UQAC
Université du Québec
à Chicoutimi

# AMÉLIORATION DU PROCESSUS DE TEST DANS LES INTERFACES WEB PAR LA RECHERCHE D'INVARIANTS

**PAR IGOR PAULO DOMINGUES MARTINS**

**MÉMOIRE PRÉSENTÉ À L'UNIVERSITÉ DU QUÉBEC À CHICOUTIMI EN VUE DE L'OBTENTION DU GRADE DE MAÎTRE ÈS SCIENCES (M.SC.) EN INFORMATIQUE**

**QUÉBEC, CANADA**

# RÉSUMÉ

L'ingénierie logicielle est un vaste domaine. L'une de ses branches pertinentes est le test appliqué aux interfaces utilisateur (IU). L'IU se connecte à une série d'éléments web pour représenter la mise en page de la page web qui est affichée à l'utilisateur. Ces éléments web composent des caractéristiques qui aident les utilisateurs dans leur expérience lorsqu'ils naviguent sur les pages web, telles que les performances et la réactivité. Cependant, les pages web peuvent contenir des bogues de mise en page qui interfèrent avec l'expérience utilisateur.

La qualité de la mise en page doit être maintenue tout au long du processus de codage. Les développeurs de logiciels peuvent rechercher visuellement et manuellement des bogues de mise en page, mais parfois ces bogues ne sont pas visibles à l'œil nu. Une autre ressource consiste à utiliser des outils pour appliquer des tests, mais les développeurs doivent toujours écrire des tests de code. Les deux stratégies peuvent être très laborieuses. Ainsi, nous proposons le développement d'une approche utilisant le concept d'invariants pour simplifier les tâches des développeurs.

Dans cette étude, nous avons développé un programme dans lequel nous avons mis en œuvre le concept d'invariants pour extraire des conditions sur les pages web. Avec ces conditions, nous pouvons identifier les violations d'invariants, qui représentent en d'autres termes des bogues de mise en page. Nous avons appliqué cette approche à un ensemble de pages web générées en laboratoire et à un autre ensemble de pages web copiées depuis Internet.

Les résultats montrent que notre programme a identifié des bogues de mise en page dans 100% des pages web générées en laboratoire et dans 60% des pages web copiées depuis Internet. Le temps nécessaire pour effectuer cette procédure d'extraction varie en fonction de la taille de la page web, mais il est raisonnable. Nous considérons que notre approche peut servir de base pour de futurs outils à commercialiser sur le marché. De plus, elle représente une contribution importante pour le monde académique, car elle ouvre des possibilités pour de futures expérimentations et améliorations en utilisant les concepts appliqués dans cette étude.

## ABSTRACT

Software engineering is a wide theme. One of its relevant branches is testing applied on user interfaces (UI). The UI connects to a series of web elements to represent the web page layout that is displayed to the user. These web elements compose characteristics that help users in their experience while they navigate web pages such as performance and responsiveness. Nevertheless, web pages may contain layout bugs that interfere with the user's experience.

Layout quality has to be maintained throughout the coding process. Software developers can visually and manually search for layout bugs, but sometimes these bugs are not visible to the naked eye. Another resource is to use tools to apply tests, but developers still need to write code tests. Both strategies might be very laboring as well. Thus, we propose the development of an approach using the concept of invariants to simplify developers' tasks.

In this study, we developed a program where we implemented the concept of invariants to mine conditions on web pages. With these conditions, we can identify invariants violations, which in other words represent layout bugs. We applied this approach in a set of laboratory-generated web pages and another set of web pages copied from the Internet.

The results show that our program identified layout bugs in 100% of laboratory-generated web pages and 60% of web pages copied from the Internet. The time taken to perform this mining procedure oscillates according to the web page size, but it is reasonable. We considered that our approach may serve as a base for future tools to be commercialized in the market. Also, it represents a relevant contribution to the academy, because it opened possibilities for future experiments and improvements using the concepts applied in this study.

## TABLE DES MATIÈRES

## LISTE DES TABLEAUX

## LISTE DES FIGURES

## LISTE DES ABRÉVIATIONS

| | |
|---|---|
| **WP** | Web page |
| **DOM** | Document Object Model |
| **UX** | User experience |
| **UI** | User interface |
| **URL** | Uniform Resource Locator |
| **API** | Application programming interface |
| **CSS** | Cascading Style Sheets |
| **RQ** | Research question |
| **WWW** | World wide web |
| **HTTP** | HyperText Transfer Protocol |
| **HTML** | HyperText Markup Language |

## DÉDICACE

*Je dédie ce travail à mes amis que j'ai rencontrés à Chicoutimi. Ils m'ont soutenu lors de mon arrivée, m'ont accueilli et m'ont introduit à cette nouvelle vie ici dans la région du Saguenay. Je le dédie également à ma famille et à mes amis dans mon pays d'origine, en particulier à ma chère mère, qui est un exemple de force et de vertu pour moi. Merci à tous pour les bonnes expériences et les bases que vous m'avez données pour surmonter mes défis.*

## REMERCIEMENTS

Je tiens à exprimer ma gratitude envers l'UQAC de m'avoir donné l'opportunité d'être l'un de vos étudiants. Je suis reconnaissant envers la ville de Chicoutimi d'avoir été mon foyer tout au long de mon programme d'études. J'exprime ma reconnaissance aux professeurs de l'Université de l'État de Santa Catarina (UDESC) au Brésil qui m'ont initié au monde de l'informatique, en particulier le professeur Fernando dos Santos qui m'a ouvert les portes de cette opportunité de maîtrise.

Je tiens à remercier le projet MITACS et le professeur Fehmi Jaafar de m'avoir offert une première opportunité de recherche. Ce démarrage m'a donné des conditions financières qui ont rendu cette expérience possible. Je suis reconnaissant au Professeur Fabio Petrillo qui m'a invité et a joué un rôle crucial dans la réalisation de cette expérience au Canada. Le Professeur Fabio est un chercheur expérimenté, auteur d'articles et réviseur de travaux scientifiques. Sans sa participation, je n'aurais pas eu la chance d'être ici, en train d'écrire ce mémoire. Le Professeur Fabio est également le codirecteur de recherche et a apporté des contributions importantes à son bon développement.

Enfin, je tiens à remercier le Professeur Sylvain Hallé d'avoir accepté d'être mon directeur de recherche. Le Professeur Sylvain a de nombreuses publications dans le domaine de l'informatique et est devenu l'un des chercheurs les plus importants de ce domaine. Il est largement reconnu et apprécié par la communauté. Ainsi, je le remercie profondément de m'avoir fourni le soutien dont j'avais besoin pour mener à bien le travail que j'ai accompli. Il m'a apporté les idées, les ressources et les orientations nécessaires pour mener à bien cette recherche. Sans les contributions de toutes les personnes mentionnées, cette expérience n'aurait pas été possible.

Je suis sincèrement reconnaissant envers toutes les personnes et institutions qui ont contribué à façonner mon parcours académique et à rendre cette expérience possible.

# CHAPITRE I

# INTRODUCTION

## 1.1 CONTEXTE

De nos jours, les applications web occupent une part de marché importante dans l'économie mondiale. Selon la source Statista [1], en 2021, on comptait près de 1,9 milliard de sites web à travers le monde. La plateforme StatsFind [2] a calculé qu'en 2023, ce nombre était passé à environ 2 milliards —soit plus de cent millions de nouveaux sites en à peine deux ans. Cette croissance est due au vaste marché du développement, qui, selon Siteefy [3], produit environ 250 000 applications web chaque année. Ce nombre important d'applications web impacte la manière dont les sociétés organisent leurs routines. Precedence Research [4] indique qu'en 2023, l'industrie des technologies de l'information est estimée à 650 milliards de dollars américains et devrait atteindre 1,7 billion de dollars en 2032. Cette croissance est le résultat de la demande, des investissements et du travail des professionnels.

Les chiffres concernant les applications web sont impressionnants. Cependant, il est important d'ajouter que de nombreuses applications de type « desktop » sont en réalité des applications web qui s'exécutent dans un conteneur. Cela est désormais possible grâce à des outils de développement multi-plateforme tels qu'Electron, Vue 3 et Vite. Cette approche produit ce que l'on appelle communément des applications web progressives, qui ne sont pas toujours prises en compte dans les statistiques à propos du marché du web.

---

1. https ://www.statista.com/chart/19058/number-of-websites-online/
2. https ://www.statsfind.com/how-many-websites-are-there-in-the-world-a-daily-calculator/
3. https ://siteefy.com/how-many-websites-are-there/
4. https ://www.precedenceresearch.com/software-market

Comme nous l'avons décrit, l'industrie des applications web a évolué et aujourd'hui, il existe des techniques professionnelles pour le développement de logiciels. Ainsi, le génie logiciel s'est développé en tant que discipline axée sur l'application des meilleures pratiques pour le développement de logiciels. Dans le cas du développement web, le code de programmation peut être divisé en deux parties : le *backend*, qui gère la communication avec les bases de données, et le *frontend*. Selon Dresher [2], le développement frontend correspond au le code côté client, qui s'exécute typiquement à travers l'interface utilisateur du navigateur. Cette interface permet la présentation logique des fonctions, et doit être utilisée en prenant en compte la compatibilité des navigateurs, leur performance et leur réactivité.

Une partie du génie logiciel vise à assurer la qualité des produits, et une composante importante de cette activité consiste à effectuer des tests. Ainsi, les développeurs de logiciels peuvent appliquer des tests sur leur code backend et frontend afin d'identifier des bogues de diverses natures. Ces tests peuvent être réalisés manuellement, par exemple lorsqu'un testeur utilise l'interface logicielle pour y saisir des données ou interagir avec les éléments d'une page. Ils peuvent également être effectués automatiquement grâce à des tests codés qui vérifient et valident les fonctionnalités développées.

Dans le développement frontend, effectuer des tests implique généralement d'interagir avec l'interface, et donc d'évaluer des conditions sur cette interface. Ceci sous-entend que l'interface peut également être sujette à des bogues, notamment des bogues relatif à l'affichage des éléments dans une page web. Dans une étude de Hallé et al. [1], on explique que les bogues dans les applications web peuvent varier du positionnement inapproprié d'éléments web sur la page à des problèmes plus graves compromettant la navigation de l'utilisateur dans l'interface. Les auteurs affirment également qu'il existe un manque d'outils pour exprimer les propriétés souhaitables du contenu et de l'affichage des éléments sur une page web.

Dans la pratique, lorsque les développeurs reçoivent un prototype d'interface, ces derniers peuvent opter pour différentes façons de l'implémenter. Cependant, les interfaces ont souvent des caractéristiques spécifiques à suivre, qui doivent correspondre aux besoins de l'utilisateur ou du client. On doit s'assurer que ces caractéristiques de mise en page soient maintenues tout au long du processus de développement.

À cet égard, le concept d'*invariant* peut jouer un rôle crucial dans la satisfaction des exigences du processus de codage. Flusser [3] définit les invariants comme des caractéristiques qui ne changent pas de valeur dans certaines fonctions. De leur côté, les auteurs de [4] définissent les invariants comme la comparaison de deux expressions abstraites que l'on affirme être égales ; ces expressions doivent donner des résultats du même type pour correspondre. Ainsi, les expressions qui donnent des nombres doivent donner les mêmes nombres ; les expressions qui donnent un objet doivent donner des objets ayant des identités égales ; les expressions qui produisent une séquence doivent énumérer leurs éléments dans le même ordre, et ainsi de suite.

Il est reconnu que le concept d'invariants est utile dans le développement frontend. Dans ce mémoire, nous proposons d'appliquer ce concept pour identifier dee manière automatique des patrons d'éléments représentant des conditions sur les pages web. Pour cette identification, nous nous sommes uniquement intéressés à la disposition de la mise en page des éléments web sur les pages web.

## 1.2 PROBLÉMATIQUE ET MOTIVATION

Dans ce mémoire, nous proposons d'appliquer ce concept pour identifier des patrons d'éléments représentant des conditions sur les pages web, en nous concentrant sur la disposition et la mise en page des éléments. En effet, identifier de tels bogues de mise en page implique généralement une inspection visuelle manuelle. Cette tâche n'est pas souhaitable car elle prend

du temps et les bogues de mise en page peuvent également échapper à l'œil nu. Il serait de loin préférable d'automatiser le processus à l'aide de cas de test pouvant être exécutés autant de fois que nécessaire.

On verra dans la suite que certains outils sont disponibles pour faciliter le processus de test. Cependant, comme pour tout type de test, les tests visant à détecter les bogues d'interface supposent qu'ils sont écrits par un développeur. Cela signifie que ces derniers doivent énumérer, sous forme d'assertions ou de conditions, tout ce qui constitue l'apparence ou le comportement attendu de la page en cours de développement. Bien entendu, s'il n'y a pas d'assertion, il est impossible de découvrir un bogue lui étant associé.

Or, ce travail est fastidieux et de très grande ampleur, même pour une page web triviale. À titre d'exemple, nous mentionnons la figure 1.1 qui fait partie d'un répertoire décrit dans l'étude citée plus tôt [1]. En haut, on peut voir une page web contenant un tableau avec 8 colonnes, mais en regardant attentivement, on constate que la dernière colonne omet certaines informations. Dans la partie inférieure de la figure, nous montrons, à l'intérieur des lignes rouges, le contenu manquant. Dans ce type de de mise en page, les utilisateurs ne se rendent pas compte du contenu manquant, même les développeurs qui ont construit la page web peuvent ne pas le réparer au début. La largeur du tableau dépasse la largeur de la page et des composants peuvent être inaccessibles.

Dans l'étude de Hallé et al. [1], les auteurs présentent un outil nommé Cornipickle, dans lequel développeurs peuvent écrire des conditions de mise en page en utilisant un langage déclaratif. Les auteurs affirment que ce langage peut simplifier l'écriture de conditions de test par rapport à l'écriture de code de plus bas niveau. Néanmoins, même avec un tel outil, le développeur doit imaginer et gérer des dizaines ou des centaines de conditions pour chaque

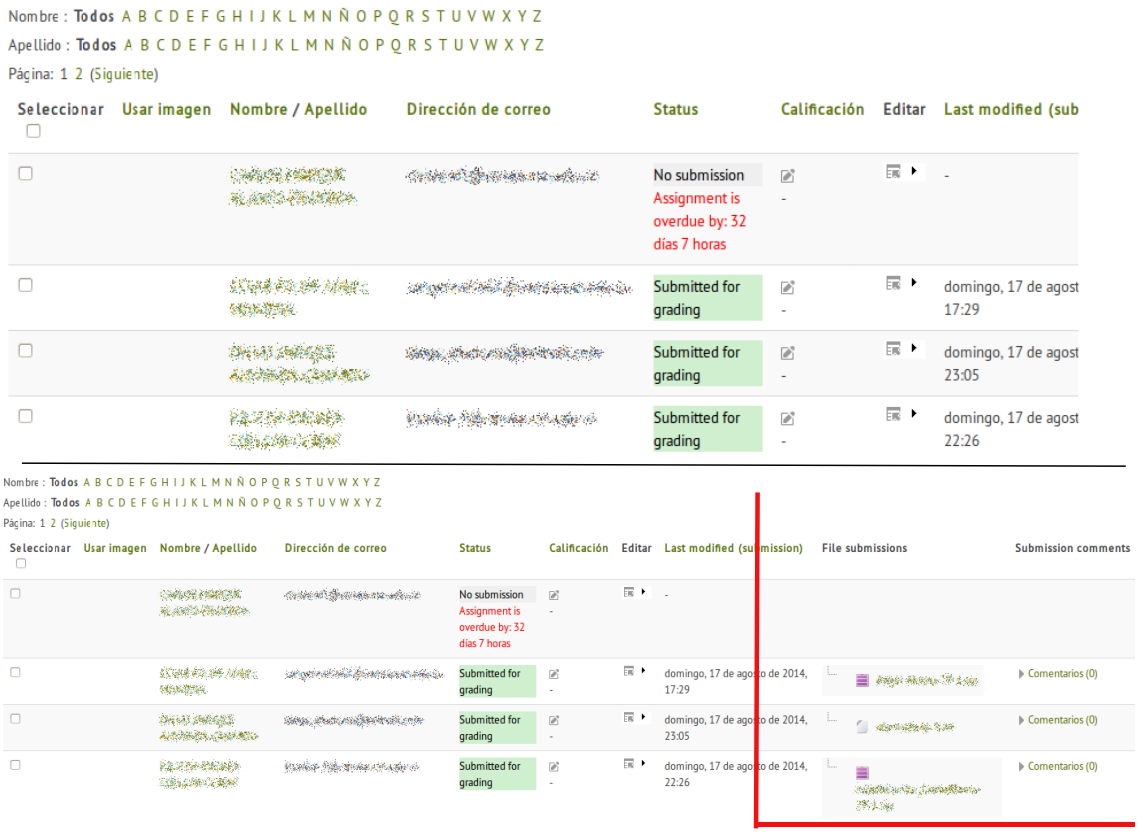



**FIGURE 1.1 : Exemple de bogue d'interface**
**©[1]**

page. Chaque condition est certes plus facile à rédiger, mais on doit malgré tout codifier un grand nombre de contraintes.

Le présent mémoire vise à résorber en partie ce problème. On rappelle l'importance des tests sur les interfaces et de la nécessité de les automatiser. En plus d'automatiser l'exécution des tests, pourrait-on également automatiser la découverte des conditions que ces tests doivent évaluer, de telle sorte que le développeur soit épargné de les écrire lui-même ?

## 1.3 OBJECTIFS ET HYPOTHÈSE

Comme on le verra plus loin dans ce mémoire, bien qu'il existe quelques études sur les invariants extraits des pages web, nous n'avons trouvé aucune référence décrivant un outil ou un banc d'essai visant à automatiser la découverte des conditions sur les pages web. L'objectif du mémoire vise à répondre à la question mentionnée ci-dessus en exploitant le concept d'invariant. Dans le cas présent, on appellera *invariant* une condition sur l'apparence d'une page web qui reste vraie dans un ensemble de pages pris comme référence.

Le principe de la solution proposée s'énonce ainsi : à partir d'un ensemble de pages fournies par un développeur, qui représente un échantillon de pages avec une apparence réputée valide, on cherchera à générer un grand nombre de conditions de mise en page et identifier celles qui sont des invariants sur cet ensemble.

L'intuition derrière ce principe est que si une condition est vraie sur toutes les pages de l'échantillon, elle a de bonnes chances d'être une contrainte réelle que les pages de l'application web développée doivent respecter. Au contraire, si une condition n'est vraie que sur une ou très peu de pages, il s'agit probablement d'une « anecdote » qui ne se généralise pas à toutes les pages du site.

Pour nous guider vers notre but, nous envisageons l'hypothèse suivante :

> ***Hypothèse :*** *Il est possible automatiser l'énonciation des conditions à vérifier en appliquant le concept d'invariants.*

Pour valider cette hypothèse, nous étudierons les questions de recherche (QR) suivantes :

**QR1 : Le temps de traitement nécessaire pour extraire les invariants d'une page web est-il acceptable ?** En répondant à cette question de recherche, notre objectif est de mesurer le temps de traitement de notre algorithme. On s'attend à ce que le processus d'extraction

soit limité à quelques minutes, sinon les utilisateurs pourraient être réticents à adopter la solution.

**QR2 : Quelle proportion des conditions générées sont des invariants sur un échantillon de pages ?** En répondant à cette question de recherche, notre objectif est d'évaluer la capacité de notre approche à détecter des invariants en distinguant les conditions vraies des fausses.

**QR3 : Les invariants violés correspondent-ils à des bogues d'interface ?** En répondant à cette question de recherche, notre objectif est de comprendre si les invariants trouvés par notre algorithme, lorsqu'ils sont violés sur une page en particulier, font référence à de réels bogues identifiés sur nos pages web. Ce faisant, nous pourrons mesurer son niveau de précision.

La réponse à ces trois questions de recherche passe par l'atteinte de trois objectifs spécifiques, définis comme suit :

- Coder un programme capable d'extraire et de signaler les invariants d'un certain page web.
- Effectuer des tests avec ce programme pour valider notre approche sur des pages web statiques.
- Recueillir des données et analyser l'efficacité pour détecter les ruptures d'invariants.

En particulier, nous avons développé un outil avec une configuration facile et rapide. Cet outil exécute un algorithme capable d'identifier automatiquement des conditions sur les pages web, distinguant les vraies des fausses. L'outil a été testé sur des ensembles de pages web et a démontré sa capacité à mettre en évidence des bogues de mise en page à l'écran

et à présenter des rapports conviviaux. Nous croyons que cette nouvelle approche proposée présente le potentiel d'aider les professionnels dans le développement des pages web.

## 1.4 ORGANISATION

Les chapitres restants de ce mémoire sont divisés comme suit :

Dans le chapitre 2 (Notions de base et bogues d'interface), nous introduisons certains concepts que nous considérons pertinents pour notre recherche. Tout d'abord, nous présentons les langages web HTML et CSS. Ensuite, nous présentons les concepts élémentaires relatifs aux bogues d'interface.

Dans le chapitre 3 (État de l'art de la détection de bogues d'interface), nous explorons l'état actuel des études connexes que nous avons trouvées au cours de notre recherche. Nous distinguons six études et les divisons en celles qui se concentrent sur les événements défectueux et celles qui se concentrent sur l'identification des bogues de mise en page.

Dans le chapitre 4 (Solution proposée), nous présentons comment nous générons des invariants à partir des conditions trouvées sur les pages web. Ensuite, nous présentons l'utilisation possible de ces invariants.

Dans le chapitre 5 (Implémentation), nous décrivons comment nous utilisons les concepts expliqués précédemment pour construire l'outil. Ainsi, nous présentons les technologies que nous avons choisies, l'architecture de l'outil, le processus d'extraction des invariants et l'identification des bogues de mise en page.

Dans le chapitre 6 (Application de l'expérience), nous décrivons le protocole expérimental que nous utilisons pour tester l'outil que nous avons développé. Ainsi, nous expliquons initialement notre échantillon de pages web générées en laboratoire, puis notre échantillon

de pages web existantes sur l'internet, puis nous exposons les résultats, les connectant à nos première, deuxième et troisième questions de recherche.

Finalement, le chapitre 7 conclut ce mémoire en résumant ses contributions, et en discutant ses limitations et les possibilités de travaux futurs.

# CHAPITRE II

# NOTIONS DE BASE ET BOGUES D'INTERFACE

Dans le dernier chapitre, nous avons introduit notre sujet et exposé la structure de notre recherche. Nous avons tout d'abord expliqué la division du développement de logiciels web entre le « backend » et le »frontend ». Ensuite, nous avons exposé nos préoccupations concernant les tests d'interface concernant la mise en page et les possibilités d'utilisation des invariants pour aider dans cette tâche. Nous également avons expliqué que les invariants permettent de comparer deux entités et de rechercher des égalités entre elles.

Ensuite, nous avons présenté notre problème de recherche, qui se concentre sur les limitations actuelles des approches de recherche de bogues de mise en page dans les sites web. Pour faire face à ces limitations, nous avons défini comme principal objectif le développement d'une approche utilisant des invariants pour valider les pages web. Cette approche sera basée sur l'implémentation d'un outil, suivi de tests et d'une collecte de données visant à en mesurer l'efficacité.

Nous émettons l'hypothèse que cette approche peut automatiser les déclarations conditionnelles à l'aide d'invariants. Les données recueillies seront utilisées pour valider cette hypothèse en répondant aux questions de recherche liées au temps de traitement, à la proportion de conditions vraies et aux liens entre les invariants trouvés et les bogues d'interface.

Dans le présent chapitre, nous prenons un pas de recul et abordons les langages du web, plus précisément le HTML et le CSS. Nous présentons également certains concepts relatifs au DOM (Document Object Model) et aux bogues d'interface.

## 2.1 LANGAGES DU WEB

Les navigateurs jouent un rôle crucial dans l'interaction avec les utilisateurs, car ils permettent le rendu du code source qui est généralement caché et incompréhensible pour la majorité des utilisateurs. Les navigateurs affichent du contenu tel que du texte, des images, des vidéos, des tableaux et divers composants d'interaction tels que des boutons et des champs de saisie.

Nous pouvons également considérer une page web comme une forme d'interface utilisateur, car elle affiche des données provenant d'un serveur et offre la possibilité d'interaction avec l'utilisateur. Pour fournir une interface conviviale, les navigateurs reposent sur certaines technologies telles que le HTML (HyperText Markup Language) et le CSS (Cascading Style Sheets). Ces deux technologies sont étroitement liés à notre recherche, et pour une meilleure compréhension, nous allons explorer davantage leurs concepts fondamentaux dans la suite.

### 2.1.1 HTML

Selon Cook et Schultz [5], le HyperText Markup Language (HTML) est un langage de codage informatique qui convertit du texte brut en un document pouvant être affiché dans un navigateur, en se connectant au World Wide Web (WWW). Le HTML est un langage ouvert et librement disponible ; il n'est pas contrôlé par une entreprise ou une personne en particulier. Il suit une syntaxe standardisée pour assurer une interprétation universelle par les ordinateurs du monde entier. Ces règles de syntaxe sont établies par l'organisation à but non lucratif appelée World Wide Web Consortium (W3C), qui est responsable de la définition des normes techniques du HTML.

Le HTML a été lancé dans les années 90 et a subi des améliorations depuis lors. De nouvelles versions ont été introduites pour relever les défis émergents. Comme mentionné

```html
<!DOCTYPE html>
<html lang="en">
  <head>
    <title>My Page</title>
  </head>
  <body>
    <h1>Hello world</h1>
  </body>
</html>
```

**FIGURE 2.1 : Représentation d'une structure HTML de base**

dans [6], le HTML5 est la dernière version du HyperText Markup Language (HTML), équipée de nouveaux outils et fonctionnalités pour gérer des technologies telles que JavaScript. Wilson ajoute que le HTML5 comprend des fonctionnalités qui s'adaptent à la gamme diversifiée d'appareils utilisés aujourd'hui, tels que les ordinateurs personnels, les téléphones intelligents et les tablettes [7]. Le HTML5 a également introduit de nouveaux éléments pour gérer les médias couramment utilisés, tels que l'audio et la vidéo, dans les pages web.

Le HTML est structuré à l'aide de balises, qui sont des mots-clés définissant comment le contenu est formaté et affiché par un navigateur. Comme l'explique Wilson [7], un élément HTML se compose d'une balise d'ouverture (<>) suivie d'un attribut d'élément, du contenu appartenant à cet élément et d'une balise de fermeture (</>). Par exemple, la balise <h1>Titre</h1> représente un élément de titre, où <h1> est la balise d'ouverture, "Titre" est le contenu et </h1> est la balise de fermeture. Grannell [8] mentionne que les balises peuvent également être imbriquées pour créer des éléments parent-enfant.

La Figure 2.1 donne un exemple simple de fichier HTML. Comme nous pouvons le voir dans la figure, la balise d'ouverture <html lang="en"> définit le début du document HTML et,

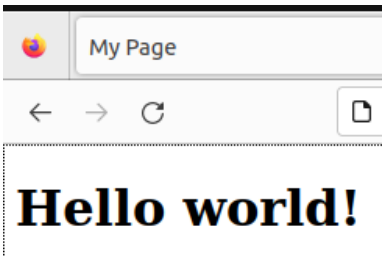



**FIGURE 2.2 : Représentation du résultat une structure HTML de base**

par conséquent, </html> la fin de celui-ci. Cette balise d'ouverture est également considérée comme l'élément « racine ».

Le document HTML peut être divisé en parties. Dans la figure 2.1, nous voyons les éléments enfants de la racine, à savoir head et body. L'élément head contient des métadonnées utilisées par les navigateurs, les robots d'exploration web et les moteurs de recherche. L'élément body contient le contenu du document : les informations qui sont rendues dans le navigateur pour les utilisateurs doivent donc être placées dans l'élément body. Le body doit suivre l'élément head, et aucun des deux ne peut être dupliqué dans le même document.

Le résultat de la structure HTML représentée dans la figure 2.1 est illustré dans la figure 2.2. Nous voyons que la balise <title>My Page</title> donne un nom au document qui apparaît dans le titre du navigateur, tandis que la balise <h1>Hello world !</h1> affiche le texte qui apparaît dans le corps du document.

Le body peut contenir des éléments enfants qui servent à organiser le contenu d'un document. La liste des éléments est longue, mais les principaux sont les suivants :

- **Header** : Il s'agit d'une balise pour la partie supérieure d'une page web, qui peut contenir des logos et des barres de navigation.
- **Footer** : Il s'agit d'une balise pour la partie inférieure d'une page web, qui contient généralement des informations sur les droits d'auteur et des liens.

- **Section** : Il s'agit d'une balise qui peut être placée dans des zones distinctes et qui contient généralement des éléments principaux tels que des listes et des images.
- **Aside** : Il est utilisé pour le contenu indirectement lié au contenu principal d'une page web. Il peut contenir une barre latérale de navigation, par exemple.
- **Div** : C'est l'élément le plus courant dans les pages Web. Il sert à diviser ou regrouper du contenu.

Il est important de comprendre que lors du développement de pages HTML, et particulièrement lors de l'écriture de règles CSS (discuté plus loin), nous pouvons utiliser certains attributs pour désigner les éléments visés. Par conséquent, une balise peut être identifiée par un attribut id ou class, comme l'explique Durocher [9] :

- **class** : Il est possible d'identifier un groupe d'éléments en définissant son attribut de classe à l'intérieur des éléments de balise visés. Exemple : <p class="callout">. Dans cet exemple, tous les éléments qui reçoivent "callout" feront partie de la même « classe » et pourront être accédés en bloc.
- **id** : L'attribut id sert à identifier un élément unique dans une page web. Il fonctionne de manière similaire à la classe, mais dans ce cas, l'identification est spécifique à l'élément qui a la reçu. Exemple : <p id="callout">.

Coulson décrit le HTML comme un document avec une structure arborescente hiérarchique [10]. Une autre manière de représenter un document HTML est avec le DOM (Document Object Model). La documentation de développement Mozilla [5] indique que le DOM est une représentation des objets qui composent la structure et le contenu d'une page web. De plus, il permet la manipulation de la structure, du style et du contenu des pages web,

5. https ://developer.mozilla.org/

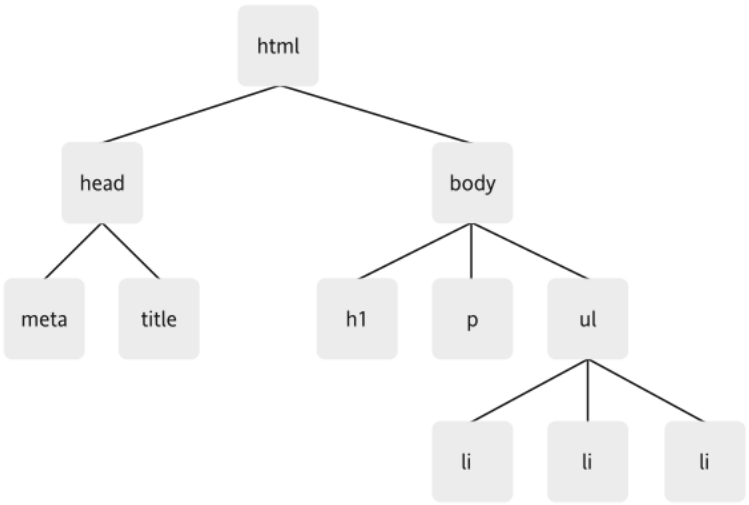



**FIGURE 2.3 : Représentation d'un arbre DOM**
**©[11]**

car c'est également une interface de programmation pour les documents web. Le DOM représente donc les documents HTML sous la forme d'un arbre d'objets ; ces objets sont également appelés nœuds. Les objets possèdent des propriétés, des méthodes et des événements liés au document HTML. La figure 2.3 illustre un exemple d'arbre DOM représentant le document auquel nous avons déjà fait référence.

Selon [12], l'objet racine d'un arbre est appelé un nœud de document et sert de point d'entrée pour une page web. Les éléments ajoutés sous le nœud de document sont appelés des nœuds d'élément. Les éléments peuvent être liés à d'autres éléments selon trois types de relations, à savoir parent, frère et enfant. Pour expliquer ces concepts, nous faisons référence à la figure 2.3 une fois de plus. Nous pouvons voir que l'élément html possède deux enfants, head et body. Par conséquent, head et body sont des nœuds frères car ils ont le même parent.

Le DOM peut être accédé à l'aide de langages de script, JavaScript étant le plus courant. Il s'agit d'un langage de programmation combinant des caractéristiques orientées objet, impératives et déclaratives. Comme nous l'avons mentionné précédemment, les langages de

```html
<!DOCTYPE html>
<html lang="en">
  <head>
    <title>My Page</title>
  </head>
  <body>
    <h1 id="h1BodyId">Hello world</h1>
  </body>
  <script>
    const elem = document.getElementById("h1BodyId");
    console.log(elem);
  </script>
</html>
```

**FIGURE 2.4 : Représentation de la structure HTML avec la méthode getElementById**

script permettent de manipuler des nœuds du DOM ; mais encore faut-il obtenir une référence à ces nœuds. Shute explique qu'il existe quatre approches pour identifier un nœud spécifique ou un groupe de nœuds [12].

La première manière consiste à utiliser l'ID d'un nœud. En JavaScript, nous pouvons utiliser la méthode document.getElementById( id ). Ainsi, chaque élément HTML peut se faire attribuer un ID, qui peut être utilisé comme paramètre dans la méthode d'identification. La méthode renvoie le nœud en fonction du paramètre qu'elle a reçu. Si aucun nœud ne possède l'ID en question, elle renvoie plutôt la valeur `null`. La figure 2.4 représente comment ce mécanisme fonctionne. La balise h1 se fait donner un ID nommé "h1BodyId". Ensuite, à l'intérieur de la balise script, nous utilisons la méthode getElementById. Elle permet d'obtenir une référence au nœud spécifié et l'affiche à l'aide de la fonction console.log(). Le résultat du code de la figure 2.4 est visible dans la figure 2.5.

Le deuxième mécanisme se réfère au nom de balise du nœud, via la méthode document.getElementsByTagName (nom). Le paramètre fait référence au nom d'une balise HTML,

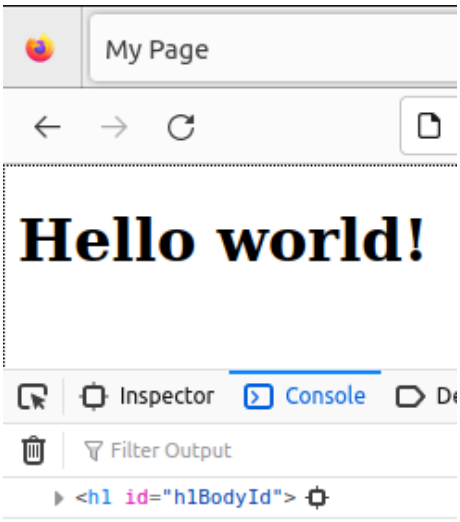



**FIGURE 2.5 : Représentation du résultat d'une structure HTML avec la méthode getElementById**

et la méthode renvoie la liste de tous les éléments ayant ce nom de balise. Comme pour la premièr méthode, nous représentons celle-ci dans un document HTML illustré dans la figure 2.6. On peut voir une balise div à l'intérieur de l'élément body. Le résultat de l'appel à la méthode est représenté dans la figure 2.7. Nous pouvons voir dans la console du navigateur qu'une liste HTMLCollection de taille un est retournée, contenant l'élément div auquel nous avons fait référence.

Le troisième moyen d'identification est la valeur de l'attribut `class`, via la méthode document.getElementByClassName(nom). Le paramètre "nom" dans la méthode recherchera toutes les éléments du document ayant un nom de classe correspondant au paramètre, ce qui renvoie une liste d'éléments. Le fonctionnement est illustré dans la figure 2.8. Cette fois, l'élément body contient deux éléments avec un nom de balise distinct, à savoir div et span. Malgré cela, la méthode d'identification est capable de récupérer les deux, car ils ont font partie de la même classe. Encore ici, le résultat est affiché dans la console du navigateur, comme le montre la figure 2.9.

```html
<!DOCTYPE html>
<html lang="en">
  <head>
    <title>My Page</title>
  </head>
  <body>
    <h1 id="h1BodyId">Hello world</h1>
  </body>
  <script>
    const elem = document.getElementsByTagName("div");
    console.log(elem);
  </script>
</html>
```

**FIGURE 2.6 : Représentation de la structure HTML avec la méthode getElementsByTagName**

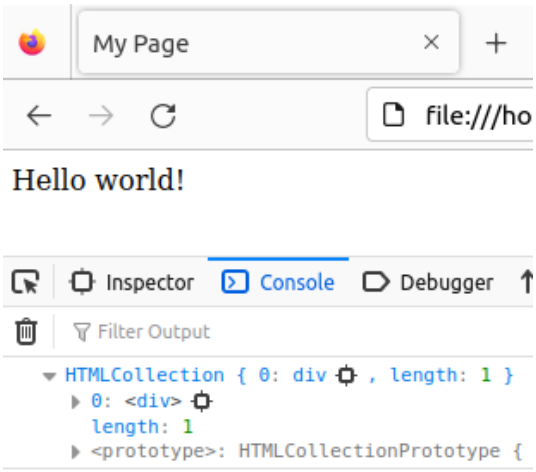



**FIGURE 2.7 : Représentation du résultat d'une structure HTML avec la méthode getElementsByTagName**

```html
<!DOCTYPE html>
<html lang="en">
  <head>
    <title>My Page</title>
  </head>
  <body>
    <div class="classSample">Hello</div>
    <span class="classSample">World!</span>
  </body>
  <script>
    const elem = document.getElementsByClassName('classSample');
    console.log(elem);
  </script>
</html>
```

**FIGURE 2.8 : Représentation de la structure HTML avec la méthode getElementByClassName**

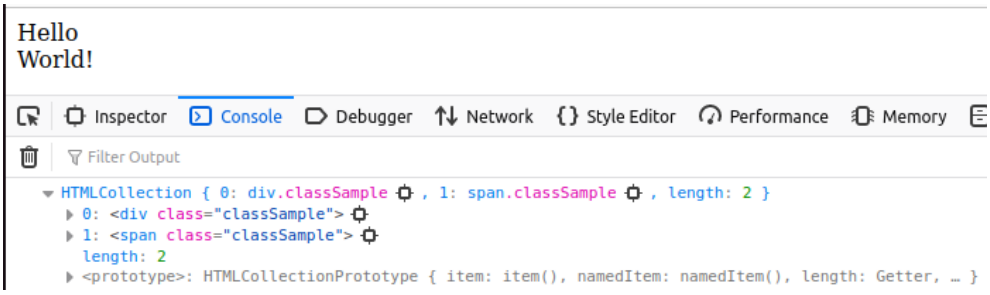



**FIGURE 2.9 : Représentation du résultat d'une structure HTML avec la méthode getElementByClassName**

Le quatrième et dernier moyen d'obtenir une référence à des éléments est le sélecteur de requête CSS. Il se décline en deux fonctions, querySelector( requête ) et querySelectorAll( requête ). La première fonction renvoie un seul élément et la seconde renvoie une liste d'éléments correspondant à la requête indiquée. La figure 2.10 montre comment l'identificateur peut être utilisé.

Dans la figure 2.10, nous avons deux éléments span dans le body, l'un avec un ID et l'autre avec une classe. À l'intérieur de la balise script, nous avons utilisé document.querySelector

```html
<!DOCTYPE html>
<html lang="en">
  <head>
    <title>My Page</title>
  </head>
  <body>
    <span id="idSample">Hello</span>
    <span class="classSample">World!</span>
  </body>
  <script>
    const elem1 = document.querySelector("#idSample");
    const elem2 = document.querySelectorAll("span");
    console.log(elem1);
    console.log(elem2);
  </script>
</html>
```

**FIGURE 2.10 : Représentation de la structure HTML avec l'identificateur requête CSS**

en passant un ID de requête comme paramètre. Dans la méthode document.querySelectorAll, nous avons passé une requête pour rechercher tous les éléments span. Le résultat nous pouvons voir dans la figure 2.11. Dans le premier console log, nous vérifions un seul élément span. Dans la deuxième console log, nous vérifions une liste de deux éléments span.

En plus de ces quatre techniques, une autre approche pour trouver un nœud consiste à naviguer à travers ses relations, à savoir les parents, les enfants et les frères. La navigation peut être effectuée en utilisant les propriétés (et non les méthodes) parentNode, childNodes, firstChild, lastChild, previousSibling et nextSibling. Ainsi donc, à partir d'un nœud, nous pouvons trouver son parent, ses enfants et ses frères.

Pour illustrer ce concept, nous faisons référence à la structure HTML de la figure 2.12. Tout d'abord, nous obtenons une référence à un élément par la méthode getElementById.

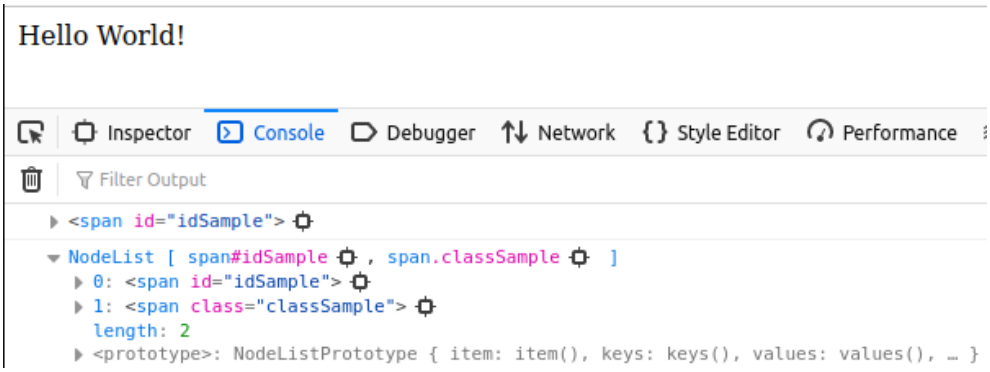



**FIGURE 2.11 : Représentation du résultat d'une structure HTML avec l'identificateur requête CSS**

Ensuite, nous naviguons vers les parents, les frères et les enfants du nœud. Dans la figure 2.13, on montre le résultat de ces opérations. Dans la première ligne se trouve le nœud que nous avons identifié. Ensuite, nous voyons son parent, qui se trouve à être l'élément body. Les lignes suivantes montrent le résultat des méthodes previousSibling, nextSibling, firstChild et lastChild. Nous voyons également un NodeList qui fait référence aux enfants du nœud.

Nous avons expliqué comment identifier et naviguer dans un arbre DOM. Maintenant, nous allons décrire comment manipuler la structure et le contenu du DOM. Shute explique que le DOM fournit des fonctions pour contrôler le HTML et le style des pages web [12]. Les modifications se produisent de manière dynamique et interactive grâce à l'utilisation de fonctions, notamment document.createElement(), Node.cloneNode(), et document.createTextNode(). Ainsi, la fonction document.createElement() reçoit en argument un nom de balise qui définit l'élément que nous souhaitons créer. La fonction cloneNode() duplique le nœud auquel nous faisons référence. Cette fonction peut recevoir l'argument true ou false. Si nous passons true, elle effectuera une copie complète du nœud et de tous ses nœuds enfants. Si nous passons false, elle clonera uniquement le nœud que nous appelons. Enfin, nous avons la fonction createTextNode(), qui est utilisée pour insérer du texte dans un nœud. Elle reçoit une chaîne

```html
<!DOCTYPE html>
<html lang="en">
  <head>
    <title>My Page</title>
  </head>
  <body>
    <div>
      <h2>Page title</h2>
    </div>
    <div id="divIdA">
      <p>Hello</p>
      <span>World!</span>
    </div>
    <div>
      <p>Content</p>
    </div>
  </body>
  <script>
    const elem1 = document.getElementById("divIdA");
    const parentNode = elem1.parentNode;
    const previousSibling = elem1.previousSibling;
    const nextSibling = elem1.nextSibling;
    const childNodes = elem1.childNodes;
    const firstChild = elem1.firstChild;
    const lastChild = elem1.lastChild;

    console.log(elem1);
    console.log(parentNode);
    console.log(previousSibling);
    console.log(nextSibling);
    console.log(childNodes);
    console.log(firstChild);
    console.log(lastChild);
  </script>
</html>
```

**FIGURE 2.12 : Représentation de la structure HTML avec la navigation des relations**

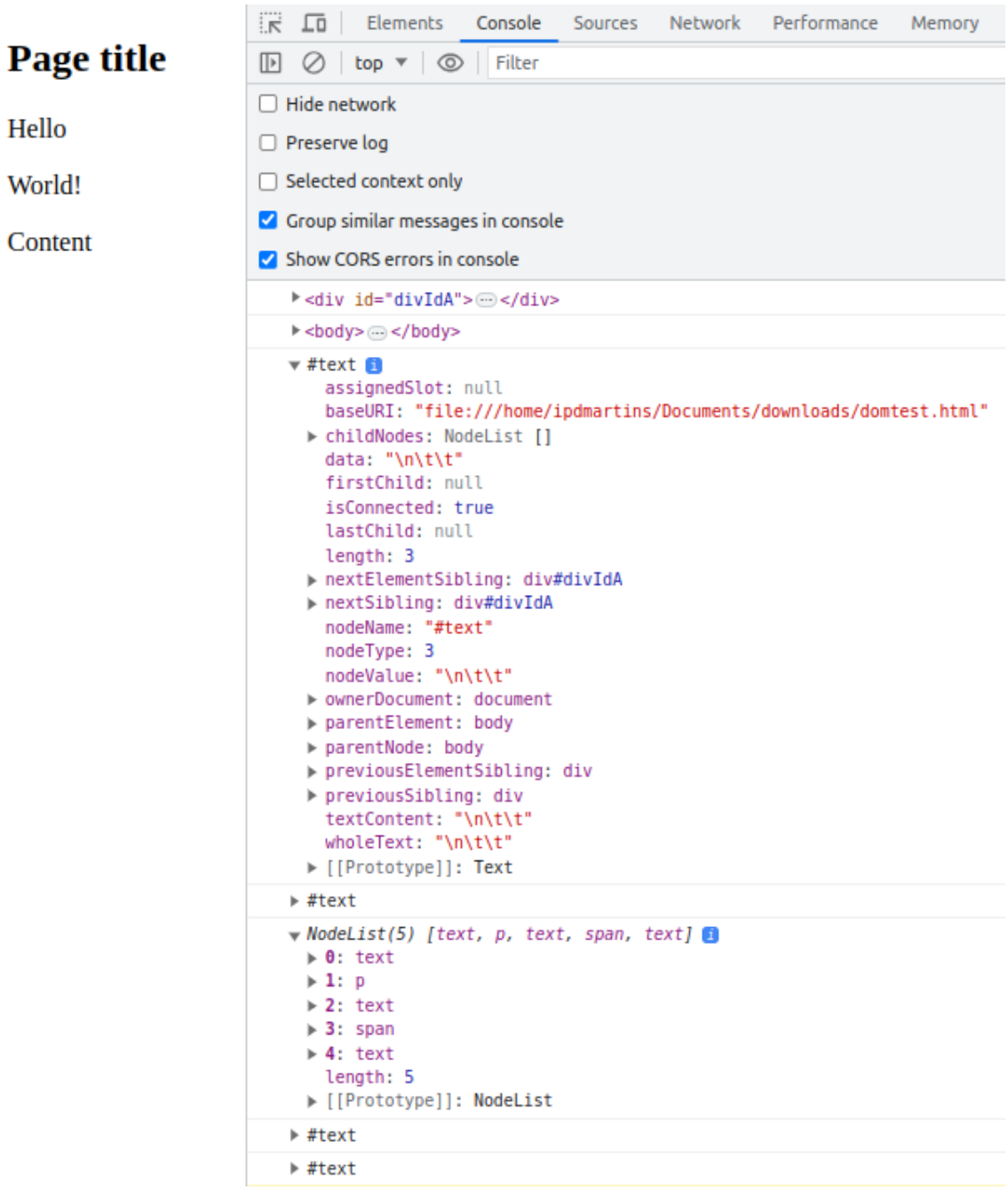



**FIGURE 2.13 : Représentation du résultat d'une structure HTML avec la navigation des relations**

```html
<!DOCTYPE html>
<html lang="en">
  <head>
    <title>My Page</title>
    <script>
      window.onload = () => {
        const createElement = document.createElement("h1");
        const elementText = document.createTextNode("Hello world!");
        createElement.appendChild(elementText);
        document.body.appendChild(createElement);

        const cloneNode = createElement.cloneNode(true);
        document.body.appendChild(cloneNode);
      }
    </script>
  </head>
  <body></body>
</html>
```

**FIGURE 2.14 : Représentation de la structure HTML avec manipulation d'éléments**

de caractères avec le contenu que l'on souhaite ajouter. Dans la figure 2.14, on peut voir une représentation de ces concepts.

Le code de la figure 2.14 montre une fonction qui s'exécute lorsque le document HTML se charge. À l'intérieur de la balise script, on peut voir un exemple d'utilisation de createElement. Cela crée un élément de balise nommé h1. Ensuite, nous utilisons la fonction createTextNode en passant la chaîne de texte "Hello world !". Ensuite, nous utilisons la fonction document.body.appendChild avec laquelle nous insérons le nœud créé dans le document. Cette fonction est essentielle car sans elle, le résultat n'apparaît pas dans la page web. Ensuite, nous utilisons la fonction cloneNode pour faire une copie complète du nœud que nous venons de créer. La figure 2.15 montre le résultat. Notez que dans le document HTML original, la balise body est vide. Malgré cela, lorsque nous utilisons la fonction appendChild pour

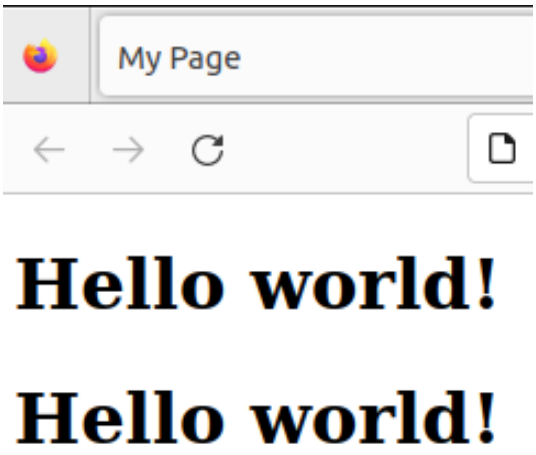



**FIGURE 2.15 : Représentation du résultat d'une structure HTML avec manipulation d'éléments**

insérer nos nœuds, ceux-ci sont ajoutés de manière dynamique par le navigateur au moment de l'exécution.

Dans cette section, nous avons exploré les principales caractéristiques de la structure HTML et de ses éléments. Nous avons appris que nous pouvons utiliser le DOM pour manipuler le HTML. Après cela, nous savons que le HTML est fondamental pour chaque page web. Dans la prochaine section, nous allons explorer le CSS, qui est complémentaire au HTML, mais c'est une technologie puissante qui se combine parfaitement avec celui-ci.

### 2.1.2 CSS

Le Cascading Style Sheets (CSS) est une technologie complémentaire au HTML. Cela signifie que le CSS ne représente pas un langage de programmation web en soi, autrement dit il ne permet pas seul de créer une page. À l'inverse, les pages web construites en HTML ont un style par défaut limité, et c'est là que le CSS entre en jeu en tant que technologie de style puissante. Comme le décrit Wilson [7], le CSS est utilisé pour personnaliser le style et la

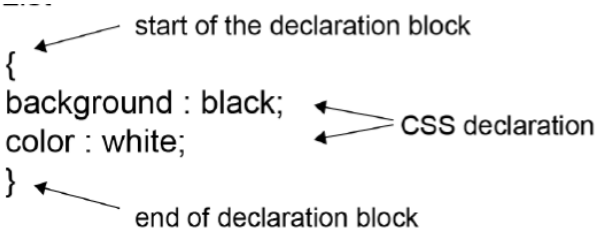



**FIGURE 2.16 : Démonstration de la syntaxe CSS**
**©[10]**

mise en page des pages web. Il utilise des sélecteurs pour modifier visuellement des parties spécifiques d'une page web.

Les modifications que le CSS peut apporter à une page web dépendent de la combinaison de règles que nous définissons. Coulson [10] explique qu'une déclaration de syntaxe CSS comprend une propriété et une valeur. La première est le nom d'un aspect du style d'un élément que l'on cherche à modifier, et la seconde est la valeur que l'on souhaite lui attribuer. Dans la figure 2.16, on voit un exemple d'un ensemble de règles. Cet ensemble commence par une accolade, suivie des propriétés background et color, qui reçoivent respectivement les valeurs black et white. Enfin, une accolade ferme l'ensemble de règles.

Comme nous le savons, un document HTML est composé d'éléments. Pour appliquer des styles CSS à des éléments nous devons utiliser une expression qui les désigne, appelée *sélecteur*. Coulson mentionne que les sélecteurs les plus courants consistent à nommer l'ID ou la classe des éléments visés. Une autre option consiste à appliquer le style CSS directement à l'intérieur de la balise d'un élément, ce qui est appelé « CSS en ligne ».

Avant de donner un exemple de CSS, dans la figure 2.17, nous montrons une structure HTML et son résultat sous forme de page web, sans style CSS.

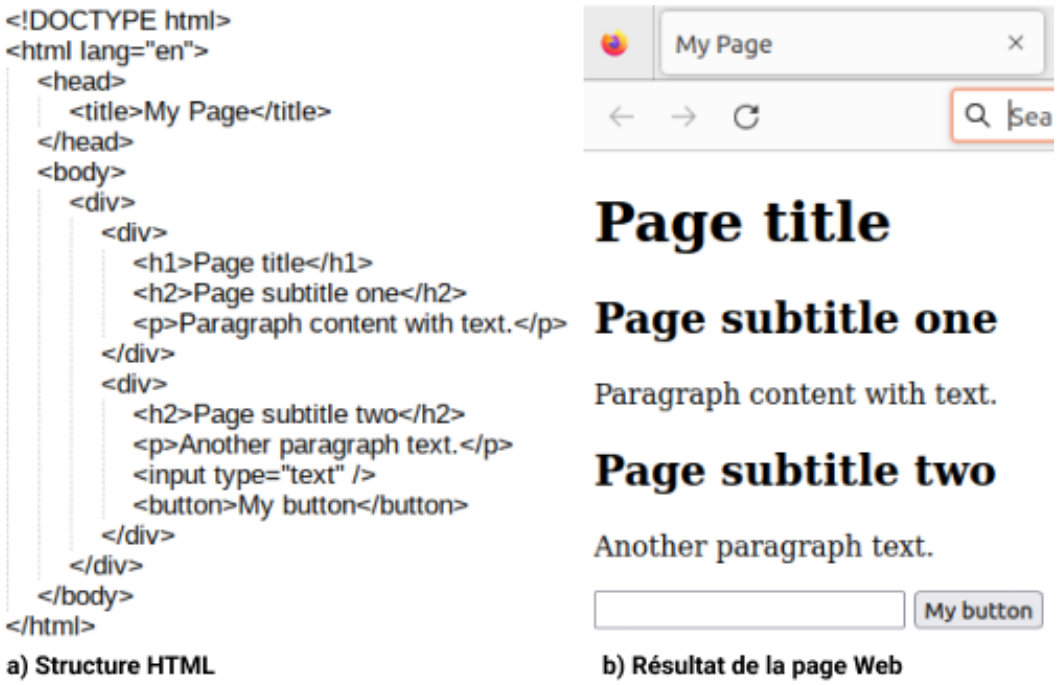



**FIGURE 2.17 : Représentation d'un document HTML sans style**

Le sélecteur de type d'élément permet de sélectionner les éléments dans le document HTML en fonction du nom de leur balise. Dans ce cas, il est possible de sélectionner un seul élément ou une liste d'éléments. Dans la figure 2.18, on peut voir la structure présentée dans la figure 2.17 avec des modifications appliquées aux styles CSS. Ainsi, on peut constater dans le code HTML qu'un style CSS en ligne a été appliqué à l'élément h1. Dans ce cas, on a changé la couleur du texte en bleu et un soulignement a été appliqué. Nous avons également ajouté une balise nommée style en haut de la structure. À l'intérieur de cette balise, qui contient du code CSS, on a utilisé le sélecteur de type unique pour désigner l'élément button, pour lequel a été défini un arrière-plan de couleur verte et une taille de police de 16 pixels. Finalement, on peut voir un exemple du sélecteur de type multiple pour les balises h2 et p. Cela signifie que nous pouvons sélectionner une liste d'éléments dans le document HTML pour appliquer le même style. Dans notre exemple, nous avons défini la couleur du texte en rouge pour tous les

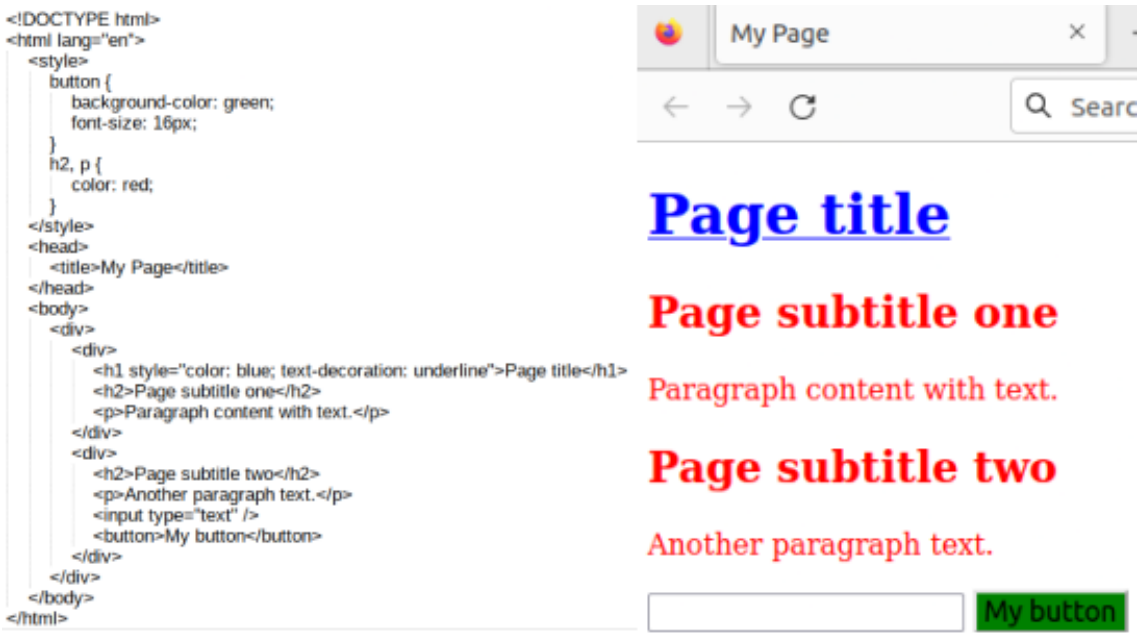



**FIGURE 2.18 : Représentation d'un document HTML avec stylisation CSS**

éléments h2 et p présents dans le document. À côté du code HTML, on peut voir le résultat de ces instructions de mise en forme.

Le sélecteur d'attribut de classe permet de styliser les éléments HTML appartenant à la même classe. Comme nous l'avons expliqué précédemment, il est possible de définir des classes pour les éléments en donnant une valeur à l'attribut class de ceux-ci. Dans le code CSS, on désigne les éléments appartenant à une classe en faisant précéder son nom d'un point.

Dans la figure 2.19, nous pouvons voir une démonstration de la stylisation en utilisant le sélecteur de classe. À l'intérieur de la balise CSS de style, nous avons identifié deux éléments div appartenant à la même classe et avons défini une marge supérieure de 10 pixels pour déplacer leur contenu vers le bas de la page. De plus, nous avons défini une bordure et une largeur pour les éléments div. Dans un deuxième temps, nous avons désigné les éléments de la classe nommée "paragraphB" et modifié leur position dans la page en les forçant à se déplacer

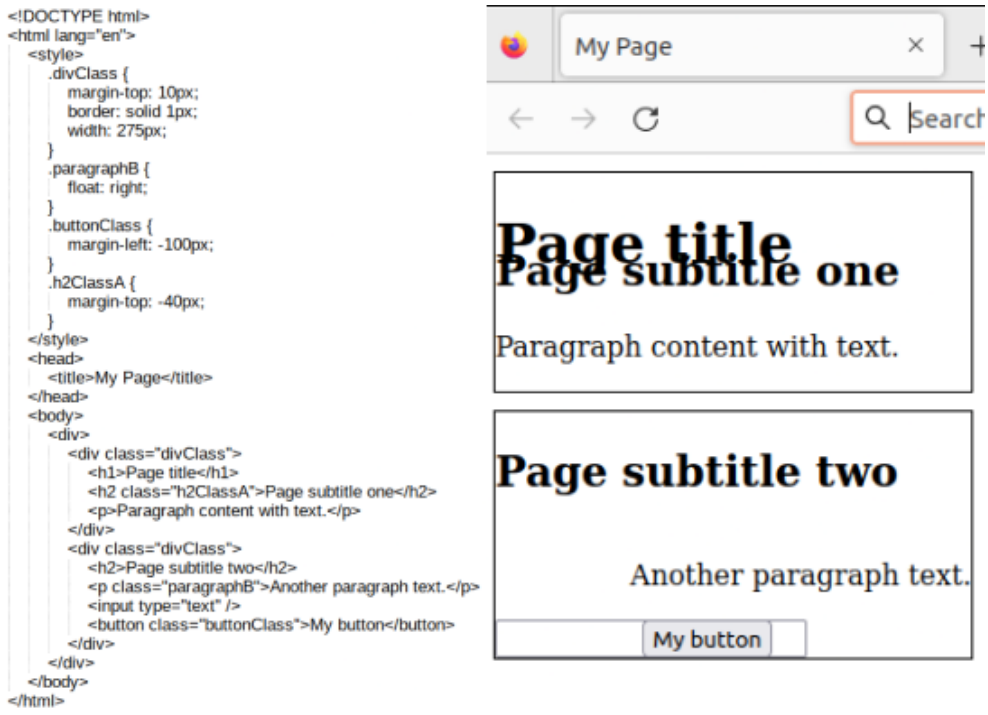



**FIGURE 2.19 : Représentation d'un document HTML avec le sélecteur d'attribut de classe**

vers le côté droit. Pour les éléments de la classe "buttonClass", nous fait en sorte que les éléments se superposent à l'élément input. Pour la classe "h2ClassA", nous avons également forcé l'élément en faisant partie à se déplacer par-dessus l'élément h1. Dans ces deux derniers exemples, on remarque que nous avons délibérément manipulé le positionnement des éléments, faisant même en sorte qu'ils occupent les zones d'autres éléments. Le résultat de la page web que nous avons affichée à côté du code HTML.

