕様変更が起きた際の影響を抑えて保守コストを低減する．

スケルトンテストコードは，動的な部分を直接のテキストもしくは属性値として持つ要素を取得するメソッドと，動的な部分のテキストのみを生成した HTML コメントから取得するメソッドを提供する．POGen のユーザはスケルトンテストコードを元にテストケースの記述を追加することで，テストコードの記述コストを削減する．また，生成したアクセサメソッドを用いて，テンプレート変数が持つ要素もしくはテキストを全て参照するようなテストコードを記述することで，テンプレート変数カバレッジが 100%を満たせる．なお，動的な部分を持つ要素は，静的な部分との結合について検証する際や，動的な部分が JavaScript で正常に変更されたかどうか検証する際に必要である．例えば，図 5 に対して図 6 のようなスケルトンテストコードが生成される．生成した `AdditionPage` クラスは我々の `AbstractPage` クラスを継承していて，コンストラクタで `FindBy` アノテーションが付与されたフィールド変数が適切に初期化される．

```
1 public class AdditionPage extends AbstractPage {
2   @FindBy(id = "_pogen_1") private WebElement _left, _right;
3   @FindBy(id = "_pogen_2") private WebElement _answer;
4   WebElement getLeftElement() {/*省略*/}   WebElement getRightElement() {/*省略*/}
5   WebElement getAnswerElement() {/*省略*/} String getLeftString() {/*省略*/}
6   String getRightString() {/*省略*/}       String getAnswerString() {/*省略*/}
7 }
```

**図 6　足し算の結果を表示するページに対して POGen が生成するスケルトンテストコード**

典型的なテストケースでは，DOM ツリーの要素を取得して操作することで，テキストボックスへの入力やボタンの押下などを通してページの遷移を行う．そして，検証したページにおいて動的な部分を持つ要素に対して，テストシナリオを実行した際に期待される状態（テキストや属性値）かどうかを検証する．検証処理は JUnit などの既存のテスティングフレームワークが提供する Assert メソッドで記述できる．例えば，図 6 に対して図 7 のような JUnit テストケースを記述できる．

```
1 public class AdditionPageTest {
2   @Test public void add1And2() {
3     AdditionPage page = new TopPage(new ChromeDriver()).goToAdditionPage(1, 2);
4     assertEquals(page.getAnswerString(), "3");
5   }
6 }
```

**図 7　POGen が生成するスケルトンテストコードを利用した JUnit テストケース**

図 8 と以下の手順で POGen による GB 結合テストの流れを示す．

1. テスト対象の Web アプリの HTML テンプレートを POGen に入力する．
2. POGen が修正 HTML テンプレートとスケルトンテストコードを生成する．
3. スケルトンテストコードにテストケースを記述してテストコードを完成させる．
4. 修正 HTML テンプレートを使用する Web アプリをデプロイする．
5. 配置した Web アプリに対して記述したテストコードを用いてテストを実施する．

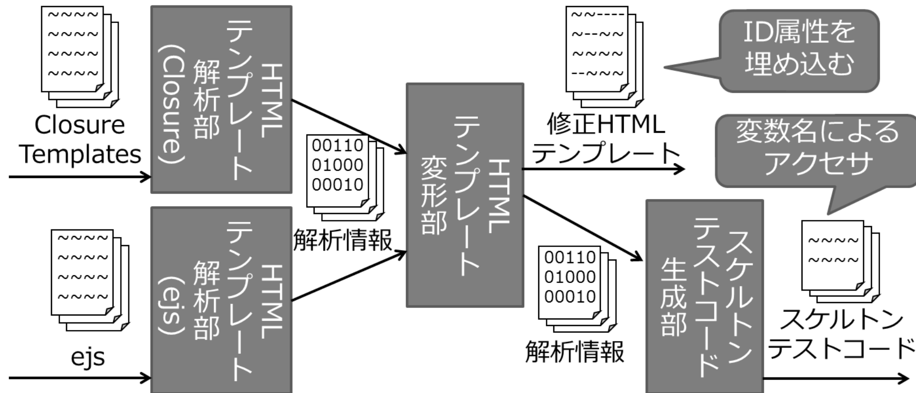


**図 8　POGen による GB 結合テストの流れ**

## 6　評価

**解決 1,2) テストオラクルが検証すべき箇所の明瞭化とテンプレート変数カバレッジ：**提案手法では HTML テンプレート中のテンプレート変数を抽出することで，動的な部分を網羅的に検出してアクセサメソッドを提供する．したがって，生成したメソッドがテストオラクルで検証すべき箇所を示しており，明瞭化に成功している．また，テンプレート変数カバレッジを利用することで，どの程度網羅的にテストオラクルが動的な部分を検証しているかどうか判断できる．そのため，テンプレート変数カバレッジを用いて属人性を排除することで，テストの品質を一定に保つことが可能である．以上から提案手法によって問題 1 と 2 の解決に成功している．