Le sélecteur d'attribut ID permet de styliser un élément HTML qui s'est fait donner une valeur à son attribut id. À l'intérieur de la balise de style, nous devons le désigner en utilisant le symbole dièse suivi du nom de l'ID. Comme nous l'avons mentionné précédemment, au plus un élément du document HTML peut posséder un ID donné.

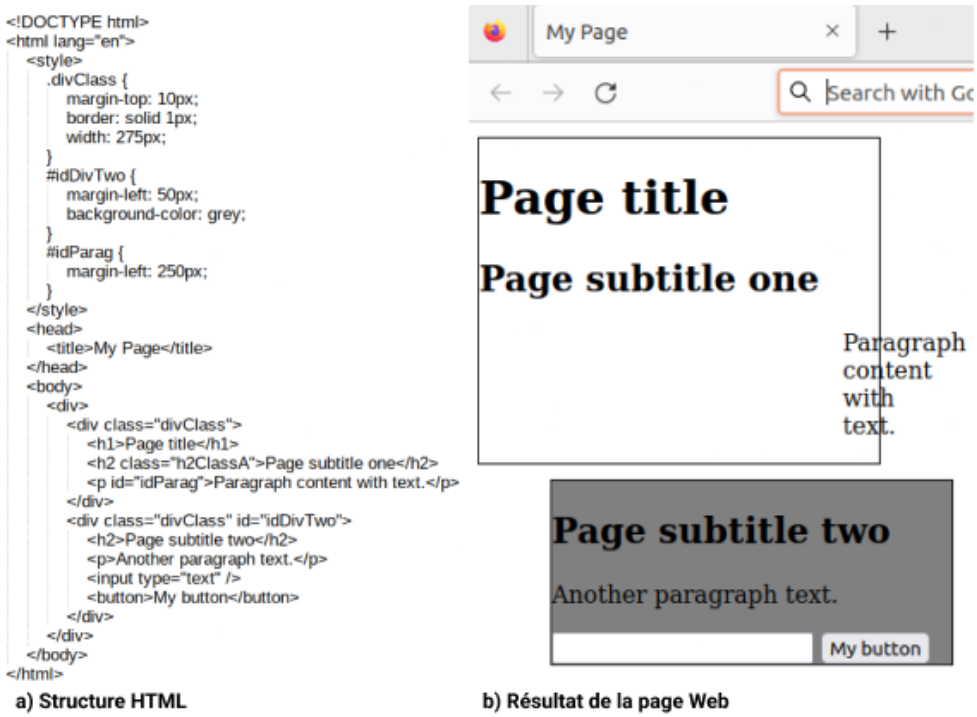



**FIGURE 2.20 : Représentation d'un document HTML avec le sélecteur d'attribut ID**

Dans la figure 2.20, nous montrons l'utilisation d'un sélecteur d'attribut ID. Nous avons conservé l'attribut de sélecteur "divClass" que nous avions défini dans l'exemple précédent. Cependant, pour le deuxième élément div, nous avons défini l'ID à "idDivTwo". À partir de son identification à l'intérieur de la balise style, nous avons défini une marge pour déplacer le contenu vers la droite, provoquant un désalignement avec l'élément div situé au-dessus. Un autre exemple pertinent concerne l'élément p, que nous avons défini avec l'ID "idParag". Nous avons défini la marge à gauche à 250 pixels, forçant cet élément à dépasser la limite de la bordure du div.

Nous avons donc vu comment l'utilisation du CSS permet de modifier la couleur, la taille et la position des éléments à l'intérieur du document HTML. CSS offre de nombreuses possibilités d'attributs de style, et nous n'avons exploré qu'une petite partie d'entre elles. Il existe d'autres formes de sélecteurs, tels que le symbole astérisque (*), les pseudo-classes

comme :hover et :active, les pseudo-éléments comme : :first-letter, et de nombreuses autres options. Avec cette vue d'ensemble de CSS, nous espérons avoir clarifié son utilité dans le design des pages web. Dans la prochaine section, nous aborderons les concepts liés aux bogues d'interface.

## 2.2 BOGUES D'INTERFACE

Dans la section précédente, nous avons exploré les technologies liées à cette recherche, à savoir le HTML et le CSS. Nous avons donné une brève explication de la structure d'une URL, de la structure d'un document HTML et de ses principaux éléments de balisage. Nous avons présenté les concepts du DOM, sa structure, les méthodes d'identification des éléments et la manière de naviguer à travers le DOM afin de trouver des nœuds et les manipuler. Pour conclure, nous avons parlé de CSS et de ses ressources pour styliser les pages web. Nous avons ainsi expliqué la syntaxe CSS et les sélecteurs.

Dans cette section, nous allons décrire ce que l'on appelle les « bogues d'interface ». On sait que dans le développement web, l'interface utilisateur affiche des composants dans la fenêtre du navigateur, avec lesquels il est possible d'interagir, ce qui peut impliquer une communication avec un serveur et une modification dynamique de l'apparence des éléments. Intéressons-nous ensuite au concept de « bogue ». Dans le développement de logiciels, un bogue est une situation qui peut mettre en évidence, par exemple, une erreur, un crash du système, un défaut ou une sortie incorrecte. Un bogue peut survenir en raison d'une mauvaise logique de programmation, d'une incohérence dans les versions d'outils tiers, d'une panne du serveur ou d'une cyberattaque, pour ne nommer que quelques exemples. Pour résumer, Samoylov affirme que *bogue* est un terme lié aux défauts de programmation qui peuvent varier en fonction de leur gravité [13].

En conséquence, nous classifions comme un bogue d'interface tout problème qui affecte négativement l'expérience utilisateur et l'interaction avec une page web. Avec la grande variété d'appareils et de tailles d'écran disponibles de nos jours, les pages web doivent s'adapter à différentes plateformes, y compris les ordinateurs, les téléphones intelligents et les tablettes. Selon Hallé et al. [1], un bogue de mise en page est un défaut visible qui peut survenir pour l'utilisateur, impactant l'apparence et la fonctionnalité des éléments Web. Ces défauts peuvent être liés à des facteurs tels que la structure du DOM, les dimensions et la mise en forme des éléments Web, et ils ne sont pas spécifiques à un navigateur particulier.

Les pages web peuvent présenter une large gamme de bogues, et ils peuvent être classés de différentes manières. Dans leur recherche, Hallé et al. [1] fournissent une liste complète de bogues, les classant en trois groupes principaux, avec des sous-catégories pour chacun d'entre eux. Examinons les catégories principales.

Le premier groupe est constitué de bogues dits de mise en page. Ce groupe concerne les erreurs causées par le mauvais alignement et le rendu incorrect des éléments dans une page web. Ceci inclut des éléments qui ne sont pas correctement alignés alors qu'ils le devraient, des éléments qui se chevauchent et provoquent des conflits visuels, des éléments qui dépassent les dimensions de leur conteneur parent et deviennent inaccessibles, et des éléments empilés de manière incorrecte où l'ordre d'affichage est inversé, provoquant un désordre dans l'apparence des éléments.

Les bogues de mise en page couvrent quatre sous-catégories, soit :

- **Éléments désalignés** : Comme son nom l'indique, ce bogue fait référence à des éléments qui devraient être alignés, mais qui, pour une raison quelconque, ne le sont pas. Parfois, ce désalignement est évident, mais parfois, il est pratiquement imperceptible à l'œil

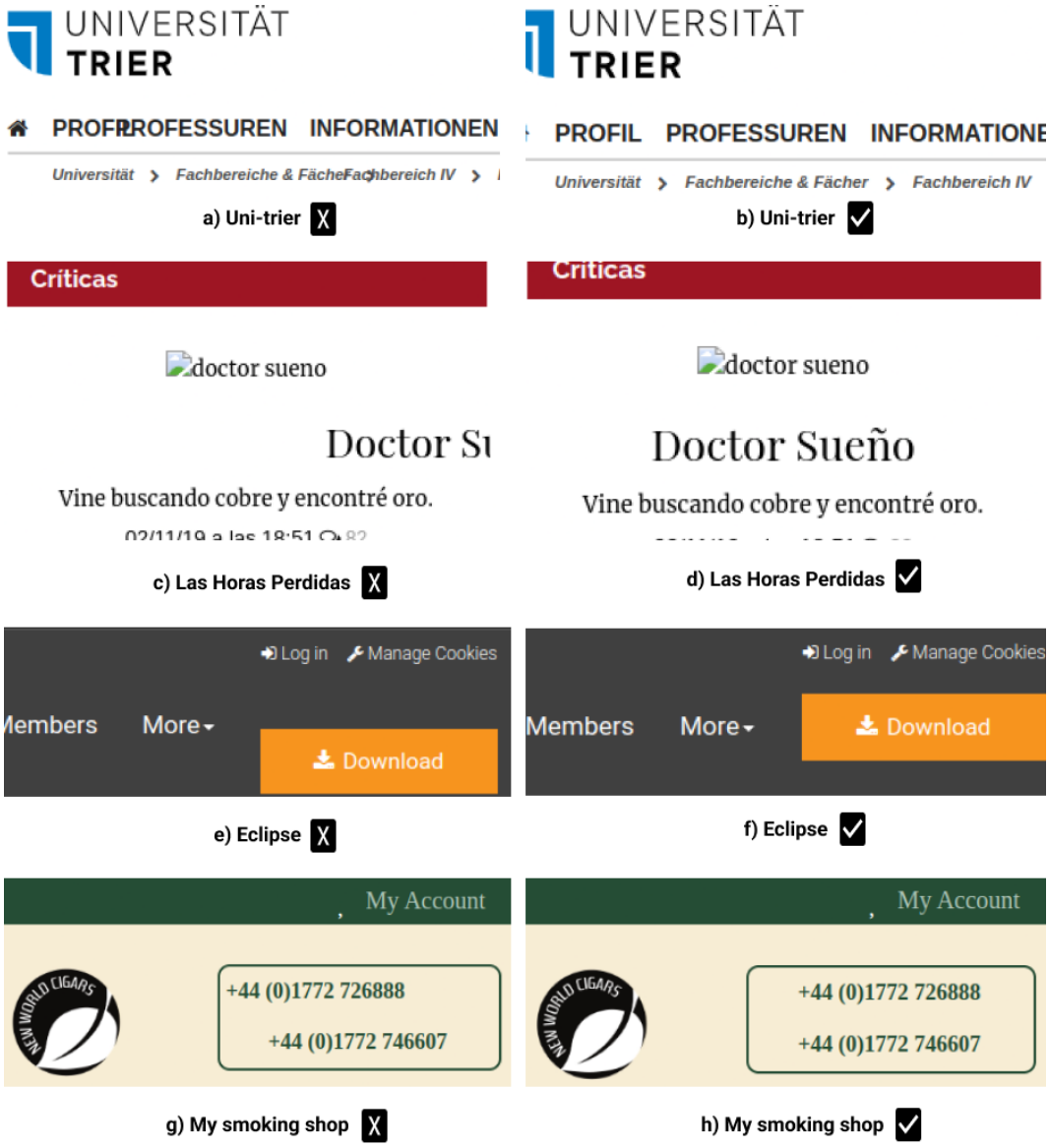



**FIGURE 2.21 : Représentation des bogues distincts trouvés sur les pages web**

nu. Cette sous-catégorie peut également se diviser en désalignements verticaux et horizontaux, comme nous l'expliquerons ci-dessous.

— Désalignement horizontal : Ce type de bogue se produit lorsqu'un élément web n'est pas correctement aligné avec ses éléments pairs. Il fait référence à un désalignement selon la référence cartésienne de l'axe X. La figure 2.21 référence e (Eclipse) montre un élément de bouton désaligné avec ses éléments de menu appa-

riés. En revanche, la référence f (Eclipse) représente un alignement correct avec ses pairs.

  — Désalignement vertical : Ce bogue de mise en page est similaire au précédent, mais son désalignement fait référence à l'axe Y du repère cartésien. La référence g de la figure 2.21 (My smoking shop) représente ce désalignement vertical dans les numéros de téléphone. La référence h (My smoking shop) représente le bon alignement de ces éléments.

- **Éléments superposés** : Survient lorsque des éléments entrent en « collision » les uns avec les autres, provoquant un chevauchement de leur contenu. Ceci peut entraîner du texte et des images superposés ou la dissimulation d'éléments, nuisant à l'utilisation d'une page web. La figure 2.21 montre en référence a (Uni-trier), un élément de menu qui chevauche un autre élément de menu. En référence b (Uni-trier), on peut voir la mise en page correcte de la même page web.
- **Éléments s'étendant à l'extérieur de leur conteneur** : Ce but fait référence à la saillie d'un l'élément. Il se produit lorsque les éléments ne se redimensionnent pas correctement. Normalement, les éléments parents doivent contenir les éléments enfants en largeur. Cependant, lorsqu'il n'y a pas assez d'espace, un élément enfant peut dépasser son contenu à l'extérieur de son parent. Cette situation peut masquer partiellement ou totalement l'élément enfant, le rendant inaccessible. Dans la figure 2.21, la référence c (Las Horas Perdidas) montre que le texte "Doctor Sueño" déborde à l'extérieur de son composant parent. Dans la référence d (Las Horas Perdidas), on peut voir un exemple de mise en page acceptable.
- **Éléments empilés de manière incorrecte** : Cette situation se produit lorsqu'un élément qui devrait être positionné au-dessus d'un autre élément est placé par erreur en dessous de lui. Ce type de problème peut généralement être résolu en attribuant correctement l'attribut CSS z-index à l'élément. La figure 2.22 représente cette situation. Nous

pouvons y voir où le menu orange apparaît en dessous du contenu de la page, rendant le menu inutile.


**FIGURE 2.22 : Représentation d'éléments mal empilés**
**©[1]**

Le deuxième groupe est constitué de bogues non liés à la mise en page. Ce groupe se concentre sur les bogues liés à l'encodage des caractères et à la mise en forme des chaînes de caractères. Les bogues non liés à la mise en page couvrent deux sous-catégories, soit :

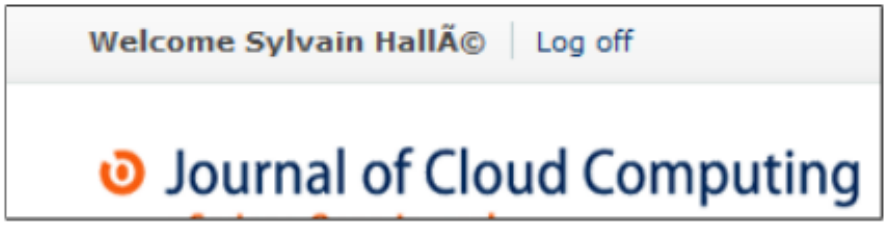
- **Mojibake et problèmes d'encodage** : Dans ce cas, les mots avec des accents ou contenant des caractères étrangers peuvent être incorrectement gérés en raison de

l'utilisation de schémas d'encodage tels que l'ASCII 7 bits, le Cp1252 ou l'UTF-8. Le système pourrait donc ne pas interpréter correctement le contenu d'une chaîne de caractères. Le mot Mojibake fait référence à un terme japonais signifiant "transformation de caractères". La figure 2.23 représente ce bogue dans le titre de (a). Dans (b), il se produit dans le nom de l'utilisateur, qui est placé à côté du champ de l'image.

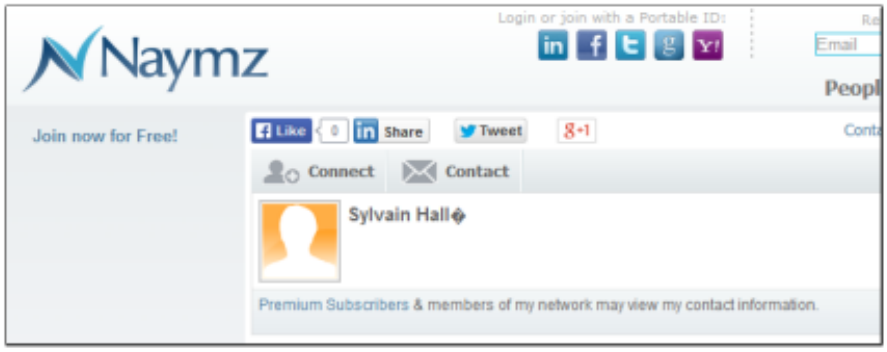
- **Problèmes d'échappement** : Dans ce cas, les chaînes de caractères contenant des caractères spéciaux échouent à être correctement encodées ou décodées entre deux applications. Cela se produit principalement lors de la procédure de lecture et d'écriture dans les bases de données –par exemple, une mauvaise gestion de l'apostrophe simple pour une apostrophe double. Un autre cas peut se produire avec des balises HTML qui peuvent être affichées directement dans la page sous forme de chaînes de caractères, plutôt que d'être interprétées comme du code HTML et rendues de telle sorte. La figure 2.24, à l'intérieur du champ "address", montre un exemple de ce bogue.

Le troisième groupe est constitué de bogues appelés comportementaux. Ce groupe considère plusieurs états successifs d'une page web, où la position d'un élément est affectée par l'interaction de l'utilisateur. Des exemples incluent des boutons dysfonctionnels généralement trouvés dans les pop-ups, un état de connexion défectueux qui échoue à valider l'accès de l'utilisateur, et des résultats de recherche incohérents affichant des informations incorrectes ou insuffisantes. Les bogues comportementaux couvrent quatre sous-catégories :

- **Éléments déplacés** : Ces bogues se produisent lorsque les éléments se déplacent vers une position inappropriée sur la page web en raison de l'interaction de l'utilisateur. Sur la figure 2.25, nous voyons que lorsqu'un utilisateur tape du texte dans la zone de texte, sa largeur est réduite et il se déplace vers le côté gauche.




(a) SpringerOpen




(b) Naymz

**FIGURE 2.23 : Représentation des problèmes d'encodage mojibake ©[1]**

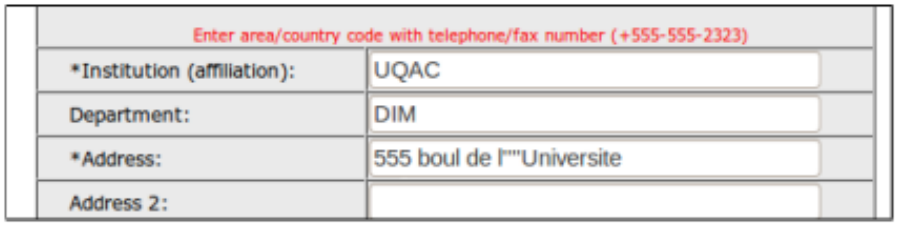



**FIGURE 2.24 : Représentation des problèmes d'échappement ©[1]**

- **Boutons défectueux** : Ce cas se produit lorsque l'utilisateur clique sur des boutons qui ne fonctionnent pas du tout. Ceci est possible, par exemple, dans le cas d'éléments superposés qui se comportent comme des fenêtres contextuelles "pop-up".
- **Confusion de l'état de connexion** : Cette situation concerne les fonctionnalités qui devraient être affichées aux utilisateurs connectés mélangées à celles affichées aux

utilisateurs déconnectés (ou à toute autre forme de confusion dans l'état de l'application). Dans la figure 2.26, nous pouvons voir le nom de l'utilisateur en haut à droite de la page (laissant croire qu'il est connecté au site) dans la page affichant le formulaire de validation de connexion (présenté aux utilisateurs qui ne sont pas déjà connectés).

- **Résultats de recherche incohérents** : Survient lorsqu'un utilisateur envoie une requête de recherche et que l'application renvoie une réponse inappropriée. Dans la figure 2.27, l'utilisateur saisit un code postal pour rechercher des magasins à proximité, mais la page suivante ne montre aucun résultat liés à son code postal.

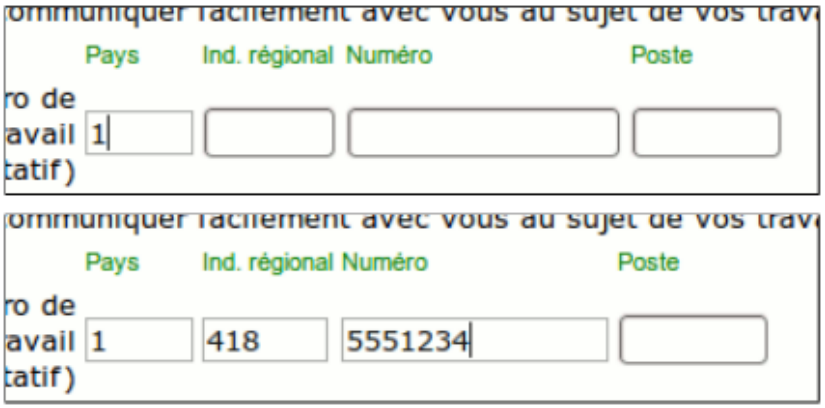



**FIGURE 2.25 : Représentation des problèmes des éléments en mouvement ©[1]**

Nous avons décrit précédemment que les pages web sont construites à l'aide du HTML combiné à CSS. Comme on vient de le voir, malgré leur solidité en tant que technologies pour le développement web, l'utilisation incorrecte des règles CSS peut entraîner des résultats erronés au niveau de l'affichage. Parfois, ces résultats sont difficiles à prévoir, ce qui représente des bogues potentiels dans la mise en page.

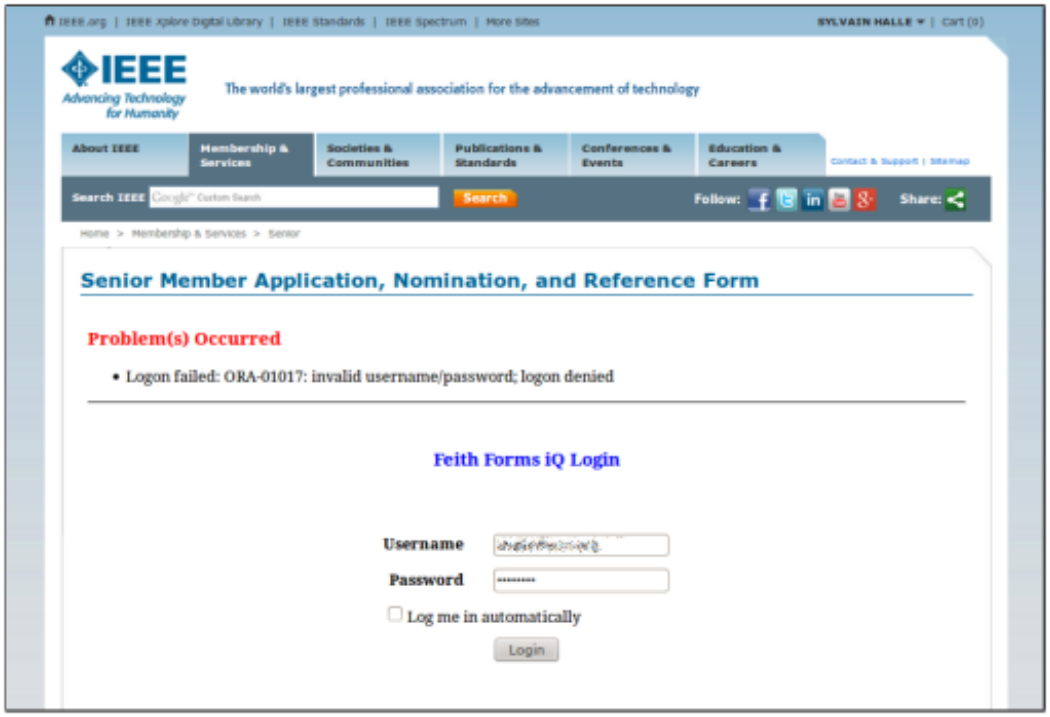



**FIGURE 2.26 : Représentation des problèmes de confusion de l'état de connexion ©[1]**

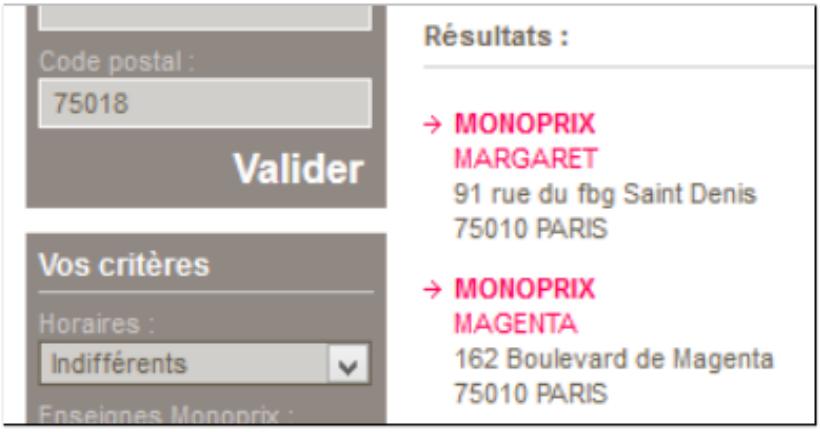



**FIGURE 2.27 : Représentation des problèmes de résultats de recherche incohérents ©[1]**

Nous venons d'étudier une liste de bogues de mise en page identifiés dans les pages web. Ces bogues peuvent entraîner des erreurs dans les applications et offrir une mauvaise expérience utilisateur, comme nous l'avons montré. Dans ce mémoire, notre recherche proposera

une approche pour identifier ces bogues, ou plus précisément de déterminer les conditions décrivant ces bugs de manière automatique.

Pour répondre à l'objectif de cette recherche, notre attention se portera sur l'étude des bogues dits de mise en page, qui peuvent se manifester dans différents scénarios, sous différentes formes, et avoir un impact sur la convivialité des pages web de différentes manières. Plus particulièrement, nous nous concentrerons sur les problèmes de désalignement, d'éléments superposés et de saillie d'éléments.

# CHAPITRE III

# ÉTAT DE L'ART EN DÉTECTION DES BOGUES D'INTERFACE

Dans le chapitre précédent, nous avons présenté les technologies de développement de pages web et les bogues d'interface. Tout d'abord, nous avons étudié le HTML et son rôle dans la création de pages web. Nous avons introduit le HTML comme un document basé sur des balises, ce qui rend le contenu lisible dans les navigateurs. Nous avons également expliqué que les balises peuvent être identifiées par des attributs comme id et class.

Nous avons également présenté les concepts du DOM et sa structure en arbre, qui représente également les documents HTML. Ainsi, le DOM offre une grande variété d'identificateurs pour les éléments d'un document HTML. Il offre également des mécanismes pour les manipuler en termes de structure et de contenu. On a ensuite abordé les concepts de CSS, qui peut identifier les éléments HTML par leur nom, leur classe et leur ID. De plus, le CSS peut modifier la couleur, la taille et la position des éléments, interférant dans leur rendu visuel sur les pages web.

Enfin, nous avons présenté une classification des bogues d'interface. Nous avons indiqué que dans [1], trois groupes majeurs de bogues ont été identifiés, à savoir les bogues de mise en page, les bogues de non-mise en page et les bogues comportementaux. Dans ces groupes, nous avons introduit des sous-groupes avec des exemples de bogues trouvés sur des pages web. De plus, nous avons souligné notre intention de nous concentrer sur les bogues de mise en page dans cette recherche.

Dans ce chapitre, nous présentons des études ayant une certaine connexion avec le sujet sur lequel nous recherchons. Tout d'abord, nous allons chercher à découvrir les liens entre ces études et notre proposition, qui vise à utiliser le concept d'invariants pour tester des conditions

sur la mise en page des sites web. Deuxièmement, nous allons déterminer si certaines de ces études a tenté appliqué la notion d'invariant pour identifier des bogues de mise en page.

Au total, nous décrirons six publications. Au cours de nos lectures, nous avons divisé ces publications en deux groupes, à savoir les approches pour les événements défectueux et les approches pour l'identification des bogues de mise en page.

## 3.1 APPROCHES POUR LES ÉVÉNEMENTS DÉFECTUEUX

Dans ce premier ensemble de publications, on s'intéresse aux changements dynamiques dans l'arbre DOM. Les travaux qu'on y retrouve ont utilisé le concept d'invariants pour détecter des « événements défectueux » dans des applications web.

### 3.1.1 INVARIANT-BASED AUTOMATIC TESTING OF AJAX USER INTERFACES ET INVARIANT-BASED AUTOMATIC TESTING OF MODERN WEB APPLICATIONS

Ces deux publications seront traitées dans la même sous-section car elles appartiennent aux mêmes auteurs, Ali Mesbah et Arie Van Deursen. La première étude a été publiée en 2009 lors de la 31e International Conference on Software Engineering, et la seconde a été publiée en 2012 dans le Scientific Journal IEEE Transactions on Software Engineering.

Dans la recherche menée par Mesbah et Deursen [14, 15], les auteurs se concentrent sur les applications AJAX, en analysant les changements d'état qui se produisent dans les arbres DOM de l'interface utilisateur (UI). Ainsi, ils expliquent que dans ce type d'application, l'état d'une UI est déterminé dynamiquement par les changements déclenchés par des événements dans le DOM du navigateur. Ainsi, ils définissent un changement d'état de l'UI AJAX comme un changement dans l'arbre DOM causé soit par des changements d'état côté serveur propagés

au client, soit par des événements côté client gérés par l'AJAX. Dans ce scénario, des erreurs peuvent survenir à partir d'événements déclenchés tels que des valeurs d'entrée fournies par les utilisateurs et des clics de souris, par exemple. Pour cette raison, ils affirment que les tests de l'UI peuvent identifier des états défectueux dans le DOM.

Suivant cette idée, les auteurs ont proposé une méthode pour tester les UI AJAX. Pour mettre en œuvre cette proposition, ils ont utilisé web crawler capable de détecter et de déclencher des événements sur des éléments cliquables. Ainsi, les auteurs ont exposé une technique et un outil d'exploration AJAX nommé CRAWLJAX. CRAWLJAX peut exécuter du code côté client et identifier des éléments cliquables capables de changer l'état dans le DOM du navigateur. Avec cette technique de crawl, les auteurs définissent un ensemble d'éléments cliquables candidats, exposés à des événements tels que le clic de souris et le survol de la souris. Par exemple, tous les éléments avec une balise div, a et span ayant l'attribut class="menuitem" pourraient être considérés comme cliquables.

Après avoir expliqué la notion d'éléments cliquables candidats, les auteurs ont défini le système d'exploration du DOM. Les changements d'état ont été enregistrés en utilisant les chemins de l'arbre DOM, ce qui leur a permis de naviguer entre les différents états. CRAWLJAX adopte des expressions XPath pour fournir une identification fiable et persistante des éléments. Ainsi, pour chaque élément modifiant l'état, l'expression XPath de cet élément renvoie son emplacement exact dans le DOM. Cette expression est enregistrée et utilisée pour trouver l'élément après un rechargement de la page web.

À partir de l'enregistrement des chemins, les auteurs déduisent un graphe d'états, qui capture les états de l'interface utilisateur et les transitions éventuelles basées sur les événements entre eux. La construction du graphe est réalisée avec l'utilisation de CRAWLJAX, qui fournit une interface de navigateur intégrée comme Mozilla et un robot pour simuler des événements de saisie utilisateur sur le navigateur. Un contrôleur a également été utilisé pour accéder

au DOM du navigateur afin d'analyser et de détecter les changements d'état. Le contrôleur gère également les actions du robot et est responsable de la mise à jour de l'état lorsque des changements pertinents se produisent dans le DOM.

Pour chaque élément cliquable candidat mentionné précédemment, le robot a déclenché un événement sur celui-ci. Après le déclenchement d'un événement, l'algorithme compare l'arbre DOM résultante avec la manière dont elle était juste avant que l'événement ne soit déclenché, pour déterminer si l'événement entraîne un changement d'état. En cas de détection d'un changement, un nouvel état a été créé et ajouté à le graphe d'états. Après la détection d'un nouvel état, la procédure de crawling est appelée récursivement pour trouver de nouveaux états possibles dans les changements partiels apportés à l'arbre DOM. CRAWLJAX calculait les différences entre l'arbre du document précédent et l'actuel.

Avec la technique expliquée, les auteurs ont eu accès à différents états du DOM, ce qui leur a permis de vérifier l'interface utilisateur par rapport à différentes contraintes. Ils ont proposé d'exprimer ces contraintes sous la forme d'invariants sur l'arbre DOM, qu'ils peuvent vérifier dans n'importe quel état. Par conséquent, ils ont classé ces invariants en trois types, à savoir les invariants génériques du DOM, les invariants de la machine d'état et les invariants spécifiques à l'application. Chaque classification était basée sur un modèle de faute représentant des fautes spécifiques à AJAX qui sont susceptibles de se produire et qui peuvent être capturées par l'invariant donné.

Le modèle se réfère au W3C, qui propose des normes telles que les directives d'accessibilité au contenu web (WCAG 1.0) ou les directives d'internationalisation et de localisation (i18n) du W3C. Par conséquent, le validateur conforme au W3C fonctionne comme un oracle pour indiquer si des erreurs et des avertissements se produisent dans un HTML, afin de valider ensuite le DOM résultant. Ci-dessous, nous donnons plus de détails sur leur classification des invariants.

Les invariants génériques du DOM concernent les invariants sur l'arbre DOM :

- **DOM validé** : Un code HTML mal formé peut être à l'origine de nombreuses vulnérabilités et problèmes de portabilité du navigateur. Les validateurs HTML s'attendent à ce que toute la structure et le contenu soient présents dans le code source HTML. Cependant, dans les applications AJAX, les modifications se manifestent sur l'interface utilisateur d'une seule page en mettant à jour partiellement le DOM avec le JAVASCRIPT. Étant donné que ces validateurs ne peuvent pas exécuter de JAVASCRIPT côté client, ils ne peuvent tout simplement pas effectuer de validation. Pour éviter les fautes, ils voulaient s'assurer qu'une application avait un DOM valide à chaque exécution possible. Ils ont utilisé l'arbre DOM obtenue après chaque changement d'état lors du crawling et l'ont transformée en l'instance HTML correspondante. Ainsi, le DOM validé est un invariant générique.
- **Pas de messages d'erreur dans le DOM** : L'état ne doit jamais contenir un patron de chaîne suggérant un message d'erreur dans le DOM. Des exemples de ces messages d'erreur sont le 404 Not Found, le 400 Bad Request, la session expirée, le 500 Internal Server Error et l'erreur MySQL. Ainsi, l'absence de messages d'erreur dans le DOM est un autre invariant générique qu'ils ont défini.
- **États sécurisés** : La détection des vulnérabilités de sécurité dans des widgets web autonomes côté client qui peuvent coexister indépendamment sur une seule page web. Ils se sont concentrés sur deux invariants de sécurité, à savoir 1) aucun widget ne peut changer le contenu (DOM) d'un autre widget, et 2) aucun widget ne peut voler des données à un autre widget et les envoyer au serveur via une requête HTTP.

Les invariants de la machine à états concernent les invariants des états de l'arbre DOM :

- **Pas d'élément cliquables mort** : Une faute courante dans les applications web classiques est l'apparition de liens morts pointant vers une URL définitivement inaccessible. En AJAX, les éléments cliquables qui sont censés changer l'état en récupérant des données du serveur, via JAVASCRIPT en arrière-plan, peuvent également être rompus. En écoutant le trafic de requêtes/réponses client/serveur après chaque événement, il est possible de détecter les liens inactifs. Ainsi, l'absence d'éléments cliquables morts est un invariant de la machine à états.
- **Bouton Retour cohérent** : Une autre faute qui survient souvent dans les applications AJAX est le dysfonctionnement du bouton Retour du navigateur. Il est possible d'enregistrer de manière programmatique chaque changement d'état avec l'historique du navigateur et des frameworks apparaissent pour traiter ce problème. Ainsi, un bouton Retour cohérent est un autre invariant de la machine à états.

Les invariants spécifiques à l'application concernent les invariants qui devraient toujours être valides et pourraient être vérifiés sur les états :

- Les auteurs ont détaillé des conditions qui pourraient être exprimées sous forme d'invariants, comme l'URL ou la visibilité des éléments DOM. Ils pouvaient exprimer des invariants sous forme d'expressions XPath, régulières ou JAVASCRIPT. La figure 3.1 montre un exemple d'expression d'un invariant XPath avec une précondition JAVASCRIPT pour vérifier si l'élément de menu sur la page d'accueil contient l'attribut de classe "menuElement".

Dans leur deuxième étude, les auteurs ont détaillé la Machine à États Spécifique à l'Application. Les invariants de cette machine concernent les contraintes sur les propriétés temporelles de l'application web.

- Ce sont des invariants qui sont censés être toujours vrais et peuvent être vérifiés sur les états. Ces invariants sont définis pour correspondre à un ensemble spécifique d'états, peu importe les modifications qu'ils subissent. Des exemples de ces états incluent le fait que cliquer sur le bouton de déconnexion devrait amener l'utilisateur à l'état déconnecté ou à la page de connexion. Un autre exemple est un état de liste de produits, lorsque l'utilisateur clique sur le lien d'aperçu, il devrait être dirigé vers l'état d'aperçu.

```
//the menu item on the home page should always have the class attribute 'menuElement'
Condition correctMenuItem = new XPathCondition("//DIV[@id='menu']/UL/LI[contains(@class, 'menuElement')]");
Condition whenAtHomePage = new JavaScriptCondition("document.title=='Home'");
crawler.addInvariant("Home page menu items", correctMenuItem, whenAtHomePage);
```

**FIGURE 3.1 : Représentation de l'expression XPath**
**©[15]**

Pour appliquer les concepts expliqués, ils ont développé un outil appelé ATUSA, qui se concentre sur l'identification des changements d'état dans les éléments du DOM du navigateur. La figure 3.2 illustre le flux de traitement d'ATUSA.

Les auteurs ont fourni un framework pour exécuter des suites de tests, générées automatiquement à l'aide d'une URL de site web. Ainsi, ils ont transformé chaque chemin trouvé en un cas de test JUnit. Chaque cas de test capture la séquence d'événements de l'état initial à l'état cible. Sur les informations contenues dans le graphe mentionné précédemment, le cas de test JUnit pouvait déclencher des événements. Après chaque invocation d'événement, l'état résultant dans le navigateur est comparé à l'état attendu dans la base de données, qui sert d'oracle, générant des rapports à partir de celui-ci. La suite de tests JUnit générée peut être utilisée avec un navigateur différent pour détecter les incompatibilités du navigateur. De plus, la suite de tests qui indique que tous les tests passent signifie qu'aucune défaillance n'a été trouvée dans l'interface utilisateur, mais si un test échoue, cela signifie que des défaillances pourraient se produire sur l'interface utilisateur.

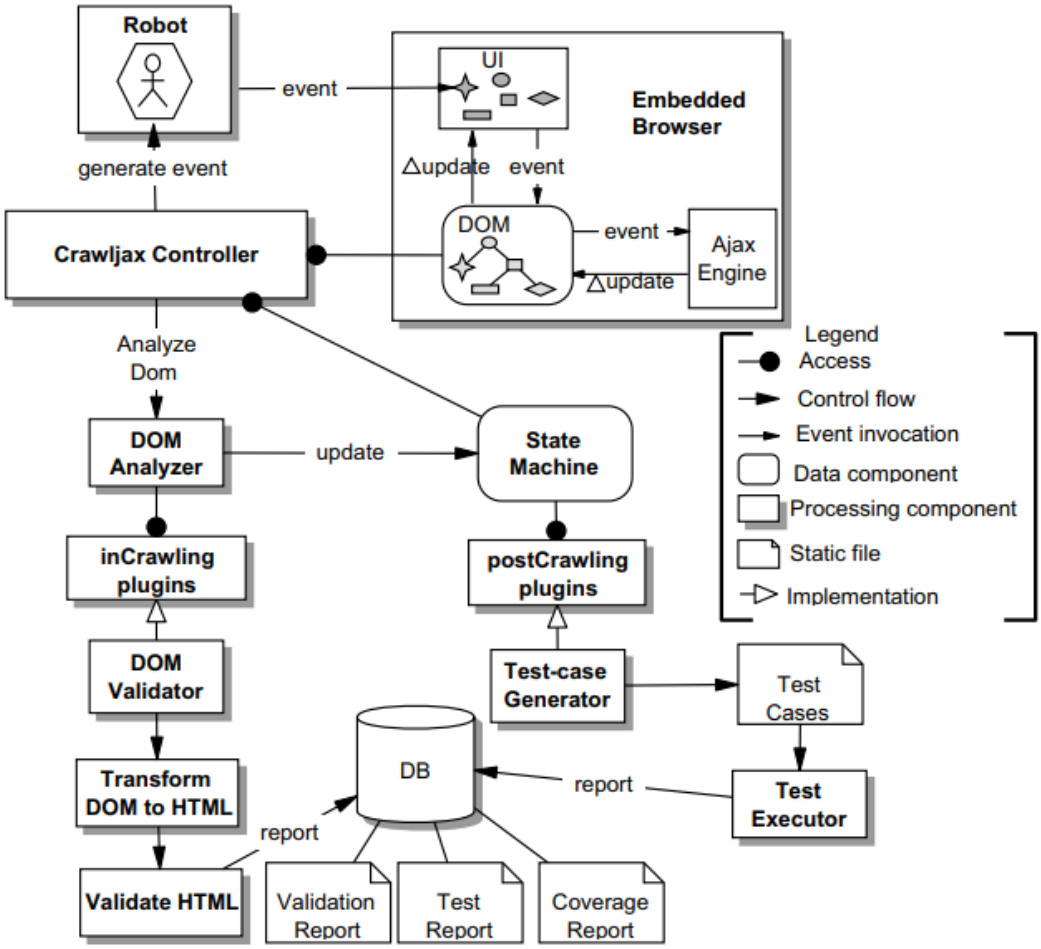



**FIGURE 3.2 : Flux de traitement d'ATUSA**
**©[14]**

Par conséquent, ATUSA offre au testeur des plugins de validation et de détection de défaillances. ATUSA propose des composants de vérification d'invariants génériques, un mécanisme de plug-in pour ajouter des validateurs d'état spécifiques à l'application et la génération d'une suite de tests à partir du graphe d'écoulement des états inféré. Chaque fois qu'une défaillance est détectée, le rapport d'erreur avec le chemin d'exécution causal est enregistré dans la base de données afin qu'il puisse être reproduit ultérieurement si nécessaire.

Dans leur étude ultérieure [15], les auteurs ont étendu leur travail précédent et apporté des améliorations à leur outil ATUSA, qui est open source et basé sur des plugins. Ils ont

travaillé sur la fonctionnalité côté client, testant les événements réels de l'utilisateur pour déduire automatiquement un modèle abstrait. Pour ce faire, ils ont utilisé un web crawler pour imiter le comportement de l'utilisateur et capturer les changements d'état.

Les données capturées ont ensuite été traitées à l'aide de comparateurs pour extraire uniquement les informations pertinentes de l'arbre DOM, en filtrant les attributs, les horodatages et les problèmes de style qui n'étaient pas critiques pour l'analyse. Le flux d'exécution des différents types de plugins utilisés dans leur outil est illustré dans la Figure 3.3.

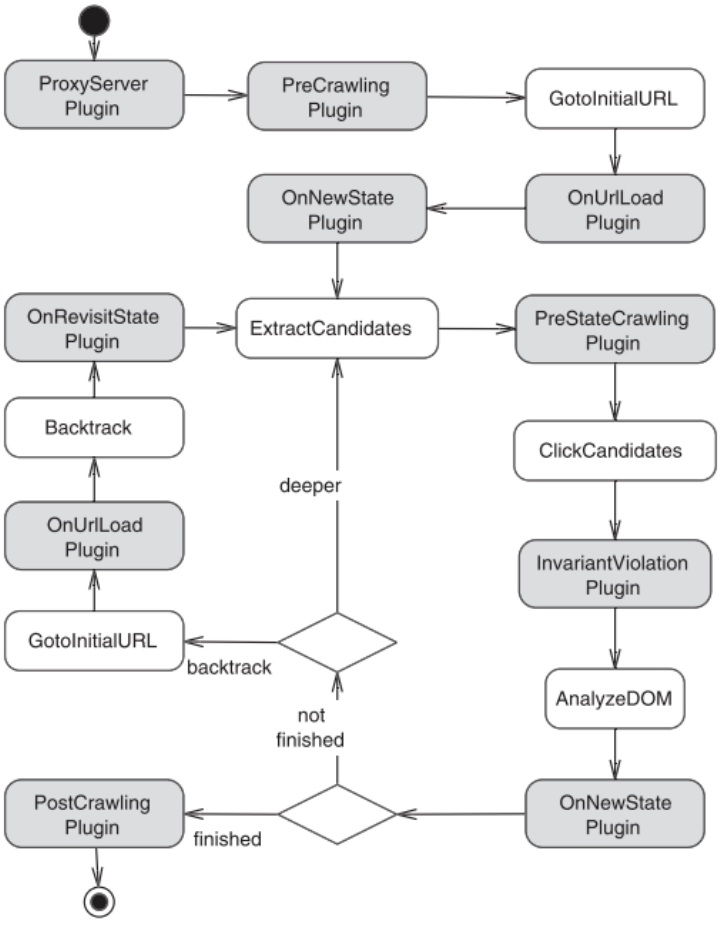



**FIGURE 3.3 : Le flux d'exécution des plugins utilisés dans l'outil ATUSA ©[15]**

Chaque cas de test capture une séquence d'événements de l'état initial à l'état final. Après chaque invocation d'événement, des assertions sont vérifiées pour s'assurer si le nouvel état correspond au résultat attendu. Ainsi, la procédure peut être résumée en pré-crawling (après le chargement du navigateur, il déclenche les plugins d'authentification et la vérification du code source), en cours de crawling (vérification des invariants après les changements d'état) et en post-crawling (après que le graphe est inféré, il peut être utilisé pour générer des cas de test).

Les auteurs ont défini trois questions de recherche pour mener leur recherche. L'objectif était d'évaluer les capacités de révélation des défauts, l'évolutivité, l'effort manuel requis et le niveau d'automatisation de leur approche. Les questions de recherche étaient les suivantes : i) Quelle est la capacité de révélation des défauts d'ATUSA ? ; ii) ATUSA fonctionne-t-il bien ? Est-il scalable ? ; iii) Quel est le niveau d'automatisation lors de l'utilisation d'ATUSA et quel est l'effort manuel impliqué dans le processus de test ?

Dans [14], ils ont réalisé deux études de cas. Le premier concerne un outil open-source AJAX appelé TUDU. Le second était un projet commercial AJAX développé par eux-mêmes. Dans [15], ils ont également testé quatre applications web, à savoir TheOrganizer, Taskfreak, Hitlist et Thetunnel.

Ils ont constaté qu'ATUSA était efficace pour détecter automatiquement des bogues dans le code AJAX qui auraient été difficiles à identifier manuellement. Ainsi, l'outil pourrait aider à révéler des défauts génériques, tels que des violations du DOM. Des exemples de ces violations incluent des images non affichées, une synchronisation rompue, un historique de navigation incohérent, des tables corrompues et des entrées manquantes.

Les performances et la scalabilité de l'outil ont été jugées acceptables, car il faut moins de 6 minutes pour crawler et tester TUDU, en analysant 332 éléments cliquables et en détectant

34 états. Il s'est avéré utile pour documenter et tester les applications AJAX et a offert une configuration rapide de l'environnement. Les auteurs ont également constaté que coder ces invariants en Java était une tâche relativement gérable, indiquant que l'effort nécessaire pour les mettre en œuvre était acceptable. De plus, les testeurs ont la flexibilité de spécifier des invariants génériques ou spécifiques.

Cette étude utilisait des URL pour démarrer un navigateur, à partir duquel ils extrayaient des XPaths pour identifier des invariants du DOM. Cependant, ils n'ont pas travaillé avec des pages web statiques, ils ont analysé les changements d'état du DOM en considérant les interactions utilisateur. De plus, nous n'avons pas identifié de préoccupations claires concernant les bugs de mise en page des pages web.

### 3.1.2 DODOM: LEVERAGING DOM INVARIANTS FOR WEB 2.0 APPLICATION RELIABILITY ET DODOM: LEVERAGING DOM INVARIANTS FOR WEB 2.0 APPLICATION ROBUSTNESS TESTING

Ces deux publications seront traitées dans la même sous-section car elles appartiennent aux mêmes auteurs, à savoir Karthik Pattabiraman et Benjamin Zorn. La première étude a été publiée en 2009 sous forme de rapport technique et la seconde a été publiée en 2010 lors du 21e International Symposium on Software Reliability Engineering.

Dans l'étude menée par Pattabiraman et Zorn [16, 17], ils ont discuté des défis liés au test des applications web en raison de leur comportement non déterministe et de la difficulté des schémas de détection d'erreurs côté client. Ils considèrent que ce non-déterminisme provient de facteurs tels que de petits changements introduits par le serveur, l'asynchronie dans les messages réseau envoyés ou reçus dans le désordre, et de petites variations dans le timing des événements côté client, pour en illustrer quelques-uns.

Pour relever ces défis, les auteurs mettent en avant l'exploration de l'arbre DOM des applications web, car plusieurs scripts côté client dans l'application partagent un état et communiquent principalement entre eux via le DOM. Par conséquent, ils estiment que la correction du DOM est essentielle pour la correction de l'application.

Pour tester la correction de l'application, ils exposent les défis auxquels sont confrontés les développeurs lorsqu'ils veulent tester la robustesse d'une application web aux fautes. Les auteurs estiment que c'est une tâche chronophage et nécessite que l'utilisateur répète les mêmes interactions avec l'application pour chaque faute injectée. De plus, ils affirment que les développeurs doivent se fier à la perception visuelle pour déterminer si une faute injectée affecte l'application. Par conséquent, ils proposent une méthode pour caractériser le comportement correct d'une application web en vue de tests de robustesse.

Les chercheurs comprennent que le test de robustesse d'une application repose sur des séquences d'actions de l'utilisateur avec une application web. Ainsi, ils proposent de multiples exécutions d'applications pour en extraire les caractéristiques qui définissent un comportement attendu. Ce comportement est lié aux modifications apportées au DOM par l'application en réponse à divers événements tels que des interactions utilisateur ou des messages réseau. Par des exécutions répétées, ils souhaitent cartographier les éléments de l'arbre DOM, en distinguant ceux qui ont subi des modifications en raison de déclencheurs d'événements et ceux qui sont restés stables après la séquence d'exécutions. Les éléments qui sont restés stables, les auteurs les ont classés comme invariants DOM. Par conséquent, un invariant DOM est un sous-arbre du DOM de l'application web partagé par plusieurs exécutions.

En résumé, l'extraction d'invariants à partir du DOM des applications web consiste à i) enregistrer une séquence d'interactions utilisateur et d'événements sur une page. ii) rejouer cet enregistrement sur plusieurs exécutions et capturer la séquence de DOM générés après

chaque événement. iii) extraire les invariants sur l'ensemble de toutes les séquences DOM à l'aide d'un processus d'apprentissage hors ligne.

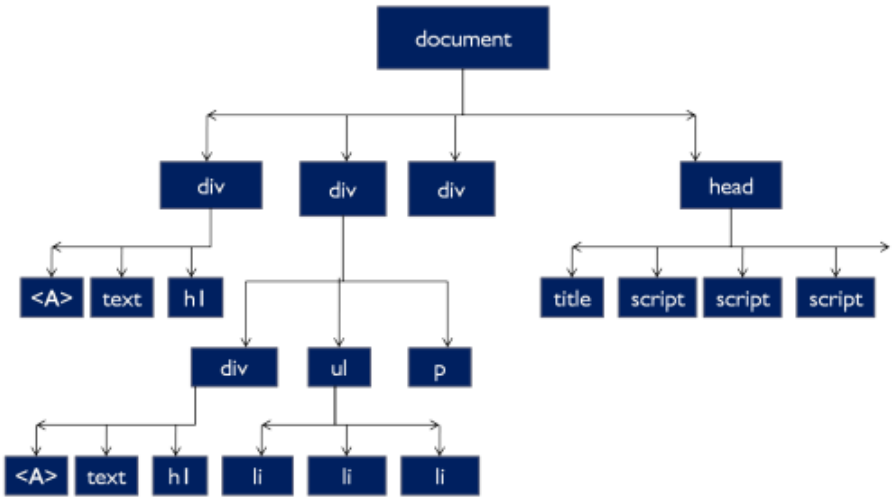



**FIGURE 3.4 : Représentation de l'arbre DOM classique**
**©[15]**

Sur la figure 3.4, les auteurs représentent un arbre DOM classique. Après plusieurs rounds d'exécution d'une application web en utilisant des événements déclenchés, ils peuvent identifier le patron d'invariants DOM. La figure 3.5 est une autre représentation de 3.4, mais avec les invariants DOM identifiés colorés en bleu. En revanche, les éléments en vert sont ce que les auteurs ont exemplifié comme non invariant car ils ont subi des changements causés par des événements déclenchés.

Les auteurs estiment que l'extraction des invariants décrite présente trois principaux défis. Premièrement, les applications web présentent de petits changements d'une exécution à l'autre en raison de variations au niveau du serveur et du client. Ainsi, l'invariant DOM ne doit pas inclure de tels changements, sinon il entraînera des faux positifs. Deuxièmement, les invariants extraits doivent conserver autant que possible le DOM d'origine. Cela garantit une couverture de détection élevée des invariants pour les violations de fiabilité et de sécurité.

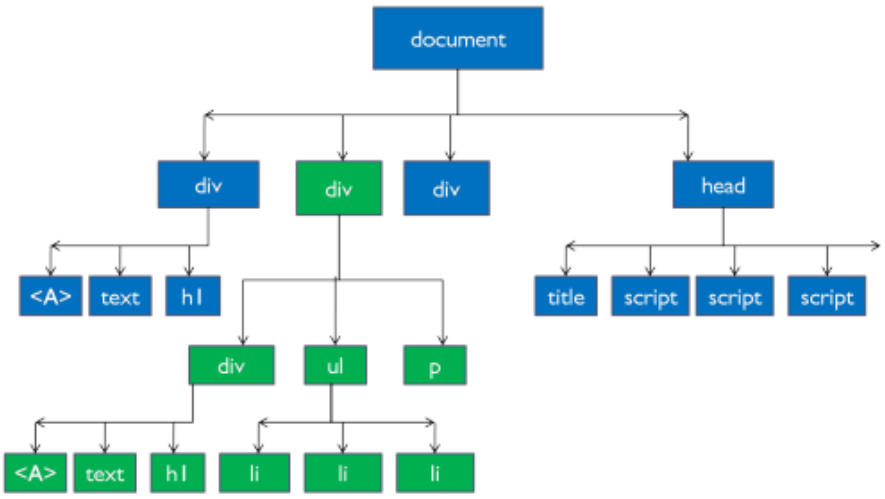



**FIGURE 3.5 : Représentation de l'arbre DOM avec les invariants DOM identifiés colorés en bleu ©[15]**

Troisièmement, le système d'extraction des invariants doit être compatible avec différents navigateurs et plates-formes.

Après avoir défini le DOM invariant, ils ont proposé l'injection de défauts dans l'application. Par exemple, en supposant qu'une application web importe un script défectueux et que le script a accès au DOM de la page web. Cependant, il n'est pas censé modifier une partie de l'arbre DOM invariant. Si tel est le cas, la modification sera détectée comme une erreur, ce qui peut caractériser un écart significatif par rapport au comportement attendu ou une violation potentielle de la sécurité.

Les principaux types de défauts proposés par les auteurs étaient les erreurs d'événement côté client et les défaillances de domaine. Ainsi, les erreurs d'événement peuvent être causées par des exceptions dans les gestionnaires d'événements correspondants ou par le fait que les événements ne sont pas déclenchés correctement. Dans certains cas, le navigateur web peut interrompre un gestionnaire d'événements s'il s'exécute trop longtemps. Les défaillances de domaine font référence à une défaillance du réseau ou à l'indisponibilité des serveurs

de domaine. Elles peuvent également être causées par des plugins côté client ou des proxys administratifs qui peuvent bloquer les scripts de certains domaines.

Pour automatiser le processus d'enregistrement d'une séquence d'interaction utilisateur avec une application web, de la rejouer et d'extraire des invariants des séquences DOM observées, les chercheurs ont introduit l'outil qu'ils ont développé, appelé DoDOM. Cet outil était capable de trouver des nœuds pendant la relecture, en utilisant à la fois le contenu d'un nœud dans le DOM et sa position relative par rapport à d'autres nœuds. La comparaison était basée sur des mesures heuristiques, garantissant que les événements étaient rejoués sur les nœuds d'origine sur lesquels ils s'étaient produits même si la page web avait subi des modifications pendant la relecture. L'outil a été construit avec JavaScript et ne nécessitait aucune modification du navigateur web.

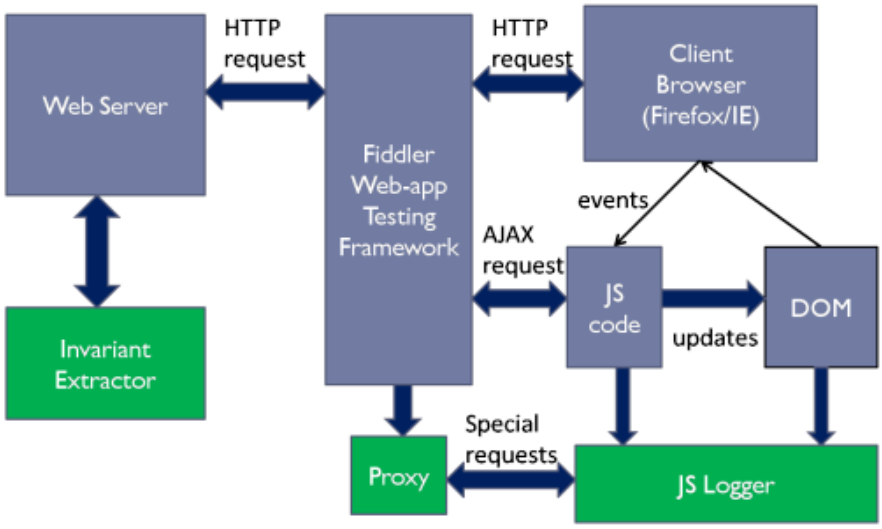



**FIGURE 3.6 : Illustration de l'architecture de DoDOM**
**©[16]**

La figure 3.6 illustre l'architecture de DoDOM. Les rectangles gris représentent une architecture web classique, tandis que les rectangles verts indiquent la mise en œuvre spécifique introduite par les auteurs. Le JS Logger joue un rôle clé dans l'exécution de l'opération

principale de DoDOM. Il s'agit d'un morceau de code Javascript qui est exécuté dans le navigateur du client. Le JS Logger peut lire/écrire dans le DOM de la page web, installer des gestionnaires d'événements et enregistrer les modifications apportées au DOM.

Le proxy s'exécute côté client en tant que plugin. Son principal objectif est d'injecter le code JS Logger dans la page web de l'application, de collecter les événements et les réponses envoyés par le JS Logger, et de les enregistrer. Ainsi, le proxy sert de pont de communication et de filtre entre le JS Logger et l'application web. L'Invariant Extractor effectue une analyse hors ligne pour trouver une séquence d'arbres DOM invariants pour chaque événement.

Les chercheurs ont utilisé DoDOM pour tester trois applications web, à savoir Slashdot, CNN et JavaPetStore. Slashdot agrège des actualités liées à la technologie en provenance de différents sites web et permet aux utilisateurs de commenter une histoire. JavaPetstore est une application gratuite qui imite un site web de commerce électronique pour l'achat d'animaux de compagnie. CNN est un site d'actualités populaire qui propose du contenu personnalisé à ses lecteurs.

Pour tester leur approche, les auteurs ont posé trois questions de recherche dans leur première étude, à savoir : i) Combien d'exécutions faut-il pour apprendre les DOM invariants pour une application web ? ii) Combien d'erreurs d'événement affectent le DOM de l'application web et quelle est l'efficacité des invariants pour détecter ces erreurs ? iii) Combien de défaillances de domaine affectent le DOM de l'application web et quelle est l'efficacité des invariants pour détecter ces erreurs ?

Pour la 1re question de recherche, leur approche consiste à choisir de manière aléatoire un sous-ensemble d'une séquence d'exécutions de l'application, qu'ils ont appelé l'ensemble d'entraînement. Avec cet ensemble, ils ont effectué un exercice en faisant varier sa taille pour comprendre à quelle vitesse les invariants convergent vers une valeur stable. Les caractéris-

tiques de l'arbre DOM qu'ils évaluent sont le nombre de nœuds, le nombre moyen d'enfants par nœud, le nombre maximal de niveaux à partir de chaque nœud, la hauteur du sous-arbre et le nombre moyen de descendants totaux par nœud.

Les auteurs ont varié la taille de l'ensemble d'entraînement de 1 à 10 sur 58 exécutions qu'ils ont effectuées et ont obtenu la séquence DOM invariante. Par conséquent, ils ont découvert que la taille de l'ensemble d'entraînement augmente et le nombre de nœuds dans le DOM diminue car de plus en plus de nœuds sont éliminés des DOM invariants. Cependant, le nombre de nœuds cesse de diminuer une fois que la taille de l'ensemble d'entraînement atteint 6 (environ 10% des 58 exécutions). Ainsi, les séquences DOM invariantes convergent vers la stabilité avec une taille d'ensemble d'entraînement de 6 exécutions.