**解決 3,4) テストコードの記述・保守コストの削減：**我々が開発する TopCoder のような競技プログラミング向け Web アプリ (http://almond-choco-test.herokuapp.com/) において，提案手法を利用したテストコードの記述・保守の実験を行った．対象 Web アプリでは，ユーザーが問題の作成と編集ができ，さらに，既存の問題に対して解法となるプログラムのコードを投稿できる．問題はタイトル・説明・入力・期待値を有しており，Web アプリはユーザーの投稿したコードをオンライン上で実行して，その出力結果が期待値と等しいか答え合わせを行う．Web アプリはテンプレートエンジンである ejs を利用していて，問題の一覧および作成画面（Index），問題の編集および投稿画面（Problem），実行結果の画面（Result）に対して，それぞれ POGen を用いてスケルトンテストコードを生成した．表 1 に生成したスケルトンテストコードについて，空行やコメントを除いた有効行数とユーザーがテストコードとして記述が必要な行数を示す．コードの行数はページクラスとページクラスを利用して記述したテストケースに分けて示す．なお，ユーザーは問題の作成，問題の編集，コードの投稿と実行の 3 種類のテストケースを記述する．

POGen はテンプレート変数を持つ要素とテキストへのアクセサメソッドとページクラスを生成するため，ユーザーがテストケースを作成する際に，必要な記述量を

**表 1　POGen が生成したスケルトンテストコードとユーザーが記述したコードの有効行数**

| | Index | Problem | Result | テストケース | 合計 |
|---|---|---|---|---|---|
| POGen による生成 | 58 | 70 | 38 | 0 | 166 |
| ユーザーによる追記 | 11 | 12 | 0 | 38 | 61 |
| 合計 | 69 | 82 | 38 | 38 | 227 |

削減することができる．なお，a, input, button, textarea タグについては，テンプレート変数の有無に関わらず，name 属性もしくはテキストから名前を決定して，アクセサメソッドを生成する．そのため，CSS セレクタや XPath などの知識がないユーザーであっても，テンプレート変数の名前などに基づくアクセサメソッドを利用して，テストで必要な要素にアクセスするコードを容易に記述できる．本実験においては，およそ 7 割のコードに関しては POGen によって自動生成できた．

さらに，テストの保守について，ページのデザインを変更したところ，POGen を利用することで，自動生成したアクセサメソッドについては一切手を加える必要がなかった．POGen では，HTML 文章の構造に依存せずに，テンプレート変数を持つ要素とテキストを取得できるため，テストコードはページデザインの変更による影響をほとんど受けない．図 9 に実行結果の画面の HTML テンプレート（ejs）について，仕様変更前の内容を上部に，仕様変更後の内容を下部に示す．HTML テンプレート中のテンプレート変数の要素とテキストをどちらも同じメソッド，例えば，result 変数のテキストであれば，getResultString メソッドから取得可能である．ただし，テンプレート変数名を変えると，アクセサメソッドの名前も変化してしまうため，保守作業が必要になる．しかし，これらの変更はデザインの変更よりも稀であると考えられる．以上から提案手法によって問題 3 と 4 の緩和に成功している．

```
1 <!-- 仕様変更前のHTMLテンプレート（ejs） -->
2 <div><%= result %></div><br />
3 actual = [<span><%= out %></span>]<br />
4 expect = [<span><%= ex %></span>]
5 <!-- 仕様変更後のHTMLテンプレート（ejs） -->
6 <p> Your answer is <b><%= result %></b>!  </p>
7 <p> Your program's ouput is [<span><%= out %></span>].  </p>
8 <p> Then, our expected ouput is [<span><%= out %></span>].  </p>
```

**図 9　実行結果の画面の仕様変更前と後の HTML テンプレート（ejs）**

## 7　提案手法の制約

**テンプレート変数への依存：**提案手法は Web アプリの動的な部分を検出するために，HTML テンプレートを解析して，テンプレート変数を HTML 文章に埋め込んでいる箇所を特定する．しかし，テンプレートエンジンを利用しなくとも，Web フレームワークや DOM の API を利用することで，サーバーサイドとクライアントサイドの両サイドで HTML 文章を動的に生成・変形できるため，そのような技術を利用する Web アプリに対して提案手法を適用できない．