Pour la deuxième question de recherche, ils ont mesuré la couverture de détection d'erreur de l'injection de fautes telles User-Event Error, Message Error, and Timeout Error. Ainsi, lorsqu'une injection de faute modifiait le sous-ensemble de l'arbre DOM considéré comme invariant DOM et lorsque cette modification était détectée par DoDOM, ils la classaient comme une détection réussie d'erreur. Ils ont découvert que le taux de détection variait en fonction de la faute injectée. Certains événements avaient des taux de détection de 0 et étaient liés aux événements de délai d'attente, et leurs gestionnaires ne mettaient pas à jour le DOM. Les autres événements étaient des clics de souris et des événements de traitement de messages, et les gestionnaires correspondants ajoutent ou suppriment des nœuds du DOM. Ainsi, les invariants détectent toutes les erreurs d'événement qui affectent le DOM.

Pour la troisième question de recherche, ils ont bloqué chaque domaine sur la page web, un à la fois en utilisant le plugin NoScript. Dans l'article, ils ont donné des exemples de domaines en tant que liens disponibles dans l'application web qui pourraient affecter l'arbre DOM. Dans ce cas, les liens Fsdn.com et mediaplex.com affectaient le DOM de Slashdot, l'une des applications web qu'ils ont testées. Ainsi, ils ont déclaré que l'un des 5 domaines

inclus par Slashdot avait un effet sur le DOM et que DoDOM fournissait une couverture à 100% pour les défaillances de ce domaine.

Les principales contributions des auteurs ont été le développement de l'outil DoDOM et la démonstration de sa capacité à apprendre des invariants en aussi peu que six exécutions. Ils ont également constaté que l'outil atteignait un taux de couverture élevé pour la détection d'erreurs approchant les 100%. De plus, ils ont découvert qu'un sous-arbre fiable, représentant les invariants DOM, pouvait être trouvé avec une taille d'ensemble d'entraînement de six ou plus. Leur étude a examiné l'arbre DOM des applications web pour en déduire des invariants. De plus, ils se sont préoccupés des tests sur ces applications web. Malgré cela, nous n'avons pas trouvé de préoccupations spécifiques concernant les bogues de mise en page détectés sur les applications web comme le fait notre recherche. Leur focus était basé sur les fautes dans ces applications.

## 3.2 APPROCHES POUR IDENTIFIER LES BOGUES DE MISE EN PAGE

Dans les études suivantes, ils ont également utilisé l'arbre DOM comme référence pour leurs enquêtes. Ils ont pris des sites web pour explorer la possible existence de bogues de mise en page en utilisant le concept d'invariants. La deuxième étude a utilisé des captures de sites web.

### 3.2.1 DECLARATIVE LAYOUT CONSTRAINTS FOR TESTING WEB APPLICATIONS

Dans l'étude menée par Hallé et al. [1], les auteurs présentent un résumé des bogues de mise en page découverts dans 35 pages et applications web. Ils fournissent des descriptions détaillées des caractéristiques des bogues identifiés et établissent des liens entre chaque bogue

et la page web correspondante. Dans la section 2.2, nous avons d'ailleurs fait référence à certains de ces bogues de mise en page.

Pour détecter les bogues, les auteurs ont développé un outil appelé Cornipickle. Il a été construit comme un langage déclaratif capable d'exprimer des conditions sur le DOM et les attributs CSS des éléments d'une page. Cornipickle a été conçu pour permettre l'écriture de spécifications conviviales pour les humains, facilitant ainsi leur adoption par le public. Une caractéristique notable de l'outil est que les développeurs peuvent écrire des spécifications en utilisant leur propre vocabulaire. Par exemple, on peut définir des prédicats en utilisant la construction "We say that... when...". Le texte entre "that" et "when" est interprété comme une chaîne littérale qui peut contenir des variables. S'ils découvrent un bogue de mise en page, les développeurs peuvent donc utiliser Cornipickle pour écrire une règle qui permettra de détecter de nouvelles occurrences de ce bogue particulier ; on appelle cela un test de régression.

Cornipickle permet l'expression de propriétés sous forme d'assertions sur le contenu et les attributs d'une capture spécifique d'une page web. Les événements utilisateur tels que les pressions de touche et les clics de souris sont eux aussi représentés sous forme d'attributs. Le langage permet la quantification du premier ordre et définit des expressions pour sélectionner des éléments semblables aux sélecteurs de CSS. La Figure 3.7 illustre la grammaire utilisée dans le langage de Cornipickle.

Cornipickle a été conçu pour respecter quatre exigences fondamentales. Premièrement, il devait s'adapter au plus grand nombre possible de navigateurs et de systèmes d'exploitation. Deuxièmement, les informations devaient être collectées côté client. Troisièmement, pour minimiser la charge dans le navigateur, l'évaluation des spécifications ne devait pas s'exécuter côté client ; les informations devaient être collectées et envoyées à un serveur backend pour le

**Cornipickle statements**

⟨*statement*⟩ ::= ⟨*foreach*⟩ | ⟨*exists*⟩ | ⟨*binary-stmt*⟩ | ⟨*negation*⟩
| ⟨*temporal-stmt*⟩ | ⟨*userdef-stmt*⟩ | ⟨*let*⟩ | ⟨*when*⟩
⟨*binary-stmt*⟩ ::= ⟨*equality*⟩ | ⟨*gt*⟩ | ⟨*lt*⟩ | ⟨*regex-match*⟩ | ⟨*and*⟩ | ⟨*or*⟩ | ⟨*implies*⟩
⟨*temporal-stmt*⟩ ::= ⟨*globally*⟩ | ⟨*eventually*⟩ | ⟨*never*⟩ | ⟨*next*⟩ | ⟨*next-time*⟩

**First-order logic**

⟨*foreach*⟩ ::= For each ⟨*var-name*⟩ in ⟨*set-name*⟩ ( ⟨*statement*⟩ )
⟨*exists*⟩ ::= There exists ⟨*var-name*⟩ in ⟨*set-name*⟩ such that ( ⟨*statement*⟩ )
⟨*when*⟩ ::= When ⟨*var-name*⟩ is now ⟨*var-name*⟩ ( ⟨*statement*⟩ )
⟨*let*⟩ ::= Let ⟨*var-name*⟩ be ⟨*property-or-const*⟩ ( ⟨*statement*⟩ )
⟨*and*⟩ ::= ( ⟨*statement*⟩ ) And ( ⟨*statement*⟩ )
⟨*or*⟩ ::= ( ⟨*statement*⟩ ) Or ( ⟨*statement*⟩ )
⟨*implies*⟩ ::= If ( ⟨*statement*⟩ ) Then ( ⟨*statement*⟩ )
⟨*negation*⟩ ::= Not ( ⟨*statement*⟩ )

**Temporal statements**

⟨*globally*⟩ ::= Always ( ⟨*statement*⟩ )
⟨*never*⟩ ::= Never ( ⟨*statement*⟩ )
⟨*next*⟩ ::= Next ( ⟨*statement*⟩ )
⟨*eventually*⟩ ::= Eventually ( ⟨*statement*⟩ )
⟨*next-time*⟩ ::= The next time ( ⟨*statement*⟩ ) Then ( ⟨*statement*⟩ )

**FIGURE 3.7 : La grammaire utilisée dans le langage de Cornipickle ©[1]**

traitement. Quatrièmement, un utilisateur devait être capable d'ajouter, de supprimer ou de modifier les spécifications évaluées par l'outil.

La figure 3.8 illustre l'architecture de Cornipickle. Elle est divisée en plusieurs étapes, mais nous les résumons comme suit :

- Un développeur informe un ensemble d'énoncés déclaratifs ;
- L'ensemble est stocké dans la mémoire de Cornipickle ;
- Cet ensemble reçoit un identifiant unique, qui peut être référencé après le chargement d'une page web ;

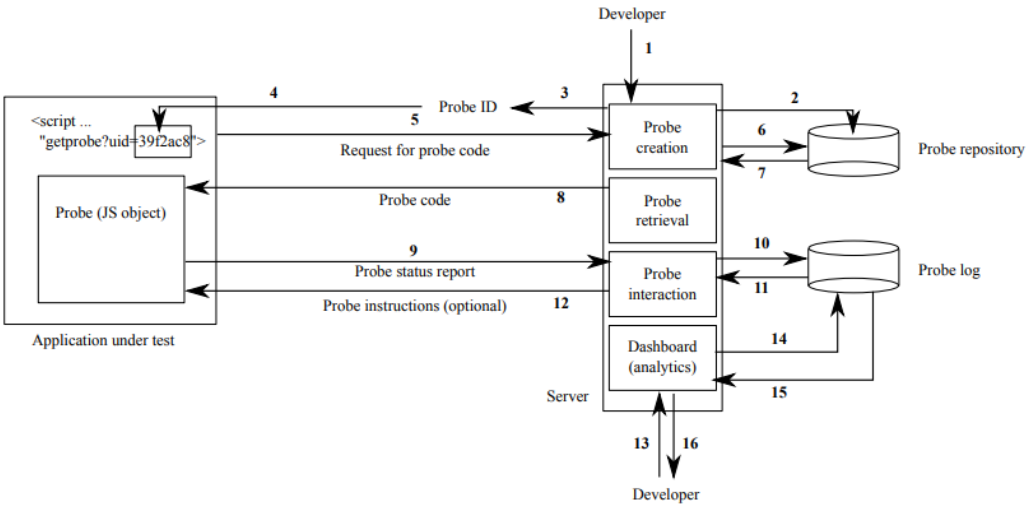



**FIGURE 3.8 : Illustration de l'architecture de Cornipickle**
**©[1]**

- Cornipickle créé dynamiquement la sonde (*probe*) JavaScript correspondant à l'ensemble d'assertions à évaluer et l'envoie au client ;
- Lorsque l'utilisateur déclenche des événements, la sonde (*probe*) collecte toutes les informations pertinentes pour le contenu de la page et transmet ces informations au serveur Cornipickle ;
- Ces informations sont également enregistrées dans un fichier des log ;
- Les informations sur l'état actuel des assertions en cours d'évaluation peuvent être renvoyées à la sonde (*probe*) ;

Selon la documentation de Mozilla [6], une sonde "est le concept général d'instrumentation du code de manière sélective pour collecter des types particuliers de données. Les *probes* peuvent consister en un code spécial compilé dans un programme et/ou un mécanisme de surveillance inséré par un programme externe". Dans le cas de Cornipickle, la sonde récupère l'état des éléments de la page envoie ces informations au serveur backend.

6. https ://wiki.mozilla.org/Performance :Probes

Ce serveur est responsable d'évaluer les assertions à partir des données fournies par la sonde, en suivant le langage de spécification. Il peut ainsi déterminer quand une assertion était fausse, et également identifier les éléments responsables de cette violation. Pour ce faire, les auteurs ont développé un système de « témoins » et de « verdicts » ; un témoin est un arbre d'éléments du DOM et un verdict est composé d'une valeur de vérité et de deux témoins : le premier composé des éléments qui expliquent pourquoi l'assertion est vraie, et l'autre des éléments qui expliquent pourquoi elle est fausse. En théorie, un seul de ces deux ensembles est non-vide pour un verdict donné.

La figure 3.9 donne un exemple. Selon l'arbre, l'expression suivante était indiquée : `For each $x in $(p) For each $y in $(p) $x's width equals $y's width`. Cette expression stipule que tous les paragraphes de la page sont de même largeur.

Dans ce cas, la sonde récupérera depuis la page la largeur de tous les paragraphes. Dans la figure 3.9, on peut voir deux éléments p avec des largeurs différentes. Lorsqu'une propriété est ainsi violée, un ensemble d'éléments est extrait du témoin résultant. L'ID unique de ces éléments est renvoyé côté client, où la sonde peut mettre en évidence les éléments impliqués. Dans la figure, éléments en cause sont affichés en rouge.

Les chercheurs ont constaté que Cornipickle impose une charge minimale sur les applications web existantes. Il est capable d'exprimer des conditions pour les bogues liés à la mise en page et détecte avec précision les bogues identifiés dans des captures de sites Web. Il fournit également un ensemble approprié d'expressions de langage rédigées pour détecter les bogues et empêcher leur réapparition.

L'étude [1] traitait du sujet de l'identification des bogues sur les pages web. Les chercheurs ont créé un outil appelé Cornipickle, qui est un langage déclaratif permettant d'exprimer

```
<html>
 <head>
  <title>My title</title>
 </head>
 <body>
  <h1>The first page</h1>
  <p style="color:red;width:400px">
   Hello world</p>
  <p style="font-size:14pt;width:200px;">
   Another <b>paragraph</b></p>
  <p style="width:400px;"></p>
 </body>
</html>
```

(a)

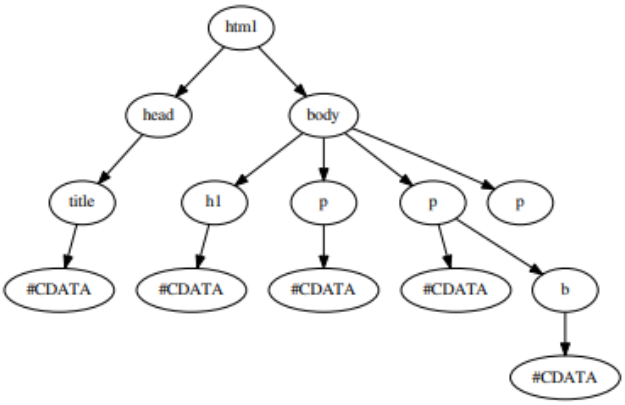



**FIGURE 3.9 : Représentation de l'analyse DOM avec Cornipickle**
**©[1]**

les propriétés du DOM et des CSS pour détecter les bogues de mise en page des sites web. Nous avons constaté qu'ils n'ont pas utilisé d'invariants pour analyser la mise en page des pages web. Cette étude se concentrait sur la démonstration des fonctionnalités de leur outil pour identifier les bogues.

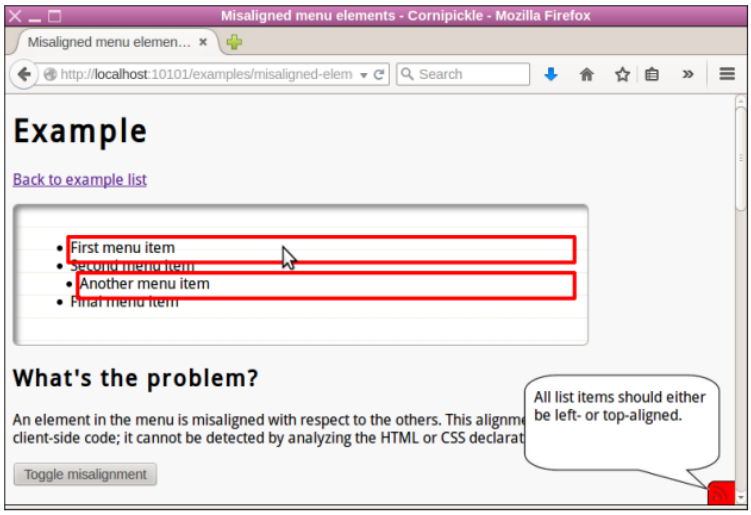



**FIGURE 3.10 : Représentation du résultat de l'analyse DOM avec Cornipickle ©[1]**

### 3.2.2 DETECTING RESPONSIVE WEB DESIGN BUGS WITH DECLARATIVE SPECIFICATIONS

Dans l'étude [18], les chercheurs ont présenté une approche pour tester la réactivité (*responsiveness*) de la mise en page des sites web, en tenant compte de la largeur de la fenêtre où ils sont affichés. En effet, les pages web sont sujettes à des changements constants en fonction de l'appareil sur lequel elles sont affichées, tels qu'un ordinateur ou une tablette. Ces variations peuvent entraîner des problèmes de mise en page, tels que le chevauchement ou le mauvais alignement d'éléments, compromettant ainsi l'expérience de l'utilisateur. L'approche définit ces bogues comme des assertions exprimées dans un langage déclaratif spécifique au domaine conçu pour les interfaces web. Ce langage a été développé dans l'étude [1] précédemment explorée. Les auteurs n'ont considéré que la dimension horizontale, car la hauteur d'une page est généralement gérée avec une barre de défilement et ne pose pas de problème particulier.

L'objectif principal de l'étude était de vérifier cinq problèmes de mise en page qui avaient été décrits par [19]. Walsh et al. [19] appellent respectivement ces cinq problèmes « collision d'éléments » (chevauchement d'éléments), « protrusion d'éléments » (élément dépassant de son élément parent), « protrusion de la fenêtre d'affichage » (éléments poussés en dehors de la fenêtre d'affichage), « empilement d'éléments » (éléments poussés sur une ligne supplémentaire) et « intervalle restreinte » (éléments affichés sur une plage limitée de largeurs). La figure 3.11 donne une représentation visuelle de ces problèmes de mise en page.

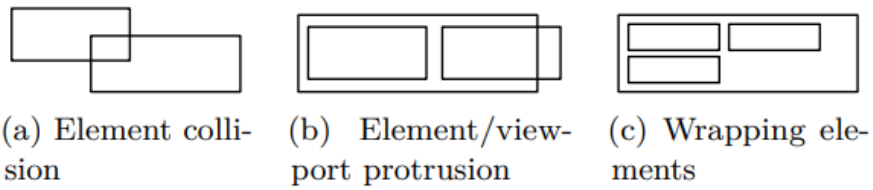



**FIGURE 3.11 : Représentation visuelle de ces problèmes de mise en page ©[18]**

La figure 3.12 illustre le schéma de l'approche proposée par les auteurs. La même page web est ouverte plusieurs fois dans un navigateur web. La première fenêtre conserve la taille de fenêtre standard, tandis que les fenêtres suivantes rapetissent progressivement en largeur. Pour chaque capture de page (*snapshot*), un état de la mise en page est enregistré, ce qui produit une séquence de pages web distinctes qui peuvent ensuite être comparées et analysées.

Afin de détecter d'éventuels problèmes, un élément ne peut pas être évalué sur une page web indépendamment des éléments d'autres pages web ; c'est pourquoi il doit être possible de faire le lien entre le même élément dans plusieurs captures de la même page. Pour gérer ce problème, les auteurs ont combiné deux outils, à savoir Cornipickle et Crawljax.

Comme on l'a vu plus tôt, Cornipickle permet aux développeurs d'écrire des spécification déclaratives. Entre autres, l'outil dispose d'opérateurs basés sur la Logique Temporelle Linéaire

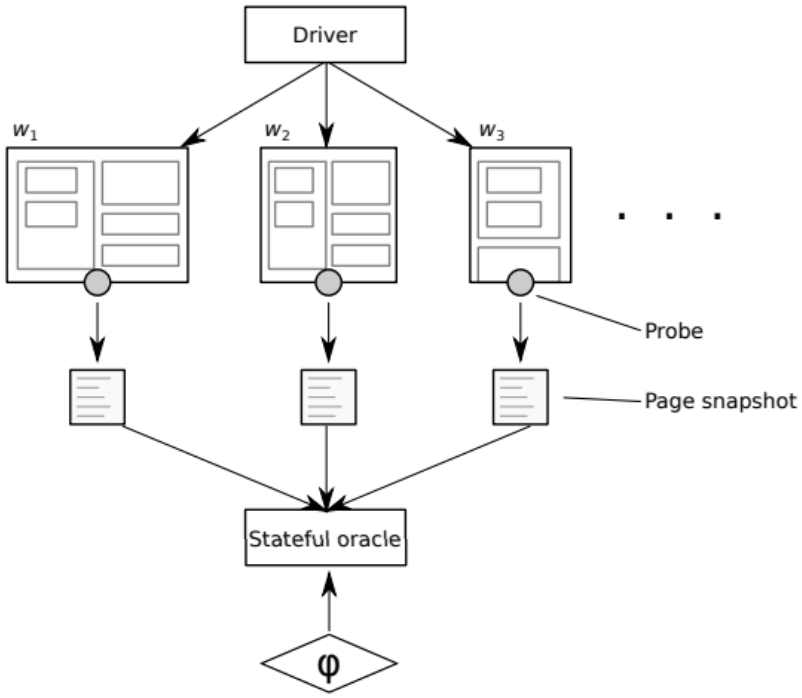



**FIGURE 3.12 : Représentation de l'approche**
**©[18]**

(LTL) [20]. Grâce à ces opérateurs, un développeur peut exprimer des assertions sur un document web au fil du temps en déclarant des déclarations telles que "When x is now y then z", qui permet de placer dans les variables symboliques *x* et *y* deux copies du même élément dans deux versions de la même page, pour ensuite comparer leurs attributs.

De son côté, Crawljax peut détecter automatiquement l'état dynamique des applications web modernes. Après chaque interaction avec une page, les modifications dans la structure de son arbre DOM produisent une nouvelle version de la page qui peut être vue comme un état dans une machine à états finis. Ce crawler peut interagir avec Cornipickle grâce à son architecture de plugin. Il remplit un rôle similaire à la sonde en sérialisant l'état de la page web et en l'envoyant à Cornipickle. Dans ce travail en particulier, Crawljax a été utilisé pour manipuler la taille de la fenêtre du navigateur web. Chaque modification de la taille de la

fenêtre était traitée comme une page web distincte, permettant des comparaisons du même élément sur plusieurs captures de page.

Pour exprimer les bogues de réactivité, les auteurs ont défini deux règles de base. Ils ont d'abord défini une construction de la forme "`$x and $y are the same`"; cette déclaration permet d'identifier si les variables $x et $y réfèrent à deux copies du même élément dans deux captures distinctes d'une page web. Ils ont ensuite défini la construction "`not the same`" comme la négation de la précédente. L'alignement de deux éléments était défini avec les constructions "top-aligned" et "left-aligned", de telle sorte qu'ils étaient alignés lorsque leurs valeurs étaient égales. La figure 3.13 montre comment les auteurs ont défini cette construction.

```
We say that $x and $y are the
same when (
  $x's cornipickleid equals
  $y's cornipickleid).

We say that $x and $y are not
the same when (
  Not ($x and $y are the same)).

We say that $x is visible when (
  Not ($x's display is "none")).

We say that $x and $y are
top-aligned when (
  $x's top equals $y's top).

We say that $x and $y are
left-aligned when (
  $x's left equals $y's left).
```

**FIGURE 3.13 : Représentation des constructions d'alignements "top-aligned" et "left-aligned" ©[18]**

La deuxième règle traite des situations de chevauchement, définies comme la présence de deux éléments qui sont à la fois visibles, et qui s'intersectent à la fois horizontalement et verticalement. La figure 3.14 représente cette construction.

Les bogues traités dans l'étude impliquent également le concept de contenance (*containment*), qui s'applique lorsque les limites d'un élément sont entièrement incluses dans les limites d'un autre. Les constructions de contenance sont présentées dans la Figure 3.15. Ces règles

```
We say that $x x-intersects $y
when (((($y's right - 1)
    is greater than $x's left) And
  (($x's right - 1)
    is greater than $y's left)).

We say that $x y-intersects $y
when (((($y's bottom - 1)
    is greater than $x's top) And
  (($x's bottom - 1)
    is greater than $y's top)).
```

```
We say that $x and $y
overlap when (
  (($x is visible) And
      ($y is visible)) And (
    ($x x-intersects $y)
    And
    ($x y-intersects $y))).

We say that $x and $y do not
overlap when (
  Not ($x and $y overlap)).
```

**FIGURE 3.14 : Représentation des constructions de chevauchent**
**©[18]**

définissent deux types de contenance : le cas d'un élément enfant contenu dans son parent, et celui d'un élément arbitraire contenu dans la fenêtre d'affichage globale du navigateur.

```
We say that $c is horizontally
  inside $p when (
  ($c's left is greater than
    ($p's left - 2))
  And
  ($c's right is less than
    ($p's right + 2))).

We say that $c is vertically
  inside $p when (
  ($c's top is greater than
    ($p's top - 2))
  And
  ($c's bottom is less than
    ($p's bottom + 2))).
```

```
We say that $c is fully
  inside $p when (
  If (($c is visible) And
    ($p is visible)) Then (
    ($c is horizontally inside $p)
    And
    ($c is vertically inside $p))).

We say that $x is fully inside
the viewport when (
  If ($x is visible) Then (
    (($x's left + 2) is greater than 0)
    And
    ($x's right is less than
      (the page's width + 2)))).
```

**FIGURE 3.15 : Représentation des constructions containment**
**©[18]**

Malgré certaines limitations des outils utilisés, les résultats ont démontré que le programme était efficace pour détecter les types de bogues identifiés par Walsh *et al.*, et qui sont liés à l'évaluation de la position relative des éléments.

### 3.2.3 CRITIQUES DES SOLUTIONS EXISTANTES

Dans notre revue de travaux connexes, nous avons identifié plusieurs études qui ont utilisé des concepts et proposé des approches liées à notre proposition. Certaines de ces études se sont concentrées sur la capture d'événements générés par l'utilisateur à l'aide de plugins et de crawlers d'analyse dynamique des applications web. Ainsi, certaines ont identifié certaines conditions sur les pages web, ce qui a abouti à des invariants. Néanmoins, ces invariants ont leur origine dans des processus externes ou ont été générés par des événements sur l'application web. Par conséquent, ils n'évaluent pas les pages web statiques comme nous le faisons dans notre recherche.

Nous avons également rencontré des travaux qui utilisent des invariants pour la détection d'erreurs et des objectifs de sécurité. Ces études ont développé des outils pour tester la résilience des applications web face aux fautes. De plus, elles ont considéré des arbres DOM distincts dérivés d'événements déclenchés sur les applications web. Dans ces cas, elles ne se sont pas préoccupées des bogues de mise en page sur les pages web.

Nous avons également trouvé une étude qui offre une description complète de divers bogues de mise en page ainsi qu'un outil d'identification et de prévention des bogues. Une autre étude a utilisé des outils pour manipuler la largeur de l'écran afin d'identifier d'éventuels bogues de mise en page. Néanmoins, nous n'avons pas identifié de proposition pour extraire automatiquement des conditions sur l'apparence d'une page afin de mettre en évidence les bogues de mise en page à l'écran.

Contrairement à ces approches parallèles qui impliquent des interactions utilisateur et l'analyse de différents arbres DOM causés par des événements déclenchés, notre solution proposée se concentre sur l'analyse de pages web statiques. En d'autres termes, nos pages web ne sont pas soumises aux interactions utilisateur et aux modifications du DOM résultant,

par exemple, du côté serveur. Nous visons à évaluer la mise en page et à identifier les violations d'invariants, qui représentent des bogues de mise en page dans notre contexte. Notre préoccupation principale est la présentation visuelle des pages web, le reporting et la mise en évidence des invariants violés pour les utilisateurs. Par conséquent, dans le prochain chapitre, nous présenterons la solution que nous proposons pour remédier à cette lacune trouvée dans les études que nous avons présentées.

# CHAPITRE IV

# LA RECHERCHE D'INVARIANTS POUR TESTER LES INTERFACES WEB

Dans le dernier chapitre, nous avons parlé de six publications liées à notre étude. Nous les avons divisées en deux sections, à savoir les approches pour les événements défectueux et les approches pour l'identification des bogues de mise en page. Nous les avons regroupées de cette manière pour faciliter la compréhension, car chaque groupe explore un domaine de recherche distinct.

En particulier, nous avons décrit l'étude de Hallé et al. [1], décrivant une classification des bogues de mise en page de pages web. Les auteurs ont développé un outil appelé Cornipickle, qui utilise un langage déclaratif pour exprimer les propriétés du DOM et du CSS. Cet outil se caractérise par son utilisation conviviale. Il s'est avéré capable d'exprimer des conditions pour les bogues basés sur la mise en page et de détecter avec précision les bogues sur les sites web.

Dans ce chapitre, nous présenterons la solution que nous avons développée. Son objectif est d'éviter que le développeur ait à rédiger les spécifications de l'apparence attendue des pages web, contrairement à ce qui est supposé dans beaucoup d'approches existantes (dont Cornipickle). Pour y parvenir, nous souhaitons générer automatiquement des conditions potentielles s'appliquant à la mise en page des documents web. Concrètement, étant donné un échantillon de pages web de référence, nous sommes à la recherche de conditions qui sont toujours vraies dans toutes les pages de cet échantillon. Intuitivement, cela signifie que ces conditions représentent des aspects de la page web qui ne changent jamais. Comme ils ne varient pas, nous les appellerons donc des *invariants*. Nous émettons l'hypothèse que le suivi de ces invariants auto-générés permet de détecter les bogues de mise en page.

## 4.1 LA GÉNÉRATION D'INVARIANTS

Dans le cadre de cette étude, nous avons concentré notre recherche sur quatre types spécifiques de bogues de mise en page. Ces bogues sont décrits plus en détail dans la section 2.2 et peuvent être résumés comme suit : éléments superposés (éléments web qui entrent en collision les uns avec les autres), protrusion d'éléments (éléments web qui dépassent leur contenu en dehors de leur parent, provoquant un débordement), désalignement horizontal (éléments web qui devraient être alignés selon la référence cartésienne de l'axe X), et désalignement vertical (éléments web qui devraient être alignés selon la référence cartésienne de l'axe Y). Chacun de ces bogues est associé à un type particulier de condition que nous souhaitons trouver avec notre solution. Nous les décrivons plus en détail ci-dessous.

- **Disjoint** : Il se réfère aux éléments en chevauchement. Un élément qui chevauche un autre élément, le cachant généralement totalement ou partiellement.
- **Contained** : Il se réfère aux éléments en protrusion. Un élément qui dépasse l'espace qu'il est censé occuper sur la page Web, provoquant un débordement.
- **Désalignement horizontal** : Un élément qui devrait être aligné avec d'autres éléments, dans les mêmes coordonnées sur l'axe X, mais qui ne l'est pas.
- **Désalignement vertical** : Un élément qui devrait être aligné avec d'autres éléments, dans les mêmes coordonnées sur l'axe Y, mais qui ne l'est pas.

Pour détecter les types de conditions décrites, nous devons identifier les conditions qui se rapportent à la taille relative des éléments, au fait que les éléments sont contenus les uns dans les autres ou en chevauchement, à leur alignement, et s'ils sont répertoriés de gauche à droite ou de haut en bas. Comme nous l'avons expliqué dans la section 2.1, nous pouvons identifier la relation entre les éléments web en naviguant dans l'arbre DOM à l'aide de méthodes et de propriétés spécifiques.

En fonction de la condition qui doit être évaluée, nous devons donc considérer diverses relations entre les nœuds de l'arbre DOM, telles que a) comparer un nœud DOM avec son parent ; b) comparer un nœud DOM avec chacun de ses enfants ; c) comparer tous les éléments DOM ayant le même parent ; d) comparer tous les éléments DOM appartenant à la même classe CSS. Nous avons donc développé un outil, dont nous donnerons les détails plus tard, pour identifier des conditions dans une page web. Dans le code source de cet outil, nous appelons ces regroupements distincts des *patrons d'éléments*. Ainsi, un patron d'élément est une collection d'éléments DOM d'une page web, caractérisés par une relation spécifique sur laquelle certaines conditions peuvent être évaluées.

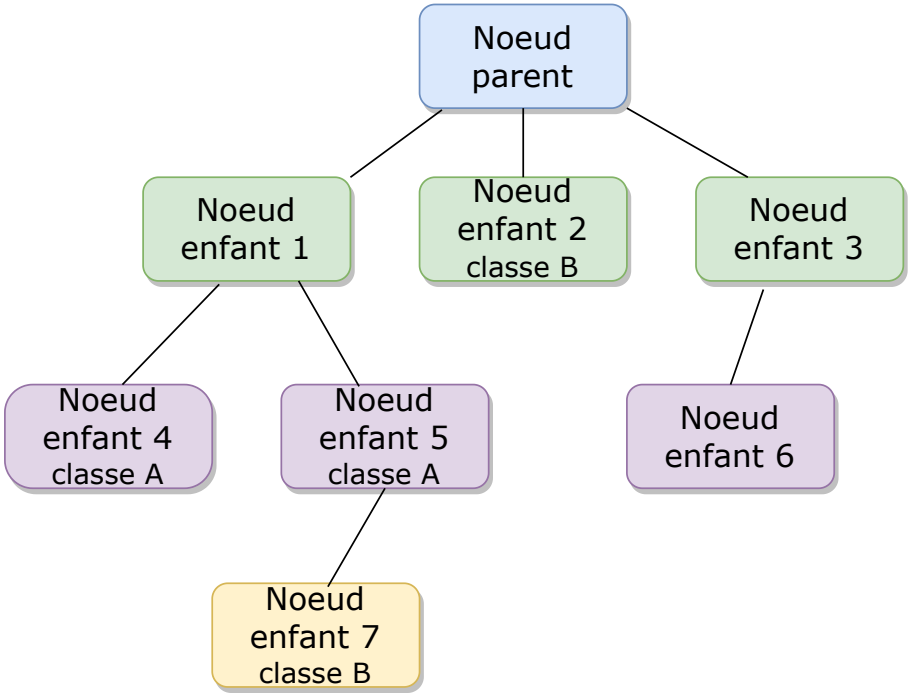



**FIGURE 4.1 : Représentation du patron d'éléments dans une arbre DOM**

Sur la figure 4.1, nous pouvons voir une représentation d'un patron d'éléments dans une arbre DOM. Par exemple, en haut, on trouve le nœud parent, qui a trois nœuds enfants, à savoir le nœud enfant 1, le nœud enfant 2 et le nœud enfant 3. Le nœud enfant 1, est également le parent de deux nœuds enfants, à savoir le nœud enfant 4 et le nœud enfant 5 —et ainsi de suite.

| Condition | Patron |
|---|---|
| Même largeur (x1, ..., xn) | Enfants d'un même parent, même classe |
| Même hauteur (x1, ..., xn) | Enfants d'un même parent, même classe |
| Alignés horizontalement (x1, ..., xn) | Enfants d'un même parent, même classe |
| Alignés verticalement (x1, ..., xn) | Enfants d'un même parent, même classe |
| Affichés gauche-droite (x1, ..., xn) | Enfants d'un même parent |
| Affichés haut-bas (x1, ..., xn) | Enfants d'un même parent |
| Contenu(x,y) | Parent-enfant |
| Plus large (x,y) | Parent-enfant |

**TABLEAU 4.1 : Assertions combinant une condition et un patron d'éléments**

Nous pouvons également observer que le nœud enfant 2 et le nœud enfant 7 appartiennent à la même classe CSS, soit la classe B. Une situation similaire se produit avec le nœud enfant 4 et le nœud enfant 5, qui font partie de la classe A.

Il est ensuite possible de combiner une condition avec un patron d'éléments ; ainsi une condition comme « les éléments ont tous la même largeur » peut s'appliquer à tous les éléments d'une même classe, ou encore à tous les enfants possédant le même parent. Chaque combinaison correspond à une assertion distincte sur le contenu d'une page web. Dans le tableau 4.1, nous montrons les diverses combinaisons de condition et de patron formant les assertions que nous considérons dans ce travail. On peut voir que ce ne sont pas toutes les combinaisons possibles qui sont admises. Ainsi, les éléments d'une même classe CSS sont probablement dispersés à plusieurs endroits de la page, et il fait donc peu de sens de chercher à affirmer que l'un d'eux soit contenu dans les autres. À l'inverse, il est fréquent qu'un élément parent dans l'arbre DOM contienne visuellement ses enfants, et cette combinaison peut donc représenter un invariant potentiel.

En réfléchissant de manière plus globale à une page web, nous devons comprendre que chaque patron d'éléments peut se produire à divers endroits à l'intérieur du même arbre DOM. Ainsi, le patron parent-enfants, dans la figure 4.1, peut être instancié à plusieurs reprises en

choisissant des nœuds différents. On doit donc trouver un moyen de distinguer ces instances de patrons. Concrètement, chaque instance du même patron peut être distinguée en examinant le chemin des éléments qui le composent.

Revenons à la figure 4.1, et expliquons comment les instances de ces patrons sont produites à partir d'un arbre DOM. Dans cet arbre, on peut utiliser la méthode DOM document.getElementsByClassName('classe') pour recherche des instances du patron d'éléments appartenant à la même classe, comme nous l'avons mentionné précédemment dans le tableau 4.1. De la même façon, on peut choisir un noeud dans l'arbre et utiliser les propriétés nextSibling et previousSibling afin de récupérer tous les éléments partageant le même parent. Pour chacun des patrons trouvés, on peut ensuite créer une assertion concrète en choisissant l'une ou l'autre des conditions s'appliquant au type de patron correspondant.

Nous illustrons ce principe au moyen d'un second scénario. Dans le même code HTML de la figure 4.2, nous avons identifié des éléments de la classe appelée « classeB », référencés comme nœud enfant 2 et 7 dans la figure 4.1. À partir du nœud enfant 2, nous avons navigué vers le nœud suivant, que nous avons nommé nœud enfant 3. Dans ce nœud, nous avons trouvé un objet où nous avons identifié l'élément web du nœud et son style. De plus, nous avons identifié son nœud enfant 6 et ses attributs de style. Avec ces informations, nous pourrions détecter la condition de contenu décrite dans le tableau 4.1. Nous présentons le résultat HTML dans la figure 4.3.

Dans la figure 4.3, nous avons délibérément inséré un débordement de contenu sur le nœud enfant 6. Cela correspond à une violation de la condition de « Contenu » dans le tableau 4.1 (la condition Contenu(x, y) est vraie lorsque y désigne une certaine boîte dans l'illustration et x est son parent dans le DOM). Supposons que précédemment nous ayons évalué une version de cette page web où cette condition était considérée comme vraie. Dans la

```html
<!DOCTYPE html>
<html lang="en">
  <style>
    .classeA {
      background-color: blue; width: 150px; height: 35px; border: 1px solid;
      margin-top: 5px;
    }
    .classeB {
      background-color: blueviolet; width: 100px; height: 20px; border: 1px solid;
      margin-top: 5px; margin-left: 10px;
    }
  </style>
  <body>
    <div
      style="
        background-color: green; width: 300px; height: 100px; padding-top: 10px;
        padding-left: 10px;
      "
    >
      <div class="classeA"></div>
      <div class="classeA">
        <div class="classeB"></div>
      </div>
    </div>
    <div class="classeB"></div>
    <div
      style="
        background-color: red; width: 300px; height: 100px; padding-top: 10px;
        padding-left: 10px; margin-top: 5px;
      "
    >
      <div
        style="
          background-color: cadetblue; width: 325px; height: 80px; margin-left: 10px;
        "
      ></div>
    </div>
  </body>
  <script>
    const classPatternA = document.getElementsByClassName('classeA');
    const siblingA = classPatternA[0].nextSibling;
    const classPatternB = document.getElementsByClassName('classeB');
    const siblingB = classPatternB[1].previousSibling;
    const siblingBB = classPatternB[1].nextSibling;
    console.log(classPatternA);
    console.log(siblingA);
    console.log(classPatternB);
    console.log(siblingB);
    console.log(siblingBB);
    console.log(siblingBB.nextSibling);
    console.log(siblingBB.nextSibling.firstElementChild);
  </script>
</html>
```

**FIGURE 4.2 : Représentation du patron d'éléments d'une arbre DOM dans une structure HTML**

version actuelle de la page, nous constatons que cette condition est maintenant fausse. Nous

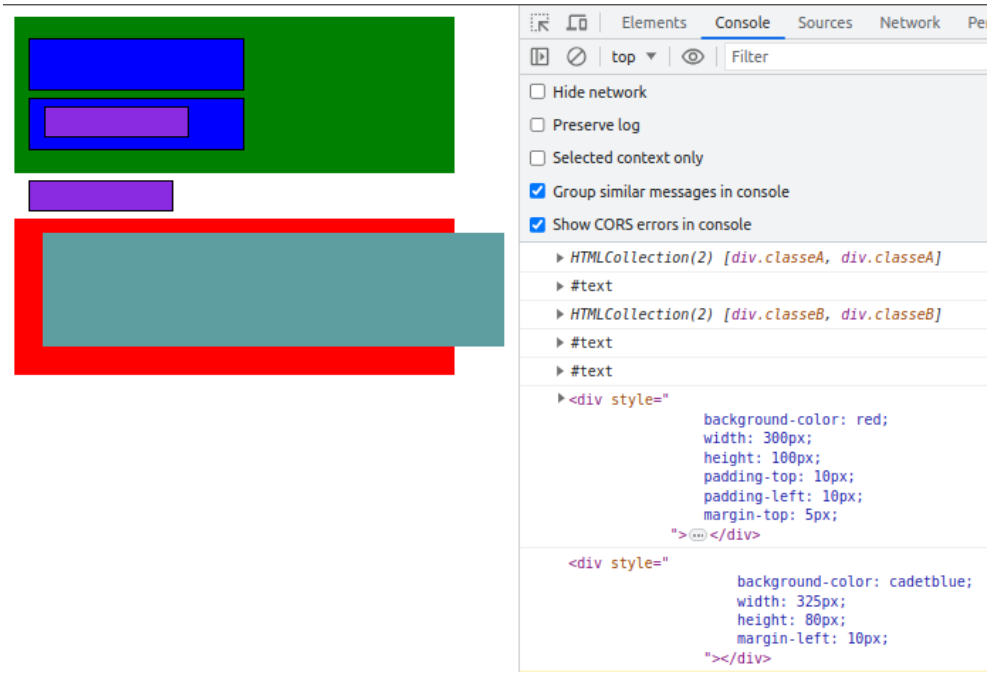



**FIGURE 4.3 : Représentation du patron d'élément d'une arbre DOM dans le navigateur**

pouvons donc émettre l'hypothèse que le débordement de contenu observé est la violation d'un invariant, c'est-à-dire d'une condition qui devrait rester vraie, mais qui est devenue fausse.

Nous appelons *invariant potentiel* une assertion correspondant à une condition précise sur une instance précise du patron d'éléments sur la page web, et telle que son évaluation renvoie la valeur vraie dans la page en question. Comme nous l'avons illustré précédemment, les nœuds enfants 1 et 3, les nœuds enfants 4 et 5, et les nœuds enfants 2 et 7 peuvent être sujets à des invariants potentiels, si l'on considère les conditions « même largeur », « même hauteur » et « alignement vertical identique », puisque ces conditions sont toutes vraies lorsqu'on les évalue avec ces patrons d'éléments dans la page.

## 4.2 APPLICATIONS AUX INVARIANTS POTENTIELS

Dans la dernière section, nous avons expliqué les types de bogues de mise en page sur lesquels nous nous concentrons, les conditions et leur connexion aux éléments du nœud dans un arbre DOM, les possibilités de navigation à travers ces nœuds, le concept de patron d'éléments, et la génération d'invariants potentiels. À ce stade, nous savons que nous pouvons détecter des conditions spécifiques sur une page web qui se transforment en invariants potentiels. Par conséquent, nous voulons montrer dans quels buts nous pouvons les utiliser.

Pour commencer, supposons que nous avons utilisé cette technique pour trouver des invariants potentiels sur une page web donnée. Ces invariants potentiels pourraient être utilisés pour évaluer de nouvelles pages web ou de nouvelles versions de la page web initiale. Cela peut se produire pendant le processus de développement d'une application web, par exemple, où il est très courant que les développeurs incrémentent des versions consécutives d'une application existante avec de nouvelles fonctionnalités, des refactorisations ou des corrections d'erreurs.

Pour illustrer différentes versions d'une application web, nous nous référons à la figure 4.4, où nous montrons un exemple d'une nouvelle fonctionnalité insérée sur une page web existante. Du côté gauche, nous pouvons voir la première version, et du côté droit la deuxième version. Dans cette dernière, le développeur a ajouté une nouvelle étiquette de texte et un nouveau champ de texte. Cependant, le nouveau champ de texte est plus étroit que le premier. De plus, le bouton a été déplacé vers le bas provoquant un débordement. Nous avons mis en évidence ces deux événements avec un cercle rouge dans la figure.

Généralement, les champs de texte de la page devraient être alignés, mais dans notre exemple, nous observons un désalignement vertical. De plus, on s'attend à ce qu'un composant comme un bouton respecte une distance minimale par rapport au bas de la page. On observe

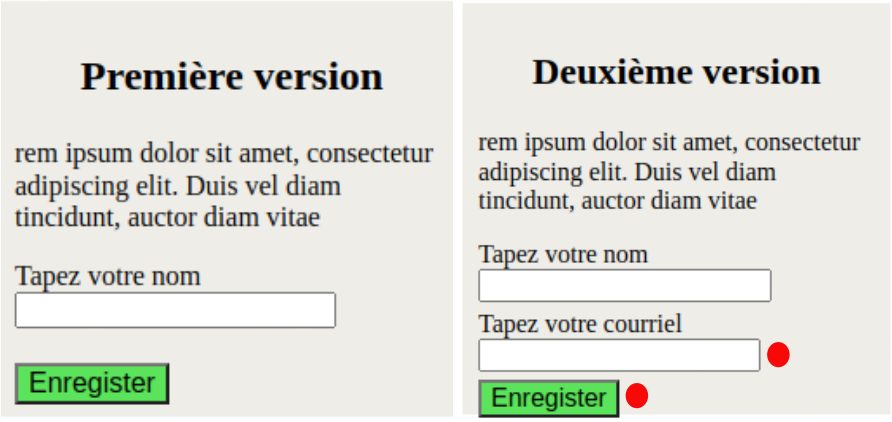



**FIGURE 4.4 : Représentation de deux versions d'une page web**

que ce n'est plus le cas dans la nouvelle version de la page. Or, imaginons que le développeur ait établi un ensemble d'assertions sur divers éléments de la page qui sont toujours demeurées vraies au fil des versions du document en développement ; on pourrait supposer que l'alignement des éléments de texte et l'espacement du bouton figurent parmi ces assertions. Si ces conditions commencent soudainement à être fausses, cela pourrait indiquer un bogue ou une régression dans la version actuelle de la page. L'avantage de cette façon de procéder est que tout ce processus peut être automatisé : il suffit de générer un grand nombre d'invariants potentiels à partir d'un échantillon initial de pages, et de déterminer lesquels deviennent faux dans des documents développés ultérieurement. Une partie de l'effort de détection des bogues peut se dérouler sans intervention manuelle.

Une autre utilisation potentielle des invariants concerne la définition d'un *gabarit*. Krause [21] affirme que les gabarits aident à la création de parties dynamiques dans les pages web, à la réduction du code générique et à éviter la répétition de balises. Ils sont présents dans de nombreux frameworks web de nos jours tels que Razor (.NET), Django (Python) et Pug (NodeJS).

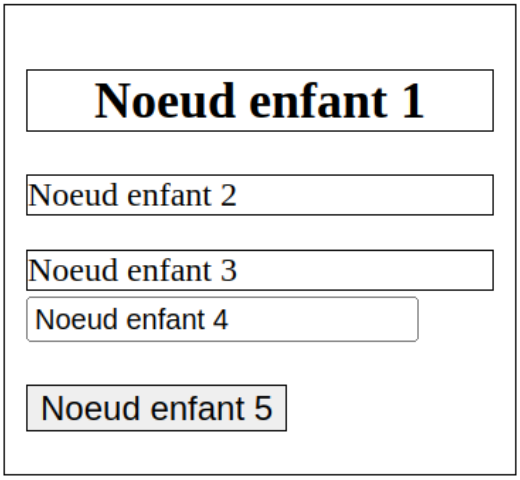



**FIGURE 4.5 : Représentation des éléments avec le nœud correspondant dans l'arbre DOM**

Un gabarit peut être vu comme une forme de modèle à suivre ; par exemple, le gabarit d'un site web peut préciser l'emplacement du menu principal, ses dimensions, l'alignement de ses éléments, la manière avec laquelle le contenu de la page est disposé à l'intérieur de cette structure, et ainsi de suite. Or, dans notre contexte, le gabarit n'est rien d'autre qu'un ensemble d'invariants impliquant des éléments précis d'un document. On peut donc affirmer que la recherche des invariants dans un ensemble de pages revient, en quelque sorte, à l'établissement d'une forme de gabarit stipulant les éléments stables de l'apparence attendue des pages d'un site. Ainsi, si une nouvelle version d'une page web enfreint le gabarit, ceci devrait normalement pouvoir être détecté par la violation d'au moins un invariant.

Nous faisons à nouveau référence à la figure 4.4. Supposons que nous puissions définir un gabarit avec la première version de la page web. En se basant sur les conditions du tableau 4.1, on pourrait donc identifier des assertions stipulant que certains éléments ont la même largeur, la même hauteur, le même alignement horizontal, etc.

Dans la figure 4.5, nous avons représenté un arbre DOM abstrait où les nœuds de la page originale ne sont désignés que par un identifiant numérique. On observe que la position

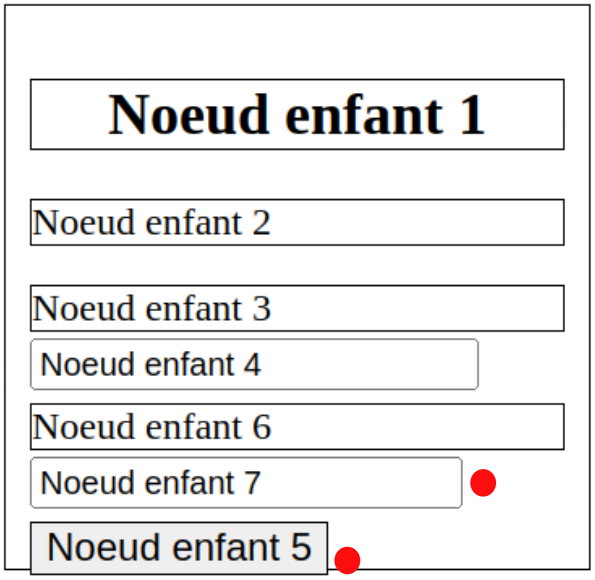



**FIGURE 4.6 : Représentation des éléments erronés avec le nœud correspondant dans l'arbre DOM**

du nœud 4 correspond au champ de texte de la page de départ, et que le nœud 5 correspond à l'élément bouton. Dans la figure 4.6, nous avons représenté la deuxième version de la page web mentionnée dans la figure 4.4, qui compte deux nœuds supplémentaires ; l'un représente le nouveau texte et l'autre le nouveau champ de texte. Suivant l'idée du gabarit que nous avons proposée précédemment, la première version serait la page web que nous avons définie comme une référence de gabarit, et la deuxième version, une page web qui devrait respecter les conditions définies par le gabarit. Ainsi, si nous comparons la deuxième page web avec la première, nous pouvons constater que certaines conditions sont devenues fausses pour les nœuds enfants 7 et 5, comme indiqué dans la figure 4.6.

Comme on peut voir, l'identification des conditions correspondant au gabarit, à partir d'une page de référence, permet en théorie la détection ultérieure de déviations de ce gabarit sur de nouvelles pages (ou de nouvelles versions de la même page). Il reste néanmoins à

automatiser le processus ; dans le prochain chapitre, nous décrirons comment nous avons utilisé ces concepts pour construire l'outil que nous avons proposé.

# CHAPITRE V
# IMPLÉMENTATION

Dans le dernier chapitre, nous avons expliqué la solution que nous proposons pour éviter aux développeurs d'avoir à écrire les spécifications de l'apparence attendue des pages web. Ainsi, dans la première section, nous avons décrit une technique pour détecter certaines conditions sur une page web. Cette détection se fait en naviguant à travers les nœuds de l'arbre DOM, pour identifier des patrons d'éléments. Ainsi, chaque condition est liée à un patron d'élément dans un arbre DOM, aboutissant à une assertion. Au moyen de ces assertions, nous pouvons identifier des invariants potentiels sur une page web.

Ensuite, nous avons décrit des applications possibles une fois un ensemble d'invariants potentiels calculé. Par exemple, ceux-ci peuvent être utilisés pour comparer différentes versions d'une page web, permettant de faire persister des conditions spécifiques tout au long d'un processus de développement. Une autre utilisation possible est liée à la notion de gabarit. Lorsque des invariants potentiels sont identifiés sur une page web donnée, cette page peut être considérée comme un modèle à suivre. Les conditions peuvent ensuite être évaluées sur d'autres pages ou d'autres versions de la page.

Dans ce chapitre, nous présentons l'outil que nous avons implémenté pour tester ces concepts. Nous allons décrire l'implémentation qui permettra, par une évaluation expérimentale, de fournir des réponses aux trois questions de recherche que nous avons établies dans le chapitre 1. Ainsi, nous décrirons les technologies que nous avons utilisées pour développer l'outil, son architecture, le processus d'extraction des invariants et la détection des bogues de mise en page au moyen de ces invariants.

## 5.1 CHOIX TECHNOLOGIQUE

Pour la mise en œuvre pratique de l'approche que nous avons proposée, nous avons opté pour certaines technologies. Nous avons développé un outil en utilisant le langage de programmation Java pour automatiser les concepts proposés. Java permet l'utilisation de bibliothèques externes. Ces bibliothèques sont généralement formées de code développé par des tiers, et qui peuvent être inclus dans un projet pour étendre les fonctionnalités du langage de base. Elles facilitent le développement en automatisant certaines tâches. Dans notre cas, nous avons utilisé des bibliothèques pour aider à l'analyse et à l'extraction des données. Les principales sont :

- **Jsoup.jar** [7] : Bibliothèque pour la manipulation de HTML ; nous l'avons utilisée uniquement pour compter les balises dans une page web.
- **Poi.jar** [8] Bibliothèque pour la manipulation de divers fichiers tels que les fichiers Office Open et Microsoft Office. Nous l'avons utilisé pour sauvegarder les données dans des fichiers de format feuille de calcul.
- **Selenium WebDriver** [9] : Bibliothèque permettant d'ouvrir et explorer une page web via son URL. Elle expose un navigateur web sous la forme d'un objet Java manipulable.
- **JavascriptExecutor** [10] : Il s'agit d'une interface de Selenium Webdriver. Son utilité consiste à exécuter des commandes JavaScript dans une page web donnée. Au moyen de cette interface, nous pouvons par exemple manipuler les couleurs des éléments dans une page web, ce qui sera utile pour fournir une rétroaction visuelle à l'utilisateur.

7. https ://jsoup.org/
8. https ://poi.apache.org/index.html
9. https ://www.selenium.dev/selenium/docs/api/java/org/openqa/selenium/WebDriver.html
10. https ://www.selenium.dev/selenium/docs/api/java/org/openqa/selenium/JavascriptExecutor.html

En développement logiciel, l'utilisation d'un système de gestion de version de code est également fondamentale. Dans notre cas, nous avons utilisé Git [11] qui est un système de gestion de version permettant de contrôler l'historique des changements dans des projets logiciels. Généralement, le système Git est utilisé avec une plateforme d'hébergement pour les fichiers et les projets. Dans notre cas, nous avons utilisé GitHub [12] pour héberger le code source de notre projet Java.

## 5.2 ARCHITECTURE

Dans cette section, nous présentons l'architecture de l'outil que nous avons développé pour mettre en œuvre les concepts que nous avons présentés dans le dernier chapitre. Pour mettre en œuvre l'approche que nous avons proposée, nous avons développé un outil avec un ensemble de classes et d'interfaces. Pour représenter cette architecture, nous nous référons à la figure 5.1, qui montre un diagramme UML avec les principales classes et leurs relations. Ces classes contiennent, entre autres, une série de méthodes pour explorer un arbre DOM.

L'outil développé est facile à configurer et peut être exécuté à l'aide d'un simple terminal de commande, ou encore en exécutant le code via l'IDE Eclipse. Le point d'entrée du programme est la classe MainIde, qui contient la méthode principale servant de point d'entrée pour l'exécution d'un programme Java. Dans MainIde, on peut renseigner l'URL de la page web à analyser, qui sera envoyée en tant que paramètre pour guider la classe Mining dans l'exécution de son rôle.

La classe Mining contrôle les instances de classes qui travaillent pour identifier des conditions sur n'importe quelle page web donnée. Elle contient également la logique pour

11. https ://git-scm.com/book/fr/v2
12. https ://github.com/

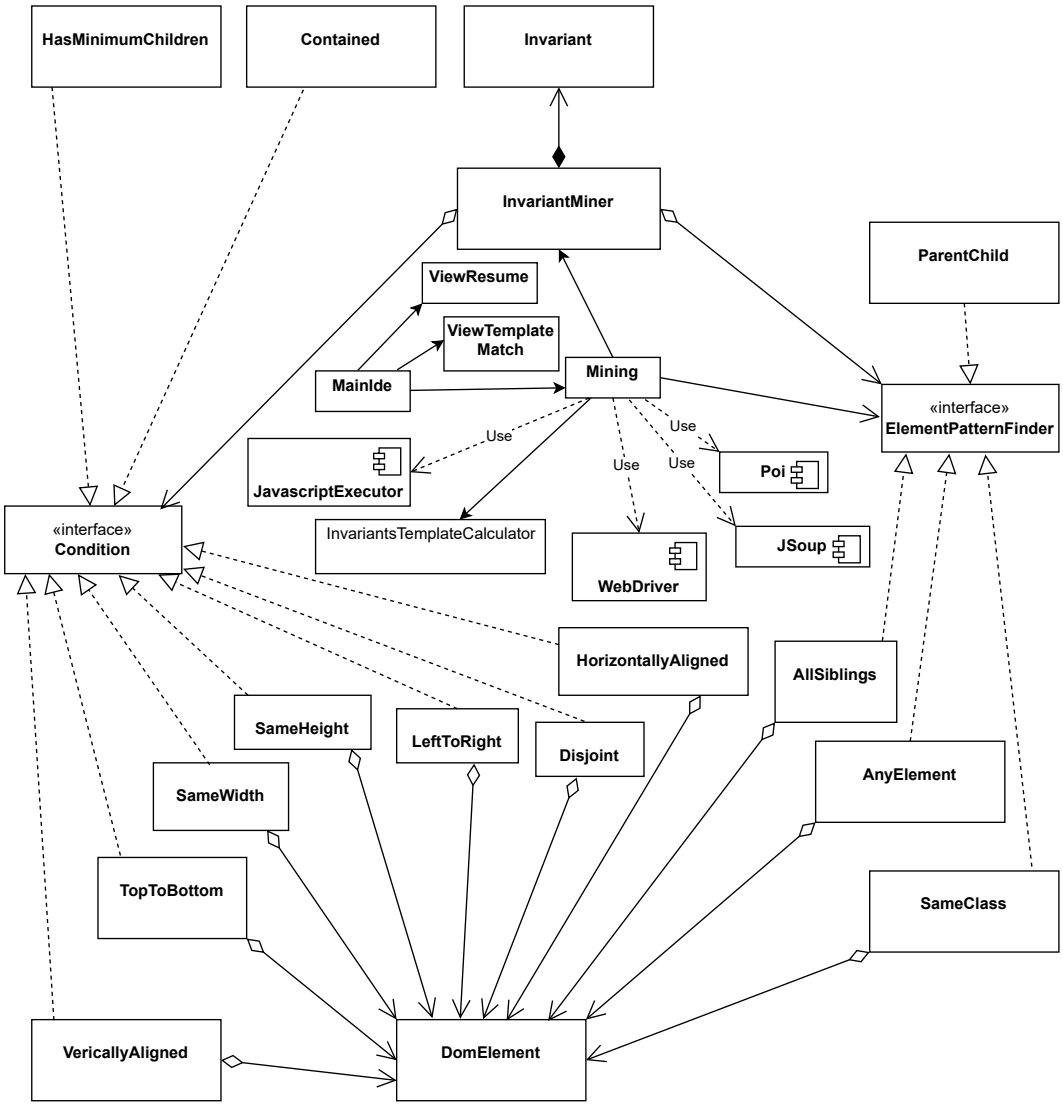



**FIGURE 5.1 : Représentation de l'architecture UML des classes de l'outil**

traiter des listes de conditions, la mise en forme des réponses et l'enregistrement des fichiers de logs. De plus, cette classe utilise fait appel à des librairies externes telles que Webdriver et Poi.

La classe Mining instancie à son tour la classe InvariantMiner, qui agrège deux interfaces, à savoir Condition et ElementPatternFinder. La réalisation de l'interface Condition est assurée par les classes Contained, HasMinimumChildren, HorizontallyAligned, Disjoint, LeftToRight,

SameHeight, SameWidth, TopToBottom et VerticallyAligned. Ces classes correspondent aux conditions génériques que nous avons présentées au chapitre précédent, et sont responsables de la détection de chaque type de condition sur une page web. Enfin, chacune de ces classes agrège la classe DomElement. Ci-dessous, nous décrirons le rôle de chacune des classes mentionnées.

- **Contained** : Effectue une comparaison vérifiant que les boîtes englobantes de toute paire d'éléments dans une liste ne débordent pas.
- **Disjoint** : Effectue une comparaison vérifiant que les boîtes englobantes de toute paire d'éléments dans une liste ne se chevauchent pas.
- **HorizontallyAligned** : Condition stipulant que tous les éléments d'une liste ont la même coordonnée X.
- **VerticallyAligned** : Condition stipulant que tous les éléments d'une liste ont la même coordonnée Y.
- **HasMinimumChildren** : Condition évaluée sur un seul élément DOM à la fois et renvoie la valeur vraie chaque fois que cet élément a au moins n enfants.
- **LeftToRight** : Condition stipulant que tous les éléments d'une liste forment une séquence croissante de coordonnées X.
- **TopToBottom** : Condition stipulant que tous les éléments d'une liste forment une séquence croissante de coordonnées Y.
- **SameHeight** : Condition stipulant que tous les éléments d'une liste ont la même hauteur.
- **SameWidth** : Condition stipulant que tous les éléments d'une liste ont la même largeur.
- **DomElement** : Désigne un nœud DOM spécifique dans une page en utilisant son chemin. Il ne conserve que le chemin d'un élément et peut donc faire référence à un emplacement commun sur plusieurs pages distinctes.

La réalisation de l'interface ElementPatternFinder est assurée par les classes ParentChild, AllSiblings, SameClass et AnyElement. Ces trois dernières agrègent également la classe DomElement. Ci-dessous, nous décrivons le rôle de chacune de ces classes.

- **ParentChild** : Correspond au patron formé d'un nœud parent et de l'un de ses enfants directs.
- **AllSiblings** : Possède une liste d'éléments correspondant à tous les frères d'un certain élément parent. Les frères sont ajoutés à la liste dans l'ordre où ils apparaissent dans le DOM (de la « gauche » vers la « droite »).
- **SameClass** : Patron contenant tous les éléments avec un nom de classe commun.
- **AnyElement** : Patron contenant un seul élément du DOM.

## 5.3 L'EXTRACTION DES INVARIANTS

Nous allons maintenant expliquer comment les invariants potentiels sont identifiés par l'outil proposé. Cette procédure est résumée dans le pseudocode de la figure 5.2.

On peut voir dans la figure 5.2 que la classe Mining contrôle l'ordre d'exécution de notre programme. Ainsi, elle contrôle le WebDriver, qui est la bibliothèque responsable de l'accès à la page web. Elle instancie également un InvariantMiner, qui coordonne la procédure d'identification des conditions. NL'InvariantMiner reçoit l'instance du WebDriver et l'URL de la page web que nous souhaitons analyser. Son rôle consiste à trouver des assertions qui sont vérifiées dans la page pointée par l'URL. Pour ce faire, on génère tout d'abord toutes les instances de patrons d'éléments que l'on peut former à partir de l'arbre DOM de la page en cours —c'est-à-dire toutes les paires parent-enfant, tous les ensembles de frères ayant un parent commun, et ainsi de suite. Une boucle « for » parcourt ensuite cet ensemble de patrons d'éléments, et crée une assertion pour chacune des conditions auxquelles ce patron peut être

```
Class Mining
    WebDriver webDriver
    InvariantMiner miner
    miner.addListCondition( condition)
    List conditions = miner(webDriver, webSitePath)
    Save conditions to log file
endclass

Class InvariantMiner
    webDriver gets webSitePath

    for elements_pattern as pattern
        condition evaluates pattern in webDriver
        if pattern contains condition
            add condition to list
        endif
    endfor

    return list
endclass
```

**FIGURE 5.2 : Pseudocode représentant le mécanisme d'exécution pour la détection des conditions**

associé. Si cette assertion est satisfaite dans la page, elle est ajoutée dans une liste . À la fin, la procédure retourne la liste des assertions trouvées sur la page web, qui représentent donc les invariants potentiels.

Par la suite, le programme utilise d'autres méthodes pour traiter et formater les assertions trouvées. Ainsi, l'ensemble des assertions peut être converti en une chaîne de caractères contenant les conditions identifiées et le patron d'éléments qui lui est associé. La figure 5.3 donne un exemple de données brutes résultant de ce processus d'exportation.

Comme on peut le voir dans la figure 5.3, l'ensemble des conditions donne une longue liste avec le nom de la condition trouvée, suivi du chemin dans l'arbre DOM des éléments du patron concerné. Lorsque sauvegardée sur un support externe, cette information brute est complémentée de données supplémentaires. Tout d'abord, nous enregistrons les horodatages

```
[vertically_aligned on //body/*[1]/*[1]/*[2]/*,
horizontally_aligned on //body/*[1]/*[1]/*[3]/*,
has_at_least 1 children on //body/*[1]/*[1], contained
on //body/*[1]/*[1]→//body/*[1]/*[1]/*[1],
vertically_aligned on //body/*[1]/*[1]/*[4]/*, contained
on //body/*[1]/*[1]→//body/*[1]/*[1]/*[5], contained
on //body/*[1]/*[1]→//body/*[1]/*[1]/*[3], contained
on //body/*[1]/*[1]/*[5]→//body/*[1]/*[1]/*[5]/*[1],
contained on //body/*[1]/*[1]/*[4]→//body/*[1]/*[1]/*[4]/
*[3], contained on //body/*[1]/*[1]/*[4]→//body/*[1]/
*[1]/*[4]/*[1], contained on //body/*[1]/*[1]→//body/
*[1]/*[1]/*[2], disjoint on //body/*[1]/*[1]/*[4]/*,
disjoint on //body/*[1]/*[1]/*[6]/*, contained on //body/
*[1]/*[1]/*[2]→//body/*[1]/*[1]/*[2]/*[1], contained
on //body/*[1]/*[1]/*[3]→//body/*[1]/*[1]/*[3]/*[2],
has_at_least 1 children on //body, disjoint on //body/
*[1]/*[1]/*, has_at_least 1 children on //body/*[1]/*[1]/
*[5], contained on //body/*[1]/*[1]/*[2]→//body/*[1]/
*[1]/*[2]/*[2], contained on //body/*[1]/*[1]/*[3]→//
body/*[1]/*[1]/*[3]/*[1], disjoint on //body/*[1]/*[1]/
*[3]/*, left_to_right on //body/*[1]/*[1]/*[6]/*,
left_to_right on //body/*[1]/*[1]/*[3]/*, contained on //
body/*[1]/*[1]/*[4]→//body/*[1]/*[1]/*[4]/*[2],
has_at_least 1 children on //body/*[1]/*[1]/*[2],
has_at_least 1 children on //body/*[1], has_at_least 1
children on //body/*[1]/*[1]/*[3], contained on //body/
*[1]/*[1]→//body/*[1]/*[1]/*[6], contained on //body/
*[1]/*[1]/*[6]→//body/*[1]/*[1]/*[6]/*[2], contained
on //body/*[1]/*[1]/*[6]→//body/*[1]/*[1]/*[6]/*[1],
top_to_bottom on //body/*[1]/*[1]/*[4]/*, contained on //
body/*[1]/*[1]→//body/*[1]/*[1]/*[4], top_to_bottom on //
body/*[1]/*[1]/*[2]/*, has_at_least 1 children on //body/
*[1]/*[1]/*[6], has_at_least 1 children on //body/*[1]/
*[1]/*[4], contained on //body/*[1]/*[1]/*[6]→//body/
*[1]/*[1]/*[6]/*[3]]
```

**FIGURE 5.3 : Chaîne de caractères contenant les conditions identifiées sur une page web**

du début et de la fin du processus, qui serviront ultérieurement à calculer la durée de chaque exploration de conditions dans la page web. Ensuite, nous enregistrons l'ensemble des conditions trouvées dans des fichiers des logs, en sauvegardant leur taille, leurs données brutes et le temps nécessaire à leur traitement. Dans la figure 5.3, on peut voir que les conditions sont extraites et ajoutées à l'ensemble dans l'ordre où elles sont trouvées dans la page. Évidemment, le même nom de condition peut être ajouté plusieurs fois à l'ensemble, mais associé à des patrons d'éléments distincts. C'est le cas de la condition vertically-aligned qui apparaît plusieurs fois

dans la figure 5.3, mais où chacune diffère par son chemin spécifique dans l'arbre DOM. On compte également le nombre de fois où chaque type de condition apparaît dans la sortie.

Dans une autre fonction, nous séparons le nom de la condition du chemin de ses éléments associés. Nous regroupons les chemins qui appartiennent à une certaine condition et les sauvegardons dans un autre fichier de log. Cette étape est également importante pour afficher un rapport présentant une vue d'ensemble du processus d'exploration, comme le montre dans la figure 5.4.

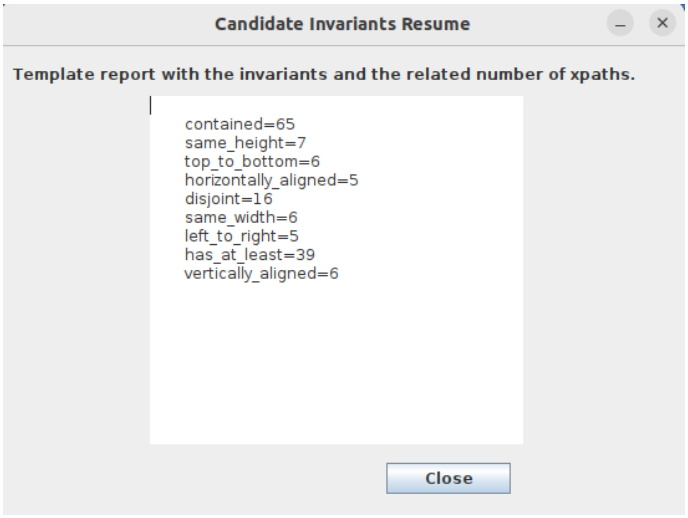



**FIGURE 5.4 : Rapport des conditions trouvées dans une page web**

Cette étape constitue la fin du processus. En somme, l'outil explore l'arbre DOM d'une page web donnée, identifie des invariants potentiels, les sauvegarde dans des fichiers de log, et enfin affiche un rapport à l'utilisateur. On dispose à ce stade d'un ensemble d'invariants potentiels trouvés sur une page web unique.

Dans la section 4.2, nous avons donné deux exemples d'utilisation de ces invariants potentiels, soit l'établissement d'un gabarit de page, ainsi que la détection de régressions dans l'évolution de la mise en page d'un site web. Par conséquent, nous avons programmé un second outil qui peut recevoir l'URL d'une autre page web, identifier ses invariants potentiels

et les comparer aux invariants de la page analysée en premier lieu. Dans un tel scénario, la page initiale est considérée comme une page de référence ou une page gabarit.

```
Class Mining
    WebDriver webDriver
    InvariantMiner miner
    miner.addListCondition( condition)
    List conditions = miner(webDriver, webSitePath)
    Save conditions to log file

    if compare two web pages
        Get saved_conditions from log file

        for saved_conditions as saved
            if conditions contains saved
                list add saved
            endif
        endfor
    endif
endclass

Class InvariantMiner
    webDriver gets webSitePath

    for elements_pattern as pattern
        condition evaluates pattern in webDriver
        if pattern contains condition
            add condition to list
        endif
    endfor

    return list
endclass
```

**FIGURE 5.5 : Pseudocode représentant le mécanisme d'exécution pour la détection des conditions et pour l'identification des invariantes**

Pour illustrer ce processus, nous avons étendu le pseudocode de la figure 5.2 en y ajoutant une nouvelle phase, comme le montre la figure 5.5. Ainsi, si nous évaluons une autre version d'une page web précédemment investiguée, nous pouvons renseigner un paramètre indiquant que l'on doit comparer les invariants potentiels des deux pages en question. Pour ce

faire, nous récupérons les conditions de la première version sauvegardée dans le fichier des logs. Ensuite, nous exécutons une boucle « for » pour vérifier si les conditions trouvées dans la deuxième version contiennent chaque condition appartenant à la première version. Si c'est le cas, elles sont ajoutées à une nouvelle liste de conditions. Cette étape élimine de l'ensemble toutes celles qui sont fausses, ne conservant que celles qui sont vraies.

Ce processus peut être répété pour un ensemble contenant un nombre arbitraire de pages. La procédure suivi est toujours la même : i) L'algorithme accède à l'arbre DOM d'une page web via son URL. ii) Il explore et identifie des conditions en naviguant à travers ses chemins. iii) Il retourne un ensemble de conditions trouvées. iv) Il classifie et organise l'ensemble des conditions. v) Il enregistre les résultats trouvés dans des fichiers des logs. Finalement, seules les assertions étant vraies dans toutes les pages analysées depuis le début sont conservées. Ce qui reste de cette opération correspond à nos invariants. Il s'agit de l'ensemble des conditions que nous considérons comme la « spécification » décrivant les pages web analysées.

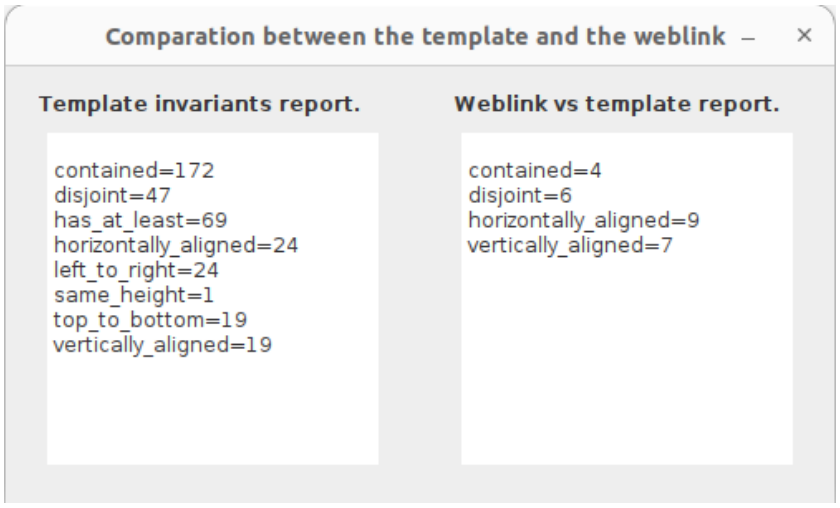



**FIGURE 5.6 : Résumé du rapport des invariants après le processus d'extraction**

Le rapport résultant de ce processus itératif est illustré dans la Figure 5.6. À gauche, on liste les catégories d'invariants trouvés dans le processus. À droite, on peut voir les conditions

qui sont classées comme fausses, c'est-à-dire celles qui sont vraies dans la première page analysée, mais pas dans toutes les autres.

À ce stade, nous savons que nous pouvons examiner des pages web distinctes à l'aide de l'outil que nous avons développé. De plus, l'outil offre un mécanisme pour identifier des invariants. Dans la section suivante, nous expliquerons comment utiliser les invariants pour identifier de potentiels bogues de mise en page dans un site web.

## 5.4 IDENTIFICATION DES BOGUES DE MISE EN PAGE

On rappelle qu'un invariant *potentiel* est une assertion spécifique qui se trouve à être vraie dans une page web donnée. Pour qu'un invariant potentiel devienne un invariant persistant, la condition qu'il représente doit rester vraie sur un ensemble de pages web. À l'aide de la figure 5.5, nous avons expliqué la distinction entre les assertions vraies et fausses. Ainsi, les assertions fausses se réfèrent à des patrons d'éléments qui devraient respecter une condition (un invariant potentiel établi précédemment), mais qui ne le font pas. Ce sont ces assertions que nous considérons comme des bogues potentiels de mise en page.

Afficher les résultats de manière plus conviviale est l'une des propositions de cette étude. Dans la dernière section, nous avons fait référence aux figures 5.4 et 5.6, comme une manière plus conviviale de présenter les résultats. Pour adopter une idée similaire pour les bogues potentiels sur une page web, nous avons développé des fonctions pour identifier et mettre en évidence ces bogues potentiels à l'utilisateur. Ainsi, après le processus de comparaison entre deux pages web représenté par la figure 5.5, nous utilisons les assertions fausses pour changer la couleur des éléments de la page web qui leur sont associés.

Pour ce faire, nous utilisons les chemins de l'arbre DOM qui appartiennent à la condition identifiée comme fausse. Par exemple, supposons que la chaîne « disjoint on //body/*[12]/*[1]/*[2]/*[2]/*[1]/*[1]/* » est une assertion fausse identifiée dans le processus.

Le chemin nous donne l'emplacement de l'élément dans l'arbre DOM. Avec ce chemin, on peut utiliser les méthodes et attributs du DOM pour atteindre l'élément et le manipuler, comme nous l'avons expliqué dans la sous-section 2.1.1. Plus précisément, avec l'aide de la bibliothèque JavascriptExecutor, nous pouvons utiliser le chemin pour identifier et manipuler la couleur de l'élément web avec CSS. Ainsi, nous récupérons chaque condition fausse, ajustons son chemin et changeons la couleur du ou des éléments DOM associés dans le navigateur qui affiche la page web.

Dans la figure 5.7, on peut voir le résultat d'un changement de couleur. Dans cette figure, nous voyons à gauche la mise en page web attendue. À droite, on peut voir une copie de la page web, dans laquelle certains textes chevauchent volontairement les images. Ce cas caractérise un bogue identifié par la catégorie Disjoint. On voit que les éléments violant l'invariant sont effectivement identifiés visuellement par un rectangle de couleur.

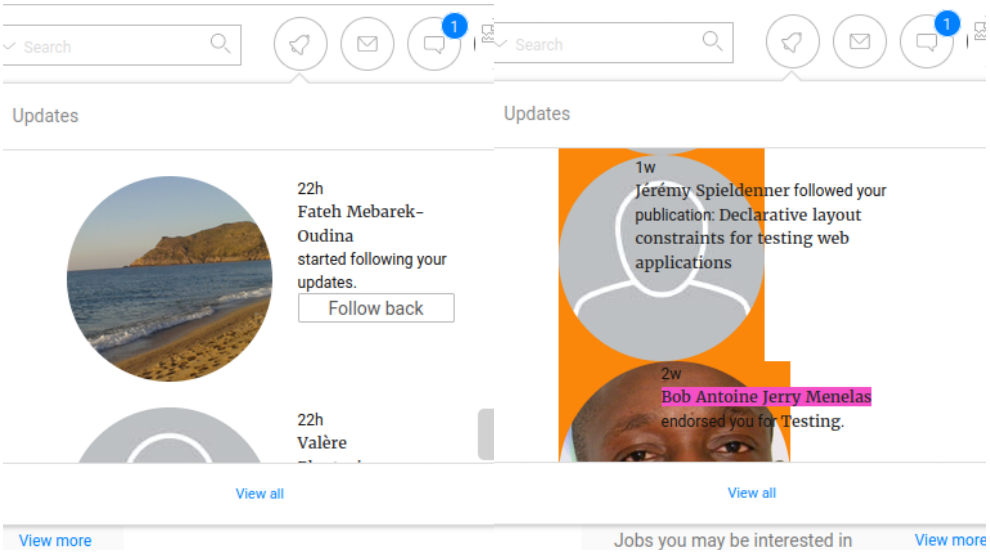



**FIGURE 5.7 : Résultat d'un changement de couleur pour montrer le chevauchement des elements ©[1]**

Dans la figure 5.8, on peut voir le résultat d'un autre changement de couleur. Dans cette figure, la mise en page web attendue est illustrée dans la partie du haut. En bas, on voit la

même page avec cette fois certains textes qui débordent de leur conteneur. Dans un des cas, on ne peut même plus voir le texte complet. Cette situation caractérise un bogue de la catégorie Contained. On voit que les deux occurrences ont été mises en surbrillance en rouge.


**FIGURE 5.8 : Résultat d'un changement de couleur pour montrer le débordement des elements ©[1]**

Dans la figure 5.9, nous montrons un dernier exemple d'une mise en évidence ; il s'agit cette fois-ci d'un élément de menu mal aligné horizontalement. Ce bogue de mise en page est pratiquement invisible à l'œil nu, mais l'outil a détecté une différence de quelques pixels par rapport aux autres éléments de menu et l'a affichée en vert. On aura compris que la couleur utilisée pour la mise en évidence dépend du type de condition violée.

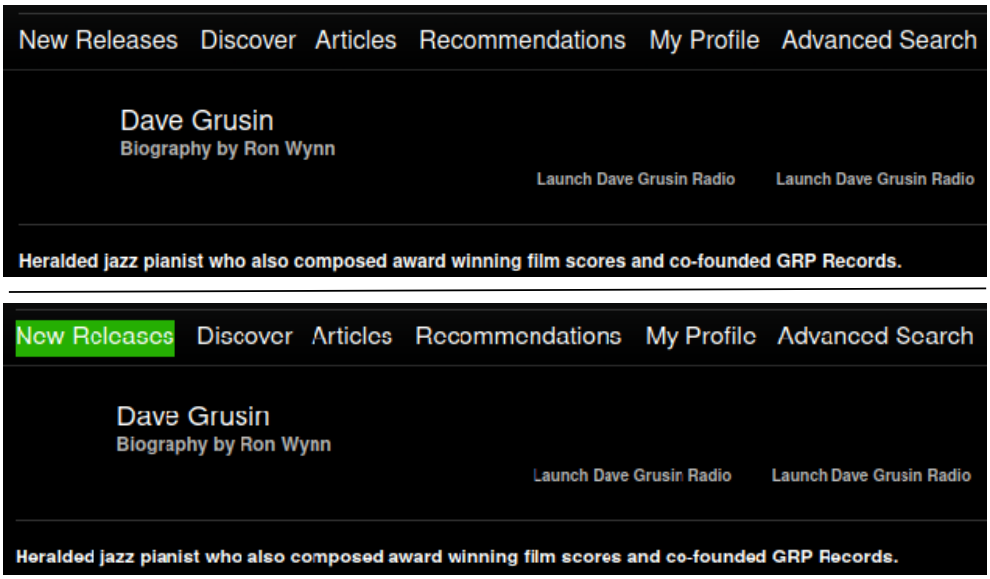



**FIGURE 5.9 : Résultat d'un changement de couleur pour montrer le désalignement horizontal d'un element**
**©[1]**

Dans cette section, nous avons donc expliqué comment nous identifions et mettons en surbrillance un bogue de mise en page sur une page web. Nous avons utilisé les conditions fausses résultant de la comparaison des pages web. Dans le prochain chapitre, nous présenterons les résultats d'une analyse expérimentale visant à répondre à nos trois questions de recherche.

# CHAPITRE VI
# APPLICATION EXPÉRIMENTALE

Dans le dernier chapitre, nous avons présenté la mise en œuvre de l'outil que nous avons proposé. Nous avons entre autres expliqué la procédure permettant de générer un grand nombre d'assertions à partir d'une page web, et d'identifier celles qui sont vraies dans cette page en particulier (formant des invariant potentiels). De plus, nous avons expliqué le processus de comparaison entre deux pages web pour valider les invariants. Pour finir, nous avons expliqué comment nous utilisons les conditions trouvées pour afficher un rapport à l'utilisateur.

Dans la dernière section, nous avons démontré comment nous identifions les bogues de mise en page en utilisant des conditions fausses. Nous avons décrit la procédure pour obtenir une condition avec son chemin d'arbre DOM afin de la formater en tant qu'une chaîne standard pour adresser un élément web. De plus, nous avons montré comment nous utilisons ce chemin pour mettre en évidence les bogues de mise en page à l'écran en changeant la couleur de l'élément web.

À ce stade, nous disposons donc d'un outil capable d'évaluer une page web, d'identifier des conditions en explorant son arbre DOM, et de comparer plusieurs versions d'une page web pour identifier d'éventuels invariants. Afin d'évaluer les capacités de l'outil et de répondre aux questions de recherche exposées dans l'introduction, nous avons appliqué un protocole expérimental que nous décrivons dans ce chapitre.

Tester l'outil développé implique de disposer d'un ensemble de pages web. Pour constituer cet ensemble, nous avons tout d'abord utilisé un script appelé Pagegen, tel que décrit par Jacquet [22]. Pagegen permet de générer des pages web synthétiques aux caractéristiques

paramétrables, nous permettant de créer des pages web de manière contrôlée, offrant ainsi une manière pratique de constituer un ensemble d'échantillons pour tester notre outil.

Cependant, pour renforcer l'expérience, nous avons également souhaité tester l'outil avec des pages web existantes se trouvant sur internet. Pour ce faire, nous avons utilisé un répertoire de pages web mentionné dans l'étude de Hallé et al. [1]. Dans cette recherche, un grand ensemble de pages web possédant des problèmes de mise en page a été identifié. Comme ce répertoire fournit un ensemble d'échantillons facilement disponible, nous avons décidé d'inclure des pages provenant de cette étude.

## 6.1 ÉCHANTILLON DE PAGES WEB SYNTHÉTIQUES

Intéressons-nous d'abord aux pages synthétiques. Comme mentionné ci-dessus, nous avons pour ce faire utilisé l'outil PageGen afin générer un ensemble de pages web. Ce logiciel est un générateur de structure d'arbre DOM aléatoire, où chaque document DOM est composé d'un ensemble de boîtes imbriquées, avec une hauteur et une largeur sélectionnées de manière aléatoire. Cet outil présente deux caractéristiques principales, que nous décrivons ci-dessous.

La première caractéristique est que Pagegen peut générer des pages web avec une disposition visuelle correcte. Ainsi, conformément à [22], il fonctionne en générant de manière récursive un élément $e$, puis en sélectionnant de manière aléatoire un nombre d'enfants. Chaque enfant peut recevoir une valeur de profondeur. Les enfants sont disposés horizontalement ou verticalement à l'intérieur de $p$. Ensuite, les dimensions de $e$ sont définies afin de former un rectangle qui englobe tous les enfants, aboutissant à un arbre de rectangles imbriqués. Cette arbre peut être exporté sous la forme d'un document HTML composé d'éléments  sans texte.

Une deuxième caractéristique de Pagegen est sa capacité à produire des pages web possédant des défauts de mise en page injectés artificiellement lors de la génération. Ainsi,

| Paramètres | Résultat |
|---|---|
| –min-depth X | Définit la profondeur minimale du document à x, où x est un nombre entier. |
| –max-depth X | Définit la profondeur maximale du document à x, où x est un nombre entier. |
| –seed x | Initialise le générateur pseudo-aléatoire avec le *seed* x. |
| –output file | Exporte le résultat vers un fichier donné. |

**TABLEAU 6.1 : Paramètres de Pagegen pour générer des pages web**

lorsque tous les éléments sont disposés à l'intérieur de leur élément parent, nous pouvons spécifier un paramètre (concrètement, une probabilité entre 0 et 1) déterminant pour chacun de ces éléments s'il doit être désaligné, sorti de son conteneur parent ou superposé à son voisin de gauche ou de droite. Le résultat final est une page possédant un nombre paramétrable de défauts de mise en page.

Nous avons établi précédemment que la scalabilité de l'outil que nous avons développé était une caractéristique importante à mesurer. Pour générer les pages web avec Pagegen, de nombreux paramètres peuvent être spécifiés, tels que la profondeur de l'arbre et le degré minimum/maximum des nœuds. Il est ainsi possible de créer un ensemble de pages contenant un nombre croissant d'éléments. Plus l'arbre DOM est grand, plus le nombre de conditions à évaluer et le nombre d'invariants potentiel est grand, ce qui peut évidemment avoir un impact sur le temps d'exécution de la procédure d'analyse. Les paramètres de Pagegen que nous avons fait varier sont donnés dans le tableau 6.1.

Nous avons organisé les paramètres pour créer des pages web en suivant la logique que les pages web devraient avoir un nombre croissant d'éléments. Pour ce faire, on ouvre une fenêtre de ligne de commandeet on saisit une commande suivant la forme suivante :

```
java -jar pagen.jar --overflow 0 --overlap 0 --misalign 0
--min-depth 1 --max-depth 2 --seed 1 --output fileName
```

Cette commande génère un document HTML sans débordement, chevauchement, ni bogue d'alignement, possédant une profondeur d'arbre entre 1 et 2. L'outil ne génère pas toujours des arbres de la même taille en raison de son caractère pseudo-aléatoire, et du fait que les paramètres –min-depth X et –max-depth X font référence à un intervalle de valeurs. Par conséquent, pour obtenir une page d'une taille donnée, on doit répéter cette commande jusqu'à obtenir une taille d'arborescence acceptable. La figure 6.1 représente la sortie que nous obtenons de la part de l'outil. Ce dernier génère une page web, qui est enregistrée dans un répertoire avec un nom de fichier se terminant par « lignxxx.html ».

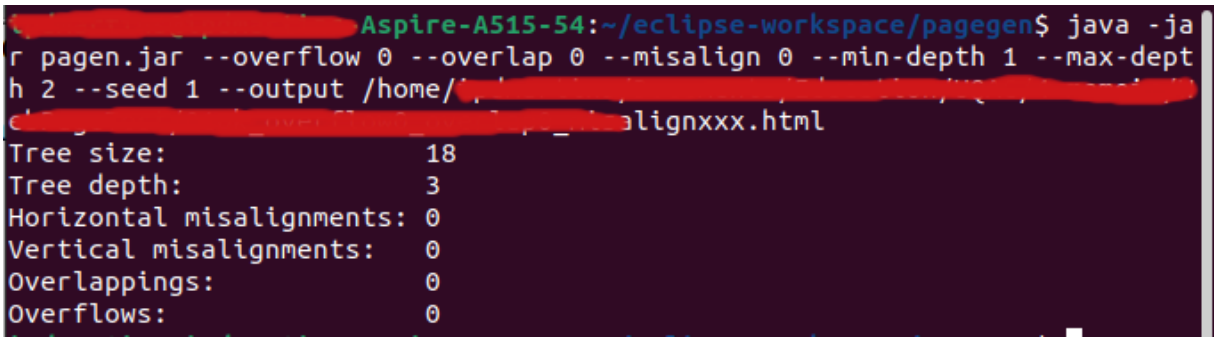



**FIGURE 6.1 : La sortie de la ligne de commande de Pagegen**

Nous avons répété la commande indiquée dans le dernier paragraphe pour générer des pages web de nouvelles tailles. Concrètement, nous avons créé des pages avec une taille d'arborescence de 18 et une profondeur d'arborescence de 3, une taille d'arborescence de 29 et une profondeur d'arborescence de 4, ainsi qu'une autre avec une taille d'arborescence de 173 et une profondeur d'arborescence de 6. À ce stade, nous avions trois ensembles de 3 pages web chacun, avec une taille croissante d'éléments et sans aucun bogue de mise en page.

Après avoir créés ces ensembles de pages, nous avons commencé à insérer artificiellement des bogues dans de nouvelles versions de ces pages. Par exemple, nous avons saisi à la ligne de commande :

```
java -jar pagen.jar --overflow 1 --overlap 0 --misalign 0
--min-depth 3 --max-depth 5 --seed 1 --output fileName
```

On obtient par cette commande une copie de la page générée dans l'exemple antérieur, mais où cette fois des éléments possèdent un débordement en raison du paramètre –overflow 1. À noter que l'on doit utiliser la même valeur de seed, autrement on obtiendra une page avec une structure différente de l'originale et qui ne permettra pas de comparaison. De la même manière, nous avons créé plus de versions avec des éléments en chevauchement et des éléments mal alignés, en utilisant –overlap 1 ou –misalign 1. La dernière page web, a été créée en combinant les 4 types de bogues de mise en page, à savoir débordement, chevauchement, alignement horizontal et vertical. Nous avons répété cette procédure pour les 3 tailles de pages web. Le tableau 6.2 présente les détails de l'échantillon que nous avons ainsi généré.

On peut voir dans le tableau que nous avons généré un total de 15 pages web, dont 3 sans bogues de mise en page et 12 avec un quelconque type de bogue. Dans la figure 6.2,

| **Référence** | **Taille** | **Prof.** | **Débordes** | **chevauchement** | **Dés. Horiz** | **Dés. Vert.** |
|---|---|---|---|---|---|---|
| NBL18A | 18 | 3 | 0 | 0 | 0 | 0 |
| BL18B | 18 | 3 | 1 | 0 | 0 | 0 |
| BL18C | 18 | 3 | 0 | 2 | 0 | 0 |
| BL18D | 18 | 3 | 0 | 0 | 1 | 1 |
| BL18E | 18 | 3 | 0 | 2 | 1 | 1 |
| NBL29A | 29 | 4 | 0 | 0 | 0 | 0 |
| BL29B | 29 | 4 | 1 | 0 | 0 | 0 |
| BL29C | 29 | 4 | 0 | 1 | 0 | 0 |
| BL29D | 29 | 4 | 0 | 0 | 1 | 3 |
| BL29E | 29 | 4 | 0 | 1 | 1 | 3 |
| NBL173A | 173 | 6 | 0 | 0 | 0 | 0 |
| BL173B | 173 | 6 | 1 | 0 | 0 | 0 |
| BL173C | 173 | 6 | 0 | 3 | 0 | 0 |
| BL173D | 173 | 6 | 0 | 0 | 16 | 19 |
| BL173E | 173 | 6 | 2 | 3 | 16 | 19 |

**TABLEAU 6.2 : Paramètres utilisés pour générer des pages web avec l'outil Pagegen**

nous présentons une comparaison de deux pages web générées avec Pagegen. Celle de gauche est sans bogue de mise en page et celle de droite possède des problèmes de débordement, de chevauchement et d'alignement. Évidemment, il s'agit de pages synthétiques n'ayant aucun sens particulier, et qui nous intéressent uniquement pour leur structure. Elles ne reflètent pas la réalité des pages web disponibles sur Internet.

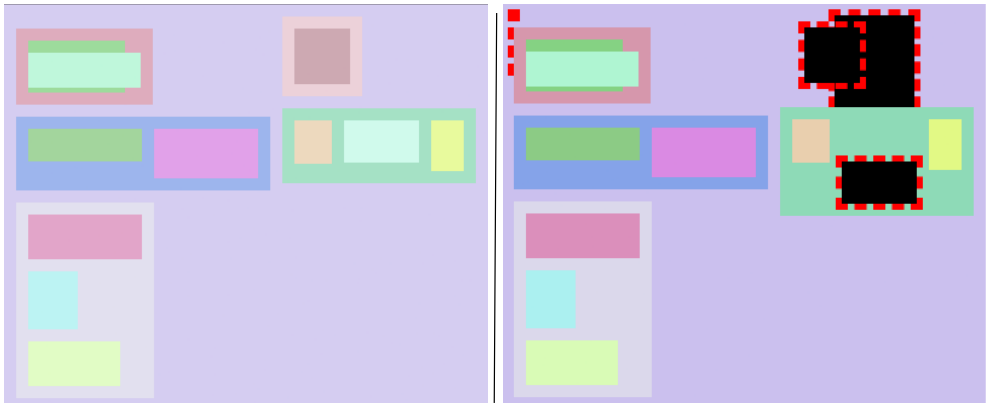


**FIGURE 6.2 : Comparaison de deux pages web générées avec Pagegen**

## 6.2 ÉCHANTILLON DE PAGES WEB EXISTANTES

Dans cette section, nous décrirons l'échantillon de pages web existantes que nous avons utilisé pour tester davantage notre outil. Ce type de page présente une grande variété d'éléments, de styles CSS, d'images, de disposition de mise en page et de taille d'arbre DOM. Nous avons décidé de constituer ce deuxième échantillon car nous avons estimé qu'il était pertinent de tester l'outil avec de vraies pages web trouvées sur Internet, sujettes à de potentiels bogues de mise en page. Par conséquent, nous croyons que cette expérience peut fournir des résultats plus riches à analyser par rapport à un ensemble limité de pages web synthétiques.

Tel que mentionné précédemment, nous avons utilisé un répertoire de pages web fourni par l'étude de Hallé et al. [1]. Dans cette recherche, on peut trouver des captures d'écran de pages web contenant différents types de bogues de mise en page, qui ont été identifiés

et classifiés. Les caractéristiques de chaque bogue ont également été consignées, et le code source complet de la page a également été sauvegardé. Ce répertoire contient une grande variété de pages web, duquel nous avons extrait uniquement celles se rapportant aux bogues considérés dans notre travail.

Nous avons expliqué dans les sections 2.2 et 4.1 que nous allions restreindre notre étude à quatre types de bogues, à savoir les éléments en chevauchement, les éléments en débordement, les éléments mal alignés verticalement et horizontalement. Ainsi, en utilisant le répertoire indiqué dans [1], nous avons retenu 11 pages possédant un bogue de chevauchement, 8 pages avec des débordements, et 2 pages avec des problèmes d'alignement, totalisant 21 pages web. Le tableau 6.2 donne la liste de ces pages et leur provenance.

On sait que l'outil que nous avons développé est capable de trouver des conditions spécifiques qui s'appliquent sur une page web. Il peut également comparer ces conditions avec d'autres conditions trouvées dans un deuxième échantillon de pages. Ainsi, pour tester les capacités de l'outil à trouver les bogues de mise en page décrits dans l'étude, nous avons analysé chacun d'eux pour comprendre le bogue de mise en page signalé. Ensuite, nous avons corrigé le bogue avec l'utilisation de CSS, afin de rendre la page web aussi proche que possible d'une mise en page standard et acceptable.

Dans les figures 5.7 et 5.9 que nous avons mentionnées précédemment, nous montrons des exemples de pages web appartenant à la liste du tableau 6.3. Du côté droit, on peut voir les échantillons pris directement de l'étude originale, et du côté gauche, les versions corrigées de ces pages que nous avons créées en modifiant les styles CSS de manière appropriée.

Comme on l'a vu, seulement 21 pages de l'étude de Hallé et al. [1] contiennent des erreurs de mise en page du type considéré dans notre étude ; en particulier, seulement deux sont des exemples d'éléments mal alignés. Nous avons donc décidé d'élargir notre ensemble

| Nom de la page |
|---|
| AllMusic |
| Slideshare-popup |
| YouTube-menu |
| Moodle-combobox |
| SSBC-footer |
| Uniform-Server |
| AgentSolo |
| ResearchGate |
| Moodle-overflow |
| Bose |
| 2 Frères |
| Evous |
| Air Canada 1 |
| Air Canada 2 |
| Moodle-overlap |
| Moodle-editor-overlap |
| SSBC-overlap |
| Time |
| RailEurope-overlap |
| Adagio2 |
| LinkedIn home |
| LiveShout |

**TABLEAU 6.3 : Références de la page tirées de [1]**

d'échantillons. Ainsi, en recherchant de nouvelles pages sur Internet, nous avons finalement ajouté dix nouvelles pages à l'ensemble original de 21. Ce nouvel ensemble est présenté dans le tableau 6.4.

Au départ, nous n'avons pas identifié de bogues de mise en page sur ces pages web. Cependant, nous voulions suivre la même logique que nous avons expliquée dans les paragraphes précédents, soit de disposer d'une page avec une mise en page correcte et la comparer avec une nouvelle version où le bogue n'est plus présent. Nous avons constaté que cette procédure était précieuse pour tester les capacités de notre outil à identifier les bogues de mise en page. Nous avons donc procédé à l'inverse de l'étude de Hallé et al., et avons manuellement inséré

| Référence | L'adresse web |
|---|---|
| web-platform-tests | https ://web-platform-tests.org/ |
| turfparadise | https ://www.turfparadise.com/owner-view.html |
| personales | http ://personales.upv.es/josilga/links.htm |
| Oxford | https ://www.ox.ac.uk/admissions |
| clinicaltrials | https ://www.clinicaltrials.gov/ct2/manage-recs/how-apply |
| las horas perdidas | https ://www.lashorasperdidas.com |
| uni-trier | https ://www.uni-trier.de |
| mysmokingshop | https ://mysmokingshop.co.uk/buy-cigarettes-mysmokingshop |
| Water | https ://water.org/about-us/why-water/ |
| eclipse | https ://www.eclipse.org/org/ |

**TABLEAU 6.4 : Liste complémentaire des pages web trouvées sur Internet**

des bogues de mise en page en utilisant du CSS dans des composants déterminés. Ainsi, nous avons ajouté 3 pages avec des bogues de chevauchement, 3 pages avec des débordements, et 4 pages avec des problèmes d'alignement. Au total, cela nous donne finalement un ensemble de 32 paires de pages boguées/corrigées pour les besoins de nos tests.

Un point important à mentionner est que, pour toutes les pages web, nous avons stocké le code source localement sur l'ordinateur où étaient effectués les tests. Cette approche est nécessaire car les pages web sur Internet peuvent subir des mises à jour. Ainsi, dans ce cas, il ne serait pas possible de tester notre solution, car les éléments web pourraient être modifiés ou les bogues de mise en page pourraient être corrigés, ce qui affecterait nos conditions.

Maintenant que les échantillons de pages utilisés pour les tests ont été décrits, nous exposerons les résultats que nous avons obtenus avec notre outil et les relierons aux questions de recherche définies en introduction de ce mémoire.

## 6.3 RÉSULTATS LIÉS À LA PREMIÈRE QUESTION DE RECHERCHE

Pour répondre à la première question de recherche, on rappelle que nous disposons tout d'abord d'une page web que nous considérons comme ayant une bonne mise en page ou

sans bogue apparent. Nous avons en plus une ou plusieurs autres versions de la même page contenant un quelconque bogue de mise en page. Nous savons également que dans l'ordre d'exécution de l'outil proposé, la page web sans bogues est analysée en premier [13], suivie des pages web boguées. Ainsi, pour faciliter la description de nos résultats, nous appellerons la première version d'une page web sans bogues le *gabarit*.

Dans cette section, nous décrivons les résultats et notre interprétation en lien avec la première question de recherche, qui est :

*Le temps de traitement nécessaire pour trouver les conditions sur une page web est-il raisonnable ?*

Dans le tableau 6.5, nous avons représenté certaines données que nous avons capturées lors du processus d'identification et d'extraction des invariants avec les pages web synthétiques. Dans la première colonne figure le nom symbolique des pages ; celles se terminant par la lettre A correspondent à une page gabarit. Celles se terminant par les lettres B, C et D sont générées en ne contenant qu'un seul type de bogue. En revanche, les pages web se terminant par la lettre E présentent une combinaison de bogues, notamment le débordement, le chevauchement, le désalignement horizontal et le désalignement vertical.

Nous avons également enregistré la taille de la page web en nombre de balises, le nombre total de conditions vraies trouvées dans la page, et le temps de traitement en secondes nécessaire pour détecter ces conditions. Ces colonnes ne concernent que le processus d'exploration d'une page web, qu'il s'agisse d'une page avec un bogue ou sans bogue. De plus, nous avons enregistré les invariants résultant de la comparaison entre une page gabarit et une page boguée. Enfin, nous avons mesuré le temps nécessaire pour effectuer cette comparaison.

Le tableau 6.5 montre que les conditions trouvées pour les gabarits NBL18A, NBL29A et NBL173A sont presque les mêmes que celles trouvées dans les versions ultérieures. Cependant,

13. Ou du moins, la première page analysée est considérée comme la référence.

le temps nécessaire pour explorer une page web avec un bogue est considérablement inférieur au temps nécessaire pour générer les conditions d'un gabarit. Ces valeurs se rapportent uniquement au processus de détection des conditions décrit dans la figure 5.2 à la section 5.3.

En examinant la dernière colonne, soit le temps de comparaison, on constate que le temps nécessaire pour détecter les conditions sur une page web boguée suivi du processus de comparaison avec les conditions extraites du gabarit (comme nous l'avons décrit dans 5.5 à la section 5.3) est inférieur. Dans tous les cas, il a fallu moins de temps pour détecter les conditions et les comparer aux conditions du gabarit que le temps nécessaire pour identifier les conditions dans un gabarit uniquement.

En ce qui concerne notre échantillon de pages web synthétiques, dans la figure 6.3, nous représentons la relation entre le nombre de balises HTML, qui sert à mesurer la taille d'une page web, et le nombre de conditions trouvées lors de chaque recherche de gabarit. Nous constatons que le nombre de conditions augmente en fonction du nombre de balises.

| Reference | Tags | Invariants | Time | Invar. match | Time match |
|---|---|---|---|---|---|
| NBL18A | 24 | 37 | 6 | - | - |
| BL18B | 24 | 36 | 2 | 2 | 5 |
| BL18C | 24 | 36 | 2 | 34 | 5 |
| BL18D | 24 | 36 | 2 | 34 | 5 |
| BL18E | 24 | 34 | 2 | 32 | 5 |
| NBL29A | 35 | 72 | 8 | - | - |
| BL29B | 35 | 71 | 3 | 69 | 7 |
| BL29C | 35 | 70 | 3 | 68 | 6 |
| BL29D | 35 | 70 | 3 | 67 | 6 |
| BL29E | 35 | 68 | 3 | 65 | 7 |
| NBL173A | 179 | 375 | 20 | - | - |
| BL173B | 179 | 372 | 16 | 370 | 19 |
| BL173C | 179 | 369 | 15 | 367 | 18 |
| BL173D | 179 | 359 | 15 | 355 | 18 |
| BL173E | 179 | 351 | 15 | 347 | 18 |

**TABLEAU 6.5 : Résultats extraits avec l'échantillon de pages web du laboratoire pour l'évaluation du temps de traitement**

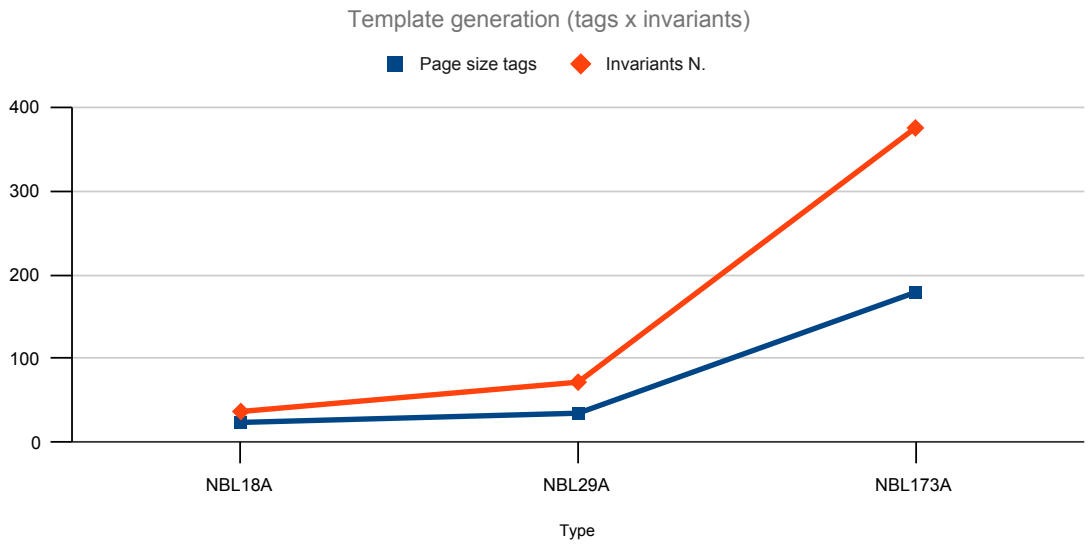



**FIGURE 6.3 : Relation entre le nombre de balises HTML et le nombre de vraies conditions trouvées dans le gabarits**

La figure 6.4 montre la relation entre le nombre d'invariants potentiels détectés et le temps d'exécution du processus de comparaison, qui suggère encore une fois une relation de proportionnalité entre ces deux quantités.

En ce qui concerne notre échantillon de pages web « réelles », on rappelle qu'on en compte 64 si nous additionnons le gabarit et la version avec des bogues. Ces pages ayant une plus grande taille que les échantillons synthétiques, nous avons choisi de ventiler les résultats en fonction du temps d'analyse, en répartissant les pages selon l'intervalle de temps pris pour que notre outil la traite. Dans le tableau 6.6, nous présentons les données relatives au processus de détection des vraies conditions sur les gabarits. La première colonne donne les intervalles de temps, qui sont divisés en tranches 60 secondes. La deuxième colonne donne le nombre de pages ayant été traitées dans l'intervalle de temps correspondant. Dans cet ensemble de pages, on liste ensuite la taille de la plus grande et de la plus petite s'y trouvant, en nombre de balises. Finalement, un autre intervalle donne le nombre le plus faible et le plus élevé de conditions vraies trouvées dans cet ensemble de pages web.

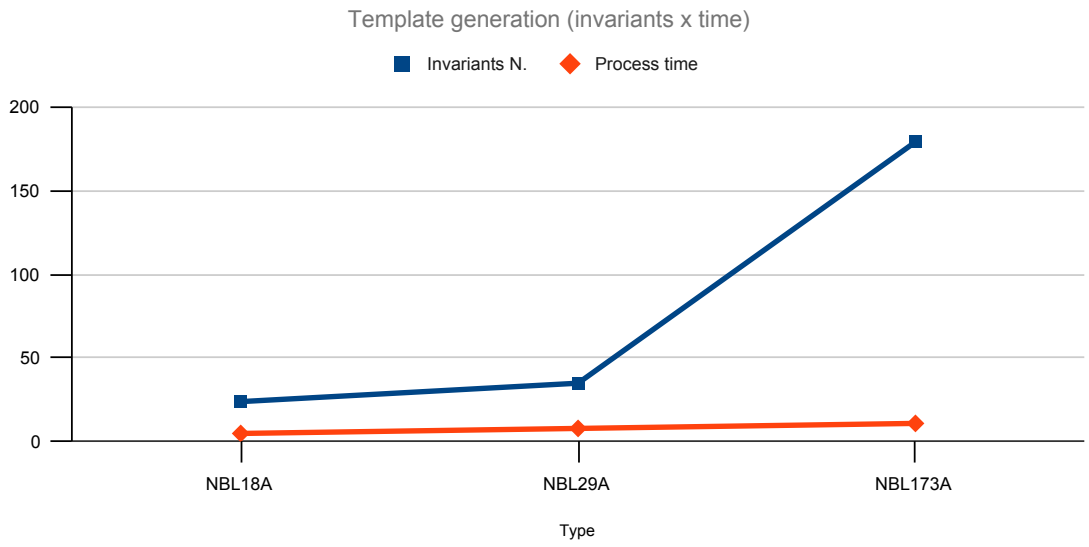



**FIGURE 6.4 : Relation entre les invariantes et le temps du processus pour comparer les pages**

Selon ce tableau, 18 pages web ont pris au plus 60 secondes pour que notre outil puisse en extraire le gabarit, ce qui représente 56,25% du total des pages web. Nous avons également constaté que cet ensemble possédant un nombre relativement faible de balises, se situant dans un intervalle de 74 à 602. Par conséquent, le nombre de conditions trouvées est plus petit, de 126 pour la plus petite page web à 1123 pour la plus grande.

D'autre part, nous avons laissé certaines lignes vides dans le tableau, car aucune page web n'est tombée dans l'intervalle de temps de traitement mentionné. La plus grande page web en nombre de balises comptait 4894 éléments, donnant lieu à 8831 conditions vraies, soit le nombre le plus élevé. Nous avons remarqué que certaines pages web, bien qu'elles soient plus petites que d'autres, ont mis plus de temps à passer à travers le processus. C'est le cas d'une page avec 408 balises, d'une autre avec 1809 et d'une autre avec 1800. À l'exception de la page avec 408 balises, qui a généré 1981 conditions vraies, nous n'avons pas trouvé de raison claire expliquant pourquoi certaines pages web plus petites prenaient plus de temps pour terminer le processus.

| Intervalle de temps | Nombre de pages | Taille en balises | Conditions vraies |
|---|---|---|---|
| 9 - 60 | 18 | 74 - 602 | 126 - 1123 |
| 61 - 120 | 4 | 804 - 1517 | 1222 - 3480 |
| 121 - 180 | 3 | 1469 - 1690 | 2779 - 3483 |
| 181 - 240 | 2 | 408 - 3322 | 1981 - 6163 |
| 241 - 300 | - | - | - |
| 301 - 360 | 2 | 1854 - 2201 | 3972 - 4428 |
| 361 - 420 | 1 | 1809 | 3559 |
| 421 - 480 | - | - | - |
| 481 - 540 | - | - | - |
| 541 - 600 | - | - | - |
| 601 - 660 | 1 | 1800 | 2977 |
| 661 - 720 | 1 | 4894 | 8831 |

**TABLEAU 6.6 : Les données relatives au processus de détection des vraies conditions sur les gabarits.**

Dans la figure 6.5, nous représentons le tableau 6.6 sous forme de graphique à bandes. Les bandes bleues représentent le temps nécessaire pour trouver les conditions vraies, qui sont représentées en vert. Les bandes rouges représentent le nombre de balises. Nous observons qu'en général, le temps nécessaire augmente en fonction du nombre de balises sur une page web. Par conséquent, le nombre de balises génère plus de conditions vraies (donc d'invariants potentiels) sur une page web. Parfois, le nombre de conditions est presque le double du nombre de balises.

Dans le tableau 6.7, nous présentons les données relatives au processus de détection des conditions vraies sur les pages web présentant un type de bogue, suivi de la comparaison avec la page gabarit. Ainsi, dans ce tableau, nous avons pris en compte le temps nécessaire pour détecter les conditions, ainsi que le temps nécessaire pour comparer deux pages web. Les résultats sont similaires au tableau précédent. Les principaux changements ont eu lieu dans les lignes 4, 5 et 6, où le regroupement des pages web a changé par rapport au tableau 6.7.

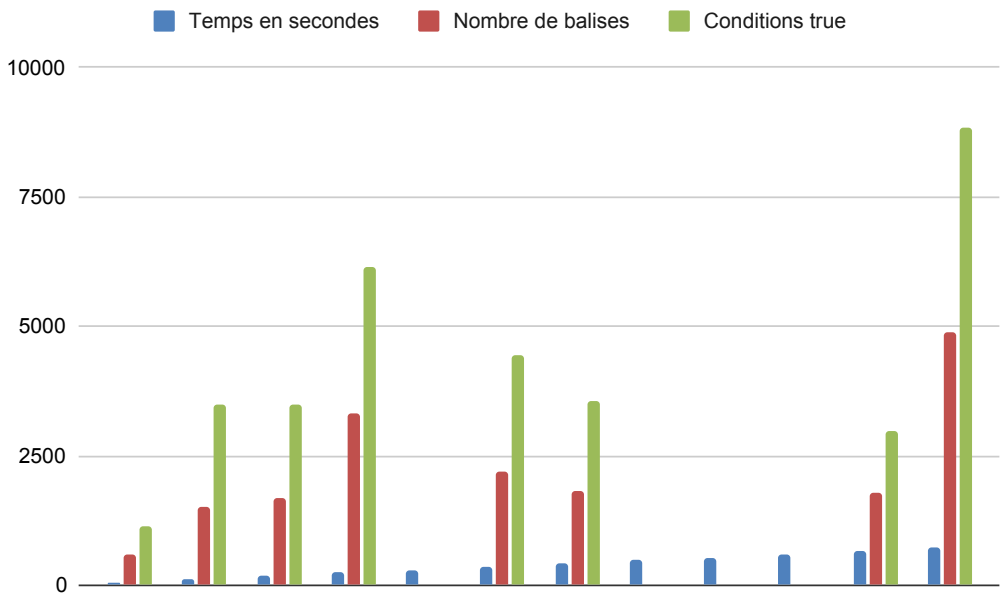



**FIGURE 6.5 : Représentation graphique du relatives au processus de détection des conditions vraies sur les gabarits**

Malgré ces changements mineurs, nous pouvons observer que le temps nécessaire pour extraire les conditions d'une version de page web avec un certain type de bogue est inférieur à celui nécessaire pour extraire les conditions d'une page gabarit. Parfois, le processus de la page gabarit était plus long et parfois l'extraction et la comparaison de la version avec bogue étaient plus longues. À noter que dans ce dernier cas, nous additionnons le temps d'extraction et de comparaison. Dans le tableau 6.8, nous présentons ces données.

En revenant à notre première question de recherche, on constate que le temps nécessaire augmente globalement en fonction de la taille de la page web. Cependant, cette tendance n'est pas stricte. Par exemple, la page d'accueil de LinkedIn avec 3321 balises a pris 219 secondes pour être analysée. De son côté, la page SlideShare-popup avec 1854 balises a pris 309 secondes dans le processus d'extraction. Finalement, une autre page web d'une taille similaire, Moodle-editor-overlap avec 1800 balises, a pris 609 secondes. Nous n'avons pas trouvé de raison claire pour ces divergences par rapport à la tendance générale.

| Intervalle de temps | Nombre de pages | Taille en balise | Conditions vraies |
|---|---|---|---|
| 6 - 60 | 18 | 74 - 602 | 126 - 1123 |
| 61 - 120 | 4 | 804 - 1517 | 1222 - 3480 |
| 121 - 180 | 3 | 1469 - 1690 | 2779 - 3480 |
| 181 - 240 | 1 | 3322 | 6166 |
| 241 - 300 | 2 | 408 - 2201 | 1979 - 4428 |
| 301 - 360 | 1 | 1854 | 3972 |
| 361 - 420 | 1 | 1809 | 3559 |
| 421 - 480 | - | - | - |
| 481 - 540 | - | - | - |
| 541 - 600 | - | - | - |
| 601 - 660 | 1 | 1800 | 2976 |
| 661 - 720 | 1 | 4894 | 8824 |

**TABLEAU 6.7 : Données relatives au processus de détection des conditions vraies sur les pages web boguées**

En combinant les résultats des pages web synthétiques et des pages web existantes, on observe que la plus grande page web possédait 4894 balises et a pris 677 secondes de temps de traitement. Cela peut sembler long ; or rappelons que le temps nécessaire pour l'analyse visuelle des pages web peut également prendre beaucoup de temps. Si l'on considère le temps nécessaire pour le codage de conditions de test par n'importe quel développeur, on peut atteindre plusieurs heures. Par conséquent, nous concluons que le temps nécessaire pour évaluer une page web, en extraire ou comparer des invariants, est raisonnable.

## 6.4 RÉSULTATS LIÉS À LA DEUXIÈME QUESTION DE RECHERCHE

Dans cette section, nous décrivons les résultats liés à notre deuxième question de recherche, qui est :

*Quelle proportion des conditions générées sont des invariants sur un échantillon de pages ?*

Pour répondre à cette deuxième question de recherche, nous avons analysé le nombre de conditions vraies détectées lors de l'extraction du gabarit. De plus, nous avons examiné les

| Nom de la page | Temps de traitement du gabarit (s) | Temps de comparaison (s) |
|---|---|---|
| Uniform-Server | 9 | 9 |
| SSBC-footer | 10 | 10 |
| personales | 14 | 15 |
| Air Canada 1 | 15 | 15 |
| SSBC-overlap | 21 | 21 |
| web-platform-tests | 23 | 22 |
| Air Canada 2 | 27 | 28 |
| Water | 29 | 30 |
| Moodle-overlap | 31 | 31 |
| LiveShout | 31 | 31 |
| uni-trier | 32 | 32 |
| AgentSolo | 33 | 34 |
| Bose | 42 | 39 |
| Oxford | 43 | 42 |
| eclipse | 45 | 43 |
| turfparadise | 56 | 52 |
| Time | 60 | 58 |
| 2 Frères | 60 | 58 |
| las horas perdidas | 61 | 64 |
| Adagio2 | 68 | 67 |
| YouTube-menu | 103 | 115 |
| Evous | 103 | 104 |
| Moodle-overflow | 126 | 131 |
| RailEurope-overlap | 151 | 165 |
| AllMusic | 167 | 161 |
| LinkedIn home | 219 | 212 |
| ResearchGate | 222 | 291 |
| clinicaltrials | 305 | 320 |
| Slideshare-popup | 309 | 278 |
| Moodle-combobox | 402 | 382 |
| Moodle-editor-overlap | 633 | 609 |
| mysmokingshop | 677 | 685 |

**TABLEAU 6.8 : Comparaison entre le temps d'extraction du gabarit par rapport à la comparaison aux versions boguées**

| Nom | N. de balises | Cond. vraies | Invar. violés | Pourcentages |
|---|---|---|---|---|
| BL18B | 24 | 36 | 1 | 97.2 |
| BL18C | 24 | 36 | 1 | 97.2 |
| BL18D | 24 | 36 | 1 | 97.2 |
| BL18E | 24 | 34 | 3 | 91.2 |
| BL29B | 35 | 71 | 1 | 98.6 |
| BL29C | 35 | 70 | 1 | 98.6 |
| BL29D | 35 | 70 | 3 | 95.7 |
| BL29E | 35 | 68 | 4 | 94.1 |
| BL173B | 179 | 372 | 3 | 99.2 |
| BL173C | 179 | 369 | 6 | 98.4 |
| BL173D | 179 | 359 | 18 | 95.0 |
| BL173E | 179 | 351 | 26 | 92.6 |

**TABLEAU 6.9 : Pourcentage de conditions qui sont invariantes dans les pages web de laboratoire**

invariants violés par la page web présentant un bogue. Pour faciliter notre analyse, nous avons créé deux tableaux avec des structures identiques, représentant chacun un ensemble distinct de pages web. Le premier tableau affiche les pages web générées de façon synthétique, tandis que le deuxième tableau concerne l'ensemble de pages web existantes. Ces tableaux incluent les informations suivantes pour chaque page web : le nom de la page web, le nombre de balises HTML, le nombre total de conditions vraies trouvées dans les pages web présentant des bogues, les invariants violés dans la page web correspondante, et le pourcentage d'invariants trouvés.

Dans le tableau 6.9, nous présentons les résultats recueillis avec les pages web générées en laboratoire. Nous avons retiré les pages web gabarit du tableau car nous n'analysons que les violations d'invariants, qui se produisent dans les pages web présentant des bogues. Nous pouvons observer une tendance où à la fois le nombre total de conditions vraies et le nombre d'invariants violés augmentent à mesure que la taille de la page web mesurée en termes de nombre de balises, augmente.

Il est noté que dans les pages web présentant un mélange de bogues, le pourcentage d'invariants violés est plus élevé par rapport aux autres pages web. En conséquence, l'intervalle d'invariants se situe entre 91,2% et 99,2%.

Le tableau 6.10 présente l'ensemble de pages web existantes. Notamment, la taille des pages, mesurée en termes du nombre de balises, est significativement plus grande dans cet ensemble par rapport à l'ensemble de pages web synthétiques. Dans ce tableau, nous avons observé des cas où le programme n'identifiait aucune violation d'invariant. D'autre part, il y a eu des cas où le programme ne détectait qu'un très petit nombre d'invariants violés, représentant une très faible fraction du nombre total d'invariants extraits. Dans la plupart des cas, le nombre d'invariants violés variait entre 1 et 3, représentant un pourcentage d'invariants entre 99,2% et 99,9%, en fonction de la taille de la page Web. Nous avons également observé certaines pages web présentant des écarts, notamment Moodle-combobox et ResearchGate, qui présentaient un nombre plus élevé d'invariants violés par rapport aux autres.