**テンプレート変数へのアクセサメソッドの再生性：**あるテンプレート変数が複数回出現する場合，各アクセサメソッドの名前に対して連番を振ることで区別する．そのため，仕様変更において既存のテンプレート変数への参照を追加する場合，メソッド名の連番が変化して，呼び出し側の修正が必要になるケースがある．

## 8 関連研究

Staats らは Gourlay らのフレームワークを拡張して，テスト対象のプログラム，テストコード，テストオラクルの関係を整理した [7]．また，彼らはテストの品質について議論する際に，テストカバレッジだけではなくテストオラクルについても言及すべきだと主張している．本論文で提案したテンプレート変数カバレッジは，テストオラクルの品質に関する観点が含まれており，彼らの主張に従っている．

Schuler らはテストオラクルに関する新しい指標として，ステートメントカバレッジを拡張したチェック済みカバレッジを提案している [8]．彼らの提案手法は Web アプリに限定しておらず，また，WB テストにおける手法であるため，本論文で議論しているような結合テストにおいて利用することは困難である．しかし，今回提案する GB 結合テストと併せて利用することで，よりテストの品質を向上できる．

小高らはアスペクト指向を用いて JSP をテストする手法を提案した [9]．彼らの提案手法では，JSP における動的な部分のテキストがユーザーが入力する期待値と一致するか検証するツールを提供する．しかし，期待値との比較しか行えないため，テキスト以外の情報を用いたテストが行えない．一方，我々の提案手法では動的な部分のテキストを持つ DOM ツリー上の要素も取得でき，その要素のテキストや属性値，兄弟や親子の要素についてテストオラクルで検証できる．そのため，当該する要素をクリックしたり状態を検証するなど，より柔軟なテストが可能である．

## 9 まとめと今後の展望

本論文では，従来の BB 結合テストにおける問題点を整理した上で，HTML 文章中の動的な部分に着目するテンプレート変数カバレッジを提案して，その上で，テンプレート変数カバレッジが 100%を満たせるような，GB 結合テストを提案した．さらに，HTML テンプレートを解析することで，PageObjects デザインパターンに基づき，動的な部分に対するアクセサメソッドを備えたスケルトンテストコードを自動生成するツール POGen を開発した．また，提案手法の有用性を示すために評価実験を行い，テストの記述・保守コストの削減に成功した．

今後は，テンプレート変数カバレッジに他のカバレッジを組み合わせることで，品質の高いテストの十分条件と成るようなカバレッジを提案する予定である．

## 参考文献


[1] Choudhary, Shauvik Roy and Prasad, Mukul R. and Orso, Alessandro: CrossCheck: Combining Crawling and Differencing to Better Detect Cross-browser Incompatibilities in Web Applications, *Proceedings of the 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation (ICST 2012)*, pp. 171–180 (2012).

[2] Jupiter Consulting: EJS - JavaScript Templates, http://embeddedjs.com/.

[3] Google: Closure Tools – Google Developers, https://developers.google.com/closure/templates/.

[4] Simon Stewart： PageObjects, http://code.google.com/p/selenium/wiki/PageObjects.

[5] Sprenkle, Sara and Pollock, Lori and Simko, Lucy: A Study of Usage-Based Navigation Models and Generated Abstract Test Cases for Web Applications, *Proceedings of the 2011 Fourth IEEE International Conference on Software Testing, Verification and Validation (ICST 2011)*, pp. 230–239 (2011).

[6] 坂本一憲, 東海政治ほか: Web アプリケーション開発における画面仕様書およびテスト仕様書の自動生成手法と開発プロセス改善の提案, ソフトウェア品質シンポジウム 2011, (2011).

[7] Staats, Matt and Whalen, Michael W. and Heimdahl, Mats P.E.: Programs, tests, and oracles: the foundations of testing revisited, *Proceedings of the 33rd International Conference on Software Engineering (ICSE 2011)*, pp. 391–400 (2011).

[8] Schuler, David and Zeller, Andreas: Assessing Oracle Quality with Checked Coverage, *Proceedings of the 2011 Fourth IEEE International Conference on Software Testing, Verification and Validation (ICST 2011)*, pp. 90–99 (2011).

[9] 小高敏裕, 上原忠弘, 片山朝子, 山本里枝子: アスペクト指向を利用した Web アプリケーションテストの自動化, *IPSJ SIG Notes 2007(33)*, pp. 97–104 (2007).