Un autre point important à souligner est la présence de conditions dans les pages web présentant des bogues qui n'appartenaient pas au gabarit. Cela a été observé sur plusieurs pages web : ResearchGate avait 12 conditions Contained, La Horas Perdidas en avait 2 Contained, Air Canada 1 en avait 1 Contained et 1 condition de désalignement vertical, Air Canada 2 en avait 1 Contained, Time en avait 1 de désalignement vertical, et Turfparadise en avait 2 Contained, 1 de désalignement horizontal et 1 de désalignement vertical. Dans le cas des pages web avec des bogues où le programme a identifié des conditions vraies qui n'appartiennent pas au gabarit, nous catégorisons ces conditions classées comme des violations d'invariants. Cette situation s'est produite dans un total de 6 pages web, représentant environ 18,7 % du total général.

| Nom de la page | N. de balises | Cond. vraies | Invar. violés | Pourcentages |
|---|---|---|---|---|
| AllMusic | 1690 | 3483 | 3 | 99.9 |
| Slideshare-popup | 1854 | 3972 | 0 | 100 |
| YouTube-menu | 1517 | 3480 | 0 | 100 |
| Moodle-combobox | 1809 | 3559 | 83 | 97.7 |
| SSBC-footer | 137 | 155 | 0 | 100 |
| Uniform-Server | 74 | 126 | 1 | 99.2 |
| AgentSolo | 395 | 698 | 3 | 99.6 |
| ResearchGate | 2201 | 4428 | 15 | 99.7 |
| Moodle-overflow | 1574 | 3157 | 2 | 99.9 |
| Water | 344 | 719 | 1 | 99.9 |
| las horas perdidas | 804 | 1441 | 0 | 100 |
| Oxford | 443 | 857 | 2 | 99.8 |
| Bose | 575 | 1029 | 2 | 99.8 |
| 2 Frères | 602 | 979 | 0 | 100 |
| Evous | 1479 | 2710 | 1 | 100 |
| Air Canada 1 | 148 | 189 | 0 | 100 |
| Air Canada 2 | 305 | 469 | 4 | 99.1 |
| Moodle-overlap | 265 | 489 | 1 | 99.8 |
| Moodle-editor-overlap | 1800 | 2977 | 1 | 100 |
| SSBC-overlap | 128 | 208 | 1 | 99.5 |
| Time | 576 | 971 | 0 | 100 |
| RailEurope-overlap | 1469 | 2779 | 0 | 100 |
| Adagio2 | 696 | 1222 | 0 | 100 |
| personales | 143 | 210 | 0 | 100 |
| clinicaltrials | 408 | 1981 | 1 | 99.9 |
| uni-trier | 400 | 823 | 2 | 99.8 |
| LinkedIn home | 3322 | 6163 | 0 | 100 |
| LiveShout | 423 | 788 | 5 | 99.4 |
| web-platform-tests | 179 | 349 | 0 | 100 |
| eclipse | 436 | 957 | 1 | 99.9 |
| turfparadise | 590 | 1123 | 0 | 100 |
| mysmokingshop | 4894 | 8831 | 7 | 99.9 |

**TABLEAU 6.10 : Pourcentages de conditions qui sont invariantes dans les pages web existantes**

## 6.5 RÉSULTATS LIÉS À LA TROISIÈME QUESTION DE RECHERCHE

Dans cette section, nous décrirons les résultats liés à notre troisième question de recherche, qui est :

*Les invariants violés correspondent-ils à des bogues d'interface ?*

Nous avons analysé visuellement les pages web pour identifier leurs bogues de mise en page en fonction de la violation d'invariants respective. Idéalement, le programme devrait identifier le même bogue de mise en page que celui que nous avons identifié par sa violation d'invariant, et terminer en le mettant en évidence avec sa couleur sur le navigateur. Cependant, nous avons obtenu des résultats différents et certains d'entre eux n'ont pas répondu à nos attentes.

En ce qui concerne tout d'abord les pages web synthétiques, nous avons commencé par extraire les invariants potentiels des pages exemptes de bogues de mise en page, afin de définir les conditions correspondant au gabarit. Ensuite, nous avons pris les quatre variantes de pages présentant un bogue de mise en page pour les comparer au gabarit. Nous avons procédé de la même manière pour les pages web de tailles différentes, à savoir avec des tailles d'arbre de 18, 29 et 173, comme nous l'avons mentionné dans le tableau 6.2 à la section 6.1.

Le tableau 6.11 montre les résultats trouvés pour les pages web générées par l'outil Pagegen. Lors de l'exécution du programme, nous avons codé des fonctions pour séparer et compter le nombre de conditions trouvées pour chaque processus d'extraction. De plus, nous avons analysé visuellement les pages générées par Pagegen pour identifier leurs bogues de mise en page. Nous avons indiqué le nombre de bogues trouvés dans les lignes nommées « Sur page ».

Une autre proposition de cette étude est d'obtenir les chemins de l'arbre DOM liés aux violations des invariants et de les identifier avec des couleurs distinctes sur les pages

| Nom | Contained | Disjoint | Horizontal | Vertical | |
|---|---|---|---|---|---|
| NBL18A | 17 | 4 | 2 | 2 | |
| BL18B | 16 | 2 | 2 | 2 | |
| | 1 | 2 | 0 | 0 | Sur page |
| | 1 | 4 | 0 | 0 | Colorée |
| BL18C | 17 | 2 | 2 | 2 | |
| | 0 | 2 | 0 | 0 | Sur page |
| | 0 | 0 | 0 | 0 | Colorée |
| BL18D | 17 | 0 | 1 | 2 | |
| | 0 | 0 | 1 | 1 | Sur page |
| | 0 | 0 | 1 | 0 | Colorée |
| BL18E | 16 | 0 | 1 | 2 | |
| | 1 | 0 | 1 | 0 | Sur page |
| | 0 | 3 | 1 | 0 | Colorée |
| NBL29A | 28 | 2 | 4 | 6 | |
| BL29B | 27 | 2 | 4 | 6 | |
| | 1 | 1 | 0 | 0 | Sur page |
| | 1 | 0 | 0 | 0 | Colorée |
| BL29C | 28 | 0 | 4 | 6 | |
| | 0 | 0 | 0 | 0 | Sur page |
| | 0 | 0 | 0 | 0 | Colorée |
| BL29D | 28 | 0 | 4 | 3 | |
| | 0 | 4 | 0 | 3 | Sur page |
| | 0 | 1 | 0 | 1 | Colorée |
| BL29E | 28 | 2 | 4 | 3 | |
| | 1 | 0 | 0 | 3 | Sur page |
| | 0 | 1 | 0 | 0 | Colorée |
| NBL173A | 172 | 1 | 24 | 19 | |
| BL173B | 169 | 0 | 24 | 19 | |
| | 3 | 1 | 0 | 0 | Sur page |
| | 3 | 0 | 0 | 0 | Colorée |
| BL173C | 171 | 3 | 24 | 19 | |
| | 1 | 1 | 0 | 0 | Sur page |
| | 1 | 2 | 0 | 0 | Colorée |
| BL173D | 172 | 1 | 15 | 12 | |
| | 0 | 1 | 12 | 13 | Sur page |
| | 0 | 0 | 3 | 5 | Colorée |
| BL173E | 168 | 0 | 15 | 12 | |
| | 4 | 1 | 11 | 14 | Sur page |
| | 1 | 0 | 3 | 3 | Colorée |

**TABLEAU 6.11 : Résultats des pages Web générées par Pagegen**

web. Ainsi, nous avons également analysé si les violations trouvées avaient une relation avec les éléments colorés sur une page web, que nous avons identifiés dans les lignes nommées « colorée », dans le tableau 6.11.

Après avoir comparé une page gabarit avec une version contenant un bogue de mise en page, le programme devrait colorer les violations des invariants. Nous avons inspecté visuellement les pages produites afin de contrôler si les éléments colorés correspondent aux bogues de mise en page que nous avions notés avant le processus. Nous avons observé que l'outil les identifie correctement dans le cas des références BL18B, BL29B, BL173B et BL173C.

En revanche, les résultats pour certaines pages web répondaient partiellement aux attentes. En ce qui concerne BL18C, nous avons noté un bogue de mise en page, mais il n'a pas été coloré par le programme. Pour BL18D, il y avait 1 désalignement horizontal généré et coloré, mais il y avait aussi 1 Vertical généré, mais non coloré par le programme. En référence à BL18E, nous avons noté trois bogues, mais un seul a été coloré. La référence BL29C avait 1 bogue de type Disjoint, mais il n'était pas coloré. La référence BL173C avait 1 invariant Contained non prévu et il était coloré par le programme comme les 5 Disjoints sur la même page web.

Pour montrer les résultats par rapport à notre échantillon de pages existant, nous avons construit trois tableaux. Dans le tableau 6.12, nous montrons uniquement les pages web pour lesquelles nous avons identifié visuellement des conditions Contained. Dans le tableau 6.13, nous montrons les pages web avec une majorité de conditions Disjoint. Enfin, dans le tableau 6.14, nous montrons celles avec des conditions de désalignement horizontal et vertical, en général.

De même que pour les pages web générées en laboratoire, nous avons analysé l'ensemble des pages web existantes pour identifier le type de bogue de mise en page qu'elles présentaient. Nous avons donc indiqué ces bogues dans les lignes appelées sur « page ». Les violations des invariants identifiées par le programme et mises en évidence sur le navigateur, sont identifiées dans les lignes appelées « colorée ». Dans la colonne appelée « type », NBP est utilisé pour désigner les pages sans bogues, et BLP pour désigner les pages avec bogues.

Nous avons effectué des tests sur 32 pages web, pour lesquelles nous avons obtenu des résultats mitigés. Pour certaines pages web, le bogue de mise en page que nous avions identifié visuellement était le même que celui identifié par le programme, qui le mettait en évidence à l'écran. C'est le cas de Uniform-Server, Water et Las horas Perdidas dans le tableau 6.12, mais aussi des pages web Bose et Moodle-editor-overlap dans le tableau 6.13 et Uni-trier dans le tableau 6.14.

Dans certains cas, le programme a coloré à l'écran, non seulement le bogue de mise en page que nous avions identifié manuellement, mais également d'autres. C'est arrivé pour Moodle-combobox, Moodle-overflow et la page web d'Oxford dans le tableau 6.12 et la page web Air Canada 2 dans le tableau 6.13. Dans ces cas, le programme a coloré des invariants supplémentaires, à savoir les désalignements vertical et horizontal et les Disjoints.

Une autre situation à laquelle nous avons été confrontés dans l'expérience concerne les pages web que nous avons identifiées visuellement comme ayant un bogue appartenant à une condition, mais que le programme a classées et colorées comme appartenant à une autre condition. Cette situation s'est produite avec Allmusic et ResearchGate, qui figurent dans le tableau 6.12. De plus, les pages web Evous, Moodle-overlap et SSBC-overlap du tableau 6.13 présentaient le même comportement. Enfin, dans le tableau 6.14, les pages web ClinicalTrials, Linkedin home, Eclipse, Turfparadise et My smoking shop présentaient la même situation.

| Reference | Type | Contained | Disjoint | Dés.Horiz. | Dés.Vert. |
|---|---|---|---|---|---|
| AllMusic | NBP | 1502 | 317 | 180 | 113 |
| | BLP | 1502 | 317 | 177 | 113 |
| | Sur page | 1 | 0 | 0 | 0 |
| | Colorée | 0 | 0 | 1 | 0 |
| Slideshare-popup | NBP | 1546 | 359 | 275 | 253 |
| | BLP | 1546 | 359 | 275 | 253 |
| | Sur page | 1 | 0 | 0 | 0 |
| | Colorée | 0 | 0 | 0 | 0 |
| YouTube-menu | NBP | 1340 | 314 | 227 | 212 |
| | BLP | 1340 | 314 | 227 | 212 |
| | Sur page | 1 | 0 | 0 | 0 |
| | Colorée | 0 | 0 | 0 | 0 |
| Moodle-combobox | NBP | 1660 | 247 | 164 | 162 |
| | BLP | 1580 | 245 | 164 | 161 |
| | Sur page | 1 | 0 | 0 | 0 |
| | Colorée | 1 | 0 | 0 | 1 |
| SSBC-footer | NBP | 65 | 16 | 5 | 6 |
| | BLP | 65 | 16 | 5 | 6 |
| | Sur page | 1 | 0 | 0 | 0 |
| | Colorée | 0 | 0 | 0 | 0 |
| Uniform-Server | NBP | 58 | 8 | 2 | 4 |
| | BLP | 57 | 8 | 2 | 4 |
| | Sur page | 1 | 0 | 0 | 0 |
| | Colorée | 1 | 0 | 0 | 0 |
| AgentSolo | NBP | 329 | 42 | 20 | 16 |
| | BLP | 329 | 42 | 19 | 16 |
| | Sur page | 1 | 0 | 0 | 0 |
| | Colorée | 0 | 0 | 0 | 0 |
| ResearchGate | NBP | 1955 | 418 | 165 | 168 |
| | BLP | 1967 | 405 | 165 | 166 |
| | Sur page | 1 | 0 | 0 | 0 |
| | Colorée | 0 | 13 | 0 | 2 |
| Moodle-overflow | NBP | 1442 | 231 | 123 | 118 |
| | BLP | 1441 | 230 | 123 | 118 |
| | Sur page | 1 | 0 | 0 | 0 |
| | Colorée | 1 | 1 | 0 | 0 |
| Water | NBP | 306 | 61 | 9 | 42 |
| | BLP | 305 | 61 | 9 | 42 |
| | Sur page | 1 | 0 | 0 | 0 |
| | Colorée | 1 | 0 | 0 | 0 |
| las horas perdidas | NBP | 601 | 162 | 34 | 38 |
| | BLP | 603 | 162 | 34 | 38 |
| | Sur page | 1 | 0 | 0 | 0 |
| | Colorée | 1 | 0 | 0 | 0 |



**TABLEAU 6.12 : Résultats des pages web existantes sur les conditions de type Contained**

| Reference | Type | Contained | Disjoint | Dés.Horiz. | Dés.Vert. |
|---|---|---|---|---|---|
| Oxford | NBP | 394 | 57 | 27 | 40 |
| | BLP | 393 | 57 | 27 | 39 |
| | Sur page | 1 | 0 | 0 | 0 |
| | Colorée | 1 | 0 | 0 | 1 |
| Bose | NBP | 412 | 79 | 24 | 43 |
| | BLP | 411 | 78 | 24 | 43 |
| | Sur page | 0 | 1 | 0 | 0 |
| | Colorée | 0 | 1 | 0 | 0 |
| 2 Frères | NBP | 472 | 46 | 33 | 24 |
| | BLP | 472 | 46 | 33 | 24 |
| | Sur page | 0 | 1 | 0 | 0 |
| | Colorée | 0 | 0 | 0 | 0 |
| Evous | NBP | 1237 | 121 | 88 | 166 |
| | BLP | 1236 | 121 | 88 | 166 |
| | Sur page | 0 | 1 | 0 | 0 |
| | Colorée | 1 | 0 | 0 | 0 |
| Air Canada 1 | NBP | 102 | 14 | 2 | 4 |
| | BLP | 103 | 14 | 2 | 5 |
| | Sur page | 0 | 1 | 0 | 0 |
| | Colorée | 0 | 0 | 0 | 0 |
| Air Canada 2 | NBP | 215 | 50 | 25 | 12 |
| | BLP | 216 | 48 | 24 | 12 |
| | Sur page | 0 | 1 | 0 | 0 |
| | Colorée | 0 | 1 | 1 | 0 |
| Moodle-overlap | NBP | 229 | 36 | 15 | 23 |
| | BLP | 229 | 36 | 15 | 22 |
| | Sur page | 0 | 1 | 0 | 0 |
| | Colorée | 0 | 0 | 0 | 1 |
| Moodle-editor-overlap | NBP | 1436 | 207 | 138 | 123 |
| | BLP | 1436 | 206 | 138 | 123 |
| | Sur page | 0 | 1 | 0 | 0 |
| | Colorée | 0 | 1 | 0 | 0 |
| SSBC-overlap | NBP | 109 | 17 | 5 | 8 |
| | BLP | 108 | 17 | 5 | 8 |
| | Sur page | 0 | 1 | 0 | 0 |
| | Colorée | 1 | 0 | 0 | 0 |
| Time | NBP | 437 | 76 | 24 | 54 |
| | BLP | 437 | 76 | 24 | 55 |
| | Sur page | 0 | 1 | 0 | 0 |
| | Colorée | 0 | 0 | 0 | 0 |
| RailEurope-overlap | NBP | 1253 | 264 | 112 | 158 |
| | BLP | 1253 | 264 | 112 | 158 |
| | Sur page | 0 | 1 | 0 | 0 |
| | Colorée | 0 | 0 | 0 | 0 |



**TABLEAU 6.13 : Résultats des pages web existantes sur les conditions de type Disjoint**

| Reference | Type | Contained | Disjoint | Dés.Horiz. | Dés.Vert. |
|---|---|---|---|---|---|
| Adagio2 | NBP | 576 | 105 | 62 | 67 |
| | BLP | 576 | 105 | 62 | 67 |
| | Sur page | 0 | 1 | 0 | 0 |
| | Colorée | 0 | 0 | 0 | 0 |
| personales | NBP | 108 | 6 | 0 | 4 |
| | BLP | 108 | 6 | 0 | 4 |
| | Sur page | 0 | 1 | 0 | 0 |
| | Colorée | 0 | 0 | 0 | 0 |
| clinicaltrials | NBP | 846 | 247 | 102 | 104 |
| | BLP | 845 | 247 | 102 | 104 |
| | Sur page | 0 | 1 | 0 | 0 |
| | Colorée | 1 | 0 | 0 | 0 |
| uni-trier | NBP | 331 | 90 | 38 | 46 |
| | BLP | 331 | 88 | 38 | 46 |
| | Sur page | 0 | 1 | 0 | 0 |
| | Colorée | 0 | 1 | 0 | 0 |
| LinkedIn home | NBP | 2500 | 605 | 245 | 283 |
| | BLP | 2500 | 605 | 245 | 283 |
| | Sur page | 13 | 0 | 1 | 0 |
| | Colorée | 0 | 1 | 0 | 0 |
| LiveShout | NBP | 308 | 76 | 37 | 42 |
| | BLP | 303 | 76 | 37 | 42 |
| | Sur page | 0 | 0 | 1 | 1 |
| | Colorée | 0 | 0 | 0 | 0 |
| eclipse | NBP | 399 | 83 | 53 | 57 |
| | BLP | 398 | 83 | 53 | 57 |
| | Sur page | 0 | 0 | 1 | 0 |
| | Colorée | 1 | 0 | 0 | 0 |
| web-platform-tests | NBP | 157 | 27 | 6 | 16 |
| | BLP | 157 | 27 | 6 | 16 |
| | Sur page | 0 | 0 | 0 | 1 |
| | Colorée | 0 | 0 | 0 | 0 |
| turfparadise | NBP | 540 | 63 | 33 | 41 |
| | BLP | 542 | 63 | 34 | 42 |
| | Sur page | 0 | 0 | 0 | 1 |
| | Colorée | 1 | 0 | 0 | 0 |
| mysmokingshop | NBP | 4648 | 305 | 162 | 156 |
| | BLP | 4642 | 304 | 162 | 156 |
| | Sur page | 0 | 0 | 0 | 1 |
| | Colorée | 1 | 0 | 0 | 0 |

**TABLEAU 6.14 : Résultats des pages web existantes sur les conditions de désalignement**

Les tests ont mis en évidence des cas où le programme n'a signalé ni coloré aucune violation d'invariant. C'est le cas des pages web Slideshare-popup, Youtube-menu, SSBC-footer et AgentSolo du tableau 6.12, les pages web 2Frères, Air Canada 1, Time, RailEurope-overlap, Adagio2 et Personales du tableau 6.13, et les pages web Liveshout et Web-platform-tests du tableau 6.14. Toutes les pages web mentionnées n'avaient aucun élément web mis en évidence dans le navigateur.

Nous avons résumé nos résultats dans le tableau 6.15. Nous avons observé des résultats partiellement similaires dans les deux échantillons de pages web. Parmi les pages synthétiques, seules 4 sur 12 répondaient au scénario idéal (le programme identifie le bogue et uniquement celui-ci), tandis que dans l'ensemble de pages du monde réel, la proportion est de 6 sur 32. Dans les deux ensembles, on a observé des cas où le programme a identifié plus d'invariants que nous ne l'avions initialement anticipé, les mettant ainsi en évidence dans le navigateur. Cette situation s'est produite avec une page synthétique et quatre pages existantes.

Une situation que nous pourrions considérer comme positive est que, pour dix pages existantes, le programme a classé les invariants différemment de notre évaluation, mais a néanmoins identifié le bogue en question. Cela signifie que nous avons peut-être commis une erreur en classifiant certains bogues de mise en page lors de notre inspection visuelle.

En ce qui concerne la précision du programme pour mettre en évidence correctement les composants web classés comme bogués, nous avons observé les résultats suivants. Pour les pages synthétiques, le programme a réussi à mettre en évidence tous les bogues de mise en page identifiés dans 5 cas, atteignant une mise en évidence complète. Dans 2 cas, le programme a partiellement mis en évidence les bogues préalablement identifiés, ce qui donne un taux de précision de 58,3%. Pour les pages web existantes, le programme a réussi à mettre en évidence les bogues de mise en page identifiés dans 20 cas, donnant un taux de précision de 62,5%.

| Résultat | Pages de laboratoire | Pages existantes |
|---|---|---|
| Le programme a identifié les invariants et les a mis en évidence | 4 | 6 |
| Le programme a identifié les invariants mais ne les a pas mis en évidence | 1 | 0 |
| Le programme a partiellement identifié les invariants et les a partiellement mis en évidence | 2 | 0 |
| Le programme a identifié plus d'invariants et ne les a pas mis en évidence | 4 | 0 |
| Le programme a identifié plus d'invariants et les a mis en évidence | 1 | 4 |
| Le programme a classé les invariants différemment et les a mis en évidence | 0 | 10 |
| Le programme n'a identifié aucun invariant | 0 | 12 |

**TABLEAU 6.15 : Résumé des résultats trouvés dans l'expérience comparant les invariants violés aux bogues se trouvant réellement dans une page**

En résumé, nous concluons que le programme est partiellement capable de mettre en évidence les violations d'invariants identifiées à l'aide du chemin de l'arbre DOM. Bien qu'il ait réussi à mettre en évidence tous les éléments concernés dans certains cas, on a également vu des cas où seule une mise en évidence partielle a été obtenue. Ces résultats donnent un aperçu des performances du programme dans l'identification et la mise en évidence précises des invariants, tant dans les pages synthétiques que dans les pages web existantes.

La raison pour laquelle le programme n'a pas atteint une correspondance de 100 pour cent avec le scénario idéal pourrait potentiellement être attribuée à une détection erronée. Par exemple, dans certaines pages web que nous avons classées comme ayant un type spécifique de bogue, l'algorithme n'a rapporté aucune violation d'invariant, même si le bogue était manifestement visible à l'oeil. Cette situation peut survenir en raison des chemins obtenus à partir de l'arbre DOM que l'algorithme a extraite. On rappelle qu'un invariant spécifie de

manière exacte le chemin dans l'arbre DOM menant à chacun des éléments du patron qui le définit. Si ces éléments changent de position relative entre la version gabarit et la version boguée d'une page, l'algorithme peut échouer à identifier une violation. Il est à noter à cet égard que nous avons pris des précautions pour ne pas introduire de nouvelles balises HTML lors de la création de copies des pages web utilisées pour générer des gabarits, évitant ainsi de modifier la position relative des éléments. Ainsi, seul le code CSS a été modifié dans ces pages web.

Un autre point important à considérer est la présence de faux positifs. Comme nous l'avons mentionné précédemment, nous avons identifié visuellement des bogues dans les pages web. Cependant, le programme a indiqué des bogues de mise en page qui n'étaient pas visuellement apparents. Cette divergence peut être attribuée au programme extrayant de manière erronée les chemins DOM pour ces éléments. Étant donné que les pages web boguées sont des copies exactes des pages web de gabarit, la seule différence étant le CSS appliqué à l'élément bogué, on peut s'attendre à ce que la structure HTML et le style CSS restent les mêmes entre les deux versions. Dans ces conditions, il est probable que le programme ait identifié de manière erronée ces éléments comme contenant des bogues en raison de sa dépendance au processus d'extraction. Malgré la structure HTML et le style CSS identiques, le programme a incorrectement déduit la présence de bogues de mise en page dans ces cas.

En somme, nous pouvons observer que le programme a réussi identifier des bogues de mise en page dans tous les cas de pages synthétiques. En revanche, pour les pages web existantes, les bogues de mise en page ont été détectés dans 60 % des pages web testées. Après avoir analysé ces résultats, nous comprenons que le programme a partiellement réussi à établir un patron pour identifier la corrélation entre les bogues de mise en page classés et les violations d'invariants.

# CHAPITRE VII
# CONCLUSION

Dans le dernier chapitre, nous avons exposé l'application de notre expérience. Nous avons expliqué comment nous avons constitué l'échantillon de pages web que nous avons utilisé pour l'expérience. Ensuite, nous avons évalué chacune de nos trois questions de recherche et les avons reliées aux résultats que nous avons obtenus. Dans ce chapitre, nous présentons nos conclusions, limitations et travaux futurs.

## 7.1 CONCLUSION

Ainsi qu'on a pu le voir, les pages web sont sujettes à des bogues de mise en page, et leur détection manuelle s'avère souvent une tâche laborieuse. Dans ce mémoire, nous avons exposé la solution que nous proposons à ce problème, les principes qui y sont liés, notre objectif avec cette recherche, notre hypothèse et nos questions de recherche. L'idée principale, on l'a vu, consiste à automatiser le processus de test en identifiant des conditions sur les pages web, préservant ainsi les développeurs de tâches de codage intensives.

Nous avons donc proposé une approche capable d'abord de trouver automatiquement des collection d'éléments DOM de pages web. Ces collections se caractérisent par une relation spécifique entre certains éléments, sur laquelle certaines conditions peuvent être évaluées. Nous avons donc développé un outil pour explorer l'arbre DOM de n'importe quelle page web, détecter des conditions vraies et fournir les chemins des éléments liés à ces conditions —ce que l'on a appelé des invariants potentiels.

Pour tester notre approche, nous avons extrait des invariants potentiels d'un ensemble de pages, et ensuite comparé ces invariants à ceux d'une autre page distincte, afin de distinguer

celles qui sont restent vraies de celles qui deviennent fausses. Les conditions qui sont restées vraies après le processus de comparaison entre les pages web ont été classées comme des invariants. Les conditions qui se sont révélées fausses ont été classées comme des violations d'invariants.

Nous avons utilisé deux ensembles d'échantillons de pages web pour tester notre approche, soit un ensemble de pages web générées de manière synthétique et un autre ensemble de pages web existantes provenant d'Internet. Nous avons d'abord identifié ce que nous avons défini comme des pages web de gabarit, exemptes de bogues de mise en page. Nous avons ensuite comparé ces modèles avec des pages web contenant n'importe quel type de bogue de mise en page. Malgré le fait que l'outil que nous avons développé n'atteigne pas le résultat idéal (soit l'identification du bogue et uniquement de celui-ci) dans de nombreuses circonstances, nous avons trouvé un nombre considérable de pages web où il a été capable d'identifier et de mettre en évidence des violations d'invariants, qui étaient généralement liées à des bogues de mise en page.

## 7.2 LIMITES

Au cours de nos recherches, nous avons rencontré certains défis qui pourraient potentiellement impacter les résultats obtenus.

### 7.2.1 MANIPULATION CSS

Le processus d'identification manuelle des bogues de mise en page dans l'ensemble des pages web existantes, suivi de leur manipulation à l'aide de CSS, peut présenter certains problèmes. En effet, la manipulation CSS effectuée sur les page web peut entraîner des modifications visuelles, notamment des désalignements ou des chevauchements d'éléments. Cependant, l'interprétation de ces modifications visuelles par l'algorithme utilisé dans l'étude

peut différer de notre compréhension subjective des changements. Comme discuté dans la section 6.5, la classification de ces modifications visuelles par l'algorithme peut ne pas correspondre à notre perception, ce qui pourrait potentiellement influencer les résultats.

À ce stade, il est important de reconnaître que nous n'avons pas pleinement évalué dans quelle mesure les changements CSS ont influencé les résultats concernés. Une enquête et une analyse supplémentaires seraient nécessaires pour déterminer l'impact précis de la manipulation CSS sur les résultats de recherche.

### 7.2.2 VARIABILITÉ DES PAGES WEB

De nos jours, nous disposons d'une large gamme d'appareils pour accéder à Internet. Ces appareils comprennent des smartphones, des ordinateurs, des tablettes, et bien d'autres. Par conséquent, les applications Web peuvent être accessibles sur différentes tailles d'écran et résolutions en fonction de l'appareil utilisé. Il est important de prendre en compte le fait qu'une même application web peut donc présenter des résultats visuels différents lorsqu'elle est utilisée sur différents appareils. Cette variation des tailles d'écran peut affecter la mise en page et la présentation de l'application, ce qui peut entraîner des différences dans l'identification des invariants de mise en page.

En outre, l'utilisation de différentes technologies, telles que le HTML pur par rapport aux frameworks JavaScript, peut également avoir un impact sur le rendu et le comportement des pages web. Bien que notre approche ait été été testée sur un seul ordinateur, il est important de reconnaître que des tests sur une plus large gamme d'appareils et de technologies pourraient donner des résultats différents. L'identification des invariants peut varier en fonction de l'environnement spécifique et des technologies utilisées.

### 7.2.3 TYPES D'INVARIANTS

Nous avons concentré nos efforts sur un nombre limité d'invariants, à savoir les conditions de type Contenu, Disjoint, ainsi que désalignement horizontal et vertical. Nous avons adopté cette stratégie en raison des caractéristiques des pages web dont nous disposions, qui contenaient des bugs de ce type. Cependant, le programme que nous avons créé est capable d'identifier d'autres invariants, que nous avons nommés « Has at least one child », « Left to right », « Same height », « Same width » et « Top to bottom ». Chaque invariant a ses propres caractéristiques et représente des bogues potentiels pouvant se produire dans une page web.

De plus, nous pensons que de nombreux autres types d'invariants peuvent être explorés en raison de la variété de bogues de mise en page qui peuvent être présents dans une page web. Par conséquent, l'exploration de différents types d'invariants peut conduire à des résultats différents, ce qui pourrait constituer une limite de notre approche.

### 7.2.4 INTERACTIONS AVEC L'UTILISATEUR

Nous avons limité nos expériences aux pages web dites « statiques ». Les pages analysées étaient même copiées directement sur notre machine locale afin d'éviter leurs mise à jour et les modifications de mise en page qui pourraient survenir si on accédait toujours à la plus récente version de la page disponible en ligne. Par conséquent, nous n'avons pas testé les interactions que l'utilisateur peut avoir dans une page web, ce qui caractérisait une expérience dynamique, similaire à celles que nous avons trouvées dans certaines études connexes.

Ces interactions entraînent typiquement l'apparition de nouveaux éléments web dans le navigateur et, par conséquent, de nouveaux invariants à générer et à suivre au fil des pages explorées. Ce scénario peut être riche pour tester les limites de notre approche.

## 7.3 TRAVAUX FUTURS

Nous avons étudié et testé l'approche d'identification des bogues de mise en page en utilisant des invariants. Cette technique a démontré une opportunité pertinente d'automatiser les tests et de faciliter le travail de développement. Notre étude comporte encore des lacunes à combler, ce qui ouvre des possibilités futures pour renforcer cette recherche.

## BIBLIOGRAPHIE


[1] S. Hallé, N. Bergeron, F. Guérin, G. Le Breton, et O. Beroual, "Declarative layout constraints for testing web applications," *Journal of Logical and Algebraic Methods in Programming*, vol. 85, n° 5, pp. 737–758, 2016.

[2] T. Dresher, A. Zuker, et S. Friedman, *Hands-On Full-Stack Web Development with ASP. NET Core : Learn end-to-end web development with leading frontend frameworks, such as Angular, React, and Vue*. Packt Publishing Ltd, 2018.

[3] J. Flusser, B. Zitova, et T. Suk, *Moments and moment invariants in pattern recognition*. John Wiley & Sons, 2009.

[4] S. Cook et J. Daniels, *Designing object systems*. Citeseer, 1994, vol. 135.

[5] C. Cook et D. Schultz, *Beginning HTML with CSS and XHTML : Modern Guide and Reference*. Apress, 2007.

[6] B. Frain, *Responsive web design with HTML5 and CSS3*. Packt Publishing Ltd, 2012.

[7] K. Wilson, "Introduction to html," dans *The Absolute Beginner's Guide to HTML and CSS : A Step-by-Step Guide with Examples and Lab Exercises*. Springer, 2023, pp. 31–39.

[8] C. Grannell, *The essential guide to CSS and HTML web design*. Apress, 2007.

[9] D. DuRocher, *HTML & CSS QuickStart Guide : The Simplified Beginners Guide to Developing a Strong Coding Foundation, Building Responsive Websites, and Mastering the Fundamentals of Modern Web Design*. ClydeBank Media LLC, 2021.

[10] L. Coulson, B. J. R. L. M. Park, et M. Zburlea, *The HTML and CSS Workshop*. Packt Publishing, 2019.

[11] J. Keith et J. Sambells, *DOM scripting : Web design with Javascript and the document object model*. Apress, 2010.

[12] Z. Shute, *Advanced JavaScript : Speed up Web Development with the Powerful Features and Benefits of JavaScript*. Birmingham : Packt Publishing Ltd, 2019.

[13] N. Samoylov, *Introduction to Programming : Learn to Program in Java with Data Structures, Algorithms, and Logic*. Packt Publishing Ltd, 2018.

[14] A. Mesbah et A. Van Deursen, "Invariant-based automatic testing of ajax user interfaces," dans *2009 IEEE 31st International Conference on Software Engineering*. IEEE, 2009, pp. 210–220.

[15] A. Mesbah, A. Van Deursen, et D. Roest, "Invariant-based automatic testing of modern web applications," *IEEE Transactions on Software Engineering*, vol. 38, n° 1, pp. 35–53, 2011.

[16] K. Pattabiraman et B. Zorn, "Dodom : Leveraging dom invariants for web 2.0 application reliability," *Microsoft Research, Tech. Rep. MSR-TR-2009-176*, 2009.

[17] ——, "Dodom : Leveraging dom invariants for web 2.0 application robustness testing," dans *2010 IEEE 21st International Symposium on Software Reliability Engineering*. IEEE, 2010, pp. 191–200.

[18] O. Beroual, F. Guérin, et S. Hallé, "Detecting responsive web design bugs with declarative specifications," dans *International Conference on Web Engineering*. Springer, 2020, pp. 3–18.

[19] T. A. Walsh, G. M. Kapfhammer, et P. McMinn, "Automated layout failure detection for responsive web pages without an explicit oracle," dans *Proceedings of the 26th ACM SIGSOFT international symposium on software testing and analysis*, 2017, pp. 192–202.

[20] A. Pnueli, "The temporal logic of programs," dans *18th Annual Symposium on Foundations of Computer Science (sfcs 1977)*. ieee, 1977, pp. 46–57.

[21] J. Krause, *Developing web components with typescript : native web development using thin libraries*. Springer, 2021.

[22] S. Jacquet, X. Chamberland-Thibeault, et S. Hallé, "Automated repair of layout bugs in

web pages with linear programming," dans *International Conference on Web Engineering*. Springer, 2021, pp. 423–439.

# APPENDICE A

# PREMIÈRE ANNEXE

# Invariant-Based Automatic Testing of Modern Web Applications

Ali Mesbah, *Member*, *IEEE Computer Society*,
Arie van Deursen, *Member*, *IEEE Computer Society*, and Danny Roest

**Abstract**— AJAX-based *Web 2.0* applications rely on stateful asynchronous client/server communication, and client-side runtime manipulation of the DOM tree. This not only makes them fundamentally different from traditional web applications, but also more error-prone and harder to test. We propose a method for testing AJAX applications automatically, based on a crawler to infer a state-flow graph for all (client-side) user interface states. We identify AJAX-specific faults that can occur in such states (related to, e.g., DOM validity, error messages, discoverability, back-button compatibility) as well as DOM-tree invariants that can serve as oracles to detect such faults. Our approach, called ATUSA, is implemented in a tool offering generic invariant checking components, a plugin-mechanism to add application-specific state validators, and generation of a test suite covering the paths obtained during crawling. We describe three case studies, consisting of six subjects, evaluating the type of invariants that can be obtained for AJAX applications as well as the fault revealing capabilities, scalability, required manual effort, and level of automation of our testing approach.

**Index Terms**—Automated testing, web applications, Ajax.

## 1 Introduction

THERE is a growing trend to move applications toward the web. Well-known examples include Google's mail and office software comprising spreadsheet, word processing, and calendar applications. The reasons for this move to the web are manifold:

- no installation effort for end users;
- automatic use of the most recent software version by all users, thus reducing maintenance and support costs;
- universal access from any browser on any machine with Internet access, not only to the application but also to user data;
- new collaboration and community building opportunities as supported by *Web 2.0* applications.

For today's web applications, one of the key technologies facilitating this move is AJAX, an acronym for "Asynchronous JAVASCRIPT and XML" [13]. With AJAX, web browsers not only offer the user navigation through a sequence of HTML pages, but also dynamic rich interaction via graphical user interface (UI) components.

While the use of AJAX technology positively affects the user-friendliness and interactiveness of web applications [27], it comes at a price: AJAX applications are notoriously error-prone due to, e.g., their stateful, asynchronous, and event-based nature, the use of (loosely typed) JAVASCRIPT, the client-side manipulation of the browser's Document-Object Model (DOM), and the use of delta communication between client and webserver [27].

In order to improve the dependability of AJAX applications, static analysis or testing techniques could be deployed. Unfortunately, static analysis techniques are not able to reveal many of the dynamic dependencies present in today's web applications. Furthermore, traditional web testing techniques are based on the classical page request/response model, not taking into account client side functionality. Recent tools such as Selenium[1] offer a capture-and-replay style of testing for modern web applications. While such tools are capable of executing AJAX test cases, they still demand a substantial amount of manual effort from the tester.

The goal of this paper is to support *automated testing of* AJAX *applications*. To that end, we propose an approach in which we automatically derive a model of the user interface states of an AJAX application. We obtain this model by "crawling" an AJAX application, automatically clicking buttons and other UI-elements, thus exercising the client-side UI functionality. In order to recognize failures in these executions, we propose the use of *invariants*: properties of either the client-side DOM tree or the derived state machine that should hold for any execution. These invariants can be generic (e.g., after any client-side change the DOM should remain W3C-compliant valid HTML) or application-specific (e.g., the home-button in any state should lead back to the starting state).

We offer an implementation of the proposed approach in an open source, plugin-based tool architecture. It consists of a crawling infrastructure called CRAWLJAX,[2] as well as a series

- *A. Mesbah is with the Department of Electrical and Computer Engineering, Faculty of Applied Science, University of British Columbia, 2332 Main Mall, Vancouver, BC V6T 1Z4, Canada. E-mail: amesbah@ece.ubc.ca.*
- *A. van Deursen and D. Roest are with the Department of Software Technology, Faculty of Electrical Engineering, Mathematics and Computer Science, Delft University of Technology, Mekelweg 4, Delft 2628CD, The Netherlands. E-mail: arie.vandeursen@tudelft.nl, d.roest@student.tudelft.nl.*

*Manuscript received 2 Mar. 2010; revised 2 Sept. 2010; accepted 3 Feb. 2011; published online 5 Feb. 2011.*
*Recommended for acceptance by J.M. Atlee and P. Inverardi.*
*For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number TSESI-2010-03-0057.*
*Digital Object Identifier no. 10.1109/TSE.2011.28.*

1. http://selenium.openqa.org.
2. http://crawljax.com.

of testing-specific extensions referred to as ATUSA. We have applied these tools to a series of AJAX applications. We report on our experiences in this paper, evaluating the proposed approach in terms of fault-finding capabilities, scalability, automation level, and the usefulness of invariants.

This paper is a substantially expanded and revised version of our paper from early 2009 [28]. Since the first publication on CRAWLJAX [24], a range of improvements to the tool and the underlying algorithms have been realized. Furthermore, the tool and testing approach have been applied to several AJAX applications (see, e.g., [33], [6]). In this paper, we provide an integrated presentation of the full approach, incorporating the most recent developments concerning the crawling algorithm, the testing approach, the available plugins, and the application of the approach to a range of different AJAX applications, of which six are covered in substantial detail.

The paper starts with a survey of related work (Section 2), followed by an analysis of AJAX testing challenges (Section 3). We then explain the crawling algorithms (Section 4) as well as the invariant-based testing approach built on top of it (Sections 5 and 6). After covering the architecture of our plugin-based tool set (Section 7), we describe three case studies, totaling six different AJAX applications (Section 8). We conclude with a discussion of our findings, a summary of our contributions, and an outlook toward future work.

## 2 Related Work

Modern web interfaces incorporate client-side scripting and user interface manipulation which is increasingly separated from server-side application logic [36]. Although the field of rich web interface testing is mainly unexplored, much knowledge may be derived from two closely related fields: traditional web testing and GUI application testing. We survey these in Sections 2.1 and 2.2. We describe current AJAX testing approaches in Section 2.3, after which we provide a short overview of the use of invariants for web testing in Section 2.4.

### 2.1 Traditional Web Testing

Benedikt et al. [4] present VeriWeb, a tool for automatically exploring paths of multipage websites through a crawler and detector for abnormalities such as navigation and page errors (which are configurable through plugins). VeriWeb uses SmartProfiles to extract candidate input values for form-based pages. Although VeriWeb's crawling algorithm has some support for client-side scripting execution, the paper provides insufficient detail to determine whether it would be able to cope with modern AJAX web applications. VeriWeb offers no support for generating test suites as we do in Section 6.

Tools such as WAVES [18] and SecuBat [19] have been proposed for automatically assessing web application security. The general approach is based on a crawler capable of detecting data entry points which can be seen as possible points of security attack. Malicious patterns, e.g., SQL and XSS vulnerabilities, are then injected into these entry points and the response from the server is analyzed to determine vulnerable parts of the web application.

Alfaro apply model-checking [9] to web applications using his tool called MCWEB [10]. His work, however, was targeted toward web 1.0 applications.

A model-based testing approach for web applications was proposed by Ricca and Tonella [31]. They introduce ReWeb, a tool for creating a model of the web application in UML, which is used along with defined coverage criteria to generate test cases. Another approach was presented by Andrews et al. [1], who rely on a finite state machine together with constraints defined by the tester. All such model-based testing techniques focus on classical multipage web applications. They mostly use a crawler to infer a navigational model of the web. Unfortunately, traditional web crawlers are not able to crawl AJAX applications [24].

Logging user session data on the server is also used for the purpose of automatic test generation [11], [34]. This approach requires sufficient interaction of real web users with the system to generate the necessary logging data. Session-based testing techniques are merely focused on synchronous requests to the server and lack the complete state information required in AJAX testing. Delta-server messages [27] from the server response are hard to analyze on their own. Most of such delta updates become meaningful after they have been processed by the client-side engine on the browser and injected into the DOM.

Exploiting static analysis of server-side implementation logic to abstract the application behavior is another testing approach. Artzi et al. [2] propose a technique and a tool called Apollo for finding faults in PHP web applications that is based on combined concrete and symbolic execution. The tool is able to detect runtime errors and malformed HTML output. Halfond and Orso [16], [17] present their static analysis of server-side Java code to extract web application request parameters and their potential values. They use [15] symbolic execution of server-side code to identify possible interfaces of web applications. Such techniques have limitations in revealing faults that are due to the complex (client-side) runtime behavior of modern rich web applications.

### 2.2 GUI Application Testing

Reverse engineering a model of the desktop (GUI) in order to generate test cases has been proposed by Memon [23]. AJAX applications can be seen as a hybrid of desktop and web applications since the user interface is composed of components and the interaction is event-based [27]. However, AJAX applications have specific features, such as the asynchronous client/server communication and dynamic DOM-based user interface, which make them different from traditional GUI applications [22], and therefore require other testing tools and techniques.

### 2.3 Current AJAX Testing Approaches

The server side of AJAX applications can be tested with any conventional testing technique. On the client, testing can be performed at different levels. Unit testing tools such as JsUnit[3] can be used to test JAVASCRIPT on a functional level.

The most commonly used AJAX testing tools are currently *capture/replay* tools such as Selenium IDE,[4] WebKing,[5] and Sahi,[6] which allow DOM-based testing by capturing events fired by user interaction. Other web

3. http://jsunit.net.
4. http://selenium.openqa.org.
5. http://www.parasoft.com/jsp/products/home.jsp?product=WebKing.
6. http://sahi.co.in/w/.

application testing tools such as WebDriver[7] or Watij[8] take a different approach: Rather than being a JAVASCRIPT application running within the browser, they use a wrapping mechanism and provide APIs to control the browser. Such tools demand, however, a substantial amount of manual effort on the part of the tester to create and maintain a test suite since every event trail and the corresponding DOM assertions have to be written by the tester.

Marchetto et al. [21] discuss a case study in which they demonstrate that traditional web testing techniques (e.g., code coverage testing [31], model-based testing [1], session-based testing [11], [34]) have serious limitations when applied to modern *Web 2.0* applications. They propose an approach for state-based testing of AJAX applications. They use traces of the application to construct a finite state machine. Sequences of semantically interacting events in the model are used to generate test cases once the model is refined by the tester.

Our approach is the first to exercise automated testing of *Web 2.0* applications by simulating real user events on the web user interface and inferring an abstract model automatically.

### 2.4 Invariants

The concept of using invariants to assert program behavior at runtime is as old as programming itself [8]. For the domain of web applications, any approach that performs validation of the HTML output (e.g., [4], [2]) could be considered to be using invariants on the DOM. This paper makes the use of invariants for testing web applications explicit by defining different types of (client side) invariants, providing a mechanism for expressing those invariants, and automatically checking them through dynamic analysis.

Automatic detection of invariants is another direction that has gained momentum. The best-known work is that of Ernst et al. on Daikon [12], a tool capable of inferring *likely invariants* from program execution traces. A more recent tool is DoDom [30], capable of inferring DOM invariants. We have also started exploring ways of automatically detecting DOM and JAVASCRIPT invariants in web applications [14].

## 3 AJAX TESTING CHALLENGES

In order to test AJAX applications automatically, we need to face the following challenges:

- Find a method to simulate a user's interaction with the web application.
- Gain access to various dynamic DOM states.
- Develop a method to assess the correctness of the obtained states.

In traditional web applications, states are explicit and correspond to pages having a unique URL. In AJAX applications, the state of the user interface is determined dynamically through event-driven changes in the browser's DOM tree that are only visible after executing the corresponding JAVASCRIPT code. Ultimately, an AJAX application could consist of a single page [25] with a single URL.

The event-driven nature of AJAX presents the first serious challenge for automation, as the event model of the browser must be manipulated, instead of just constructing and sending appropriate URLs to the server. Thus, simulating user events on AJAX interfaces requires an environment equipped with all the necessary technologies, e.g., JAVASCRIPT, DOM, and the XMLHttpRequest object used for asynchronous communication.

In addition, any response to a client-side event can be injected into the single-page interface and therefore faults propagate to and are manifested at the DOM level. Hence, access to the dynamic runtime DOM is a necessity in order to be able to analyze and detect the propagated errors.

One way to simulate a web user and gain access to dynamic states of AJAX applications automatically is by adopting a web crawler capable of detecting and firing events on clickable elements on the web interface. Such a crawler should be able to crawl through different UI states and infer a model of the navigational paths and states. In addition, executing different sequences of events can also trigger an incorrect state. Therefore, we should be able to generate and execute different event sequences as well as different (random or user specified) input data. We have proposed such a crawler for AJAX, called CRAWLJAX [24], which is substantially extended for this work and explained in Section 4.

Automating the process of assessing the correctness of test case output is a challenging task, known as the oracle problem [38]. The problem is even more demanding when we consider AJAX in which all the state changes are manifested through modifications on the DOM tree. Ideally, a tester acts as an oracle who knows the expected output, in terms of DOM tree elements and their attributes, after each state change. When the state space is huge, this manual approach becomes practically impossible. An approach taken in practice is to use a version of the application to obtain a baseline, also known as the Gold Standard [7]. The shortcoming of this approach is that it presumes that the baseline represents a correct version of the system from which initial states can be collected and reused as oracles in subsequent test executions. The web testing literature has mainly used HTML comparators [35] and validators [2] for testing web applications. Such validators are, however, not capable of capturing complex faulty DOM states that are present in modern web applications. To automate the test oracles, we propose to use generic and application-specific invariants on the DOM tree.

The details of our solutions for the challenges mentioned in this section are presented in the following sections.

## 4 DERIVING AJAX STATES

The testing method we propose is based on CRAWLJAX,[9] a crawler capable of automatically deriving a state machine from an AJAX web application, which we originally proposed in early 2008 [24]. One year later (early 2009), we described how this crawler can be applied for testing purposes. Since then, we have used CRAWLJAX in a range of projects (see, e.g., [33], [6]), resulting in numerous improvements to the crawler and the testing approach. In

7. http://seleniumhq.org/docs/09_webdriver.html.
8. http://watij.com/.
9. http://crawljax.com.

the subsequent sections, we provide an integrated presentation of our most recent developments concerning the crawling algorithm.

Algorithms 1 and 2 show the overall crawling process. Central to our approach is the automatic inference of a *state-flow graph* from the web application. To infer such a graph automatically, we open the web application in a web browser, we examine the DOM-tree looking for *candidate elements* to fire events on, and we detect user interface state changes. We conduct the analysis and *navigation* part recursively for all possible states. For *input fields*, we provide random data if no custom data is available. And finally, we provide various options for *controlling the crawling phase*. The following sections discuss these steps in more detail.

**Algorithm 1.** Crawling process with pre/postCrawling hooks

```
 1: procedure START (url, Set tags)
 2: browser ← initEmbeddedBrowser(url)
 3: robot ← initRobot()
 4: sm ← initStateMachine()
 5: preCrawlingPlugins(browser)
 6: crawl(null)
 7: postCrawlingPlugins(sm)
 8: end procedure
 9: procedure CRAWL (State ps)
10: cs ← sm.getCurrentState()
11: Δupdate ← diff(ps, cs)
12: f ← analyseForms(Δupdate)
13: Set C ← getCandidateClickables(Δupdate, tags, f)
14: for c ∈ C do
15:     generateEvent(cs, c)
16: end for
17: end procedure
```

**Algorithm 2.** Firing events and analyzing AJAX states

```
 1: procedure GENERATEEVENT (State cs, Clickable c)
 2: robot.enterFormValues(c)
 3: robot.fireEvent(c)
 4: dom ← browser.getDom()
 5: if stateChanged(cs.getDom(), dom) then
 6:     xe ← getXpathExpr(c)
 7:     ns ← sm.addState(dom)
 8:     sm.addEdge(cs, ns, Event(c, xe))
 9:     sm.changeToState(ns)
10:     runOnNewStatePlugins(ns)
11:     testInvariants(ns)
12:     if stateAllowedToBeCrawled(ns) then
13:         crawl(cs)
14:     end if
15:     sm.changeToState(cs)
16:     if browser.history.canGoBack then
17:         browser.history.goBack()
18:     else
19:         {We have to back-track by going to the initial
            state.}.
20:         browser.reload()
21:         List E ← sm.getPathTo(cs)
22:         for e ∈ E do
23:             re ← resolveElement(e)
24:             robot.enterFormValues(re)
25:             robot.fireEvent(re)
26:         end for
27:     end if
28: end if
29: end procedure
```

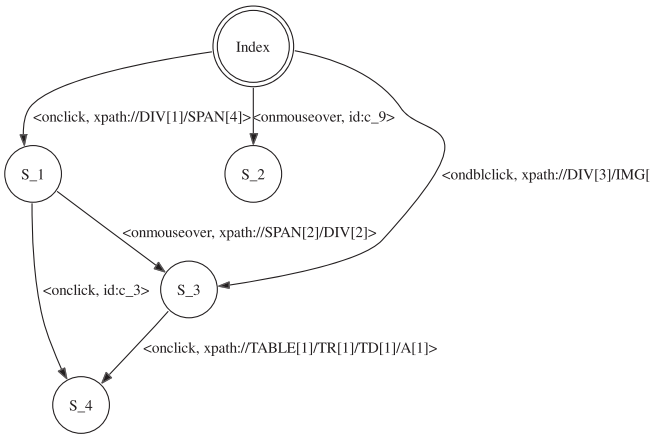



Fig. 1. An inferred state-flow graph.

### 4.1 The State-Flow Graph

The crawler we propose is a tool that can exercise client-side code and identify elements[10] that change the state within the browser's dynamically built DOM. From these state changes, we infer a *state-flow graph*, which captures the states of the user interface and the possible event-based transitions between them.

**Definition 1.** *We define an AJAX UI state change as a change on the DOM tree caused either by server-side state changes propagated to the client or client-side events handled by the AJAX engine.*

We model such UI changes in a directed graph by recording the paths (events) to the DOM changes to be able to navigate between the different states. For that purpose we define a *state-flow graph* as follows:

**Definition 2.** *A* ***state-flow graph*** *for an AJAX site* **A** *is a 3-tuple* $\langle \boldsymbol{r}, \boldsymbol{V}, \boldsymbol{E} \rangle$ *where:*

1. $\boldsymbol{r}$ *is the root node (called Index) representing the initial state after* $\boldsymbol{A}$ *has been fully loaded into the browser.*
2. $\boldsymbol{V}$ *is a set of vertices representing the UI states. Each* $\boldsymbol{v} \in \boldsymbol{V}$ *represents a unique runtime DOM state in* $\boldsymbol{A}$.
3. $\boldsymbol{E}$ *is a set of edges between vertices. Each* $(\boldsymbol{v}_1, \boldsymbol{v}_2) \in \boldsymbol{E}$ *represents a clickable* $\boldsymbol{c}$ *connecting two states if and only if state* $\boldsymbol{v}_2$ *is reached by executing* $\boldsymbol{c}$ *in state* $\boldsymbol{v}_1$.

As an example, Fig. 1 displays the state-flow graph of a simple AJAX site. From the index page three different states can be reached directly. The edges between states are labeled with an identification (e.g., XPath expression) of the element and the event type to reach the next state.

10. For the sake of simplicity, we call such elements "clickables" in this paper; however, they could have any type of DOM event-listener such as ondblclick or onmouseover.

```
1 <script>
2  $(".news").click(function() {
3    $("#content").load("news.html");
4 });
5 </script>
6 <div class="news">...</div>
```

Fig. 2. Attaching an onClick event listener to a DIV element with attribute class="news."

### 4.2 Inferring the State Machine

The state machine (line 4 Algorithm 1) is created incrementally. Initially, it only contains the root state and new states are created and added as the application is crawled and state changes are analyzed (lines 7-8 Algorithm 2). The following components participate in the construction of the graph:

- CRAWLJAX uses an *embedded browser* interface (with different implementations: IE, Firefox, and Chrome) supporting all technologies required by modern dynamic web applications;
- a *robot* is used to simulate user input (e.g., `click`, `hover`, text input) on the embedded browser;
- the *finite state machine* is a data component maintaining the state-flow graph, as well as a pointer to the current state;
- the *controller* has access to the browser's DOM and analyzes and detects state changes. It also controls the robot's actions and is responsible for updating the state machine when relevant changes occur on the DOM tree.

As can be seen in Algorithm 1, the browser, robot and state machine are initialized first (lines 2-4) and from there the `crawl` procedure is called. The steps taken by the crawl procedure to infer the state machine are discussed below.

### 4.3 Detecting Clickables

To illustrate the difficulties involved in crawling AJAX, consider Fig. 2. This is a highly simplified example, showing how an `onclick` event listener can be attached to a `DIV` element at runtime through JAVASCRIPT. Traditional crawlers as used by search engines simply ignore all such clickables. Finding these clickables at runtime is a nontrivial task for any modern crawler.

To tackle this challenge, CRAWLJAX implements an algorithm in which a set of *candidate elements* (line 13 Algorithm 1) are exposed to an event type (e.g., click, mouseover) (line 3 Algorithm 2).

In an automatic mode, the crawler examines all elements of the type `A`, `DIV`, `INPUT`, and `IMG` since these elements are often used to attach event listeners. If the user wishes to define their own criteria for selection, this list can be extended or adapted. The candidate clickables can be labeled as such based on their HTML tag element name and attribute constraints. For instance, all elements with a tag `SPAN` having an attribute `class="menuitem"` can be set to be considered as candidate clickable. For each detected candidate element on the DOM tree, the crawler fires an event on the element in the browser to analyze the effect. A candidate clickable becomes an actual clickable if the event fired on the element causes a DOM change in the browser. In that case, using the clickable element an edge is created and added to the state machine, which connects the previous state to the current state (lines 5-9 Algorithm 2).

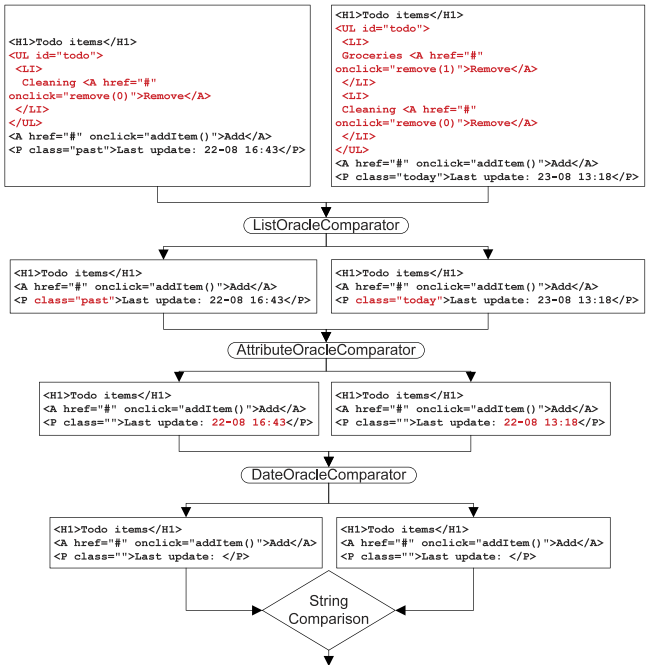



Fig. 3. Pipelining state comparators.

The general approach to using CRAWLJAX is to select a large set of elements to examine (for good coverage) and to exclude elements that are not of importance or can cause problems (delete items or log the user out).

### 4.4 Creating and Comparing States

After firing an event on a candidate clickable, the algorithm inspects the resulting DOM tree to see if the event results in a modified state (line 5 Algorithm 2). If a similar state is part of the state flow graph already, merely an edge is created, identifying the type of click and the location clicked. If the next state is not part of the graph already, a new state is created and added first (line 7 Algorithm 2).

The level of abstraction achieved in the resulting state-flow graph is largely determined by the algorithm used to compare DOM trees (which reflect the states in the flow graph). A generic and effective way is to use a simple string edit distance algorithm such as Levenshtein [20]. This has the advantage that it does not require application-specific knowledge and that the algorithm can be fine-tuned by means of a similarity threshold (between 0 and 1).

Alternatively, we propose the use of a series of "comparators" that each can compare specific aspects of two DOM trees. Each comparator can eliminate specific parts of the DOM tree, such as (irrelevant) attributes, time stamps, or styling issues. The resulting simplified DOM tree is subsequently pipelined to the next comparator. Fig. 3 shows an example of how the pipelining works by stripping the differences and passing the result forward. At the end, after all the desired differences are removed, a simple string comparison determines the equality of the two DOM strings.

Web applications often contain structures with repeating patterns, such as tables and lists. Since the actual elements are not always relevant, we propose comparators that can abstract from the specific elements. In particular, our

technique scans the DOM tree for elements that recur within a structure, and automatically generates a template capturing the pattern. The comparators can use these templates to ignore the given repeating patterns.

While state comparators can be a powerful means to ignore nondeterministic differences, they can also be too permissive, grouping DOM trees that should be considered different. To address this problem, state comparators are equipped with preconditions: Boolean predicates that the test engineer can add to ensure that a given comparator is only used on states meeting certain constraints.

Finally, to enable fast comparison with existing states, the resulting stripped DOM tree is used to calculate a hashcode which is used in all subsequent comparisons in the state machine.

### 4.5 Processing Document Tree Deltas

After a new state has been detected, the crawling procedure is recursively called (line 13 Algorithm 2) to find new possible candidate elements in the partial changes made to the DOM tree. CRAWLJAX computes the differences between the previous document tree and the current one (line 11 Algorithm 1) by means of an enhanced *Diff* algorithm to detect AJAX partial updates, which may be due to a server request call that injects new elements into the DOM. This differencing method is an optimization step, needed to scan and search for candidate clickables merely in the changed parts of the DOM-tree, as opposed to examining all of the elements of the new DOM-tree.

### 4.6 Navigating the States

Upon completion of the recursive call, the browser should be put back into the previous state. A dynamically changed DOM state does not register itself with the browser history engine automatically, so triggering the "Back" function of the browser is usually insufficient. To deal with this AJAX crawling problem, we save information about the elements (line 6-7 Algorithm 2) and the order in which their execution results in reaching a given state. We can then reload the application and follow and execute the elements from the initial state to the desired state (lines 16-27 Algorithm 2).

CRAWLJAX adopts XPath to identify the clickable elements. After a reload or state change, DOM elements, can easily be deleted, changed, or replaced. As a consequence, the XPath expression used for navigation can become invalid. To tackle this problem, our approach uses a mechanism called Element Resolver (line 23 Algorithm 2), which examines the clickable elements before they are used to make state transitions. This examination is needed to make sure we have access to the correct element. To detect the intended element persistently, we use various (saved) properties of the element such as their attributes and text value. Using a combination of these properties, our element resolver searches the DOM for a match, which gives us some degree of reliability in case clickables are removed or changed. Note that despite our element resolving mechanism, because of side effects of server-side state there is no guarantee that we find the same element on the DOM-tree and can reach the exact same state.

TABLE 1
Random Values for Form Input Elements

| Type | Value |
|---|---|
| Text fields | Random string of 8 alpha characters. |
| Checkboxes | Checked with $p < 0.5$ |
| Radio buttons | Checked with $p < 0.5$ |
| Lists | Random item from the list |

### 4.7 Data Entry Points

Besides clicks to proceed along links, buttons, etc., certain user interface states will require data entered by the user. In order to provide input values in AJAX web applications, we have adopted a reverse engineering process to extract all exposed data entry points. To this end, we have extended our crawler with the capability of detecting *input elements* on each newly detected state (line 12 Algorithm 1).

While crawling, before the robot clicks on an element, it checks the DOM for input elements and enters the corresponding values (line 2 Algorithm 2). The related input fields are saved with the clickable element, causing the state transition so that the crawler knows which values to enter in which fields the next time it clicks on the element (while backtracking). For supplying values in the input fields, our approach considers three categories:

**Random input values.** Automation is an important aspect of our approach. Therefore, our tool enters random values in the form elements by default. With this approach, many states that need input values can be reached without any human effort. Table 1 shows the random values used while crawling if no custom values are provided.

If there is already some value in an input field, that value is retrieved and used instead of a random value.

**Custom input values.** Specific input values are often needed for testing or to reach certain states. For example, a valid e-mail address as input is needed to add a contact. With our approach it is possible to provide custom input values by specifying them for the crawler.

**Multiple custom input values.** Entering multiple values for input fields can be useful for testing or to reach more states. For example, entering a normal string, an empty string, and a string with non-alpha-numeric characters in a field that requires a text value could be required for testing.

The challenge here is to know when to enter which value. Our current approach is based on grouping the input elements on each state by specifying the related clickable element (e.g., submit button). For each input value, $n$ number of values are provided. These values are inserted into the input fields and the associated clickable is clicked $n$ times, where $n$ is the position of the provided input values. A more complete approach would be to try every combination of the input values at the cost of increasing the running/testing time.

### 4.8 Controlling the Crawling Phase

In order to have more control on the crawling paths, we use *crawl conditions*, conditions that check whether a state should be visited. A state is crawled only if the crawl conditions are satisfied (line 12 in Algorithm 2). Fig. 4 shows an example of a crawl condition that enforces that only states within the `crawljax.com` domain are crawled.

```
UrlCondition condition = new UrlCondition("crawljax.com");
crawler.addCrawlCondition("Only crawl the Crawljax web site
    ", condition));
```

Fig. 4. A URL crawl condition for only crawling the CRAWLJAX webpage.

In addition, to give more controllability, CRAWLJAX has a set of options, such as the maximum number of states, maximum crawling time, waiting time after each reload (for the page to load fully), waiting time after an event is fired (for the DOM-tree to get updated), and the crawl depth.

## 5 TESTING AJAX STATES THROUGH INVARIANTS

With access to different dynamic web states we can check the user interface against different constraints. We propose to express those as *invariants*, which we can use as an oracle to automatically conduct sanity checks in any state. Although the notion of invariants has predominantly been applied to programming languages for software evolution [12] and verification [3], we believe that invariants can also be adopted for testing modern web applications to specify and constrain DOM elements' properties, their relations, and occurrences.

In this work, we distinguish between *generic* and *application-specific* invariants on the DOM-tree, between DOM-tree states, and on the runtime JAVASCRIPT variables. Each invariant is based on a fault model [7], representing AJAX-specific faults that are likely to occur and which can be captured through the given invariant.

### 5.1 Generic DOM Invariants

#### 5.1.1 Validated DOM

Malformed HTML code can be the cause of many vulnerability and browser portability problems. Although browsers are designed to tolerate HTML malformedness to some extent, such errors have led to browser crashes and security vulnerabilities [2]. All current HTML validators expect all the structure and content to be present in the HTML source code. However, with AJAX, changes are manifested on the single-page user interface by partially updating the dynamic DOM through JAVASCRIPT. Since these validators cannot execute client-side JAVASCRIPT, they simply cannot perform any kind of validation.

To prevent faults, we must make sure that the application has a valid DOM on every possible execution path and modification step. We use the DOM tree obtained after each state change while crawling and transform it into the corresponding HTML instance. A W3C HTML validator serves as an oracle to determine whether errors or warnings occur.

#### 5.1.2 No Error Messages in DOM

A client-site web state should never contain a string pattern that suggests an error message [4] in the DOM tree. Error messages that are injected into the DOM as a result of client-side (e.g., 404 Not Found, 400 Bad Request) or server-side errors (e.g., Session Timeout, 500 Internal Server Error, MySQL error) can be detected automatically. The prescribed list of potential fault patterns should be configurable by the tester.

TABLE 2
Expressing State Invariants

| | Satisfied if and only if |
|---|---|
| XPath expression | the XPath expression returns at least one DOM element |
| Regular expression | the regular expression is found in the DOM string |
| JAVASCRIPT expression | the JAVASCRIPT expression evaluates to true |
| URL condition | the current browser's URL contains the specified string |
| Visible condition | the specified DOM element is visible |

#### 5.1.3 Accessibility and i18n Compliant DOM

Many modern AJAX web applications pose accessibility challenges to people with disabilities due to their dynamic content and advanced user interface components. Evaluating the dynamic states against W3C standards such as the web content accessibility guidelines (WCAG 1.0)[11] or the recent accessible rich internet applications suite (ARIA)[12] can help find accessibility faults automatically.

The same is true for checking each DOM state against W3C internationalization and localization (i18n)[13] guidelines.

#### 5.1.4 Secure States

Testing modern web applications for security vulnerabilities is far from trivial. Capturing web security requirements in terms of generic invariants that can be checked automatically is very promising. Recently, we applied this technique [6] for automatically detecting security vulnerabilities in client-side self-contained web widgets that can coexist independently on a single webpage. We focused on two security invariants, namely, 1) no widget is able to change the content (DOM) of another widget, and 2) no widget can steal data from another widget and send it to the server via an HTTP request, with promising detection results.

In addition, security vulnerabilities such as Cross-Site Scripting (XSS) in AJAX applications can be captured in the same manner. It is worth mentioning that detecting DOM-based XSS requires an analysis of the runtime generated DOM (which we have access to) and not just the pages' syntax [37].

### 5.2 Application-Specific State Invariants

We can define invariants that should always hold and could be checked on the dynamic states, specific to our AJAX application in development. In our case studies, Section 8, we describe a number of application-specific invariants.

Constraints over the DOM-tree can be easily expressed as invariants. Typically, this can be coded into one or two simple Java methods. The resulting invariants can be used to dynamically search for invariant violations.

Table 2 shows the different generic ways the invariants can be expressed. We currently have support for expressing invariants in XPath, regular, or JAVASCRIPT expressions. In addition, we support conditions such as the URL or visibility of DOM elements, which can be used to express invariants. The logical operators NOT, OR, AND, and NAND can also be applied, on or between the invariants, for more flexibility. In addition, each invariant type can be constrained to a specific set of states using preconditions.

11. http://www.w3.org/TR/WAI-WEBCONTENT/.
12. http://www.w3.org/WAI/intro/aria.
13. http://www.w3.org/International/publications.

```
//the menu item on the home page should always have the class attribute 'menuElement'
Condition correctMenuItem = new XPathCondition("//DIV[@id='menu']/UL/LI[contains(@class, 'menuElement')]");
Condition whenAtHomePage = new JavaScriptCondition("document.title=='Home'");
crawler.addInvariant("Home page menu items", correctMenuItem, whenAtHomePage);
```

Fig. 5. Example of an XPATH invariant with a JAVASCRIPT precondition.

While crawling through the different states of the web application, since we have access to the runtime JAVASCRIPT we can also specify invariants on the values of any JAVASCRIPT variable.

Fig. 5 shows an example of expressing an XPATH invariant with a JAVASCRIPT precondition for checking whether the menu item on the home page contains the class attribute "menuElement."

The generated templates capturing DOM patterns (discussed in Section 4.4) can also be augmented and used as invariants on the DOM tree. Fig. 6 shows a DOM invariant template that checks the structure of the list, whether the item is between 2 and 50 alphanumeric characters, and whether an item's identifier is always an integer value.

### 5.3 Generic State Machine Invariants

Besides constraints on the DOM-tree in individual states, we can identify requirements on the state machine and its transitions. Some of the generic invariants that can be defined on any state machine inferred by our tool consist of the following:

#### 5.3.1 No Dead Clickables

One common fault in classical web applications is the occurrence of *dead links*, which point to a URL that is permanently unavailable. In AJAX, clickables that are supposed to change the state by retrieving data from the server, through JAVASCRIPT in the background, can also be broken. Such error messages from the server are mostly swallowed by the AJAX engine, and no sign of a dead link is propagated to the user interface. By listening to the client/server request/response traffic after each event (e.g., through a proxy), dead clickables can be detected.

#### 5.3.2 Consistent Back Button

A fault that often occurs in AJAX applications is the broken Back button of the browser. As explained in Section 4, a dynamically changed DOM state does not register itself with the browser history engine automatically, so triggering the "Back" function makes the browser completely leave the application's webpage. It is possible to programatically register each state change with the browser history and frameworks are appearing which handle this issue. However, when the state space increases, errors can be made and some states may be ignored by the developer to be registered properly. Through crawling, upon each new state one can compare the expected state in the graph with the state after the execution of the Back button and find inconsistencies automatically.

```
<UL id="todo">
  ([\s]*<LI>[\s]+
    [a-zA-Z0-9 ]{2,50}<A href="#" onclick="remove([0-9]+)">
        Remove</A>[\s]*
  </LI>[\s]*)*
</UL>
```

Fig. 6. An augmented template that can be used as an invariant.

### 5.4 Application-Specific State Machine Invariants

Besides generic invariants on the state machine, we can also define constraints on the temporal properties of the web application using application-specific invariants. These temporal properties are usually in a `Source - Action -> Target` format, and can be checked as invariants after the state machine is fully inferred, using, for instance, temporal logic model checking. Examples include:

- from any state, clicking on the `logoff` button should bring us to the logged off state (or the login page);
- from state product list, clicking on the `overview` link should take us to the overview state.

## 6 TESTING AJAX PATHS

While running the crawler to derive the state machine can be considered as a first full test pass, the state machine itself can be further used for testing purposes. For example, it can be used to execute different paths to cover the state machine in different ways. In this section, we explain how to derive a test suite (implemented in JUnit) automatically from the state machine, and how this suite can be used for testing purposes.

### 6.1 Test Suite Generation

To generate a test suite, we use the $K$ *shortest paths* [39] algorithm, which is a generalization of the shortest path problem in which several paths in increasing order of length are sought. We collect all sinks in our graph, and compute the shortest path from the index page to each of them. Loops are included once. This way, we can easily achieve all transitions coverage.

Given a rooted directed graph $G$ with nonnegative edge weights, a positive integer $K$, and two vertices $v_1$ and $v_2$, the problem asks for the $K$ shortest paths from $v_1$ to $v_2$, in nondecreasing order of length. In our algorithm, first the set of *sink* vertices (with no outgoing edges) in $G$ is calculated. Then, we use each sink in $\{s_1, s_2, \ldots, s_n\}$ to find the $K$ shortest paths from the root (`index`) state to $s_i$. Loops are included once.

Next, we transform each path found into a JUnit test case, as shown in Fig. 7. Each test case captures the sequence of events from the initial state to the target state. The JUnit test case can fire events since each edge on the state-flow graph contains information about the event-type and the element the event is fired on to arrive at the target state. We also provide all the information about the clickable element, such as tag name and attributes, as code comments in the generated test method. The test class provides API's to access the DOM (`browser.getDom()`)

```
@Test
public void testcase1() {
browser.goToUrl(url);

/*Element-info: SPAN class=expandable-hitarea */
Clickable c1 = new Eventable(new Identification(
  "xpath", "//DIV[1]/SPAN[4]"), "onclick");

assertPresent(c1);
browser.enterRelatedInputValues(c1);
assertTrue(browser.fireEvent(c1));
assertEquals(oracle.getState("S_1").getDom(),
  browser.getDom());

/*Element-info: DIV class=hitarea id=menuitem2 */
Clickable c2 = new Eventable(new Identification(
  "xpath", "//SPAN[2]/DIV[2]"), "onmouseover");

assertPresent(c2);
assertTrue(browser.fireEvent(c2));
assertEquals(oracle.getState("S_3").getDom(),
              browser.getDom());
...
}
```

Fig. 7. A generated JUnit test case.

and elements (`browser.getElementBy(how, value)`) of the resulting state after each event, as well as its contents.

If a clickable element is associated with input fields, input values are first inserted in the browser's DOM tree before triggering the event.

After each event invocation the resulting state in the browser is compared with the expected state. The comparison can take place at different levels of abstraction ranging from textual [35] to schema-based similarity [25]. The states are currently compared with our oracle comparator pipelining mechanism as discussed in Section 4.4.

### 6.2 Test-Case Execution

Usually, extra coding is necessary for simulating the environment where the tests will be run, which contributes to the high cost of testing [5]. We provide a framework to run all the generated tests automatically using a real web browser and generating success/failure reports. At the beginning of each test case, the embedded browser is initialized with the URL of the AJAX site under test. For each test case, the browser is first put in its initial index state. From there, events are fired on the clickable elements (and forms filled if present). After each event invocation, assertions are checked to see if the expected results are seen on the web application's new UI state.

In short, a test case succeeds if:

1. every transition (edge) element can be successfully found in the state;
2. the corresponding event can be fired on the transition element;
3. there are no time-outs when loading each state;
4. the invariants are satisfied;
5. every visited state in the browser is equivalent to the expected state in the state machine.

### 6.3 Applications

The generated JUnit test suite can be used in several ways. First, it can be run as is on the current version of the AJAX application, but, for instance, with a different browser to detect browser incompatibilities.

Furthermore, the test suite can be applied to altered versions of the AJAX application to support regression testing: For the unaltered user interface, the test cases should pass, and only for altered user interface code might failures occur (also helping the tester to understand what has truly changed). For further details on how this technique is used for regression testing AJAX applications, we refer to our recent paper [33].

The typical use of the derived test suite will be to take apart specific generated test cases, and augment them with application-specific assertions. In this way, a small test suite arises capturing specific fault-sensitive click trails.

```
CrawlSpecification crawler = new CrawlSpecification(URL);
crawler.click("a");
crawler.when(aCondition).click("div").withText("close");
crawler.dontClick("a").withAttribute("class", "info").
     withText("delete");
crawler.dontClick("a").underXPath("//DIV[@id='header']");
crawler.addInvariant(new XPathCondition(xpath, preCondition
     );
crawler.addPlugin(new TestSuiteGenerator());
...
```

Fig. 8. Tool configuration example.

## 7 Tool Implementation

### 7.1 The Testing Framework

Our approach, called Automatically Testing UI States of AJAX (ATUSA), is implemented in Java. It is based on the crawling capabilities of our open-source crawler CRAWLJAX and provides plugin *hooks* for testing AJAX applications at different levels. More implementation details of the crawler can be found on the CRAWLJAX website.[14] The *state-flow graph* is based on the JGrapht[15] library. Apache Velocity templates assist us in the code generation process of JUnit test cases.

ATUSA offers generic invariant checking components, a plugin-mechanism to add application-specific state validators, and generation of a test suite from the inferred state-flow graph.

Furthermore, ATUSA provides a number of generic comparators (see Section 4.4), each of which is responsible for ignoring merely one type of difference. The list of comparators currently available includes Whitespace, Attributes, Style, Datetime, Structure, List, Table, Regex, and XPathExpression, each addressing a particular way of eliminating tree differences.

ATUSA supports looking for many different types of faults in AJAX-based applications, from errors in the DOM instance to errors that involve the navigational path, e.g., constraints on the length of the deepest paths [4] or number of clicks to a certain state. Whenever a fault is detected, the error report along the causing execution path is saved so that it can be easily reproduced later.

ATUSA explores a large number of execution paths that may result from unpredictable user behavior. This is thus complementary to that of capture/replay testing tools, which are useful for testing the correctness of a few specific paths in the web application.

ATUSA offers implemented a Java-based *API* for configuring the tool with merely a few lines, as depicted in Fig. 8.

14. http://crawljax.com.
15. http://jgrapht.sourceforge.net.

TABLE 3
Plugin Types

| Plugin Type | Execution Phase | Examples |
|---|---|---|
| ProxyServer | At the initialization phase | Loading a custom proxy configuration in the embedded browser |
| PreCrawling | Before the crawling | Authentication plugins to log onto the system |
| OnNewState | When a new state is found | Validate DOM, Create Screenshots |
| OnRevisitState | When a state is revisited | Benchmarking |
| OnUrlLoadPlugin | After the initial URL is (re)loaded | Reset back-end state |
| OnInvariantViolation | When an invariant assertion fails | Report builder, test generation |
| PreStateCrawling | Before a new state is crawled | Logging candidate elements |
| PostCrawling | After the crawling process, the state-flow graph is fully inferred | Generating test cases from the state machine |

This figure shows how 1) the user can include (click) and exclude (dontClick) certain element types from the crawling process, 2) invariants can be added for testing, 3) plugins can be added for analysis and test suite generation.

### 7.2 Plugins

ATUSA provides the tester with APIs to implement plugins for validation and fault detection. The main interface for extending the framework is *Plugin*, which is extended by the different *types* of plugins. Each plugin type serves as an extension point that is called in a different phase of the crawling execution. Table 3 summarizes the main plugin types and their invocation phases. Fig. 9 depicts the execution flow of each type of plugin and Fig. 10 depicts the processing view of ATUSA, showing only the DOM Validator and TestSuite Generator as examples of possible plugin implementations.

The list of currently available plugins is shown in Table 4. Most of these plugins are open source.[16]

Understanding why a test case fails is very important to determine whether a reported failure is caused by a real fault or a legal change. To that end, our toolset generates a detailed web report that visualizes the failures. We format and pretty-print the DOM trees without changing their structure and use XMLUnit[17] to determine the DOM differences. The elements related to the differences are highlighted with different colors in the DOM trees. We also capture a snapshot of the browser at the moment the test failure occurs and include that in the report. Other important data, such as the sequence of fired events, JAVASCRIPT debug variables and the list of applied state comparators, are also displayed.

## 8 Empirical Evaluation

In order to assess the usefulness of our approach in supporting modern web application testing, we have conducted a number of case studies, set up following Yin's guidelines [40].

### 8.1 Goal and Research Questions

Our goal in this experiment is to evaluate the fault revealing capabilities, scalability, required manual effort, and level of automation of our approach. Our research questions can be summarized as:

- RQ1. What kind of meaningful invariants can be obtained for AJAX applications and how can they be expressed in ATUSA?

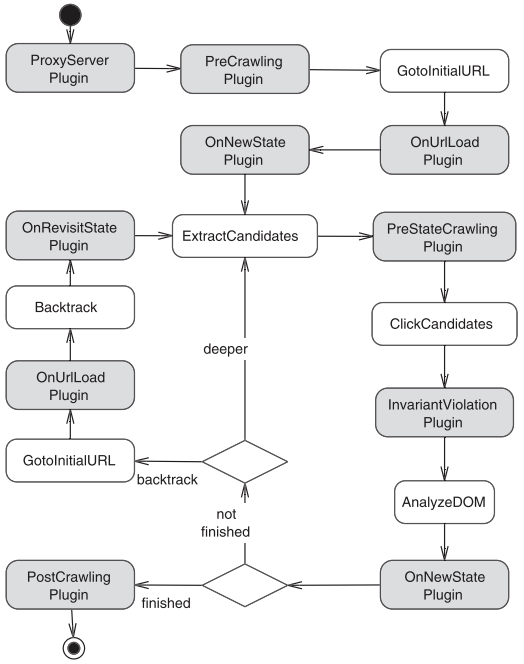



Fig. 9. Plugins invocation flow.

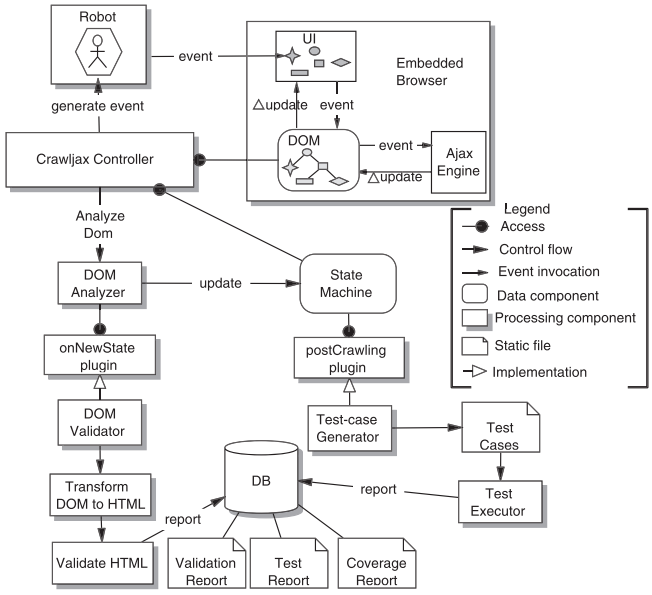



Fig. 10. Processing view of ATUSA, showing only the DOM Validator and TestSuite Generator as examples of possible plugin implementations.

16. http://crawljax.com/plugins/.
17. http://xmlunit.sourceforge.net.

TABLE 4
Available Plugins

| Plugin Name | Functionality | Type |
|---|---|---|
| TestSuite Generator | generated, in JUnit format, a test suite from the inferred state machine | PostCrawling |
| Crawl Overview | generates, in HTML format, an overview of the dynamic states visited | OnNewState, PostCrawling |
| WebScarab Wrapper [6] | wraps around the WebScarab proxy. Used for modifying passing HTTP request/response | ProxyServer |
| SFG Exporter | exports the state-flow graph to different formats (e.g., dot, GraphML, GML, Visio) | PostCrawling |
| Benchmark | creates performance measurement graphs of crawl sessions | PostCrawling |
| Error Reporter | makes a report, in HTML format, of the detected violations | OnInvariantViolation |
| LogIn | utility plugin to help with logging in | PreCrawling |
| Mirror Generator [24] | generates a static linked HTML version of the dynamic states | PostCrawling |
| DOM Validator | Validates each dynamic DOM state against the W3C standard | OnNewState |
| InvarScope [14] | detects JavaScript and DOM invariants dynamically | ProxyServer, OnNewState, PostCrawling |
| CrossBrowser Tester [26] | checks each state in three different browsers looking for cross-browser incompatibilities | PostCrawling |
| RegressionTester [33] | conducts regression tests | PostCrawlng |

- RQ2. What is the fault revealing capability and effectiveness of our testing approach?
- RQ3. What is the performance of the proposed approach, and how well does it scale?
- RQ4. What is the automation level and how much manual effort is involved in the testing process?

## 8.2 Study 1: Invariants

In our fist study, our goal is to assess to what extent meaningful invariants can be obtained for AJAX applications (RQ1). To thatn end, we analyze four open-source AJAX applications.

### 8.2.1 Case Study Setup

Our assessment involves the following steps:

1. We run our tool on the subject system (using the default configurations) to obtain a state-flow graph. We visualize the graph with the *Crawl Overview* plugin.
2. We analyze the graph manually to assess its completeness and to see if the most important user interface states are covered. If necessary, we adjust the crawler's configuration parameters and settings to increase the coverage, for example by providing specific elements that should be clicked or input values that need to be filled in.
3. We inspect the states from the graph, analyzing the DOM (with tools such as Firebug[18]) and JAVASCRIPT code in search of candidate invariants.
4. To identify candidate invariants, we use tools such as Firefinder[19] and our own regular expression tool to extract and evaluate XPath and regular expressions over the DOM-tree.
5. We express the selected invariants in Java using ATUSA's invariant expression mechanisms (see Section 5.2).
6. We run our tool to check the invariants at runtime automatically.

### 8.2.2 THEORGANIZER

THEORGANIZER[20] is an open-source web application that can be used as a task manager and organizer. It is written as a J2EE application using WebWork, Spring JDBC, and the Prototype AJAX library.

The configuration setup for THEORGANIZER was straightforward: include all images as candidate clickables and use the random input-value generator for form inputs. THEORGANIZER requires authentication; thus we wrote a plugin to log into the web application automatically.

Fig. 11 depicts some parts of the inferred graph visualized by the *CrawlOverview* plugin. This graph shows the outgoing and incoming edges from each state. In addition, we can zoom into each state by clicking on the snapshot image of the state (taken during crawling) to conduct further examination.

After manually inspecting the states, we documented five invariants for THEORGANIZER, listed as invariants O1-O5 in Table 5. These consisted of one generic, one XPath, and one Regular expression state invariant, as well as two application-specific state machine invariants (see Section 5.4).

We detected violations through invariants O2, O4, O5, as well as generic invariant A1, which was relevant for all cases. The most interesting violation was O4, in which the expected behavior was that after clicking on the logoff button, we would land on a logged off state. This invariant failed when we used Firefox as our embedded browser. Closer inspection revealed that the logoff element has an

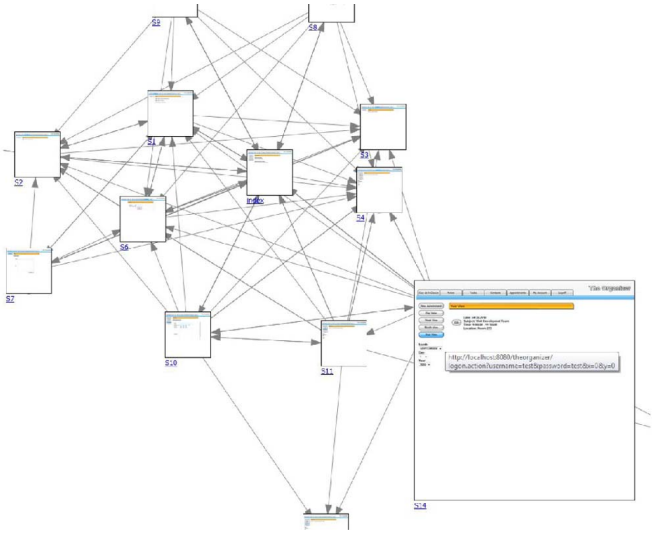


Fig. 11. The graph overview for THEORGANIZER, generated by the CrawlOverview plugin.

18. http://getfirebug.com/.
19. https://addons.mozilla.org/en-US/firefox/addon/11905/.
20. http://www.apress.com/book/downloadfile/2931.

TABLE 5
Invariants for Study 1

| Inv. # | Subject System | Inv. Description | Type |
|---|---|---|---|
| O1 | THEORGANIZER | The Year View state `//img[contains(@src, 'head_yearView')]` contains at least one appointment item `//input[@id='edit']` | XPath expr. state invariant |
| O2 | THEORGANIZER | 'Failed to populate the list properly!' | Generic ('Fail%') DOM invariant |
| O3 | THEORGANIZER | Appointment item (view) structure | Reg. expr. state invariant |
| O4 | THEORGANIZER | Clicking on the logoff button `//img[contains(@id, 'logoff')]` results in a state with `//div[contains(text(), 'You have logged out')]` | application-specific SM inv. |
| O5 | THEORGANIZER | Clicking on `//img[@id="X"]` results in a state with `//img[contains(@src, 'head_X')]`, where X is a string | application-specific SM inv. |
| T1 | TASKFREAK | Top level body contains `div[@id='header']`, `div[@id='container']`, and at most one `div[@id='calendar']` | XPath expr., state invariant |
| T2 | TASKFREAK | All pages displaying current tasks via `table[@id='taskSheet']` match a given template | Reg. expr. state invariant |
| T3 | TASKFREAK | Reload button in any state found via `img[@id='frk-status']` should lead to state displaying current tasks | application-specific SM inv. |
| H1 | HITLIST | Contact template, shown in Figure 13 | Reg. expr. (template) state invariant |
| U1 | THETUNNEL | global variable `alive` is true during the game, and false after player fails | JAVASCRIPT invariant |
| U2 | THETUNNEL | position of ship must be 32 times higher than the wall `ship_x+32 >= right_wall` | JAVASCRIPT invariant |
| U3 | THETUNNEL | the background value must be between 0 and 20 | JAVASCRIPT invariant |
| A1 | All systems | Back button | Generic SM inv. |

`onclick` event listener attached to it which calls a JAVASCRIPT function called `logoff()`. Firefox seems to have a conflict with this function name. One explanation could be that the word *logoff* is a hidden keyword in Firefox.

Invariant 5 is also worth mentioning. This invariant captures a pattern: Clicking on an image element with `id=X` results in a state with an image as header having `src=head_X`, where X is, for instance, "Tasks" or "Contacts." This invariant passes for all states except for "Appointments." Clicking on the *Appointments* element takes us to a state with an image as header having `src=head_dayView` as the header. Such inconsistencies in dynamic web applications are commonplace and usually difficult to spot manually. With automated invariant testing they can be detected and fixed in a systematic manner. The manual effort for THEORGANIZER case was less than 30 minutes.

### 8.2.3 TASKFREAK

TASKFREAK[21] is a simple task management and to-do list application written in PHP. Configuring CRAWLJAX required specifying the username and password to be used, as well as the HTML input fields where these needed to be entered, which was done through a simple *OnUrlLoad* plugin (Section 7.2). Furthermore, a quick inspection of the first page revealed that table data is clickable in TASKFREAK, which is why we specified that `td`- and `th`-elements are candidate clickables (Section 4.3). Since TASKFREAK includes a ticking clock in its page, we enabled the *DateOracleComparator* (Section 4.4), thus removing the current time before determining DOM-equality.

The random data for input fields (Section 4.7) works well for most of the data entry points for TASKFREAK. In order to permit reaching additional user interface states, we configured CRAWLJAX with custom values for a valid e-mail address as well as an invalid one. Furthermore, when attempting to change the password we ensured that the password entered the second time was the same as the one entered the first time. Identifying these data entry points requires a manual exploration of the application and the derived graph obtained through the *CrawlOverview* plugin, which for TASKFREAK took less than one hour.

A selection of the application-specific invariants for TASKFREAK is listed in Table 5. The first invariant T1 expresses the high-level design decision that at any time the top-level `body`-element of TASKFREAK consists of three `div`-elements: a *header* for the top navigation menu, a *container* for the actual list of to-do items, and an optional *pop-up* area for, e.g., data entry in a calendar popping up. While simple in nature, this invariant already reveals an issue in TASKFREAK: After closing a pop-up, the corresponding `div` in the DOM-tree should be removed. In TASKFREAK, however, this is only correctly done for the calender pop-up when the user presses the *save* button: If *cancel* is pressed instead, the `div`-entry is not removed, leading to a (slowly) growing DOM tree. The invariant that at most one calendar pop-up can exist at any time spots this problem.

Note that the pop-up problem corresponds to a common AJAX-idiom: Parts of the DOM tree can be rendered invisible and can be used for representing data, user-interface elements, and so on. It is the programmer's responsibility to manage these DOM-elements and to "garbage collect" them in order to avoid endlessly growing DOM trees. Invariants can be used to express constraints over these parts of the DOM tree, ensuring proper DOM-tree management.

Other invariants include the use of a template (see Section 5.2) to ensure that all states displaying the list of actions have the same structure (T2), as well as invariants on the state machine expressing that the *reload* button always leads to the required state (T3), and that the browser's *Back* button behaves as expected (A1, which is violated in TASKFREAK).

### 8.2.4 HITLIST

Our third experimental subject for this study is the AJAX-based open-source *HitList*,[22] which is a task manager based on PHP and jQuery.

21. http://www.taskfreak.com., TaskFreak! Original, v0.6.4.

22. HitList Version 0.01.09b, http://code.google.com/p/hit-list/.

```
//check on Add Contact page
preCondition = new RegexCondition("Add Contact");
//check whether 'John Doe' does not already exist
condition = new JavaScriptCondition("$.ajax({ url: 'ajax/
     home.php', async: false }).responseText.indexOf('John
     Doe')==-1");
crawlConditions.add(new CrawlCondition("AddOnce", "Only add
      John Doe once", condition, preCondition));
```

Fig. 12. A JAVASCRIPT crawl condition, with regular expression precondition, for adding a contact once in HITLIST.

For HITLIST, the configuration of our tool consisted of including all anchor tags as well as all input elements having a class attribute equal to `add_button` as candidate clickables. Furthermore, we excluded from crawling all the elements that deleted items from the application (e.g., `<a title=Delete...>`, `<a class=delete-button...>`). To ignore subtle DOM differences, we pipelined the generic *Table* and *List* oracle comparators. These comparators abstract away the differences in structures of the HITLIST tables and lists.

To constrain the state space, we created a *CrawlCondition* that ensures a contact can only be added once during the crawling phase. This was done by checking a JAVASCRIPT condition in the *Add Contact* state, as shown in Fig. 12. When the precondition (a state containing the text "Add Contact") is satisfied, the JAVASCRIPT condition retrieves the list of contacts from the server and checks whether there are no contacts present during execution. In that case it returns true, allowing our tool to add a new contact.

From the output of the *Crawl Overview* plugin we generated a regular expression for the contact state. We manually augmented this generated template and created a custom *Contact* regular expression invariant for the contact list. This template, shown in Fig. 13, serves as a DOM invariant, since it checks the structure as well as the validity of the contact's name, phone number, e-mail, and *id*.

With H1, we were able to detect a violation in a regression version of HITLIST, namely, leading zeros in phone numbers were missing (e.g., `0641288822` was saved as `641288822`).

#### 8.2.5 THETUNNEL

Our last subject for this study is an open-source web-based implementation of a tunnel game.[23] In this game, the player controls an airplane and the objective is to avoid hitting a moving wall. It is written using jQuery.

For this web application, we were interested in documenting JAVASCRIPT invariants. Therefore, we analyzed the JAVASCRIPT source code manually and documented a number of invariants on the global variables that could be used as assertions to test the program state.

A few of the invariants we obtained are listed in Table 5. These invariants were turned into assertions and used for regression testing the JAVASCRIPT code automatically. Assertions on global variables can be checked through ATUSA's invariant checking APIs. For instance, Fig. 14 shows how U2 in Table 5 can be checked through ATUSA.

Checking local variables can be done by injecting the assertion code into the JAVASCRIPT source code through a proxy. The details of this technique can be found in [14].

23. http://arcade.christianmontoya.com/tunnel/.

```
<TABLE class="contact">\s*
 <TBODY>\s*
  <TR>\s*
   <TD height="[0-9]+" width="[0-9]+">\s*
    <IMG src="public/images/[a-z]+.png"/>\s*
   </TD>\s*
   <TD style="[^"]*">\s*
    [a-zA-Z ]{3,100}[^<]*<BR/>(.*?)
    [a-zA-Z0-9\._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,4}[^<]*<
        BR/>(.*?)
    [0-9]{10,15}[^<]*<BR/>\s*
    <A class="action view-details" href="javascript:;" id
        ="[0-9]+">(.*?)view details(.*?)</A>\s*
   </TD>\s*
  </TR>\s*
 </TBODY>\s*
</TABLE>
```

Fig. 13. Regular expression (template) invariant for HITLIST contact list.

```
crawler.addInvariant(new JavaScriptCondition("ship_x + 32
     >= right_wall"));
```

Fig. 14. Checking invariants on JAVASCRIPT global variables.

#### 8.2.6 Findings

Going back to our first research question (RQ1), from the four case studies described, we conclude the following:

- Writing invariants captures and requires an understanding of the design of the web application. The automatically generated crawl overview helps us in the process of program comprehension.
- Invariants over the relations of elements and their attributes on the DOM tree can be naturally expressed using XPath expressions. Invariants capturing the structure of the elements can be expressed using template-based regular expressions. Those constraining actions and their consequences can be captured in state machine invariants. Finally, code-level JAVASCRIPT design contracts can be easily expressed as JAVASCRIPT expressions.
- Invariants can be used to find various faults, including DOM memory leaks (TASKFREAK), cross-browser inconsistencies (THEORGANIZER), and regressions (HITLIST).
- The manual effort involved in configuring CRAWLJAX and writing the described invariants in this study, is minimum, amounting to less than one hour for each of the cases covered.

### 8.3 Study 2: TUDU

In this study, we are particularly concerned with assessing the fault revealing capabilities, scalability, and required manual effort our approach (RQ2-RQ4).

#### 8.3.1 Subject System

Our experimental subject in this study is the AJAX-based open-source TUDU web application[24] for managing personal to-do lists, which has also been used by other researchers [22]. The server side is based on J2EE and consists of around 12K lines of Java/JSP code, of which around 3K forms the presentation layer we are interested in. The client side extends on a number of AJAX libraries such as DWR[25] and Scriptaculous,[26] and consists of

24. http://tudu.sourceforge.net.
25. http://directwebremoting.org.
26. http://script.aculo.us.

TABLE 6
TUDU Case Study

| LOC Server-side | LOC Client-side | DOM string size | Candidate Clickables | Detected Clickables | Detected States | Detected Entry Points | DOM Violations | Back-button | Generated Test Cases | Coverage Server-side | Coverage Client-side | Detected Faults | Manual Effort | Performance |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 3k | 11k (ext)<br>580 (int) | 24908<br>(byte) | 332 | 42 | 34 | 4 forms<br>21 inputs | 182 | false | 32 | 73% | 35% (ext)<br>75% (int) | 80% | 26.5<br>(minutes) | 5.6<br>(minutes) |

around 11k LOC of external JAVASCRIPT libraries and 580 internal LOC.

### 8.3.2 Case Study Setup

For RQ2 and RQ3, we configured ATUSA (1 minute), setting the URL of the deployed site, the tag elements that should be included (`A, DIV`) and excluded (`A:title=Log out`) during the crawling process, the depth level (2), the similarity threshold (0.89), and a maximum crawling time of 60 minutes. Since TUDU requires authentication, we wrote (10 minutes) a `preCrawling` plugin to log into the web application automatically. We use the *TestSuite Generator* plugin in this case study to automatically generate a test suite from the inferred state machine. To address RQ4, we report the time spent on parts that required manual work.

As shown in Table 6, we measure average DOM string size, number of candidate elements analyzed, detected clickables and states, detected data entry points, detected faults, number of generated test cases, and performance measurements, all of which are printed in a log file by ATUSA after each run.

In the initial run, after the login process, ATUSA crawled the TUDU application, finding the doorways to new states and detecting all possible data entry points recursively. We analyzed the data entry points and provided each with custom input values (15 minutes to evaluate the input values and provide useful values). For the second run, we activated (50 seconds) the DOM Validator, Back Button, Error Detector, and Test Case Generator plugins and started the process. ATUSA started crawling and, when forms were encountered, the custom input values were automatically inserted into the browser and submitted. Upon each detected state change, the invariants were checked and reports were generated if any inconsistencies were found. At the end of the crawling process, a test suite was generated from the inferred state-flow graph.

To the best of our knowledge, there are currently no tools that can automatically test AJAX dynamic states. Therefore, it is not possible to form a baseline for comparison using, for instance, external crawlers. To assess the effectiveness of the generated test suite, we measure code coverage on the client as well as the presentation tier of the server. Although the effectiveness is not directly implied by code coverage, it is an objective and commonly used indicator of the quality of a test suite [16].

To that end, we instrumented the presentation part of the server-side Java code (`tudu-dwr`) with Clover. We exclude the server-side business logic and database layers since we are merely interested in the user interface parts. For the client side, we instrumented JAVASCRIPT libraries and custom code with JSCoverage,[27] and deployed the whole web application to an application server (Apache Tomcat). For each test run, we bring the TUDU database to the original state using a SQL script. We run all the test cases against the instrumented application, through ATUSA's embedded browser, and compute the amount of coverage achieved for server and client-side code. In addition, we manually seeded 10 faults, capable of causing inconsistent states (e.g., DOM malformdness, adding values longer than allowed by the database, adding duplicate to-do items, removing all items instead of one), and measured the percentage of faults detected.

### 8.3.3 Findings

The results of this study are presented in Table 6. Based on these observations we conclude that:

- The use of ATUSA can help to reveal generic faults, such as DOM violations, automatically.
- As far as RQ2 is concerned, the generated test suite can give us useful code coverage: 73 percent of server-side presentation code and 75 percent of client-side JAVASCRIPT custom code. Note that only partial parts of the external JAVASCRIPT libraries are actually used by TUDU resulting in a low-coverage percentage (35 percent). ATUSA revealed most of the DOM-based faults: 8 of the 10 seeded faults were detected, two faults were undetected because, during the test execution, they were silently swallowed by the JAVASCRIPT engine and did not affect the DOM. It is worth mentioning that increasing the depth level to 3 significantly increased the measured crawling time past the maximum 60 minutes, but did not influence the fault detection results. The code coverage, however, improved by approximately 10 percent.
- The performance and scalability of the crawling and testing process is very acceptable: It takes ATUSA less than 6 minutes to crawl and test TUDU, analyzing 332 clickables and detecting 34 states (RQ3).
- The manual effort involved in setting up ATUSA (less than half an hour in this case) is minimal (RQ4).

27. http://siliconforks.com/jscoverage/.

```
//case one: warn about collapsible divs within expandable items
String xpathCase1 =  "//LI[contains(@class,'expandable')]/DIV[contains(@class,'collapsable')]";
crawler.addInvariant(new NotXPathCondition(xpathCase1);

//case two: warn about collapsible items within expandable items
String xpathCase2 = "//LI[contains(@class,'expandable')]/UL/LI[contains(@class,'collapsable')]";
crawler.addInvariant(new NotXPathCondition(xpathCase2);
```

Fig. 15. Application-specific invariants expressed using XPath.

TABLE 7
Faults Found in CJZ-AJAX

| Failure | Cause | Violated Invariant | Invariant type |
|---|---|---|---|
| Images not displayed | Base URL in dynamic load | Dead Clickables | Generic |
| Broken synchronization in IE | Invalid HTML id | DOM-validator | Generic |
| Inconsistent history | Issue in `listen` library | Back-Button | Generic |
| Broken synchronization in IE | Backslash versus slash | Consistent current page | Specific |
| Corrupted table | Coding error | `treeview` invariants, Consistent current page | Specific |
| Missing TOC Entries | Incomplete input data | Consistent current page | Specific |

## 8.4 Study 3: Finding Real-Life Bugs

Our final case study involves the development of an AJAX user interface in a small commercial project. We use this case study to evaluate the manual effort required to use ATUSA (RQ4) and to assess the capability of ATUSA to find faults that actually occurred during development (RQ2).

### 8.4.1 Subject System

The case at hand is Coachjezelf (CJZ, "Coach Yourself"),[28] a commercial application allowing high school teachers to assess and improve their teaching skills. CJZ is currently in use by 5,000-6,000 Dutch teachers, a number that is growing with approximately 1,000 paying users every year.

The relevant part for our case is the interactive table of contents (TOC), which is to be synchronized with an actual content widget. In older versions of CJZ this was implemented through a Java applet; in the new version this is to be done through AJAX in order to eliminate a Java virtual machine dependency.

The two developers working on the case study spent around one week (two person-weeks) building the AJAX solution, including requirements elicitation, design, understanding and evaluating the libraries to be used, manual testing, and acceptance by the customer.

The AJAX-based solution made use of the `jQuery`[29] library, as well as the `treeview`, `history-remote`, and `listen` plugins for `jQuery`. The libraries are comprised of around 10,000 lines of JAVASCRIPT and the custom code is around 150 lines of JAVASCRIPT, as well as some HTML and CSS code.

### 8.4.2 Case Study Setup

The developers were asked 1) to try to document their design and technical requirements using invariants, and 2) to write the invariants in ATUSA plugins to detect errors made during development. After the delivery of the first release, we evaluated 1) how easy it was to express these invariants in ATUSA, and 2) whether the (generic or application-specific) plugins were capable of detecting faults.

28. See www.coachjezelf.nl for more information (in Dutch).
29. jquery.com.

### 8.4.3 Application-Specific Invariants

Two sets of invariants were proposed by the developers. The first essentially documented the (external) `treeview` component, capable of (un)folding tree structures (such as a table of contents).

The `treeview` component operates by setting HTML class attributes (such as `collapsible`, `hit-area`, and `lastExpandable-hitarea`) on nested list structures. The corresponding style sheet takes care of properly displaying the (un)folded (sub)trees, and the JAVASCRIPT intercepts clicks and rearranges the class attributes as needed.

Invariants were devised to document constraints on the class attributes. As an example, the `div`-element immediately below a `li`-element that has the class `expandable` should have class `expandable-hitarea`. Another invariant is that expandable list items (which are hidden) should have their CSS display type set to "none."

The second set of invariants specifically dealt with the code written by the developers themselves. This code took care of synchronizing the interactive display of the table of contents with the actual page shown. Clicking links within the page affects the display of the table of contents and vice versa.

This resulted in essentially two invariants: one to ensure that within the table of contents at most one path (to the current page) would be open and the other that, at any time, the current page as marked in the table of contents would actually be displayed in the content pane.

Expressing such invariants on the DOM tree was quite easy, requiring a few lines of Java code using XPath. An example is shown in Fig. 15.

### 8.4.4 Failures Detected

At the end of the development week, ATUSA was used to test the new AJAX interface. For each type of application-specific invariant, an `inCrawling` plugin was added to ATUSA. Six types of failures were automatically detected: three through the generic plugins, and three through the application-specific plugins just described. An overview of the type of failures found and the invariant violations that helped to detect them is provided in Table 7.

The application-specific failures were all found through two invariant types: the *Consistent current page*, which

expresses that in any state the table and the actual content should be in sync, and the *treeview invariants*. Note that for certain types of faults, for instance, the treeview corrupted table, a very specific click trail had to be followed to expose the failure. ATUSA gives no guarantee of covering the complete state of the application; however, since it tries a huge combination of clickables recursively, it was able to detect such faults, which were not seen by developers when the application was tested manually.

#### 8.4.5 Findings

Based on these observations we conclude that:

- The use of ATUSA can help to reveal bugs that are likely to occur during AJAX development and are difficult to detect manually (RQ2).
- Application-specific invariants can help to document and test the essence of an AJAX application, such as the synchronization between two widgets (RQ1-RQ2).
- The manual effort in expressing such invariants in Java and using them in ATUSA is minimal (RQ4).

## 9 Discussion

### 9.1 Automation Scope

User interface testing is a broad term, dealing with testing how the application and the user interact. This typically is manual in nature, as it includes inspecting the correct display of menus, dialog boxes, and the invocation of the correct functionality when clicking them. The type of user interface testing that we propose does not replace this manual testing, but augments it: Our focus is on finding programming faults, manifested through failures in the DOM tree. As we have seen, the highly dynamic nature and complexity of AJAX make it error-prone, and our approach is capable of finding such faults automatically.

### 9.2 Invariants

Our solution to the oracle problem is to include invariants (as also advocated by, e.g., Meyer [29]). AJAX applications offer a unique opportunity for specifying invariants thanks to the central DOM data structure. Thus, we are able to define generic invariants that should hold for all AJAX applications, and we allow the tester to use the DOM to specify generic or application-specific invariants. Furthermore, the state machine derived through crawling can be used to express invariants, such as correct Back-button behavior. Again, this state machine can be accessed by the tester to specify his or her own invariants. These invariants make our approach much more sophisticated than *smoke tests* for user interfaces (as proposed by, e.g., Memon [23])—which we can achieve thanks to the presence of the DOM and state machine data structures. Note that just running CRAWLJAX would correspond to conducting a smoke test: The difficulty with web applications (as opposed to, e.g., Java Swing applications) is that it is very hard to determine when a failure occurs—which is solved in ATUSA through the use of invariants.

### 9.3 Generated versus Hand-Coded JavaScript

The case studies we conducted involve two different popular JAVASCRIPT libraries (i.e., jQuery and Prototype) in combination with hand-written JAVASCRIPT code. Alternative frameworks exist, such as Google's web toolkit (GWT)[30] in which most of the client-side code is generated. ATUSA is entirely independent of the way the AJAX application is written, so it can be applied to such systems as well. This will be particularly relevant for testing the custom JAVASCRIPT code that remains to be handwritten, and which can still be tricky and error-prone. Furthermore, ATUSA can be used by the developers of such frameworks to ensure that the generated DOM states are correct.

### 9.4 Manual Effort

The manual steps required to run ATUSA consist of configuration, plugin development, and providing custom input values, which for the cases conducted took less than an hour. The hardest part is deciding which application-specific invariants to adopt. This is a step that is directly connected with the *design* of the application itself. Making the structural invariants explicit not only allows for automated testing, it is also a powerful design documentation technique. Admittedly, not all web developers will be able to think in terms of invariants, which might limit the applicability of our approach in practice. Those capable of documenting invariants can take advantage of the framework ATUSA provides to actually implement the invariants.

### 9.5 Performance and Scalability

The state space of any realistic web application is huge and can cause the well-known *state explosion problem*. To constrain the state space, we provide the tester with a set of configurable options. These constraints include the maximum search depth level, similarity threshold for comparing states, maximum number of states per domain, maximum crawling time, and the option of ignoring external links and links that match some predefined set of regular expressions. The main component that can influence the performance and scalability is the crawling part. The performance of crawling an AJAX site depends on many factors, such as the speed at which the server can handle requests, how fast the browser and client-side JAVASCRIPT can update the interface, and the size of the DOM tree.

### 9.6 Application Size

The six experimental subjects involve around 20,000 lines of JAVASCRIPT library code, several hundred lines of custom application code, and several thousand dynamic DOM states. One might wonder whether the size of the subjects counts against the external validity of our study. Our results, however, are based on *dynamic* analysis rather than static code analysis; hence, the amount of JAVASCRIPT code is not the determining factor in our view. The number of dynamic states is, in this case, a more realistic measure. The limiting factor for the number of states to be examined is the amount of memory available and the size of the DOM tree. Based on our experiments, the maximum number of states can be calculated by $\frac{sizeOf(memory)}{3 \times sizeOf(DOM)}$. The average DOM size of enterprise applications is around 0.25 MB [24]. On a workstation with 4 GB of memory, this

30. http://code.google.com/webtoolkit/.

would result in around 5,460 states, which is sufficient for most enterprise web applications.

### 9.7 Threats to Validity

Some of the issues concerning the *external* validity of our empirical evaluation have been covered in the above discussion on scope, generated code, application size, and scalability. The main goal of Study 1 (Section 8.2) was to demonstrate what type of invariants can be found in modern web applications and how different types of invariants can be expressed in ATUSA for automated testing. We have merely provided a few examples of each type and the list is not exhaustive. The small number of examples could, however, be a threat to external validity. More studies need to be done to extend the instances of each type.

With respect to *internal* validity, we minimized the chance of ATUSA errors by including a rigorous JUnit test suite. ATUSA, however, also makes use of many (complex) third party components, and we did encounter several problems in some of them. While these bugs do limit the current applicability of our approach, they do not affect the validity of our results. As far as the choice of faults in the second study (Section 8.3) is concerned, we selected them from the TUDU bug tracking system, based on our fault models, which we believe are representative of the types of faults that occur during AJAX development. The choice is therefore not biased toward the tool but possibly toward the fault models we have. With respect to *reliability*, our tools and all the subject systems of studies 1 and 2 are open source, making these cases fully reproducible.

### 9.8 Ajax Testing Strategies

ATUSA is a first, but essential step in automating the testing process of AJAX applications. Thanks to the plugin-based architecture of ATUSA, it now becomes possible to extend, refine, and evaluate existing software testing strategies (such as evolutionary, state-based, category-partition, and selective regression testing) for the domain of AJAX applications.

In our recent work [33], we have presented how our approach can be used for conducting *regression testing* of highly dynamic web applications. The initial results are very promising: Through a number of case studies, we show how generated test suites can detect regressions in different versions of a web application through oracle comparator pipelining.

Another direction involves the application to *security testing* of *Web 2.0* widget interactions [6], which we have conducted in close collaboration with the industry.[31]

Application in the area of *accessibility testing* involves compliance to W3C accessibility standards. Initial results in this area involve an application to Google's AdSense Front End, 3.0[32] through an internship at Google (London) [32].

Further, the technique is currently being applied by Fujitsu Laboratories of America to a number of industrial web applications. The approach is also being adopted for web model-checking using the inferred state machine. Recently, we have used the approach to automate *cross-browser compatibility* testing of modern web applications [26].

## 10 Concluding Remarks

In this paper, we have proposed a method for testing AJAX applications automatically. Our starting point for supporting AJAX-testing is CRAWLJAX, a crawler for AJAX applications that we proposed in our earlier work [24], which can dynamically make a full pass over an AJAX application. Our current work consists of extending the crawler substantially for supporting automated testing of modern web applications. We developed a series of plugins, collectively called ATUSA, for invariant-based testing and test suite generation.

To summarize, this paper makes the following contributions:

1. A series of fault models that can be automatically checked on any user interface state, capturing different categories of errors that are likely to occur in AJAX applications (e.g., DOM violations, error message occurrences), through (DOM-based) generic and application-specific invariants that serve as oracles.
2. A series of generic invariant types (e.g., XPath, template-based Regular Expression, JAVASCRIPT expression) for expressing web application invariants for testing.
3. An algorithm for deriving a test suite achieving all transitions coverage of the state-flow graph obtained during crawling. The resulting test suite can be refined manually to add test cases for specific paths or states, and can be used to conduct regression testing of AJAX applications.
4. An extension of our open-source AJAX crawler, CRAWLJAX and the implementation of the testing approach ATUSA, offering generic invariant checking components as well as a plugin-mechanism to add application-specific state validators and test suite generation.
5. An empirical evaluation, by means of three case studies, of the fault revealing capabilities and the scalability of the approach, as well as the level of automation that can be achieved and manual effort required to use the approach.

Given the growing popularity of AJAX applications, we see many opportunities for using ATUSA in practice. Furthermore, the open source and plugin-based nature makes our tool a suitable vehicle for other researchers interested in experimenting with other new techniques for testing AJAX applications.

Our future work will include conducting further case studies, as well as the development of more testing plugins for spotting development errors and security vulnerabilities in *Web 2.0* applications. Automatically detecting dynamic structural and JAVASCRIPT invariants in modern web applications [14] is another route we will be pursuing in future work.

31. Exact http://www.exact.com.
32. http://www.google.com/adsense.

## ACKNOWLEDGMENTS

The authors are grateful to Cor-Paul Bezemer, Stefan Lenselink, and Frank Groeneveld for their contributions in improving the CRAWLJAX core implementation. They would also like to thank the anonymous reviewers for their valuable and constructive comments.

## REFERENCES

[1] A. Andrews, J. Offutt, and R. Alexander, "Testing Web Applications by Modeling with FSMs," *Software and Systems Modeling,* vol. 4, no. 3, pp. 326-345, July 2005.

[2] S. Artzi, A. Kieżun, J. Dolby, F. Tip, D. Dig, A. Paradkar, and M.D. Ernst, "Finding Bugs in Dynamic Web Applications," *Proc. Int'l Symp. Software Testing and Analysis,* pp. 261-272, 2008.

[3] M. Barnett, R. Deline, M. Fahndrich, K. Rustan, M. Leino, and W. Schulte, "Verification of Object-Oriented Programs with Invariants," *J. Object Technology,* vol. 3, no. 6, pp. 1-30, 2004.

[4] M. Benedikt, J. Freire, and P. Godefroid, "VeriWeb: Automatically Testing Dynamic Web Sites," *Proc. 11th Int'l Conf. World Wide Web,* pp. 654-668, 2002.

[5] A. Bertolino, "Software Testing Research: Achievements, Challenges, Dreams," *Proc. ICSE Future of Software Eng.,* pp. 85-103, 2007.

[6] C.P. Bezemer, A. Mesbah, and A. van Deursen, "Automated Security Testing of Web Widget Interactions," *Proc. Seventh Joint Meeting of the European Software Eng. Conf. and the ACM SIGSOFT Symp. the Foundations of Software Eng.,* pp. 81-91, 2009.

[7] R.V. Binder, *Testing Object-Oriented Systems: Models, Patterns, and Tools.* Addison-Wesley, 1999.

[8] L.A. Clarke and D.S. Rosenblum, "A Historical Perspective on Runtime Assertion Checking in Software development," *ACM SIGSOFT Software Eng. Notes,* vol. 31, no. 3, pp. 25-37, 2006.

[9] L. de Alfaro, "Model Checking the World Wide Web," *Proc. 13th Int'l Conf. Computer Aided Verification,* pp. 337-349, 2001.

[10] L. de Alfaro, T.A. Henzinger, and F.Y.C. Mang, "MCWEB: A Model-Checking Tool for Web Site Debugging," *Proc. World Wide Web Conf.,* posters, 2001.

[11] S. Elbaum, G. Rothermel, S. Karre, and M. Fisher II, "Leveraging User-Session Data to Support Web Application Testing," *IEEE Trans. Software Eng.,* vol. 31, no. 3, pp. 187-202, Mar. 2005.

[12] M.D. Ernst, J. Cockrell, W.G. Griswold, and D. Notkin, "Dynamically Discovering Likely Program Invariants to Support Program Evolution," *IEEE Trans. Software Eng.,* vol. 27, no. 2, pp. 99-123, Feb. 2001.

[13] J. Garrett, "Ajax: A New Approach to Web Applications," adaptive path, http://www.adaptivepath.com/publications/essays/archives/000385.php, Feb. 2005.

[14] F. Groeneveld, A. Mesbah, and A. van Deursen, "Automatic Invariant Detection in Dynamic Web Applications," Technical Report TUD-SERG-2010-037, Delft Univ. of Technology, 2010.

[15] W. Halfond, S. Anand, and A. Orso, "Precise Interface Identification to Improve Testing and Analysis of Web Applications," *Proc. 18th Int'l Symp. Software Testing and Analysis,* pp. 285-296, 2009.

[16] W. Halfond and A. Orso, "Improving Test Case Generation for Web Applications Using Automated Interface Discovery," *Proc. Sixth Joint Meeting of the European Software Eng. Conf. and the ACM SIGSOFT Symp. the Foundations of Software Eng.,* pp. 145-154, 2007.

[17] W. Halfond and A. Orso, "Automated Identification of Parameter Mismatches in Web Applications," *Proc. 16th Int'l Symp. Foundations of Software Eng.,* pp. 181-191, 2008.

[18] Y.W. Huang, C.H. Tsai, T.P. Lin, S.K. Huang, D.T. Lee, and S.Y. Kuo, "A Testing Framework for Web Application Security Assessment," *J. Computer Networks,* vol. 48, no. 5, pp. 739-761, 2005.

[19] S. Kals, E. Kirda, C. Kruegel, and N. Jovanovic, "Secubat: A Web Vulnerability Scanner," *Proc. 15th Int'l Conf. World Wide Web,* pp. 247-256, 2006.

[20] V.L. Levenshtein, "Binary Codes Capable of Correcting Deletions, Insertions, and Reversals," *Cybernetics and Control Theory,* vol. 10, pp. 707-710, 1996.

[21] A. Marchetto, F. Ricca, and P. Tonella, "A Case Study-Based Comparison of Web Testing Techniques Applied to Ajax Web Applications," *Int'l J. Software Tools for Technology Transfer,* vol. 10, no. 6, pp. 477-492, 2008.

[22] A. Marchetto, P. Tonella, and F. Ricca, "State-Based Testing of Ajax Web Applications," *Proc. IEEE First Int'l Conf. Software Testing Verification and Validation,* pp. 121-130, 2008.

[23] A. Memon, "An Event-Flow Model of GUI-Based Applications for Testing: Research Articles," *Software Testing, Verification and Reliability,* vol. 17, no. 3, pp. 137-157, 2007.

[24] A. Mesbah, E. Bozdag, and A. van Deursen, "Crawling Ajax by Inferring User Interface State Changes," *Proc. Eighth Int'l Conf. Web Eng.,* pp. 122-134, 2008.

[25] A. Mesbah and A. van Deursen, "Migrating Multi-Page Web Applications to Single-Page Ajax Interfaces," *Proc. 11th European Conf. Software Maintenance and Reeng.,* pp. 181-190, 2007.

[26] A. Mesbah and M. Prasad, "Automated Cross-Browser Compatibility Testing," *Proc. 33rd Int'l Conf. Software Eng.,* 2011.

[27] A. Mesbah and A. van Deursen, "A Component- and Push-Based Architectural Style for Ajax Applications," *J. Systems and Software,* vol. 81, no. 12, pp. 2194-2209, 2008.

[28] A. Mesbah and A. van Deursen, "Invariant-Based Automatic Testing of Ajax User Interfaces," *Proc. IEEE 31st Int'l Conf. Software Eng.,* pp. 210-220, 2009.

[29] B. Meyer, "Seven Principles of Software Testing," *Computer,* vol. 41, no. 8, pp. 99-101, Aug. 2008.

[30] K. Pattabiraman and B. Zorn, "DoDOM: Leveraging DOM Invariants for Web 2.0 Application Robustness Testing," *Proc. IEEE 21st Int'l Conf. Software Reliability Eng.,* 2010.

[31] F. Ricca and P. Tonella, "Analysis and Testing of Web Applications," *Proc. 23rd Int'l Conf. Software Eng.,* pp. 25-34, 2001.

[32] D. Roest, "Automated Regression Testing of Ajax Web Applications," master's thesis, Delft Univ. of Technology, Feb. 2010.

[33] D. Roest, A. Mesbah, and A. van Deursen, "Regression Testing Ajax Applications: Coping with Dynamism," *Proc. Third Int'l Conf. Software Testing, Verification and Validation,* pp. 128-136, 2010.

[34] S. Sprenkle, E. Gibson, S. Sampath, and L. Pollock, "Automated Replay and Failure Detection for Web Applications," *Proc. IEEE/ACM 20th Int'l Conf. Automated Software Eng.,* pp. 253-262, 2005.

[35] S. Sprenkle, L. Pollock, H. Esquivel, B. Hazelwood, and S. Ecott, "Automated Oracle Comparators for Testing Web Applications," *Proc. IEEE 18th Int'l Symp. Software Reliability,* pp. 117-126, 2007.

[36] B. Stepien, L. Peyton, and P. Xiong, "Framework Testing of Web Applications Using TTCN-3," *Int'l J. Software Tools for Technology Transfer,* vol. 10, no. 4, pp. 371-381, 2008.

[37] G. Wassermann and Z. Su, "Static Detection of Cross-Site Scripting Vulnerabilities," *Proc. 30th Int'l Conf. Software Eng.,* pp. 171-180, 2008.

[38] E.J. Weyuker, "On Testing Non-Testable Programs," *The Computer J.,* vol. 25, no. 4, pp. 465-470, 1982.

[39] J.Y. Yen, "Finding the k Shortest Loopless Paths in a Network," *Management Science,* vol. 17, no. 11, pp. 712-716, 1971.

[40] R.K. Yin, *Case Study Research: Design and Methods,* third ed. Sage Publications, Inc., 2003.

**Ali Mesbah** received the PhD degree in computer science from Delft University of Technology in 2009. He is an assistant professor of software engineering at the University of British Columbia (UBC). From 2001 to 2005, he was a software engineer at WEST consulting in The Netherlands, where he was responsible for development and maintenance of software systems. His main area of research is software engineering, with an emphasis on software analysis and testing of modern web-based systems. He was the recipient of an ACM SIGSOFT Distinguished Paper Award at the ACM/IEEE International Conference on Software Engineering (ICSE '09). He is a member of the IEEE Computer Society.

**Arie van Deursen** received the MSc degree in computer science in 1990 from the Vrije Universiteit, Amsterdam, and the PhD degree from the University of Amsterdam in 1994. He is a full professor at Delft University of Technology, where he heads the Software Engineering Research Group. From 1996 to 2006, he was a research leader at CWI, the Dutch National Institute for Research in Mathematics in Computer Science. His research interests include reverse engineering, program comprehension, and software architecture. He was a program chair of the Working Conferenceon Reverse Engineering (WCRE) in 2002 and 2003, and has served on numerous program committees in the areas of software evolution, maintenance, and software engineering in general. He is a member of the editorial board of the *Empirical Software Engineering Journal* and a member of the IEEE Computer Society.

**Danny Roest** received the MSc degree in computer science from Delft University of Technology in 2010. He is currently a software engineer at a software company in The Netherlands. He has a great interest in developing web applications and testing in general. Creating user friendly applications and frameworks of high quality are among his passions.

# APPENDICE B

# DEUXIÈME ANNEXE

# Invariant-Based Automatic Testing of AJAX User Interfaces

Ali Mesbah
*Software Engineering Research Group*
*Delft University of Technology*
*The Netherlands*
A.Mesbah@tudelft.nl

Arie van Deursen
*Software Engineering Research Group*
*Delft University of Technology*
*The Netherlands*
Arie.vanDeursen@tudelft.nl

## Abstract

*AJAX-based Web 2.0 applications rely on stateful asynchronous client/server communication, and client-side runtime manipulation of the DOM tree. This not only makes them fundamentally different from traditional web applications, but also more error-prone and harder to test.*

*We propose a method for testing AJAX applications automatically, based on a crawler to infer a flow graph for all (client-side) user interface states. We identify AJAX-specific faults that can occur in such states (related to DOM validity, error messages, discoverability, back-button compatibility, etc.) as well as DOM-tree invariants that can serve as oracle to detect such faults. We implemented our approach in ATUSA, a tool offering generic invariant checking components, a plugin-mechanism to add application-specific state validators, and generation of a test suite covering the paths obtained during crawling. We describe two case studies evaluating the fault revealing capabilities, scalability, required manual effort and level of automation of our approach.*

## 1 Introduction

Recently, many new web trends have appeared under the *Web 2.0* umbrella, changing the web significantly, from read-only static pages to dynamic user-created content and rich interaction. Many *Web 2.0* sites rely heavily on AJAX (Asynchronous JAVASCRIPT and XML) [8], a prominent enabling technology in which a clever combination of JAVASCRIPT and Document Object Model (DOM) manipulation, along with asynchronous client/server delta-communication [16] is used to achieve a high level of user interactivity on the web.

With this new change comes a whole set of new challenges, mainly due to the fact that AJAX shatters the metaphor of a web 'page' upon which many classic web technologies are based. One of these challenges is testing such applications [6, 12, 14]. With the ever-increasing demands on the quality of *Web 2.0* applications, new techniques and models need to be developed to test this new class of software. How to automate such a testing technique is the question that we address in this paper.

In order to detect a fault, a testing method should meet the following conditions [18, 20]: *reach* the fault-execution, which causes the fault to be executed, *trigger* the error-creation, which causes the fault execution to generate an incorrect intermediate state, and *propagate* the error, which enables the incorrect intermediate state to propagate to the output and cause a detectable output error.

Meeting these reach/trigger/propagate conditions is more difficult for AJAX applications compared to classical web applications. During the past years, the general approach in testing web applications has been to request a response from the server (via a hypertext link) and to analyze the resulting HTML. This testing approach based on the page-sequence paradigm has serious limitations meeting even the first (reach) condition on AJAX sites. Recent tools such as Selenium[1] use a capture/replay style for testing AJAX applications. Although such tools are capable of executing the fault, they demand a substantial amount of manual effort on the part of the tester.

Static analysis techniques have limitations in revealing faults which are due to the complex run-time behavior of modern rich web applications. It is this dynamic run-time interaction that is believed [10] to make testing such applications a challenging task. On the other hand, when applying dynamic analysis on this new domain of web, the main difficulty lies in detecting the various doorways to different dynamic states and providing proper interface mechanisms for input values.

In this paper, we discuss challenges of testing AJAX (Section 3) and propose an automated testing technique for finding faults in AJAX user interfaces. We extend our AJAX crawler, CRAWLJAX (Sections 4–5), to infer a state-flow

[1] http://selenium.openqa.org

*ICSE'09*, May 16–24, 2009, Vancouver, Canada
978-1-4244-3452-7/09/$25.00 © 2009 IEEE

graph for all (client-side) user interface states. We identify AJAX-specific faults that can occur in such states and generic and application-specific invariants that can serve as oracle to detect such faults (Section 6). From the inferred graph, we automatically generate test cases (Section 7) that cover the paths discovered during the crawling process. In addition, we use our open source tool called ATUSA (Section 8), implementing the testing technique, to conduct a number of case studies (Section 9) to discuss (Section 10) and evaluate the effectiveness of our approach.

## 2 Related Work

Modern web interfaces incorporate client-side scripting and user interface manipulation which is increasingly separated from server-side application logic [23]. Although the field of rich web interface testing is mainly unexplored, much knowledge may be derived from two closely related fields: traditional web testing and GUI application testing.

**Traditional Web Testing.** Benedikt *et al.* [3] present VeriWeb, a tool for automatically exploring paths of multi-page web sites through a crawler and detector for abnormalities such as navigation and page errors (which are configurable through plugins). VeriWeb uses SmartProfiles to extract candidate input values for form-based pages. Although VeriWeb's crawling algorithm has some support for client-side scripting execution, the paper provides insufficient detail to determine whether it would be able to cope with modern AJAX web applications. VeriWeb offers no support for generating test suites as we do in Section 7.

Tools such as WAVES [10] and SecuBat [11] have been proposed for automatically assessing web application security. The general approach is based on a crawler capable of detecting data entry points which can be seen as possible points of security attack. Malicious patterns, e.g., SQL and XSS vulnerabilities, are then injected into these entry points and the response from the server is analyzed to determine vulnerable parts of the web application.

A model-based testing approach for web applications was proposed by Ricca and Tonella [19]. They introduce ReWeb, a tool for creating a model of the web application in UML, which is used along with defined coverage criteria to generate test-cases. Another approach was presented by Andrews *et al.* [1], who rely on a finite state machine together with constraints defined by the tester. All such model-based testing techniques focus on classical multi-page web applications. They mostly use a crawler to infer a navigational model of the web. Unfortunately, traditional web crawlers are not able to crawl AJAX applications [14].

Logging user session data on the server is also used for the purpose of automatic test generation [7, 21]. This approach requires sufficient interaction of real web users with the system to generate the necessary logging data. Session-based testing techniques are merely focused on synchronous requests to the server and lack the complete state information required in AJAX testing. Delta-server messages [16] from the server response are hard to analyze on their own. Most of such delta updates become meaningful after they have been processed by the client-side engine on the browser and injected into the DOM.

Exploiting static analysis of server-side implementation logic to abstract the application behavior is another testing approach. Artzi *et al.* [2] propose a technique and a tool called Apollo for finding faults in PHP web applications that is based on combined concrete and symbolic execution. The tool is able to detect run-time errors and malformed HTML output. Halfond and Orso [9] present their static analysis of server-side Java code to extract web application request parameters and their potential values. Such techniques have limitations in revealing faults that are due to the complex run-time behavior of modern rich web applications.

**GUI Application Testing.** Reverse engineering a model of the desktop (GUI), to generate test cases has been proposed by Memon *et al.* [13]. AJAX applications can be seen as a hybrid of desktop and web applications [16], since the user interface is composed of components and the interaction is event-based. However, AJAX applications have specific features, such as the asynchronous client/-server communication and dynamic DOM-based user interface, which make them different from traditional GUI applications [12], and therefore require other testing tools and techniques.

**Current AJAX Testing Approaches.** The server-side of AJAX applications can be tested with any conventional testing technique. On the client, testing can be performed at different levels. Unit testing tools such as JsUnit[2] can be used to test JAVASCRIPT on a functional level. The most popular AJAX testing tools are currently capture/replay tools such as Seleninum, WebKing[3], and Sahi[4], which allow DOM-based testing by capturing events fired by user (tester) interaction. Such tools have access to the DOM, and can assert expected UI behavior defined by the tester and replay the events. Capture/replay tools demand, however, a substantial amount of manual effort on the part of the tester [13].

Marchetto *et al.* [12] have recently proposed an approach for state-based testing of AJAX applications. They use traces of the application to construct a finite state machine. Sequences of semantically interacting events in the model are used to generate test cases once the model is refined by the tester. In our approach, we crawl the AJAX application, simulating real user events on the user interface and infer the abstract model automatically.

[2] http://jsunit.net

[3] http://www.parasoft.com/jsp/products/home.jsp?product=WebKing

[4] http://sahi.co.in/w/

## 3 AJAX Testing Challenges

In AJAX applications, the state of the user interface is determined dynamically, through event-driven changes in the browser's DOM that are only visible after executing the corresponding JAVASCRIPT code. The resulting challenges can be explained through the reach/trigger/propagate conditions as follows.

**Reach.** The event-driven nature of AJAX presents the first serious testing difficulty, as the event model of the browser must be manipulated instead of just constructing and sending appropriate URLs to the server. Thus, simulating user events on AJAX interfaces requires an environment equipped with all the necessary technologies, e.g., JAVASCRIPT, DOM, and the XMLHttpRequest object used for asynchronous communication.

One way to *reach* the fault-execution automatically for AJAX is by adopting a web crawler, capable of detecting and firing events on clickable elements on the web interface. Such a crawler should be able to exercise all user interface events of an AJAX site, crawl through different UI states and infer a model of the navigational paths and states. We proposed such a crawler for AJAX, discussed in our previous work [14], which will be briefly explained in Section 4.

**Trigger.** Once we are able to derive different dynamic states of an AJAX application, possible faults can be triggered by generating UI events. In addition input values can cause faulty states. Thus, it is important to identify input data entry points, which are primarily comprised of DOM forms. In addition, executing different sequences of events can also trigger an incorrect state. Therefore, we should be able to generate and execute different event sequences.

**Propagate.** In AJAX, any response to a client-side event is injected into the single-page interface and therefore, faults propagate to and are manifested at the DOM level. Hence, access to the dynamic run-time DOM is a necessity to be able to analyze and detect the propagated errors.

Automating the process of assessing the correctness of test case output is a challenging task, known as the oracle problem [24]. Ideally a tester acts as an oracle who knows the expected output, in terms of DOM tree, elements and their attributes, after each state change. When the state space is huge, it becomes practically impossible. In practice, a baseline version, also known as the Gold Standard [5], of the application is used to generate the expected behavior. Oracles used in the web testing literature are mainly in the form of HTML comparators [22] and validators [2].

## 4 Deriving AJAX States

Here, we briefly outline our AJAX crawling technique and tool called CRAWLJAX [14]. CRAWLJAX can exercise client side code, and identify clickable elements that change the state within the browser's dynamically built DOM. From these state changes, we infer a *state-flow graph*, which captures the states of the user interface, and the possible event-based transitions between them.

We define an AJAX UI state change as a change on the DOM tree caused either by server-side state changes propagated to the client, or client-side events handled by the AJAX engine. We model such changes by recording the paths (events) to these DOM changes to be able to navigate between the different states.

**Inferring the State Machine.** The state-flow graph is created incrementally. Initially, it only contains the root state and new states are created and added as the application is crawled and state changes are analyzed. The following components participate in the construction of the graph: CRAWLJAX uses an *embedded browser* interface (with different implementations: IE, Mozilla) supporting technologies required by AJAX; A *robot* is used to simulate user input (e.g., `click`, `mouseOver`, text input) on the embedded browser; The *finite state machine* is a data component maintaining the state-flow graph, as well as a pointer to the current state; The *controller* has access to the browser's DOM and analyzes and detects state changes. It also controls the robot's actions and is responsible for updating the state machine when relevant changes occur on the DOM. The algorithm used by these components to actually infer the state machine is discussed below: the full algorithm along with its testing-specific extensions is shown in Algorithms 1 and 2 (Section 8).

**Detecting Clickables.** CRAWLJAX implements an algorithm which makes use of a set of *candidate elements*, which are all exposed to an event type (e.g., click, mouseOver). In automatic mode, the candidate clickables are labeled as such based on their HTML tag element name and attribute constraints. For instance, all elements with a tag `div`, `a`, and `span` having attribute `class="menuitem"` are considered as candidate clickable. For each candidate element, the crawler fires a click on the element (or other event types, e.g., mouseOver), in the embedded browser.

**Creating States.** After firing an event on a candidate clickable, the algorithm compares the resulting DOM tree with the way as it was just before the event fired, in order to determine whether the event results in a state change. If a change is detected according to the Levenshtein edit distance, a new state is created and added to the state-flow graph of the state machine. Furthermore, a new edge is created on the graph between the state before the event and the current state.

**Processing Document Tree Deltas.** After a new state has been detected, the crawling procedure is recursively called to find new possible states in the partial changes made to the DOM tree. CRAWLJAX computes the differences between the previous document tree and the current one, by means of an enhanced *Diff* algorithm to detect AJAX par-

tial updates which may be due to a server request call that injects new elements into the DOM.

**Navigating the States.** Upon completion of the recursive call, the browser should be put back into the previous state. A dynamically changed DOM state does not register itself with the browser history engine automatically, so triggering the 'Back' function of the browser is usually insufficient. To deal with this AJAX crawling problem, we save information about the elements and the order in which their execution results in reaching a given state. We then can reload the application and follow and execute the elements from the initial state to the desired state. CRAWLJAX adopts XPath to provide a reliable, and persistent element identification mechanism. For each state changing element, it reverse engineers the XPath expression of that element which returns its exact location on the DOM. This expression is saved in the state machine and used to find the element after a reload. Note that because of side effects of the element execution and server-side state, there is no guarantee that we reach the exact same state when we traverse a path a second time. It is, however, as close as we can get.

# 5 Data Entry Points

In order to provide input values on AJAX web applications, we have adopted a reverse engineering process, similar to [3, 10], to extract all exposed data entry points. To this end, we have extended our crawler with the capability of detecting *DOM forms* on each newly detected state (this extension is also shown in Algorithm 1).

For each new state, we extract all form elements from the DOM tree. For each form, a `hashcode` is calculated on the attributes (if available) and the HTML structure of the input fields of the form. With this hashcode, custom values are associated and stored in a database, which are used for all forms with the same code.

If no custom data fields are available yet, all data, including input fields, their default values, and options are extracted from the DOM form. Since in AJAX forms are usually sent to the server through JAVASCRIPT functions, the `action` attribute of the form does not always correspond to the server-side entry URL. Also, any element (e.g., A, DIV) could be used to trigger the right JAVASCRIPT function to submit the form. In this case, the crawler tries to identify the element that is responsible for form submission. Note that the tester can always verify the *submit* element and change it in the database, if necessary. Once all necessary data is gathered, the form is inserted automatically into the database. Every input form provides thus a data entry point and the tester can later alter the database with additional desired input values for each form.

If the crawler does find a match in the database, the input values are used to fill the DOM form and submit it. Upon submission, the resulting state is analyzed recursively by the crawler and if a valid state change occurs the state-flow graph is updated accordingly.

# 6 Testing AJAX States Through Invariants

With access to different dynamic DOM states we can check the user interface against different constraints. We propose to express those as invariants on the DOM tree, which we thus can check automatically in any state. We distinguish between invariants on the DOM-tree, between DOM-tree states, and application-specific invariants. Each invariant is based on a fault model [5], representing AJAX-specific faults that are likely to occur and which can be captured through the given invariant.

## 6.1 Generic DOM Invariants

**Validated DOM.** Malformed HTML code can be the cause of many vulnerability and browser portability problems. Although browsers are designed to tolerate HTML malformedness to some extent, such errors have led to browser crashes and security vulnerabilities [2]. All current HTML validators expect all the structure and content be present in the HTML source code. However, with AJAX, changes are manifested on the single-page user interface by partially updating the dynamic DOM through JAVASCRIPT. Since these validators cannot execute client-side JAVASCRIPT, they simply cannot perform any kind of validation.

To prevent faults, we must make sure that the application has a valid DOM on every possible execution path and modification step. We use the DOM tree obtained after each state change while crawling and transform it to the corresponding HTML instance. A W3C HTML validator serves as oracle to determine whether errors or warnings occur. Since most AJAX sites rely on a single-page interface, we use a *diff* algorithm to prevent duplicate occurrences of failures that may be the result of a previous state.

**No Error Messages in DOM.** Our state should never contain a string pattern that suggests an error message [3] in the DOM. Error messages that are injected into the DOM as a result of client-side (e.g., 404 Not Found, 400 Bad Request) or server-side errors (e.g., Session Timeout, 500 Internal Server Error, MySQL error) can be detected automatically. The prescribed list of potential fault patterns should be configurable by the tester.

**Other Invariants.** In line with the above, further generic DOM-invariants can be devised, for example to deal with accessibility, link discoverability, or security constraints on the DOM at any time throughout the crawling process. We omit discussion of these invariants due to space limitations.

### 6.2 State Machine Invariants

Besides constraints on the DOM-tree in individual states, we can identify requirements on the state machine and its transitions.

**No Dead Clickables.** One common fault in classical web applications is the occurrence of *dead links* which point to a URL that is permanently unavailable. In AJAX, clickables that are supposed to change the state by retrieving data from the server, through JAVASCRIPT in the background, can also be broken. Such error messages from the server are mostly swallowed by the AJAX engine, and no sign of a dead link is propagated to the user interface. By listening to the client/server request/response traffic after each event (e.g., through a proxy), dead clickables can be detected.

**Consistent Back-Button.** A fault that often occurs in AJAX applications is the broken Back-button of the browser. As explained in Section 4, a dynamically changed DOM state does not register itself with the browser history engine automatically, so triggering the 'Back' function makes the browser completely leave the application's web page. It is possible to programatically register each state change with the browser history and frameworks are appearing which handle this issue. However, when the state space increases, errors can be made and some states may be ignored by the developer to be registered properly. Through crawling, upon each new state, one can compare the expected state in the graph with the state after the execution of the Back-button and find inconsistencies automatically.

### 6.3 Application-specific Invariants

We can define invariants that should always hold and could be checked on the DOM, specific to our AJAX application in development. In our case study, Section 9.2, we describe a number of application-specific invariants. Constraints over the DOM-tree can be easily expressed as invariants in Java, for example through an XPath expression. Typically, this can be coded into one or two simple Java methods. The resulting invariants can be used to dynamically search for invariant violations.

## 7 Testing AJAX Paths

While running the crawler to derive the state machine can be considered as a first full test pass, the state machine itself can be further used for testing purposes. For example, it can be used to execute different paths to cover the state machine in different ways. In this section, we explain how to derive a test suite (implemented in JUnit) automatically from the state machine, and how this suite can be used for testing purposes.

```
@Test
public void testcase1() {
browser.goToUrl(url);

/*Element-info: SPAN class=expandable-hitarea */
browser.fireEvent(new Eventable(new Identification(
  "xpath", "//DIV[1]/SPAN[4]"), "onclick"));

Comp.AssertEquals(oracle.getState("S_1").getDom(),
  browser.getDom());

/*Element-info: DIV class=hitarea id=menuitem2 */
browser.fireEvent(new Eventable(new Identification(
  "xpath", "//SPAN[2]/DIV[2]"), "onmouseover"));

Comp.AssertEquals(oracle.getState("S_3").getDom(),
              browser.getDom());

/*Element-info: Form, A href=#submit */
handleForm(2473584);

Comp.AssertEquals(oracle.getState("S_4").getDom(),
              browser.getDom());
}

private void handleForm(long formId) {
 Form form = oracle.getForm(formId);
 if (form != null) {
  FormHandler.fillFormInDom(browser, form);
  browser.fireEvent(form.getSubmit());
 }
}
```

**Figure 1. A generated JUnit test case.**

To generate the test suite, we use the *K shortest paths* [25] algorithm which is a generalization of the shortest path problem in which several paths in increasing order of length are sought. We collect all sinks in our graph, and compute the shortest path from the index page to each of them. Loops are included once. This way, we can easily achieve all transitions coverage.

Next, we transform each path found into a JUnit test case, as shown in Figure 1. Each test case captures the sequence of events from the initial state to the target state. The JUnit test case can fire events, since each edge on the state-flow graph contains information about the event-type and the element the event is fired on to arrive at the target state. We also provide all the information about the clickable element such as tag name and attributes, as code comments in the generated test method. The test class provides API's to access the DOM (`browser.getDom()`) and elements (`browser.getElementBy(how, value)`) of the resulting state after each event, as well as its contents.

If an event is a form submission (annotated on the edge), we generate all the required information for the test case to retrieve the corresponding input values from the database and insert them into the DOM, before triggering the event.

After each event invocation the resulting state in the browser is compared with the expected state in the database which serves as oracle. The comparison can take place at different levels of abstraction ranging from textual [22] to schema-based similarity [15].

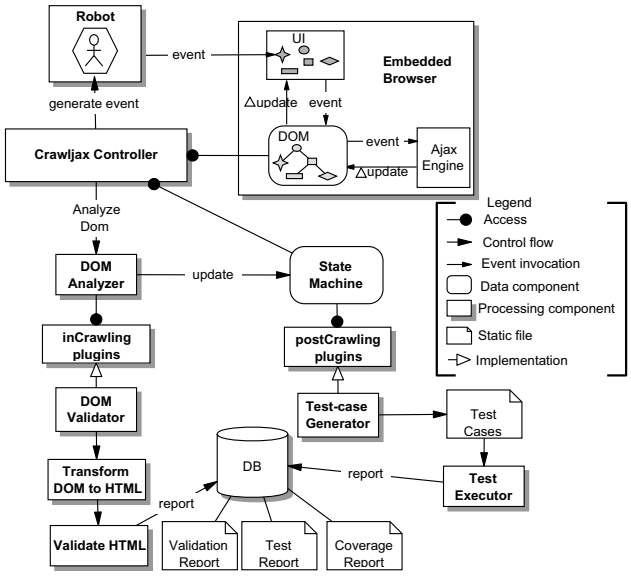



**Figure 2. Processing view of ATUSA.**

**Algorithm 1** Pre/postCrawling hooks

```
 1: procedure START (url, Set tags)
 2: browser ← initEmbeddedBrowser(url)
 3: robot ← initRobot()
 4: sm ← initStateMachine()
 5: preCrawlingPlugins(browser)
 6: crawl(null)
 7: postCrawlingPlugins(sm)
 8: end procedure
 9: procedure CRAWL (State ps)
10: cs ← sm.getCurrentState()
11: Δupdate ← diff(ps, cs)
12: analyseForms(Δupdate)
13: Set C ← getCandidateClickables(Δupdate, tags)
14: for c ∈ C do
15:     generateEvent(cs, c)
16: end for
17: end procedure
18: procedure ANALYSEFORMS (State cs)
19: for form ∈ cs.getForms() do
20:     id ← getHashCode(form)
21:     dbForm ← database.getForm(id)
22:     if dbForm == null then
23:         extractInsertForm(form, id)
24:     else
25:         fillFormInDom(browser, dbForm)
26:         generateEvent(cs, dbForm.getSubmit())
27:     end if
28: end for
29: end procedure
```

**Test-case Execution.** Usually extra coding is necessary for simulating the environment where the tests will be run, which contributes to the high cost of testing [4]. We provide a framework to run all the generated tests automatically using a real web browser and generate success/failure reports. At the beginning of each test case the embedded browser is initialized with the URL of the AJAX site under test. For each test case, the browser is first put in its initial index state. From there, events are fired on the clickable elements (and forms filled if present). After each event invocation, assertions are checked to see if the expected results are seen on the web application's new UI state.

The generated JUnit test suite can be used in several ways. First, it can be run as is on the current version of the AJAX application, but for instance with a different browser to detect browser incompatibilities. Furthermore, the test suite can be applied to altered versions of the AJAX application to support regression testing: For the unaltered user interface, the test cases should pass, and only for altered user interface code failures might occur (also helping the tester to understand what has truly changed). The typical use of the derived test suite will be to take apart specific generated test cases, and augment them with application-specific assertions. In this way, a small test suite arises capturing specific fault-sensitive click trails.

## 8 Tool Implementation: ATUSA

We have implemented our testing approach in an open source tool called ATUSA (Automatically Testing UI States of AJAX), available through our website.[5] It is based on the crawling capabilities of CRAWLJAX and provides plugin *hooks* for testing AJAX applications at different levels. Its architecture can be divided into three phases:

**preCrawling** occurs after the application has fully been loaded into the browser. Examples include authentication plugins to log onto the system and checks on the HTML source code.

**inCrawling** occurs after each detected state change, different types of invariants can be checked through plugins such as Validated DOM, Consistent Back-button, and No Error Messages in DOM.

**postCrawling** occurs after the crawling process is done and the state-flow graph is inferred fully. The graph can be used, for instance, in a plugin to generate test cases from.

Algorithms 1 and 2 show the hooks along the crawling process. For each phase, ATUSA provides the tester with specific APIs to implement plugins for validation and fault detection. ATUSA offers generic invariant checking components, a plugin-mechanism to add application-specific state validators, and generation of a test suite from the inferred state-flow graph. Figure 2 depicts the processing view of ATUSA, showing only the DOM Validator and Test Case Generator as examples of possible plugin implementations.

ATUSA supports looking for many different types of faults in AJAX-based applications, from errors in the DOM instance, to errors that involve the navigational path, e.g.,

[5] http://spci.st.ewi.tudelft.nl/atusa/

**Algorithm 2** Incrawling hook while deriving AJAX states

```
1:  procedure GENERATEEVENT (State cs, Clickable c)
2:  robot.fireEvent(c)
3:  dom ← browser.getDom()
4:  if distance(cs.getDom(), dom) > τ then
5:      xe ← getXpathExpr(c)
6:      ns ← State(dom)
7:      sm.addState(ns)
8:      sm.addEdge(cs, ns, Event(c, xe))
9:      sm.changeState(ns)
10:     inCrawlingPlugins(ns)
11:     crawl(cs)
12:     sm.changeState(cs)
13:     if browser.history.canBack then
14:         browser.history.goBack()
15:     else
16:         browser.reload()
17:         List E ← sm.getPathTo(cs)
18:         for e ∈ E do
19:             robot.fireEvent(e)
20:         end for
21:     end if
22: end if
23: end procedure
```

constraints on the length of the deepest paths [3], or number of clicks to a certain state. Whenever a fault is detected, the error report along the causing execution path is saved in the database so that it can be reproduced later easily.

**Implementation.** ATUSA is implemented in Java 1.6. The *state-flow graph* is based on the JGrapht library. The implementation details of the crawler can be found in [14]. The plugin architecture is implemented through the Java Plugin Framework (JPF) and we use Hibernate to store the data in the database. Apache Velocity templates assist us in the code generation process of JUnit test cases.

## 9 Empirical Evaluation

In order to assess the usefulness of our approach in supporting modern web application testing, we have conducted a number of case studies, set up following Yin's guidelines [26].

**Goal and Research Questions.** Our goal in this experiment is to evaluate the fault revealing capabilities, scalability, required manual effort and level of automation of our approach. Our research questions can be summarized as:

**RQ1** What is the fault revealing capability of ATUSA?

**RQ2** How well does ATUSA perform? Is it scalable?

**RQ3** What is the automation level when using ATUSA and how much manual effort is involved in the testing process?

### 9.1 Study 1: TUDU

Our first experimental subject is the AJAX-based open source TUDU [6] web application for managing personal todo lists, which has also been used by other researchers [12]. The server-side is based on J2EE and consists of around 12K lines of Java/JSP code, of which around 3K forms the presentation layer we are interested in. The client-side extends on a number of AJAX libraries such as DWR[7] and Scriptaculous[8], and consists of around 11k LOC of external JAVASCRIPT libraries and 580 internal LOC.

To address RQ3 we report the time spent on parts that required manual work. For RQ1-2, we configured ATUSA through its `properties` file (1 minute), setting the URL of the deployed site, the tag elements that should be included (`A, DIV`) and excluded (`A:title=Log out`) during the crawling process, the depth level (2), the similarity threshold (0.89), and a maximum crawling time of 60 minutes. Since TUDU requires authentication, we wrote (10 minutes) a `preCrawling` plugin to log into the web application automatically.

As shown in Table 1, we measure average DOM string size, number of candidate elements analyzed, detected clickables and states, detected data entry points, detected faults, number of generated test cases, and performance measurements, all of which are printed in a log file by ATUSA after each run.

In the initial run, after the login process, ATUSA crawled the TUDU application, finding the doorways to new states and detecting all possible data entry points recursively. We analyzed the data entry points in the database and provided each with custom input values (15 minutes to evaluate the input values and provide useful values). For the second run, we activated (50 seconds) the DOM Validator, Back-Button, Error Detector, and Test Case Generator plugins and started the process. ATUSA started crawling and when forms were encountered, the custom values from the database were automatically inserted into the browser and submitted. Upon each detected state change, the invariants were checked through the plugins and reports were inserted into the database if faults were found. At the end of the crawling process, a test suite was generated from the inferred state-flow graph.

To the best of our knowledge, there are currently no tools that can automatically test AJAX dynamic states. Therefore, it is not possible to form a base-line for comparison using, for instance, external crawlers. To assess the effectiveness of the generated test suite, we measure code coverage on the client as well as the presentation-tier of the server. Although the effectiveness is not directly implied by code coverage, it is an objective and commonly used indicator of the quality of a test suite [9]. To that end, we instrumented the presentation part of the server code (`tudu-dwr`) with Clover and the client-side JAVASCRIPT libraries with JSCoverage[9], and

[6] `http://tudu.sourceforge.net`

[7] `http://directwebremoting.org`

[8] `http://script.aculo.us`

[9] `http://siliconforks.com/jscoverage/`

| LOC Server-side | LOC Client-side | DOM string size | Candidate Clickables | Detected Clickables | Detected States | Detected Entry Points | DOM Violations | Back-button | Generated Test Cases | Coverage Server-side | Coverage Client-side | Detected Faults | Manual Effort | Performance |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 3k | 11k (ext)<br>580 (int) | 24908<br>(byte) | 332 | 42 | 34 | 4 forms<br>21 inputs | 182 | false | 32 | 73% | 35% (ext)<br>75% (int) | 80% | 26.5<br>(minutes) | 5.6<br>(minutes) |

**Table 1.** TUDU **case study.**

deployed the web application. For each test run, we bring the TUDU database to the original state using a SQL script. We run all the test cases against the instrumented application, through ATUSA's embedded browser, and compute the amount of coverage achieved for server- and client-side code. In addition, we manually seeded 10 faults, capable of causing inconsistent states (e.g., DOM malformdness, adding values longer than allowed by the database, adding duplicate todo items, removing all items instead of one) and measured the percentage of faults detected. The results are presented in Table 1.

**Findings.** Based on these observations we conclude that: The use of ATUSA can help to reveal generic faults, such as DOM violations, automatically; The generated test suite can give us useful code coverage (73% server-side and 75% client-side; Note that only partial parts of the external libraries are actually used by TUDU resulting in a low coverage percentage) and can reveal most DOM-based faults, 8 of the 10 seeded faults were detected, two faults were undetected because during the test execution, they were silently swallowed by the JAVASCRIPT engine and did not affect the DOM. It is worth mentioning that increasing the depth level to 3 significantly increased the measured crawling time passed the maximum 60 minutes, but dit not influence the fault detection results. The code coverage, however, improved by approximately 10%; The manual effort involved in setting up ATUSA (less than half an hour in this case) is minimal; The performance and scalability of the crawling and testing process is very acceptable (it takes ATUSA less than 6 minutes to crawl and test TUDU, analyzing 332 clickables and detecting 34 states).

## 9.2 Study 2: Finding Real-Life Bugs

Our second case study involves the development of an AJAX user interface in a small commercial project. We use this case study to evaluate the manual effort required to use ATUSA (RQ3), and to assess the capability of ATUSA to find faults that actually occurred during development (RQ1).

**Subject System.** The case at hand is Coachjezelf (CJZ, "Coach Yourself"),[10] a commercial application allowing high school teachers to assess and improve their teaching skills. CJZ is currently in use by 5000-6000 Dutch teachers, a number that is growing with approximately 1000 paying users every year.

The relevant part for our case is the interactive table of contents (TOC), which is to be synchronized with an actual content widget. In older versions of CJZ this was implemented through a Java applet; in the new version this is to be done through AJAX, in order to eliminate a Java virtual machine dependency.

The two developers working on the case study spent around one week (two person-weeks) building the AJAX solution, including requirements elicitation, design, understanding and evaluating the libraries to be used, manual testing, and acceptance by the customer.

The AJAX-based solution made use of the `jQuery`[11] library, as well as the `treeview`, `history-remote`, and `listen` plugins for `jQuery`. The libraries comprise around 10,000 lines of JAVASCRIPT, and the custom code is around 150 lines of JAVASCRIPT, as well as some HTML and CSS code.

**Case study setup.** The developers were asked (1) to try to document their design and technical requirements using invariants, and (2) to write the invariants in ATUSA plugins to detect errors made during development. After the delivery of the first release, we evaluated (1) how easy it was to express these invariants in ATUSA; and (2) whether the (generic or application-specific) plugins were capable of detecting faults.

**Application-Specific Invariants.** Two sets of invariants were proposed by the developers. The first essentially documented the (external) `treeview` component, capable of (un)folding tree structures (such as a table of contents).

The `treeview` component operates by setting HTML class attributes (such as `collapsible`, `hit-area`, and `lastExpandable-hitarea`) on nested list structures. The corresponding style sheet takes care of properly displaying the (un)folded (sub)trees, and the JAVASCRIPT intercepts clicks and re-arranges the class attributes as needed.

Invariants were devised to document constraints on the class attributes. As an example, the `div`-element immediately below a `li`-element that has the class `expandable` should have class `expandable-hitarea`. Another invari-

[10] See `www.coachjezelf.nl` for more information (in Dutch).

[11] `jquery.com`

| Failure | Cause | Violated Invariant | Invariant type |
|---|---|---|---|
| Images not displayed | Base URL in dynamic load | Dead Clickables | Generic |
| Broken synchronization in IE | Invalid HTML id | DOM-validator | Generic |
| Inconsistent history | Issue in `listen` library | Back-Button | Generic |
| Broken synchronization in IE | Backslash versus slash | Consistent current page | Specific |
| Corrupted table | Coding error | `treeview` invariants, Consistent current page | Specific |
| Missing TOC Entries | Incomplete input data | Consistent current page | Specific |

**Table 2. Faults found in CJZ-AJAX.**

```
//case one: warn about collapsible divs within expandable items
String xpathCase1 =  "//LI[contains(@class,'expandable')]/DIV[contains(@class,'collapsable')]";

//case two: warn about collapsible items within expandable items
String xpathCase2 = "//LI[contains(@class,'expandable')]/UL/LI[contains(@class,'collapsable')]";
```

**Figure 3. Example invariants expressed using XPath in Java.**

ant is that expandable list items (which are hidden) should have their CSS display type set to "none".

The second set of invariants specifically dealt with the code written by the developers themselves. This code took care of synchronizing the interactive display of the table of contents with the actual page shown. Clicking links within the page affects the display of the table of contents, and vice versa.

This resulted in essentially two invariants: one to ensure that within the table of contents at most one path (to the current page) would be open, and the other that at any time the current page as marked in the table of contents would actually be displayed in the content pane.

Expressing such invariants on the DOM-tree was quite easy, requiring a few lines of Java code using XPath. An example is shown in Figure 3.

**Failures Detected.** At the end of the development week, ATUSA was used to test the new AJAX interface. For each type of application-specific invariant, an `inCrawling` plugin was added to ATUSA. Six types of failures were automatically detected: three through the generic plugins, and three through the application-specific plugins just described. An overview of the type of failures found and the invariant violations that helped to detect them is provided in Table 2.

The application-specific failures were all found through two invariant types: the *Consistent current page*, which expresses that in any state the table and the actual content should be in sync, and the *treeview invariants*. Note that for certain types of faults, for instance the treeview corrupted table, a very specific click trail had to be followed to expose the failure. ATUSA gives no guarantee of covering the complete state of the application, however, since it tries a huge combination of clickables recursively, it was able to detect such faults, which were not seen by developers when the application was tested manually.

**Findings.** Based on these observations we conclude that: The use of ATUSA can help to reveal bugs that are likely to occur during AJAX development and are difficult to detect manually; Application-specific invariants can help to document and test the essence of an AJAX application, such as the synchronization between two widgets; The manual effort in coding such invariants in Java and using them through plugins in ATUSA is minimal.

## 10 Discussion

**Automation Scope.** User interface testing is a broad term, dealing with testing how the application and the user interact. This typically is manual in nature, as it includes inspecting the correct display of menus, dialog boxes, and the invocation of the correct functionality when clicking them. The type of user interface testing that we propose does not replace this manual testing, but augments it: Our focus is on finding programming faults, manifested through failures in the DOM tree. As we have seen, the highly dynamic nature and complexity of AJAX make it error-prone, and our approach is capable of finding such faults automatically.

**Invariants.** Our solution to the oracle problem is to include invariants (as also advocated by, e.g., Meyer [17]). AJAX applications offer a unique opportunity for specifying invariants, thanks to the central DOM data structure. Thus, we are able to define generic invariants that should hold for all AJAX applications, and we allow the tester to use the DOM to specify dedicated invariants. Furthermore, the state machine derived through crawling can be used to express invariants, such as correct Back-button behavior. Again, this state machine can be accessed by the tester to specify his or her own invariants. These invariants make our approach much more sophisticated than *smoke tests* for user interfaces (as proposed by e.g., Memon [13]) — which we can achieve thanks to the presence of the DOM and state machine data structures. Note that just running CRAWLJAX would correspond to conducting a smoke test: the difficulty with web applications (as opposed to, e.g., Java Swing applications) is that it is very hard to determine when a failure

occurs – which is solved in ATUSA through the use of invariants.

**Generated versus hand-coded JAVASCRIPT.** The case studies we conducted involve two different popular JAVASCRIPT libraries in combination with hand-written JAVASCRIPT code. Alternative frameworks exist, such as Google's Web Toolkit (GWT)[12] in which most of the client-side code is generated. ATUSA is entirely independent of the way the AJAX application is written, so it can be applied to such systems as well. This will be particularly relevant for testing the custom JAVASCRIPT code that remains to be hand-written, and which can still be tricky and error-prone. Furthermore, ATUSA can be used by the developers of such frameworks, to ensure that the generated DOM states are correct.

**Manual Effort.** The manual steps required to run ATUSA consist of configuration, plugin development, and providing custom input values, which for the cases conducted took less than an hour. The hardest part is deciding which application-specific invariants to adopt. This is a step that is directly connected with the *design* of the application itself. Making the structural invariants explicit not only allows for automated testing, it is also a powerful design documentation technique. Admittedly, not all web developers will be able to think in terms of invariants, which might limit the applicability of our approach in practice. Those capable of documenting invariants can take advantage of the framework ATUSA provides to actually implement the invariants.

**Performance and Scalability.** Since the state space of any realistic web application is huge and can cause the well-know *state explosion problem*, we provide the tester with a set of configurable options to constrain the state space such as the maximum search depth level, the similarity threshold, maximum number of states per domain, maximum crawling time, and the option of ignoring external links and links that match some pre-defined set of regular expressions.The main component that can influence the performance and scalability is the crawling part. The performance of ATUSA in crawling an AJAX site depends on many factors such as the speed at which the server can handle requests, how fast the client-side JAVASCRIPT can update the interface, and the size of the DOM tree. ATUSA can scale to sites comprised of thousands of states easily.

**Application Size.** The two case studies both involve around 10,000 lines of JAVASCRIPT library code, and several hundred lines of application code. One might wonder whether this is too small to be representative. However, our results are based on *dynamic* analysis rather than static code analysis, hence the amount of code is not the determining factor. Instead, the size of the derived state machine is the factor limiting the scalability of our approach, which is only moderately (if at all) related to the size of the JAVASCRIPT code.

**Threats to Validity.** Some of the issues concerning the *external* validity of our empirical evaluation have been covered in the above discussion on scope, generated code, application size, and scalability. Apart from the two case studies described in the paper, we conducted two more (on TaskFreak[13] and the Java PETSTORE 2.0[14]), which gave comparable results. With respect to *internal* validity, we minimized the chance of ATUSA errors by including a rigorous JUnit test suite. ATUSA, however, also makes use of many (complex) third party components, and we did encounter several problems in some of them. While these bugs do limit the current applicability of our approach, they do not affect the validity of our results. As far as the choice of faults in the first case study is concerned, we selected them form the TUDU bug tracking system, based on our fault models which we believe are representative of the types of faults that occur during AJAX development. The choice is, therefore, not biased towards the tool but the fault models we have. With respect to *reliability*, our tools and the TUDU case are open source, making the case fully reproducible.

**Ajax Testing Strategies.** ATUSA is a first, but essential step in testing AJAX applications, offering a solution for the reach/trigger/propagate problem. Thanks to the plugin-based architecture of ATUSA, it now becomes possible to extend, refine, and evaluate existing software testing strategies (such as evolutionary, state-based, category-partition, and selective regression testing) for the domain of AJAX applications.

## 11 Concluding Remarks

In this paper we have proposed a method for testing AJAX applications automatically. Our starting point for supporting AJAX-testing is CRAWLJAX, a crawler for AJAX applications that we proposed in our earlier work [14], which can dynamically make a full pass over an AJAX application. Our current work resolves the subsequent problems of extending the crawler with data entry point handling to *reach* faulty AJAX states, *triggering* faults in those states, and *propagating* them so that failure can be determined. To that end, this paper makes the following contributions:

1. A series of fault models that can be automatically checked on any user interface state, capturing different categories of errors that are likely to occur in AJAX applications (e.g., DOM violations, error message occurrences), through (DOM-based) generic and application-specific invariants which server as oracle.

[12] http://code.google.com/webtoolkit/

[13] http://www.taskfreak.com

[14] https://blueprints.dev.java.net/petstore/

2. An algorithm for deriving a test suite achieving all transitions coverage of the state-flow graph obtained during crawling. The resulting test suite can be refined manually to add test cases for specific paths or states, and can be used to conduct regression testing of AJAX applications.
3. An open source tool called ATUSA implementing the approach, offering generic invariant checking components as well as a plugin-mechanism to add application-specific state validators and test suite generation.
4. An empirical validation, by means of two case studies, of the fault revealing capabilities and the scalability of the approach, as well as the level of automation that can be achieved and manual effort required to use the approach.

Given the growing popularity of AJAX applications, we see many opportunities for using ATUSA in practice. Furthermore, the open source and plugin-based nature of ATUSA makes it a suitable vehicle for other researchers interested in experimenting with other new techniques for testing AJAX applications.

Our future work will include conducting further case studies, as well as the development of ATUSA plugins, capable of spotting security vulnerabilities in AJAX applications.

## References


[1] A. Andrews, J. Offutt, and R. Alexander. Testing web applications by modeling with FSMs. *Software and Systems Modeling*, 4(3):326–345, July 2005.

[2] S. Artzi, A. Kieżun, J. Dolby, F. Tip, D. Dig, A. Paradkar, and M. D. Ernst. Finding bugs in dynamic web applications. In *Proc. Int. Symp. on Software Testing and Analysis (ISSTA'08)*, pages 261–272. ACM, 2008.

[3] M. Benedikt, J. Freire, , and P. Godefroid. VeriWeb: Automatically testing dynamic web sites. In *Proc. 11th Int. Conf. on World Wide Web (WWW'02)*, 2002.

[4] A. Bertolino. Software testing research: Achievements, challenges, dreams. In *ICSE Future of Software Engineering (FOSE'07)*, pages 85–103. IEEE Computer Society, 2007.

[5] R. V. Binder. *Testing object-oriented systems: models, patterns, and tools*. Addison-Wesley, 1999.

[6] E. Bozdag, A. Mesbah, and A. van Deursen. Performance testing of data delivery techniques for Ajax applications. *Journal of Web Engineering*, 0(0), 2009. To appear.

[7] S. Elbaum, S. Karre, and G. Rothermel. Improving web application testing with user session data. In *Proc. 25th Int Conf. on Software Engineering (ICSE'03)*, pages 49–59. IEEE Computer Society, 2003.

[8] J. Garrett. Ajax: A new approach to web applications. Adaptive path, February 2005. `http://www.adaptivepath.com/publications/essays/archives/000385.php`.

[9] W. Halfond and A. Orso. Improving test case generation for web applications using automated interface discovery. In *Proceedings of the ESEC/FSE conference*, pages 145–154. ACM, 2007.

[10] Y.-W. Huang, C.-H. Tsai, T.-P. Lin, S.-K. Huang, D. T. Lee, and S.-Y. Kuo. A testing framework for web application security assessment. *Journal of Computer Networks*, 48(5):739–761, 2005.

[11] S. Kals, E. Kirda, C. Kruegel, and N. Jovanovic. Secubat: a web vulnerability scanner. In *Proc. 15th int. conf. on World Wide Web (WWW'06)*, pages 247–256. ACM, 2006.

[12] A. Marchetto, P. Tonella, and F. Ricca. State-based testing of Ajax web applications. In *Proc. 1st IEEE Int. Conference on Sw. Testing Verification and Validation (ICST'08)*, pages 121–130. IEEE Computer Society, 2008.

[13] A. Memon. An event-flow model of GUI-based applications for testing: Research articles. *Softw. Test. Verif. Reliab.*, 17(3):137–157, 2007.

[14] A. Mesbah, E. Bozdag, and A. van Deursen. Crawling Ajax by inferring user interface state changes. In *Proc. 8th Int. Conference on Web Engineering (ICWE'08)*, pages 122–134. IEEE Computer Society, 2008.

[15] A. Mesbah and A. van Deursen. Migrating multi-page web applications to single-page Ajax interfaces. In *Proc. 11th Eur. Conf. on Sw. Maintenance and Reengineering (CSMR'07)*, pages 181–190. IEEE Computer Society, 2007.

[16] A. Mesbah and A. van Deursen. A component- and push-based architectural style for Ajax applications. *Journal of Systems and Software*, 81(12):2194–2209, 2008.

[17] B. Meyer. Seven principles of software testing. *IEEE Computer*, 41(8):99–101, August 2008.

[18] L. Morell. Theoretical insights into fault-based testing. In *Proc. 2nd Workshop on Software Testing, Verification, and Analysis*, pages 45–62, 1988.

[19] F. Ricca and P. Tonella. Analysis and testing of web applications. In *ICSE'01: 23rd Int. Conf. on Sw. Eng.*, pages 25–34. IEEE Computer Society, 2001.

[20] D. Richardson and M. Thompson. The RELAY model of error detection and its application. In *Proc. 2nd Workshop on Software Testing, Verification, and Analysis*, pages 223–230, 1988.

[21] S. Sprenkle, E. Gibson, S. Sampath, and L. Pollock. Automated replay and failure detection for web applications. In *ASE'05: Proc. 20th IEEE/ACM Int. Conf. on Automated Sw. Eng.*, pages 253–262. ACM, 2005.

[22] S. Sprenkle, L. Pollock, H. Esquivel, B. Hazelwood, and S. Ecott. Automated oracle comparators for testing web applications. In *Proc. 18th IEEE Int. Symp. on Sw. Reliability (ISSRE'07)*, pages 117–126. IEEE Computer Society, 2007.

[23] B. Stepien, L. Peyton, and P. Xiong. Framework testing of web applications using TTCN-3. *Int. Journal on Software Tools for Technology Transfer*, 10(4):371–381, 2008.

[24] E. J. Weyuker. On testing non-testable programs. *The Computer Journal*, 25(4):465–470, 1982.

[25] J. Y. Yen. Finding the k shortest loopless paths in a network. *Manag. Sci.*, 17(11):712–716, 1971.

[26] R. K. Yin. *Case Study Research: Design and Methods*. SAGE Publications Inc, 3d edition, 2003.

# APPENDICE C

# TROISIÈME ANNEXE

# DoDOM: Leveraging DOM Invariants for Web 2.0 Application Reliability

Karthik Pattabiraman and Benjamin Zorn,
Microsoft Research (Redmond)

Contact: Karthik.Pattabiraman@gmail.com, zorn@microsoft.com

Technical Report MSR-TR-2009-176

Microsoft Research
One Microsoft Way Redmond, WA 98052, USA
http://research.microsoft.com

# DoDOM: Leveraging DOM Invariants for Web 2.0 Application Reliability

Karthik Pattabiraman and Benjamin Zorn, Microsoft Research (Redmond)

## Abstract

*Web 2.0 applications are increasing in popularity and are being widely adopted. However, they are prone to errors due to their non-deterministic behavior and the lack of error-detection mechanisms on the client side. This paper presents* DoDOM*, an automated system for detecting errors in a Web 2.0 application using dynamic analysis. DoDOM repeatedly executes the application under a given sequence of user actions and observes its behavior. Based on the observations, DoDOM extracts a set of invariants on the web application's DOM structure. We show that invariants exist for real applications and can be learned within a reasonable number of executions. We further demonstrate the use of the invariants in detecting errors in web applications due to failures of events and the unavailability of domains.*

## 1 Introduction

The web has evolved from a static medium that is viewed by a passive client to one in which the client is actively involved in the creation and dissemination of content. The evolution is enabled by the use of technologies such as JavaScript and Flash, which allow the execution of client-side scripts in the web browser to provide a rich, interactive experience for the user. Applications deploying these technologies are called *Rich Internet Applications* or *Web 2.0* applications [2, 28]. They are being rapidly adopted by popular web sites [33] such as Gmail, Bing and Facebook.

Unfortunately, Web 2.0 applications suffer from a number of potential reliability issues. First, as in any distributed application, the division of the application's logic between the server and client makes it difficult to understand their behavior. Further, web applications are non-deterministic and asynchronous, which makes them difficult to debug.

Second, typical Web 2.0 applications aggregate information from multiple domains, some of which may be untrusted or unreliable. The domains interact by calling each other's code or by modifying a shared representation of the page called the Document Object Model(DOM). Therefore, an error in any one domain can propagate to other domains.

Finally, web applications suffer from a lack of visibility into the behavior of the application at the client. Although tools such as AjaxScope [22] provide visibility into the execution of client side code, they do not provide visibility into the behavior of the DOM. However, users ultimately see and interact with the web application through its DOM. Further, the client-side code of an application interacts with the DOM and hence the DOM plays a key part in the end-to-end correctness of an application.

This paper introduces an approach to detect errors in Web 2.0 applications using their DOM structures. The approach analyzes multiple executions of a Web 2.0 application (also called the training set) corresponding to a given sequence of user actions and extracts dynamic invariants based on their DOM structures.

Prior work has shown that dynamic invariants can be used in general-purpose programs for software understanding, testing and error detection [7, 14, 18]. We show that dynamic invariants (1) exist in web applications, (2) can be learned automatically and (3) are useful in detecting errors. *To the best of our knowledge, our work is the first to derive dynamic invariants based on the DOM and apply them for error detection in Web 2.0 applications*.

The main drawback of using dynamic invariants for error detection is the potential for false positives, i.e., it is possible that executions of the application that do not conform to the invariants are deviant executions rather than erroneous ones. However, as we show, false positives can be minimized by using a representative set of executions for training. The main contributions of the paper are as follows:

1. We show that DOM structures can be used for checking the correctness of a web application.

2. We show that Web 2.0 applications exhibit invariants over their DOM structures and that the invariants can be learned dynamically.

3. We build a tool DoDOM, to repeatedly iterate over the sequence of events in an application to obtain its invariants (and hence the name).

4. We demonstrate the invariant extraction capabilities of DoDOM for three real Web 2.0 applications (Slashdot, CNN, and Java Petstore). We show that the invariants can be learned within ten executions of each application.

5. We further show (for Slashdot) that the derived invari-

ants provide close to 100% coverage for failures of event handlers and domains that impact the DOM.

## 2 Overview

In this section, we outline the potential reliability issues with Web 2.0 applications and present our proposed solution of dynamically extracting invariants. In this paper, we apply the invariants for error detection. However, the invariants we extract can be applied in a wide variety of circumstances which we outline in Section 6.

### 2.1 Background

Typical Web 2.0 applications consist of both server-side and client-side code. The server-side code is written in traditional web-development languages such as PHP, Perl, Java and C. The client-side code is written using dynamic web languages such as JavaScript (JS) which are executed within the client's browser. Unlike in the Web 1.0 model where the application executes primarily at the server with the browser acting as a front-end for rendering and displaying the server's responses, in a Web 2.0 application the client is actively involved in the application's logic. This reduces the amount of data that must be exchanged with the server and makes the application more interactive to the user. Henceforth, when we say web applications, we mean Web 2.0 applications unless we specify otherwise.

Typical web applications are event driven, i.e., they respond to user events (e.g., mouse clicks), timeouts and receipt of messages from the server. The developer writes handlers for each of these events using JavaScript. The event-handlers in turn can (1) invoke other functions or write to global variables stored on the heap, (2) read/modify the web page's contents through its DOM, or (3) send asynchronous messages to the server through a technology known as AJAX (Asynchronous JavaScript and XML) and specify the code to be executed upon receiving a response. The above actions are executed by client-side code (scripts).

A web page is represented internally by the browser as its Document Object Model (DOM) [23]. The DOM consists of the page's elements organized in a hierarchical format. Each DOM node represents an element of the page. Figure 1 shows an example of a DOM for a web page. In the figure, the web page consists of multiple HTML div elements, which are represented in the top-level nodes of the tree. The div elements are logical partitions of the page, each of which consists of nodes representing text and link elements. Further, the web page has a head node with three scripts as its child nodes.

JavaScript code executing in the browser can read and write to the DOM through special APIs. Any changes made to the DOM are reflected in the web page rendered by the browser. User actions are converted into events by the browser and sent to the nodes of the DOM on which they are triggerred.

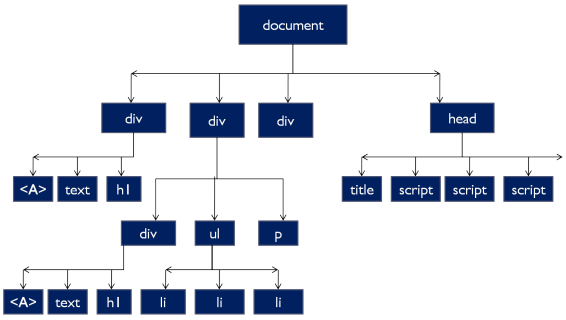



**Figure 1. Example of a DOM tree for a web application**

Browsers allow the separation of scripts into inline frames and provide isolation between scripts that do not belong to the same frame/domain (also called the same-origin-policy [36]). However, this separation severely curtails the ability of scripts to interact with each other and hence many web applications include scripts into a page using the *script* tag. This allows scripts to interact with each other through the DOM or by directly calling each other's functions. Unfortunately, this practice also means that failures of one domain can affect other domains.

Finally, failures of web applications are handled differently from failures of traditional applications. When a traditional application throws an exception or behaves unexpectedly, it is terminated by the operating system (in most cases). Browsers however allow web applications to continue executing even if an event handler fails or if a domain is unavailable (this behavior is necessary for backward compatibility reasons and is also mandated by the standard). This can lead to semantic violations of the application's behavior that are difficult to detect.

### 2.2 Fault Model

This section discusses the faults of web applications considered in this study and their potential causes. This is not an exhaustive list of faults in web applications, but only a first cut at characterizing potential errors in these applications (we are not aware of any previous work in this space).

**Event drops**: These correspond to events being dropped in the client-side code of the application. We consider three kinds of event drops and their causes.

(A) *User-event drops*: Caused by exceptions in event handlers, or the browser terminating the handler because it runs for too long (both Internet Explorer and Firefox do this). It is also possible that a DOM node fails to propa-

gate an event to its parent nodes as required to do so by the DOM [23] specification.

(B) *Server message drops*: Caused by many factors, including (1) network losing the message, (2) server dropping the request due to overload, (3) a message handler throwing an exception or not being invoked in the first place.

(C) *Timeout drops*: This can happen due to (1) the timeout handler throwing an exception, (2) timeout handler not being set correctly by the timer initialization routing, or (3) a browser bug that prevents timeouts from being triggered.

**Domain failures**: These correspond to cases where a domain included by the application is unavailable. This can happen due to a network failure or the servers of the domains being taken offline. It can also occur due to client-side plugins such as NoScript [24] or administrative proxies which may block scripts from certain domains.

## 2.3 Scenario

Web applications can be subject to different kinds of faults as shown in Section 2.2. Consider an application developer who wants to test the resilience of the application to faults. She would execute the application on the client, inject a fault and check if the application behaves as expected after the injection. However, this approach is time consuming and requires the user to repeat the same interactions with the application for each injected fault. Further, the user needs to rely on visual perception to determine whether an injected fault affects the application (while it can be argued that faults that are not perceived by the user do not matter, it may be that a different user perceives the fault). Finally and most importantly, web applications exhibit variations in their behavior from one execution to another as we show later in this paper. Such variations occur as a result of (1) systematic changes introduced by the server (e.g., in showing different advertisements on the page), (2) asynchronous behaviour at the client (e.g., in delivering user actions as events), and (3) non-determinism in network packet delivery (e.g., messages received out of order). In the face of such variations, it becomes challenging to identify whether a perceived difference is the result of a fault or if it is due to the natural behavior of the application. Further, the variations can occur in the middle of a user-interaction sequence and not necessarily at application load time, which makes it challenging to reason about the effects of faults during testing.

The problem we address in this paper is a concrete way to test the resilience of a web application to faults. We assume that the web developer has one or more user interaction sequences under which she wants to exercise the application. Our solution involves extracting a invariant characteristic of the web application's DOM from multiple executions of the application. We characterize the expected behavior of the application and consider significant deviations from this behavior as an error.

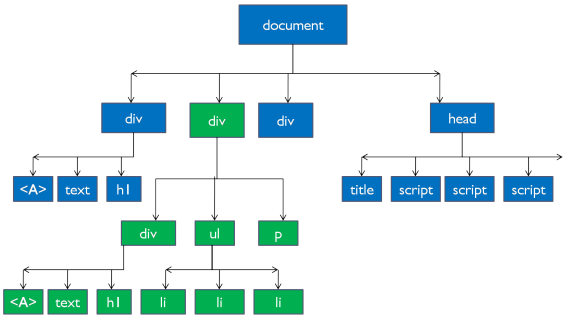



**Figure 2. Proposed solution in the context of the example DOM**

## 2.4 Dynamic Invariant Extraction

This section illustrates our proposed solution for the problem illustrated in Section 2.3. The crux of the solution is in characterizing the invariant portions of the DOM for the web application under a given sequence of user interactions. Specifically, we characterize the common portion of the application's DOM across multiple executions (each of which corresponds to the user-interaction sequence), and the changes made to the DOM by the application in response to various events (i.e., user actions, timeouts and network messages). After each event executes, we check the conformance of the resulting DOM to the invariants. A deviation between the trees indicates an error.

Figure 2 shows the invariant portion of the DOM for the example considered in Figure 1. In the figure, the blue(dark) nodes represent the invariant portions of the tree (also called the web page's backbone) while the green(light) nodes represent the non-invariant portions. We consider two example faults to illustrate the error-detection process.

First, consider the case where the user clicks on a specific DOM node which in turn triggers an event handler on the node. The event handler is supposed to update the left most 'A' element in the invariant DOM but fails to do so due to an error (e.g., the handler throws an exception). This error will be detected as the resulting DOM would deviate from the invariant DOM[1] for the event (mouse-click event).

Second, assume that the web application imports a faulty script (from a different domain) using the 'script' tag. The script has access to the entire web page's DOM. However, it is not expected to modify any portion of the invariant DOM. Assume that the faulty script modifies the center div element in the tree. This element is not a part of the invariant

[1]The invariant DOM will have the modification while the resulting DOM will not because of the error.

| Executions | Event 1 | Event 2 | ... | Event n |
|---|---|---|---|---|
| Execution 1 | $T_{1_1}$ | $T_{2_1}$ | ... | $T_{n_1}$ |
| Execution 2 | $T_{1_2}$ | $T_{2_2}$ | ... | $T_{n_2}$ |
| ... | ... | ... | ... | ... |
| Execution M | $T_{1_M}$ | $T_{2_M}$ | ... | $T_{n_M}$ |
| Invariants | $T_{1_I}$ | $T_{1_I}$ | ... | $T_{n_I}$ |

**Table 1. Invariant DOMs**

DOM and hence the modification is not considered an error. On the other hand, assume that the script attempts to modify the div element in the far left branch of the tree. This element is part of the invariant DOM and hence the modification will be detected as an error.

In the above example, the invariant DOM in Figure 2 represents a snapshot of the DOM during the course of its evolution in response to various events. In reality, the technique will capture the entire sequence of invariant DOMs and use the invariant sequence to check for deviations. We now give a more precise definition of the invariant DOM.

An invariant DOM is a sub-tree of the web application's DOM that is shared by multiple executions. We derive an invariant DOM for each event in the application corresponding to a given event seqeunce. Table 1 shows the DOM trees obtained during multiple executions of the web application for each event in the sequence. The rows of the table indicate different executions of the web application, while the columns indicate the events in the sequence. The DOMs are indicated by $T_{i_j}$, where $i$ is the event after which it is obtained and $j$ is the corresponding execution. The invariant DOMs $T_{i_I}$ are derived from the DOM trees of individual executions $T_{i_j}$ corresponding to the event $i$.

### 2.5 Challenges

There are three major challenges in extracting the invariant DOMs for error detection. First, web applications exhibit small changes from one execution to another due to variations at the server and at the client (as explained in Section 2.3). The invariant DOM must not include such changes as otherwise it will incur false positives.

Second, the invariants extracted should retain as much of the original DOM as possible. This is to ensure high detection coverage of the invariants for reliability and security violations. We show that our approach achieves both high detection coverage and low false-positive rates (Section 5).

Finally, the invariant-extraction system should be compatible with different browsers and platforms, i.e., it should not require adding new functionalty to existing browsers. Further, it should not require any modifications to the server-side code of web applications[2].

[2]The last constraint may not be as important if the server-side code is under our control, but this is often not the case

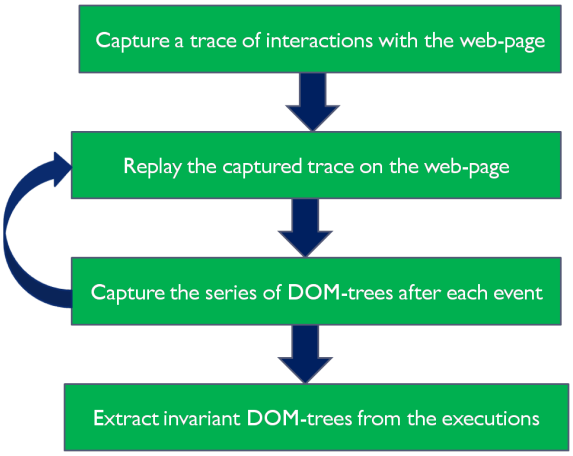



**Figure 3. Steps in learning DOM invariants**

## 3 Approach

The overall approach for extracting and learning invariants over the DOMs of web applications is as follows (shown in Figure 3). First, we record a sequence of user interactions and events on a page (the sequence of events is called a trace). We then replay this trace over multiple executions and capture the sequence of DOMs generated after each event in the trace. We then extract invariants over the set of all DOM sequences using an offline learning process. The recording, replay and capture process is performed using a tool we built called DoDOM. In this section, we describe the architecture of DoDOM and its operation. We also discuss the design choices and the trade-offs made in DoDOM.

### 3.1 DoDOM Architecture

Figure 4 shows the architecture of the *DoDOM* system. In the figure, the gray rectanges represent existing code bases and tools and the green rectangles are the components we added for implementing DoDOM. They are as follows.

The **proxy** is written as a plugin in the Fiddler web application testing framework [9] using the .Net environment. Its main purpose is to inject the JS logger code into the web page(s) of the application, collect the events and responses sent by the JS logger and write them to the file system. Further, the proxy intercepts messages sent between the client and the server and records them. The JS logger communicates with the proxy by appending a special suffix to its messages and sending it through AJAX. The proxy identifies these special messages and takes appropriate action without forwarding them to the server. Hence, the server-side code does not need to be modified to deploy DoDOM.

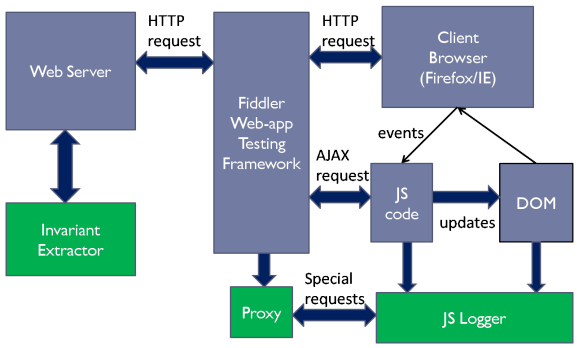



**Figure 4. Architecture of DoDOM system**

The **JS logger** is a piece of JS code that is executed on the client's browser. A separate instantiation of the JS Logger is inserted for each inline frame in the page. The JS logger can read/write to the web page's DOM, install event handlers that trap the page's handlers and log changes to the DOM. It can also intercept messages sent by the client through the XMLHttp interface and the corresponding response (i.e., AJAX messages). Finally, it can intercept all communications between the webpage's JS code and the *window* object, e.g., timeout events. The JS logger performs the core operation of DoDOM and runs within the browser. Since it is written using JS, it is highly portable.

The goal of the **invariant extractor** is to perform offline analysis of multiple executions recorded by the proxy and to extract the invariant DOMs. It runs outside the browser.

## 3.2 Operation

The **proxy** injects the JS logger script into every page loaded by the browser (a page is defined as any entity that consists of a head tag). The proxy also assigns to each JS logger script a unique tag which is used to identify the script in interactions with the proxy. The JS logger script is instantiated at the client after the page completes loading (after the *onLoad* event), upon which it performs the following actions (in sequence): (1) Traverses the DOM of the web page, creates a compact representation of the DOM, and sends it to the proxy. (2) Installs a new replacement handler for all DOM elements that have event handlers installed and stores the old handler as part of the element. (3) Replaces the *setTimeout* and *setInterval* API calls in the window object with custom versions (after storing the old handler) to intercept timeouts. (4) Replaces the *XMLHttpRequest* object with a custom version that intercepts all messages sent to the server using the AJAX interface and their corresponding responses. (5) Installs change handlers on each element of the DOM tree to track any additions, modifications and removals of the sub-tree rooted at the element.

The **JS logger** operates in two modes: record and replay. We first describe the operation of the system in the record mode and summarize the main differences during replay.

During *record mode*, the user interacts normally with the page by moving the mouse, clicking on objects etc. The browser translates the user's actions into user-events and invokes the replacement event handlers installed by the JS logger on the corresponding DOM nodes. The handlers create a snapshot of each event and send it to the proxy, which in turn adds the events to a global queue for the page. The JS logger periodically polls the proxy for outstanding events, upon which the enqueued events are sent to the client (one at a time). The JS logger then invokes original handers corresponding to the event stored in the element. The proxy also writes the events to an event log before sending it to the client.

During *replay*, the above sequence of steps is repeated, but with three differences. First, instead of waiting for events from the JS logger, the proxy reads in the list of events from the event log and populates the global queue. When the JS logger polls for events, the proxy retrieves the events from the queue one at a time and sends them to the client with the corresponding time-delay. Second, the web page may have undergone small changes between the recording session and the replay session, and the DOM node that triggered the event during the recording session may be in a different position during replay. DoDOM uses the contents of a node and its pre-order traversal index to match the node during replay. In the case of a mismatch, it searches the DOM for the node with the closest match and replays the event at the node [3]. Third, only those timeouts that expired during recording are allowed to expire during replay and only if the timeout handlers match. This ensures that the effects of a timeout expiring during replay are faithful to the effects observed during recording.

The proxy records the changes made to the web page's DOM tree after every event (i.e., user action, timeout or received messages). The **invariant extractor** post processes these traces to obtain a sequence of invariant DOM trees for each event. This sequence represents the invariant DOM shared by each of the executions (modulo small differences due to variations among replays). Each tree in the invariant sequence is learned independent of the other trees based on the corresponding trees in the individual executions.

The **algorithm** for learning the invariant DOM trees from a set of execution traces is given below (the pseudocode for the complete algorithm can be found in 3.2).

We set the initial invariant tree to the tree obtained from the first execution trace. For each execution trace, we compare its tree with the corresponding invariant tree recursively starting from the root nodes and traversing the tree in post order. A node is removed from the invariant tree due

[3] The closest match is the node(s) in which the highest number of fields and values match with the original.

to any of the following three conditions. (1) the contents of a node of the invariant tree does not match the corresponding nodes in the execution tree, (2) a node in the invariant tree has more children than its corresponding node in the execution tree, (3) the invariant tree has nodes that are not present in the execution tree. The comparison among nodes' contents is based on the fraction of its field-value pairs that match each other. If this fraction is higher than a value known as the *match threshold*, then the nodes are considered to match.

The match threshold thus determines how much of the invariant tree is pruned away due to differences among the DOMs of individual executions. A high match threshold means that only nodes that match closely across executions will be retained in the invariant tree. On the other hand, a low match threshold indicates that that the invariant tree may contain nodes that exhibit high variation in their contents among executions.

### 3.3 Discussion

For portability and ease of deployment, DoDOM is implemented predominantly in JS with a small part implemented as a client-side proxy. However, the use of JS incurs certain limitations as follows.

- The ordering of events in the JavaScript Virtual Machine (VM) cannot be observed or controlled by DoDOM. These may impact the behavior of some applications. However, a robustly designed page must not depend on the ordering of events by a specific VM, and hence this is not a significant limitation.

- DoDOM traps events by hooking into the event-handlers of DOM nodes and replacing the existing functions with a wrapper function. This requires that the web page be written using the DOM 1.0 event-model. The DOM 2.0 event model does not provide any way to remove a handler from the chain of event-listeners on a DOM-node, or to ensure that the event-handlers are invoked in a specific order. However, most of the web pages we observed were written using the DOM 1.0 model for comaptibility reasons. Further, the DOM 3.0 specification remedies thi situation, but is not yet implemented by most browsers.

- DoDOM assumes that the communications between the JS logger script and the client-side proxy is First-In-First-Out (FIFO), i.e. messages sent by the proxy are delivered in-order to the JS logger code. If this assumption is violated, it may not be possible to isolate the effects of an event. However, we found that this assumption was satisfied on our platform, namely Firefox on Windows.

```
int MatchScore(HashCode H1, HashCode H2)
{
  int score = 0, total = 0;
  for (Field F, Value V) in H1 do:
    if H2.hasField(F) and (H2.getValue(F)==V)
       score = score + 1;
    total = total + 1;
  return (score / total);
}

Set CheckMatch(TreeNode N1, TreeNode N2)
{
   Set diffSet = empty;
   HashCode H1 = getHash(N1);
   HashCode H2 = getHash(N2);
   if (MatchScore(H1, H2) < matchThreshold)
     diffSet.add( N1, N2, "content" )
     Children C1 = getChildren(N1);
     Children C2 = getChildren(N2);
     numChildren = min( C1.length(), C2.length() )
     for (i = 0; i<numChildren; ++i) {
        deltaSet = CheckMatch( C1, C2 );
        diffSet = diffSet Union deltaSet;
    }
    if (C1.length > numChildren) {
       for (i=numChildren; i < C1.length; ++i)
          diffSet.add( N1, N2, "remove" )
    } else if (C2.length > numChildren) {
         for (i=numChildren; i < C2.length; ++i)
            diffSet.add( N1, N2, "append" )
    }
    return diffSet;
}

Tree pruneInvariants(Tree InvTree,
   Tree TraceTree)
{
    Set diffSet = CheckMatch(InvTree.root,
                        TraceTree.root);
    for (element in diffSet) do {
       (node1, node2, diffType) = element
       if (diffType == 'remove')
          removeFromTree(InvTree, node1);
       else if (diffType == 'content')
          removeFromHash(InvTree, node1);
       }
       return InvTree;
}
```

**Figure 5. Pseudo-code for extracting invariants from DOMs**

- DoDOM cannot capture the state of the JS heap as it is implemented entirely in JS. While it is possible to route all object allocations on the JS heap through DoDOM, such a mechanism would incur high overheads. Hence, we do not consider the JS heap in DoDOM. This is not a significant limitation as changes to the JS heap that impact the page will be reflected in the DOM, server messages or timeouts and would hence be captured by DoDOM. As for the other changes, they do not matter from the point of view of the page and are hence not captured.

- If for any reason, the JavaScript VM crashes or hangs, then so does DoDOM. This is not a significant limitation as most JS VM's are reasonably robust[20]. We do however provide a heartbeat service to detect hangs of the JS logger and reload the page if it hangs.

# 4 Experimental Setup

This section describes the experiments performed and the benchmarks used in our evaluation. The experiments were performed using Firefox on a Core-2-Duo Intel processor machine (running at 3 GigaHertz each) with 4 GigaBytes RAM and running Windows (Vista).

We summarize the main research questions answered by the experiments.

*1. How many executions do we need to learn the invariant DOMs for a web application?*

*2. How effective are the invariants in detecting errors due to dropped events?*

*3. How effective are the invariants in detecting failures of domains in the web application?*

*4. How much variation do the invariants exhibit from one day to another ?*

## 4.1 Invariant Extraction

The goal of this experiment is to answer research question 1. From the set of all executions (replay sequences), we randomly choose a training set, which is a subset of executions used to learn the invariants. In these experiments, we vary the training set size in order to understand how quickly the invariants converge to a stable value. We also vary the match threshold described in Section 3.2 to understand how much content similarity is present among the executions.

We measure the following characteristics of the DOM tree in order to measure its convergence. (1) total number of nodes, (2) average number of children per node, i.e., its fanout, (3) maximum number of levels from each node, i.e., the height of the sub-tree, and (4) average number of total descendants per node (not only its immediate children).

| Fault Type | Injection Method |
|---|---|
| User-Event Drop | Do not replay the event at the client |
| Message Drop | Do not forward the message to the server |
| Timeout Drop | Do not replay the timeout at the client |

**Table 2. Faults injected and their characterization**

We also compare the invariant DOM sequence with the DOM sequences from all executions (even if they are not in the training set). If any of the DOMs in its sequence exhibits a mismatch with the corresponding DOM in the invariant sequence, we consider the execution a false positive.

## 4.2 Event Drops

The goal of this experiment is to answer research question 2. We measure the error-detection coverage of the invariant sequences for event drops corresponding to those in Section 2.2. We observe their effects on the web page's DOM. Table 2 shows the types of faults introduced and the injection method. The fault injector is implemented as an enhancement of the replay mechanism in the DoDOM system and can be configured through an external file. Each run injects at most one fault to ensure that its effects can be uniquely determined. After a fault is injected, the sequence of DOMs corresponding to the execution is compared with the sequence of invariant DOMs. We classify the execution as a successful detection if any of the DOM trees in its sequence exhibits a mismatch with the corresponding DOM in the invariant sequence.

## 4.3 Domain Failures

The goal of this experiment is to answer research question 3. It emulates the effect of domain failures as described in Section 2.2 using the NoScript plugin in Firefox [24]. First, the invariant DOM sequence is obtained from multiple executions as shown in section 4.1. Then, each domain in the web page is blocked one at a time and the corresponding DOM sequences are obtained. The DOM sequence for a blocked domain is then compared to the invariant DOM sequence. A mismatch indicates that the domain's failure is detected by the invariants.

## 4.4 Diurnal Variations

The goal of this experiment is to answer research question 4.

Because web pages may exhibit day-to-day changes, we gather executions of the application over multiple days to study the effect of diurnal variations in the effects of the

blocked domains on the web page. The invariants obtained on each day are compared with the executions obtained by blocking a domain (on a particular day) to determine whether the blocked domain impacts the backbone of the web page on a given day. This allows us to understand whether the effects of a domain are local to a given day or whether they span multiple days.

### 4.5 Benchmarks

Table 3 summarizes the characteristics of the three web applications. It shows the number of domains in the application, the total number of lines of JS code (obtained with Firefox's Phoenix plugin [29]) and the number of events in the recorded trace for the application. We use the Slashdot web site (*http://slashdot.org*) as the primary source of measurements in this paper.

Slashdot aggregates technology-related news from different websites and allows users to comment on a news story. Slashdot also allows users to navigate among different comments and view/expand comments on demand. We interact with a Slashdot news story normally and replay the interactions using DoDOM. The results reported are for a specific news story on Slashdot[4] with close to 300 comments. We perform a total of 58 replays for the story (over the course of 1 hour). We also repeated the experiments with a different story with about 50 events, but the results were similar and are hence not reported.

We also evaluate the invariant extraction capabilities of DoDOM on two other web applications, namely CNN and Java Petstore. Java Petstore is a freely available Web 2.0 application that mimics an e-commerce website for buying pets [10][5]. It allows the user to browse through pet listings and choose a pet corresponding to the user's preferences. We interact with the website by moving over and clicking on different elements of the page. CNN is a widely read news website that delivers customized content to its readers using JavaScript. We study the main page of CNN, which is highly dynamic. Our interaction with CNN also consists of moving the mouse over various elements of the page and clicking on news stories of interest.

Table 3 shows that Java Petstore is the least complex application in terms of the number of lines of JS code while CNN is the most complex. We choose to focus on Slashdot for our experiments as it presents a middle ground among the three applications.

[4]The story is "http://hardware.slashdot.org/story/09/07/30/2147246/ARM-Hopes-To-Lure-Microsoft-Away-From-Intel"

[5]Java PetStore is written using the Dojo programming framework, which registers event handlers for almost all events in the page, and hence the large number of events in this application

| Website | Lines of JS | No. domains | No. events |
|---|---|---|---|
| Java Petstore | 499 | 1 | 211 |
| Slashdot | 9647 | 5 | 13 |
| CNN | 15603 | 9 | 9 |

**Table 3. Characteristics of the web applications**

## 5 Results

This section presents the results of the experiments described in Section 4. We focus on the Slashdot application for the bulk of the experiments and summarize the results for the CNN and Java PetStore applications at the end. We also measure the performance overhead of DoDOM.

We summarize the main results first.

**Convergence of invariants** (Section 5.1): We show that the invariant DOM converges with a training set size of 6 executions, which corresponds to 10% of the total executions.

**Coverage for event drops** (Section 5.2): The invariants detect 100% of the drops of events whose handlers affect the DOM.

**Fault impact** (Section 5.3): We find that most faults have little or no impact on future events. However, faults that do affect future events are more likely to impact events closer to their origin in the execution trace.

**Coverage for domain failures** (Section 5.5): We also find that failures of 4 of the 5 domains included by Slashdot have no effect on the DOM. Hence, they are not detected by the invariants. However, the invariants detect 100% of the failures of the one domain that affects the DOM.

**Impact of day-to-day variations** (Section 5.6): Finally, we find considerable variations among the invariants obtained on a day-to-day basis for Slashdot. However, DoDOM is able to learn a consistent set of invariants for the application across multiple days with no false positives on any day.

**Other applications(Section 5.7)**: Finally, we find that the other two applications, namely Java PetStore and CNN, also exhibit invariants that can be learned within six executions (similar to Slashdot).

### 5.1 Invariant Extraction

In this experiment, we vary the training set size from 1 to 10 over 58 executions and obtain the invariant DOM sequence. The characteristics of the invariant DOM corresponding to the metrics listed in Section 4 are shown in Figure 6 (a) through (d).

In the figures, the X-axis represents the event number and the Y-axis represents a metric corresponding to the

| Training Set Size | match threshold | | |
|---|---|---|---|
| | 0.95 | 0.50 | 0.05 |
| 2 | 53 | 53 | 53 |
| 4 | 29 | 29 | 29 |
| 6 | 1 | 1 | 0 |
| 8 | 1 | 1 | 0 |
| 10 | 1 | 1 | 0 |

**Table 4. False positives versus training set size**

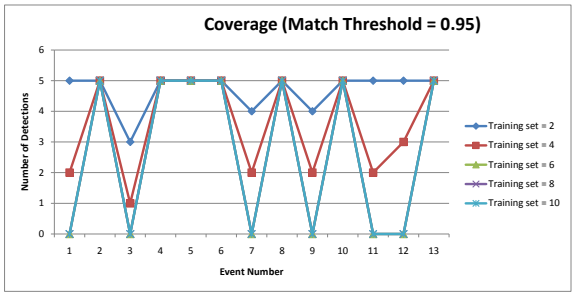



**Figure 7. Error-detection coverage of the DOM invariants**

event. The lines represent the invariants obtained with a training set of a specific size. The figures show that the number of nodes monotonically increases with the event number, while the maximum number of levels monotonically decrease. The other two metrics, namely the number of children and the number of descendants show no consistent trend. This shows that the events are progressively adding nodes to the tree although the number of descendants does not change much because the number of leaf nodes in the tree correspondingly increase.

Figure 6 (a) shows that as the training set size increases, the number of nodes in the DOM decreases because more and more nodes are eliminated from the invariant DOMs. However, it stops decreasing once the training set reaches a size of 6 (roughly 10% of the executions). Nonetheless, the number of nodes converges to a value that is within 1% of its original value (i.e., the number of DOM nodes for any given execution). Similarly, the the average number of descendants steadily increase with increasing the training set size, but also stabilizes at a training set size of 6. This shows that the invariant DOMs can be learned with a relatively modest training set of 6 executions.

To further confirm the convergence of the invariants, we measure the false-positive rate for the executions. Table 4 shows the false-positive rate as a function of the size of the training set for different values of the match threshold (introduced in Section 3.2). In the figure, the X-axis represents the training set size and the Y-axis represents the number of false positives. As can be seen in the figure, the false-positive rate initially starts out high when the training set is very small (2 executions), but quickly decreases with increase in the training set size. For a training set size of 6 or more, the false-positive rate is nearly zero (see below). This confirms the earlier observation that a training set size of 6 is the point at which the invariant DOMs stabilize.

Interestingly, the false positives do not drop to 0 when the match thresholds are 0.95 and 0.50, but remain stable at 1 up to a total of 10 executions (maximum in this experiment). This suggests that one of executions exhibits significant differences from the invariant DOM. Nonetheless, the false positives drop to 0 when the match threshold is decreased to 0.05, suggesting that the deviation is due to content differences among the DOMs rather than structural differences. We are investigating this case further.

## 5.2 Coverage for Event Drops

We measure the error-detection coverage of the invariants for event drops through fault-injection experiments shown in Table 2. For each of the 13 faults (each corresponding to an event in the trace), we perform 5 fault-injection runs and compare the resulting DOM sequence with the invariant DOM sequence. A mismatch among the sequences indicates that the fault was successfully detected. Figure 7 shows the error-detection coverage as a function of the fault (event-number) that was injected. Based on the previous results for false positives (Section 5.1), we focus on the invariants derived with training sets of 6 or more.

Figure 7 shows that for invariants with training sets of 6 or more, the detection rate is either 0% (0 detections) or 100% (5 detections) depending on the injected fault. The reason for the differences is as follows. Either an event-handler affects the DOM or it does not. The events that have 0 detection rates, namely 1, 3, 7, 9, 11 and 12, are timeout events and these events only update the global JavaScript heap and not the DOM[6]. The other events are mouse-clicks and message-handling events and the corresponding handlers add or remove nodes from the DOM. Thus, the invariants detect all event drops that affect the DOM.

The match threshold was set to 0.95 for these experiments. However, the above results are not affected by varying the match threshold from 0.95 to 0.05 (not shown in figure). This suggests that the coverage is due to structural differences among the DOMs rather than content differences.

Further, a match threshold of 0.05 has identical coverage to a match threshold of 0.95, although it has a lower false-positive rate (Section 5.1). Therefore, a match threshold of

[6]We confirmed this observation by reading their JavaScript code.

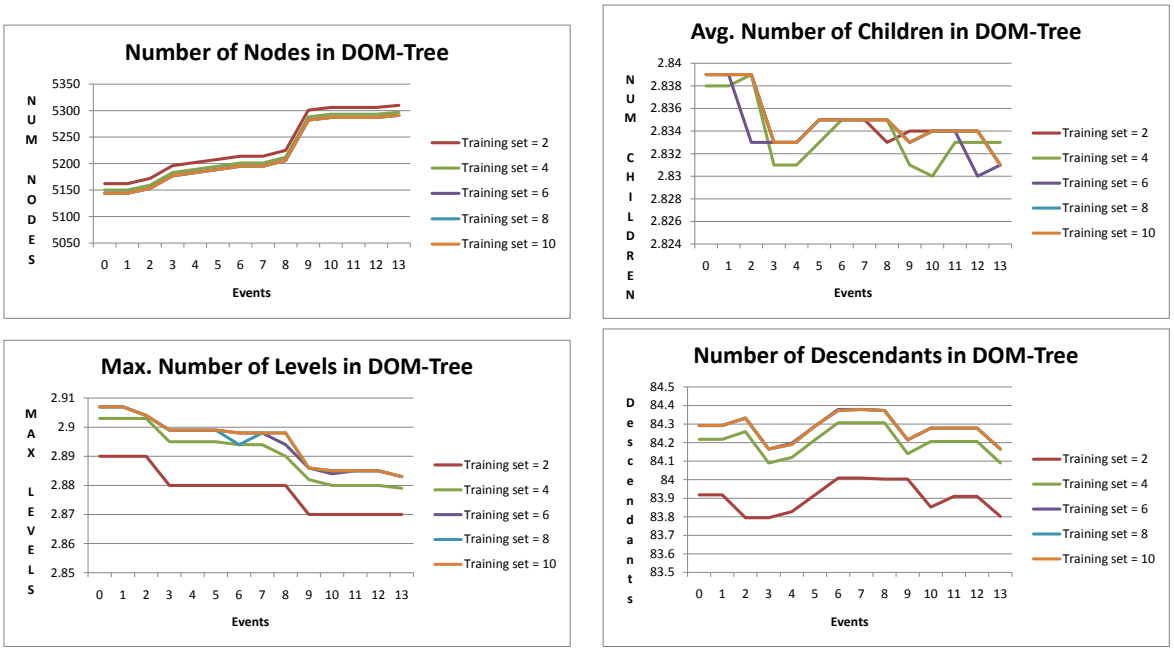



**Figure 6. Invariant Characteristics of the DOM: (a) Total number of nodes, (b) Average number of children per node, (c) Maximum number of levels in each node and (d) Average number of descendants per node**

0.05 represents an optimal tradeoff between detection coverage and false-positive rate.

### 5.3 Fault Impact

In this study, we measure the impact of a fault by comparing the DOM sequence from the fault-injected executions with the invariant DOM sequence. The training set size for the invariants used in this study was fixed at 6 and the match threshold set to 0.95. For each fault injected execution, Figure 8 shows the number of nodes in the DOM that are different from the corresponding invariant DOM sequence as a function of the event in which they differ. Each line in the graph represents the average of five injection runs for a particular fault (identified based on the event number that is dropped). Only the faults that were detected in Figure 7 are shown in Figure 8.

The following observations may be made from Figure 8. First, the total number of DOM nodes that differ from the invariant DOMs is typically less than 25 for each event and fault. This is relatively small compared to the total number of nodes in the tree (over 5000), suggesting that the impact of a fault is confined to a small fraction of nodes. Second, for all faults except fault 2 and fault 8, the number of DOM nodes that differ from the invariant tree is a constant independent of the event number. This suggests that the portion of the DOM affected by the corresponding events is not influenced by future events. Finally, for faults 2 and 8, the number of DOM nodes affected by the fault first increases and then decreases before converging to a constant value. This suggests that the events corresponding to the faults influence the behavior of the events that immediately follow them in the trace. However, the impact decreases with increasing distance from the fault's origin and tapers to a constant value (after about three events in each case).

### 5.4 Aggregate Invariants

In this paper, we have derived invariants based on both the structure and content of DOM trees as shown in Section 3. In this section, we consider only the aggregate properties of the invariant DOMs and evaluate their error detection capabilities. The aggregate measures used correspond to those in Section 5.1. We also measured the false-positive rate for the aggregate invariants. The results for coverage and false-positive rates are shown in Figure 9 and Figure 10. Each bar in the figures represents the aggregate invariants derived using training set sizes ranging from 2 to 10. In the false-positive graph, the total number of executions on the Y-axis was 58, while in the coverage graph, the total number of executions on the Y-axis was 65 (13 faults, 5 runs each).

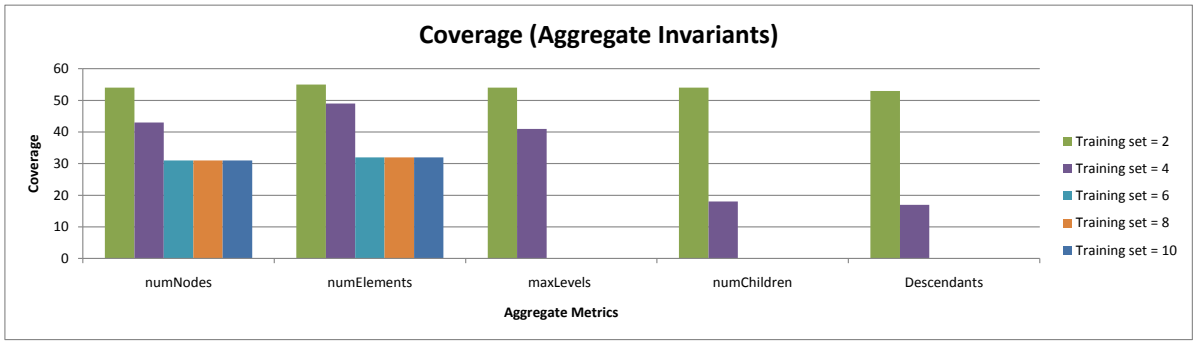



**Figure 9. Error-detection coverage of aggregate invariants**

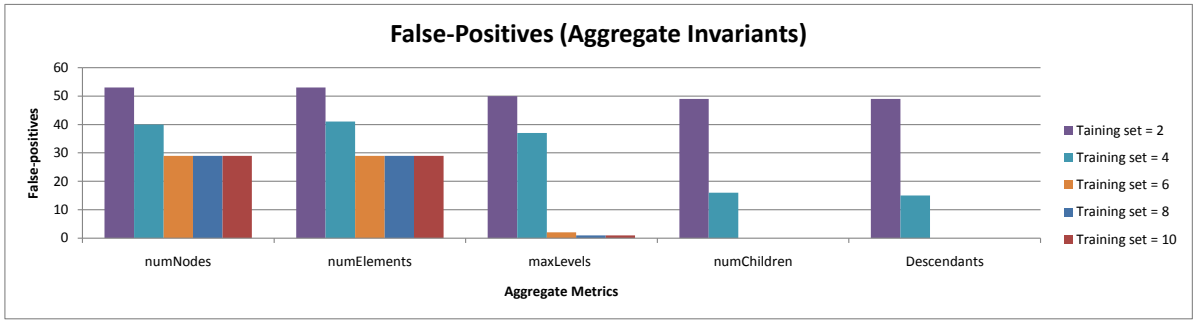



**Figure 10. False-positive rates of aggregate invariants**

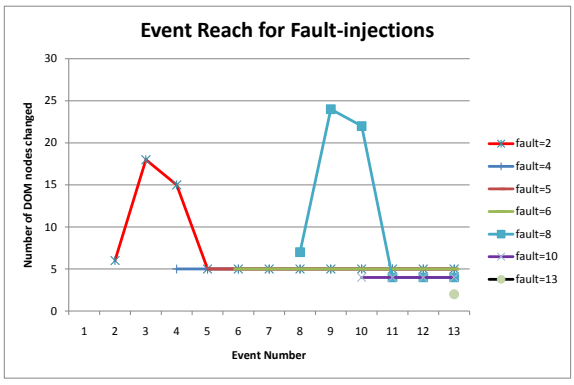



**Figure 8. Impact of injected faults on the DOM**

The main results from Figures 9 and 10 are summarized here. First, we see that both the false-positive rate and the error-detection coverage decrease as the training set increases from 2 to 6 after which they more or less stabilize (with the exception of false positives incurred by the maxLevels metric). This is in line with the results reported in Section 5.2. Therefore, we consider only training sets of 6 or more in this discussion. Further, both the number of nodes (numNodes) and number of elements (numNodes) have high error detection coverage, but also exhibit a high false-positive rate. Therefore, they are unsuitable as detectors because they exhibit high variations from one execution of the web application to another. However, the other three metrics, namely, the maximum number of levels (maxLevels), average number of children (numChildren) and number of descendants (numDescendants) have a low false-positive rate, but also have almost zero error-detection coverage. Hence, they are ineffective at detecting errors as the information they capture is too coarse-grained to detect small changes in the DOM due to errors (as we showed in Section 5.2 the injected faults resulted in changes in relatively few DOM nodes). Therefore, we do not consider aggregate metrics in this paper, although we do not rule out the possibility that there may exist aggregate metrics other

| Domain | Total executions | No. of mismatches |
|---|---|---|
| doubleClick.net | 16 | 0 |
| Fsdn.com | 27 | 27 |
| Google Syndication | 31 | 0 |
| mediaplex.com | 81 | 2 |
| 2mdn.com | 25 | 0 |
| No domain | 90 | 0 |

**Table 6. Domain failures for Slashdot**

than the ones we have considered that may be effective at detecting errors. This is a direction for future investigation.

## 5.5 Domain Failures

The goal of this study is to measure the error-detection coverage of the invariant DOMs for failures of the domains in Slashdot. The Slashdot application imports scripts from a number of different domains as shown in Table 5. Table 5 shows that some of the scripts are obfuscated and use unsafe practices such as the introduction of new code into the page. The results of the study are shown in Table 6 and summarized below.

First, the number of executions is different for different blocked domains. This is because we capped the total time for running each experiment to 30 minutes for each blocked domain. Blocking different domains has different effects on the time taken for each execution. For example, domains containing a lot of complex JS code take longer to execute and blocking these domains substantially speeds up the execution of the web application.

Second, among 90 executions in which no domain was blocked, none affected the invariants for the page, showing that the false-positive rate for the experiment was 0. Hence, the invariants were stable for the training set (as in previous experiments, the match threshold of 0.95).

Finally, each domain either impacts all the executions in which it is blocked, or it does not impact any execution[7]. Of the five domains, only fsdn.com affects the DOM tree and when it is blocked, its failure is detected by the invariants. The failures of other domains are not detected as they do not affect the DOM. Hence, the invariants detect 100% of the failures of the domain(s) that impact the DOM.

## 5.6 Diurnal Variations

In this section, we attempt to understand the day-to-day variations of the invariants obtained in section 5.1. To this end, we collected execution traces from the application by replaying a set of recorded events over three different days (we did not block domains for this experiment).

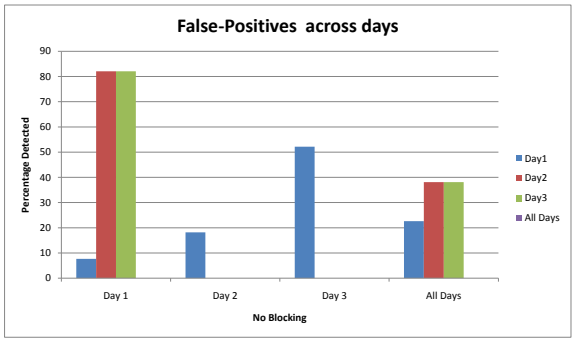



**Figure 11. Results of day-to-day variations for Slashdot**

We collected a total of 90 executions, of which each day contributed 30 executions. We then derive four sets of invariants from the traces. Three sets of invariants are derived from a training set of 10 executions each drawn from a specific day, and one set of invariants is derived by combining together the training sets of three days to form a training set of 30 executions. We then compare each set of invariant DOMs with the DOMs of the executions and measure the false-positive rate with respect to each set of invariants (since no faults are injected, all deviations are false positives). The results are shown in Figure 11. The X-axis shows the executions categorized by the day on which they were obtained while the Y-axis shows the percentage of false positives. Each bar in the graph corresponds to the false positives with respect to the one of the four invariant sets described above. The match threshold used was 0.95.

The following results may be observed from Figure 11. First, the invariants obtained across all three days do not exhibit false positives for any of the executions. Second, the invariants for days 2 and 3 do not exhibit false positives for the executions obtained during those days. This suggests that the web application did not change significantly during those two days. However, the invariants for day 1 exhibit false-positive rates of 20% and 50% for the executions obtained during days 2 and 3 respectively. Likewise, the invariants for days 2 and 3 exhibit false-positive rates of about 80% each for the executions obtained during day 1. This suggests that the web application underwent significant changes between days 1 and 2. This result is not surprising, as web applications are in constant flux and may change on a day-to-day basis. However, even with the changes, DoDOM was able to extract a stable set of invariants for the application.

[7]Mediaplex.com is an exception - it differs in 2 of 81 executions from the invariant DOM. We believe these executions are false positives.

| Domain Name | Lines of JS code | Comments |
|---|---|---|
| *doubleclick.net* | 555 | 1. Uses document.write to insert new scripts and function calls into the document,<br>2. Writes into global objects on the JavaScript heap shared by all scripts on the page. |
| *c.fsdn.com* | 7460 | 1. Complicated code that is obfuscated and compressed using JSCrunch,<br>2. Adds and removes DOM tree nodes based on user's input |
| *Google syndication* | 1451 | Shows advertisements on the page using the Google advertisement service |
| *s.fsdn.com* | 181 | Storage of media files and scripts for Slashdot |
| *Variable domains* | Unknown | These domains vary for different news stories or even when the same story is reloaded.<br>Examples are eyewonder.com, mediaplex.com, 2mdn.com |

**Table 5. Domains from which Slashdot imports scripts**

### 5.7 Other applications

In this section, we summarize aggregate results from running DoDOM on two other web applications, namely *CNN* and *Java PetStore*. We measure the convergence of the invariant DOMs for the two applications as a function of the training set size. Table 7 shows the results for both applications. The table focuses on the final events in the respective applications' traces. As can be seen in the table, the invariant DOMs stabilize with a training set size of 6 for both applications.

### 5.8 DoDOM Performance Overhead

In this section, we measure the performance overheads introduced by the DoDOM tool. The overhead consists of three main components.

**Page load**: The load time of a page depends on the amount of data that is downloaded by it. This includes both HTML and JavaScript code. DoDOM consists of about 1500 lines of uncompressed JS code, which corresponds to 16.5 KB in compressed form. The amount of JS code loaded by Web 2.0 sites ranges from 200 KB to 1.9 MB [31] and hence DoDOM adds less than 10% to the code size.

**Record Mode**: After the page has loaded, DoDOM performs a complete traversal of the web page's DOM and sends it to the proxy (in a compact form). This operation takes approximately 3.5 seconds for the Slashdot page (we use the Firebug profiler for the measurements). The time taken to run the initial scripts on the page without DoDOM is approximately 9.5 seconds. Therefore, the initialization overhead of DoDOM for the Slashdot application is 36%. DoDOM also introduces delays in capturing user interactions as they must be sent to the proxy. However, we did not observe any noticeable delay when using DoDOM for the applications considered in the paper.

We currently implement DoDOM using JavaScript, which is not optimized for performance. A more efficient implementation as a browser plugin would substantially speed up its operation and is a direction for future work.

**Invariant Extraction**: The invariant extraction process is done offline and is not in the critical path of the application. The overhead is on the order of a few minutes.

### 5.9 Threats to Validity

In this section, we discuss some of the threats to the validity of the results obtained in the study. First, the key assumption made in this study is that the invariant DOM does not change substantially from one execution to another. However, this assumption may be violated if either the server-side code undergoes upgrades, or if the content of the web page undergoes substantial changes. If this happend, then the invariants need to be relearned.

The second threat to validity is the assumption of consistency in the behavior of web applications under failures. For example, we assume that a significant deviation from the invariant DOM is due to an event being dropped (Section 5.2) or due to a domain being blocked (Section 5.5). However, it is possible that factors such as load on the server or measurement noise at the client result in spurious deviations. Future work will focus on eliminating spurious deviations.

## 6 Discussion

Although not the focus of this paper, the invariants extracted by DoDOM have uses beyond error detection. We outline some of the uses in this section.

**Web Applications Testing**: One of the challenges in testing Web 2.0 applications is in determining the correct or acceptable behavior of the application under different test inputs. ATUSA [26] uses programmer-specified invariants on the DOM to exhaustively test all paths in a Web 2.0 application and ensure their conformance. However, writing invariants is time and effort intensive. DoDOM can be used to automate the extraction of invariants for a tool such as ATUSA to check during its exploration.

**Security Enforcement**: The invariants learned by DoDOM can also be used to check if a web page has been tampered with either during transmission or rendering at the

| *Training* | **Java PetStore** | | | | **CNN** | | | |
|---|---|---|---|---|---|---|---|---|
| *Size* | *NumNodes* | *NumChildren* | *MaxLevels* | *Descendants* | *NumNodes* | *NumChildren* | *MaxLevels* | *Descendants* |
| 2 | 397 | 3.04 | 2.81 | 44.29 | 2413 | 2.470 | 2.632 | 48.485 |
| 4 | 397 | 3.04 | 2.94 | 44.29 | 2408 | 2.475 | 2.636 | 48.548 |
| 6 | 387 | 3.10 | 2.94 | 45.57 | 2407 | 2.475 | 2.636 | 48.548 |
| 8 | 387 | 3.10 | 2.94 | 45.57 | 2407 | 2.475 | 2.636 | 48.548 |
| 10 | 387 | 3.10 | 2.94 | 45.57 | 2407 | 2.475 | 2.636 | 48.548 |

**Table 7. Results for Java PetStore and CNN applications**

client. This is similar to the Web Tripwire project [33], with the difference that we can apply it to arbitrary web applications that execute client-side code. Further, the invariants can also help identify if a web page has been permanently defaced (for example, through a Type 2 XSS attack [13]).

**Better Domain Filtering**: Section 5.5 shows that failures of the majority of domains do not impact the invariant DOM for Slashdot. We believe this is also likely to be the case for many web applications that include multiple domains. We could filter such domains at the client for advertisement blocking and performance optimization. The NoScript plugin [24] already allows domain filtering but leaves it to the user to decide which domains to block. With DoDOM, we can automate the decision making process based on whether the domain impacts the DOM.

## 7 Related Work

**Dynamic invariant derivation**: DAIKON [14] and DIDUCE [18] derive program invariants based on dynamic executions. DAIKON derives invariants over multiple test inputs while DIDUCE does the same over multiple stages of a program's execution. These techniques attempt to learn invariants over program structures, and do not apply to Web 2.0 applications as these applications are typically data centric [27]. Hence, web applications require techniques that infer data-centric invariants.

HeapMD [7] infers invariants over data-structures on the heap and identifies significant violations of the properties as potential bugs. DoDOM also falls under the category of data-invariant detection techniques. However, it infers invariants over the web page's DOM, which are different from data-structure invariants in three aspects. First, data structure invariants typically encode the connectivity properties of data structures, while DOM invariants encode both the content and structure of the DOM. Second, data structure invariants are predicates over the nodes in the data structure, while DOM invariants are parts of the DOM tree. Finally, DOM invariants evolve with respect to specific events and actions in the application, while data structure invariants are typically fixed for the duration of the program.

**Web application testing**: A number of approaches have been developed to test web applications that execute primarily at the server, i.e., Web 1.0 applications [3, 34]. An example of this approach is Veriweb [3], which systematically explores a web site by navigating to each of its pages. However, Veriweb cannot be applied to Web 2.0 applications which often execute within a single page.

Marchetto et al. [25] propose an approach to test Web 2.0 applications using an abstract state machine model provided by the developer. Mesbah and Deursen [26] extend this work to infer the state machine model automatically by finding clickable elements in the application and emulating clicks on them. Similar to our work, they use invariants on the DOM tree to check the validity of a state explored by their tool called ATUSA. ATUSA differs from DoDOM in two aspects. First, the invariants used in ATUSA correspond to generic invariants on the validity of the DOM (e.g., invalid HTML), and are not specific to the web application being tested[8]. Second, ATUSA uses heuristics to find clickable elements on the page, and explores paths corresponding to the clicks. However, this exploration may not model a real user-interaction sequence.

Recently, Bezemer et al. [4] present a dynamic approach for security testing of widgets used in Web 2.0 applications. Widgets are small pieces of JavaScript code and/or HTML that co-exist within a single web page. The main differences between this approach and ours are that (1) we can analyze the entire web application rather than only widgets and (2) we do not require an explicit security policy specification.

AjaxScope [22] is a tool for remotely monitoring web 2.0 applications and changing their behavior on the fly. Similar to our approach, AjaxScope interposes on the web application using a proxy server and rewrites the application's JavaScript code. AjaxScope can be used for performing drill-down performance analysis or for debugging of complex bugs such as memory leaks. However, the intrumentation is at the level of JavaScript code and cannot be easily extended for DOM elements. Furthermore, AjaxScope does not in and of itself derive invariants and it would be non-trivial to extend it to do so.

**Static Analysis**: Static analysis approaches can also

[8] ATUSA allows the programmer to specify application-specific invariants, but does not derive the invariants.

be used for detecting errors in web applications. However, Web 2.0 applications are written in languages such as JavaScript, which are difficult for static analysis approaches to handle. This is because the JavaScript language has features such as dynamic/loose types, on the fly code creation and lack of separation between code and data [12]. Static analysis approaches for analyzing JavaScript [8, 16, 17] ignore these features in the interest of accuracy. However, many real web sites extensively use these unsafe features [39] and hence cannot be statically analyzed.

**Real World Studies**: Kalyanakrishnan et al. [21] study the availability of popular websites from an end-user perspective. Their study focused on the network connectivity between the client and the server. Further, they focus on the availability of a web site rather than its reliability. Chen et al. [6] present Pinpoint, a tool to automatically determine the root causes of failures in large internet service infrastructures deployed on the J2EE platform. Pinpoint targets the backend of web services. Pertet and Narasimhan [30] study downtime incidents of web services to understand their root causes. The study was confined to server failures.

**Fault injection into web applications**: Reinecke et al. [32] use fault injection to evaluate the resilience of reliable messaging standards in web services. Vieira et al. [38] use fault injection to evaluate the robustness of web-services' infrastructure. These approaches target errors in communication protocols and server code respectively.

Huang et al. [19] evaluate the security of web applications using fault-injection experiments. The injected faults correspond to SQL injection attacks and cross-site scripting (XSS) attacks. Fonseca et al. [15] use vulnerability and attack injection to test the resilience of web applications to attacks. However, these approaches operate on the application's server-side code and cannot target client-side code. Further, they are not concerned with non-malicious that result in the application deviating from its correct behavior.

Finally, Bagchi et al. [1] and Automatic failure path inference(AFPI) [5] use fault injection to track dependencies in server applications. We use fault injection for resilience testing rather than dependency discovery.

**Dynamic Checking**: Web applications have also used dynamic invariant approaches for detecting security attacks. Swaddler [11] derives dynamic invariants on the server code of web applications written using the PHP language. Subsequently, Swaddler uses the inferred invariants to detect attacks that attempt to bypass the application's workflow and force the application into an inconsistent state. Swaddler's analysis and enforcement is implemented on the server side and hence cannot be used for Web 2.0 applications.

Blueprint [37] is an approach to enforce the integrity of a web application's DOM at the client in order to prevent script injection (XSS attacks). The server encodes the intended DOM of the application as part of the document sent to the client, which is then compared with the actual DOM generated by the client's HTML parser. A mismatch indicates that the web page has been injected with untrusted content, a classic XSS attack. Robertson and Vigna [35] enforce the integrity of a web application's DOM using static typing at the server end to prevent both XSS attacks and SQL injection attacks. Both approaches require apriori knowledge of which portions of an application's DOM tree are trusted, and this information may not always be known to the developer. Robertson and Vigna's approach provides a way to automate the generation of such specifications, but requires the developer to use their static type system.

Web tripwire [33] is a system to detect unintended modifications of web pages made in transit or at the client. The server inserts JavaScript code into the web page to compare the page's HTML source after it has been loaded at the client with the original page's HTML. A mismatch indicates that the web page has been tampered with, either in transit to the client (e.g., by a malicious router) or by the client's web browser (e.g., by a malicious browser plugin). However, web tripwire requires exact knowledge of the page's source at the client which is hand coded and inserted into the page. Therefore, it cannot be applied to generic web pages. Furthermore, Web tripwire cannot be used to detect malicious modifications to the page after the check has been performed or by dynamic events in Web 2.0 applications.

## 8 Conclusions

This paper presents a novel approach to enhance the reliability of Web 2.0 applications using DOM-based invariants. The approach dynamically derives invariants on web applications' DOMs and uses them to detect errors. We present *DoDOM*, an automated tool to extract DOM invariants over multiple executions of the application.

We show that (1) DOM invariants exist in real web applications, (2) can be learned using DoDOM within a reasonable number of executions, and (3) are useful in detecting failures of events and domains that impact the DOM.

As future work, we plan to (1) integrate DoDOM as a part of the web browser, (2) extract invariants across different user-interaction sequences, and (3) explore the use of invariants to detect security attacks on web applications.

## Acknowledgements

We thank Trishul Chilimbi, Emre Kiciman, Benjamin Livshits, and Nikhil Swamy for their insightful comments on this paper.

## References


[1] S. Bagchi, G. Kar, and J. Hellerstein. Dependency analysis in distributed systems using fault injection: Application to problem determination in an e-commerce environment. In *Proc. 12th Intl. Workshop on Distributed Systems: Operations & Management*, 2001.

[2] W.I.C. Be. Rich Internet Applications.

[3] M. Benedikt, J. Freire, and P. Godefroid. VeriWeb: Automatically testing dynamic web sites. In *11th International World Wide Web Conference (WWW)*. ACM, 2002.

[4] Cor-Paul Bezemer, Ali Mesbah, and Arie van Deursen. Automated security testing of web widget interactions. In *Foundations of Software Engineering Symposium (FSE)*, pages 81–90. ACM, 2009.

[5] G. Candea, M. Delgado, M. Chen, and A. Fox. Automatic failure-path inference: A generic introspection technique for internet applications. In *3rd IEEE Workshop on Internet Applications (WIAPP)*, 2003.

[6] Mike Y. Chen, Emre Kiciman, Eugene Fratkin, Armando Fox, and Eric Brewer. Pinpoint: Problem determination in large, dynamic internet services. *International Conference on Dependable Systems and Networks (DSN)*, 0:595, 2002.

[7] T.M. Chilimbi and V. Ganapathy. HeapMD: Identifying heap-based bugs using anomaly detection. *ACM SIGPLAN Notices*, 41(11):228, 2006.

[8] R. Chugh, J.A. Meister, R. Jhala, and S. Lerner. Staged information flow for JavaScript. In *ACM SIGPLAN conference on Programming language design and implementation*, pages 50–62. ACM, 2009.

[9] Microsoft Corporation. Fiddler: Web debugging proxy.

[10] Sun Microsystems Corporation. Java Petstore 2.0.

[11] M. Cova, D. Balzarotti, V. Felmetsger, and G. Vigna. Swaddler: An approach for the anomaly-based detection of state violations in web applications. *Lecture Notes in Computer Science*, 4637:63, 2007.

[12] D. Crockford. *JavaScript: the good parts*. O'Reilly Media, Inc., 2008.

[13] S. Di Paola and G. Fedon. Subverting AJAX. In *23rd Chaos Communication Congress*, 2006.

[14] Michael D. Ernst, Adam Czeisler, William G. Griswold, and David Notkin. Quickly detecting relevant program invariants. In *22nd international conference on Software engineering(ICSE)*, pages 449–458, New York, NY, USA, 2000. ACM.

[15] J. Fonseca, M. Vieira, and H. Madeira. Vulnerability and attack injection for web applications. In *IEEE/IFIP International Conference on Dependable Systems and Networks(DSN)*, pages 93–102, 2009.

[16] S. Guarnieri and B. Livshits. GATEKEEPER: Mostly Static Enforcement of Security and Reliability Policies for JavaScript Code. In *Usenix Security Symposium*. Usenix Association, 2009.

[17] A. Guha, S. Krishnamurthi, and T. Jim. Using static analysis for AJAX intrusion detection. In *18th international conference on World wide web*, pages 561–570. ACM, 2009.

[18] S. Hangal and M.S. Lam. Tracking down software bugs using automatic anomaly detection. In *International Conference on Software Engineering*, pages 291–301, 2002.

[19] Yao-Wen Huang, Fang Yu, Christian Hang, Chung-Hung Tsai, Der-Tsai Lee, and Sy-Yen Kuo. Securing web application code by static analysis and runtime protection. In *13th international conference on World Wide Web(WWW)*, pages 40–52. ACM, 2004.

[20] D. Ingalls. The Lively Kernel: just for fun, let's take JavaScript seriously. In *Proceedings of the 2008 symposium on Dynamic languages*, page 9. ACM, 2008.

[21] M. Kalyanakrishnan, R. K. Iyer, and J. Patel. Reliability of internet hosts - a case study from the end user's perspective. In *6th International Conference on Computer Communications and Networks*, page 418, Washington, DC, USA, 1997. IEEE Computer Society.

[22] Emre Kiciman and Benjamin Livshits. Ajaxscope: a platform for remotely monitoring the client-side behavior of web 2.0 applications. pages 17–30, 2007.

[23] A. Le Hors, P. Le Hégaret, L. Wood, G. Nicol, J. Robie, M. Champion, and S. Byrne. Document Object Model (DOM) level 3 core specification. *W3C Recommendation*, 2004.

[24] G. Maone. Noscript-whitelist JavaScript blocking for a safer Firefox experienc e.

[25] A. Marchetto, P. Tonella, and F. Ricca. State-based testing of Ajax web applications. In *IEEE International Conference on Software Testing Verification and Validation (ICST)*, 2008.

[26] A. Mesbah and A. van Deursen. Invariant-based automatic testing of AJAX user interfaces. In *IEEE 31st International Conference on Software Engineering (ICSE)*, pages 210–220. IEEE, 2009.

[27] M. Ohara, P.N.Y. Ueda, and K. Ishizaki. The Data-centricity of Web 2.0 Workloads and its Impact on Server Performance. *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 133–142, 2009.

[28] T. OReilly. What is Web 2.0: Design patterns and business models for the next generation of software.

[29] pc facile.com. Phoenix 1.6.0.

[30] S. Pertet and P. Narasimhan. Causes of failure in web applications. *Parallel Data Laboratory, Carnegie Mellon University, CMU-PDL-05-109*, 2005.

[31] P. Ratanaworabhan, B. Livshits, D. Simmons, and B. Zorn. Jsmeter: Characterizing real-world behavior of JavaScript programs. *Technical Report, MSR-TR-2009-173*, 2009.

[32] P. Reinecke, A.P.A. van Moorsel, and K. Wolter. The fast and the fair: A fault-injection-driven comparison of restart Oracles for reliable Web services. In *QEST*, volume 6, pages 375–384. ACM, 2005.

[33] C. Reis, S.D. Gribble, T. Kohno, and N.C. Weaver. Detecting in-flight page changes with web tripwires. In *5th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 31–44. USENIX Association, 2008.

[34] F. Ricca and P. Tonella. Analysis and testing of web applications. In *International Conference on Software Engineering*, volume 23, pages 25–36, 2001.

[35] W. Robertson and G. Vigna. Static Enforcement of Web Application Integrity Through Strong Typing. In *Usenix Security Symposium*. Usenix Association, 2009.

[36] J. Ruderman. The Same Origin Policy.

[37] M. Ter Louw and VN Venkatakrishnan. Blueprint: Robust prevention of cross-site scripting attacks for existing browsers. In *30th IEEE Symposium on Security and Privacy (Oakland)*, pages 331–346, 2009.

[38] M. Vieira, N. Laranjeiro, and H. Madeira. Benchmarking the Robustness of Web Services. In *The 13th IEEE Pacific Rim Dependable Computing Conference (PRDC 2007), Melbourne, Victoria, Australia*, 2007.

[39] C. Yue and H. Wang. Characterizing insecure JavaScript practices on the web. In *18th international conference on World wide web*, pages 961–970. ACM, 2009.

# APPENDICE D

# QUATRIÈME ANNEXE

2010 IEEE 21st International Symposium on Software Reliability Engineering

# DoDOM: Leveraging DOM Invariants for Web 2.0 Application Robustness Testing

**Karthik Pattabiraman**
University of British Columbia
*karthikp@ece.ubc.ca*

**Benjamin Zorn**
Microsoft Research (Redmond)
*ben.zorn@microsoft.com*

***Abstract*—Web 2.0 applications are increasing in popularity. However, they are also prone to errors because of their dynamic nature. This paper presents *DoDOM*, an automated system for testing the robustness of Web 2.0 applications based on their Document Object Models (DOMs). DoDOM repeatedly executes the application under a trace of recorded user actions and observes the client-side behavior of the application in terms of its DOM structure. Based on the observations, DoDOM extracts a set of invariants on the web application's DOM structure. We show that invariants exist for real applications and can be learned within a reasonable number of executions. We further use fault-injection experiments to demonstrate the uses of the invariants in detecting errors in web applications. The invariants are found to provide high coverage in detecting errors that impact the DOM, with a low rate of false positives.**

***Keywords*-Web 2.0, Dynamic Invariants, Robustness Testing, Error Detection**

## I. Introduction

The web has evolved from a static medium that is viewed by a passive client to one in which the client is actively involved in the creation and dissemination of content. The evolution is enabled by the use of dynamic technologies such as JavaScript, Silverlight, and Flash, which allow the execution of client-side scripts in the web browser to provide a rich, interactive experience for the user. Applications deploying these technologies are called *Rich Internet Applications* or *Web 2.0* applications. Web 2.0 applications, such as Gmail and Facebook, require little installation or maintenance and have the ability to run on a wide variety of platforms [1]. Consequently, they are increasing in popularity and are being rapidly adopted.

Unfortunately, Web 2.0 applications are also complex and prone to errors. First, the distributed nature of the application's logic between the server and client makes it difficult to understand and debug such applications. This problem is compounded by the asynchronous behavior of the application. Second, web applications often integrate data and code from multiple domains, and the failure of any of these domains can make the application unstable. Finally, unlike desktop applications, web applications are rarely designed with a fail-fast philosophy. Rather, they tend to continue executing even if some component of the application experiences an error (e.g., an event-handler throws an exception). This behavior has its roots in the historical evolution of the web and is advantageous from the point of view of compatibility and co-existence. However, it makes it difficult to contain and localize faults and is hence disadvantageous from the point of view of reliability. Therefore, it is important to test the robustness of web applications using fault-injection experiments.

An important challenge in testing Web 2.0 applications' robustness is lack of determinism from one execution to another (even with the same input sequence). This is due to a number of factors, including (1) small changes introduced by the server, (2) asynchrony in network messages being sent or received out of order, and (3) small variations in the timing of events at the client. The non-determinism makes it difficult to ascertain if an observed change in the execution of a web application was because of an injected fault. Non-determinism has been addressed in the context of Web 1.0 applications by controlling server-side execution [2]. However, these solutions address only the first source of non-determinism in Web 2.0 applications.

In this paper, we propose a first step towards characterizing the client-side behavior of web applications with the goal of testing their robustness using fault injection. Our approach characterizes the structure of the Document Object Model (DOM) data-structure in the application using dynamically derived invariants, and considers deviations from the derived structure as erroneous executions. The invariants are derived dynamically by recording a sequence of user interactions with the application and observing the changes to the DOM by repeatedly replaying the sequence. Because we repeatedly execute the web application and derive invariants over its DOM, we call this approach DoDOM (the DO symbolizes iteration as in a do-while loop).

The DOM is the central data structure maintained by the web browser for representing and displaying a web application's output. It is organized hierarchically with each node in the DOM representing an entity of the web page corresponding to the application (e.g., list elements, text). Only objects in the DOM can be displayed by the web browser. Hence, the DOM is what users ultimately see and interact with in a web application. Further, multiple client-side scripts in the application share state and communicate with one another predominantly through the DOM. *Thus, the correctness of the DOM is essential for the correctness of the application and hence we focus on DOM-based invariants in this paper.*

DoDOM extracts invariants over the DOM structure of a web application by capturing a user-interaction sequence with the application, replaying it multiple times and observing the DOM after each execution. This dynamic approach of

1071-9458/10 $26.00 © 2010 IEEE
DOI 10.1109/ISSRE.2010.17

learning invariants is language-neutral and does not need to statically analyze the code of the web application. This is important as (1) languages and frameworks for writing web applications are rapidly evolving and a single application may combine code from multiple languages and frameworks, and (2) client-side languages such as JavaScript are notoriously difficult to analyze statically due to the presence of dynamic constructs [3].

Prior work has shown that dynamic invariants can be used in general-purpose programs for testing and error detection [4], [5], [6]. However, there has been relatively little work on deriving dynamic invariants for web applications. We show that DOM-based invariants (1) exist in web applications, (2) can be learned automatically, and (3) are useful in detecting errors. *To the best of our knowledge, DoDOM is the first technique to automatically extract invariants in Web 2.0 applications for the purpose of error detection*[1].

The main contributions of the paper are as follows:

- We show that Web 2.0 applications exhibit invariants over their DOM structures, and build a tool called DoDOM to replay web applications and extract their invariants.
- We demonstrate the invariant extraction capabilities of DoDOM for three real Web 2.0 applications: Slashdot, CNN, and Java Petstore. We show that the invariants can be learned within 6 executions of each application.
- Using fault-injection experiments, we find that the error-detection coverage of the invariants is close to 100% for errors that impact the web page's DOM. The remaining errors do not have any effect on the DOM and are hence not detected by the technique[2].

## II. Overview

In this section, we outline the reliability issues with Web 2.0 applications and present our proposed solution. We first present a brief background on Web 2.0, which may be skipped by the reader familiar with this paradigm.

### A. Background

Typical Web 2.0 applications consist of both server-side and client-side code. The server-side code is written in traditional web-development languages such as PHP, Perl, Java and C. The client-side code is written using dynamic web languages such as JavaScript (JS) which are executed within the client's browser. Unlike in the Web 1.0 model, where the application executes primarily at the server with the browser acting as a front-end for rendering and displaying the server's responses, in a Web 2.0 application the client is actively involved in the application's logic. This reduces the amount of data exchanged with the server and makes the application more interactive. *Henceforth, when we say web applications, we mean Web 2.0 applications unless we specify otherwise*.

[1]ATUSA [7] proposes the use of invariants for error detection in Web 2.0 applications but requires the programmer to define the invariants.

[2]Such errors may be detected through checks in the application's backend, but are outside this paper's scope.

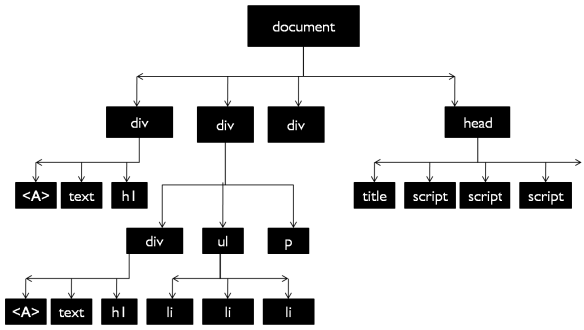



Fig. 1. Example of a DOM tree for a web application.

Typical web applications are event-driven, i.e., they respond to user events (e.g., mouse clicks), timeouts and receipt of messages from the server. The developer writes handlers for each of these events. The event-handlers can (1) invoke other functions or write to global variables stored on the heap, (2) read/modify the web page's contents through its DOM, or (3) send asynchronous messages to the server through a technology known as AJAX (Asynchronous JavaScript and XML) and specify the code to be executed upon receiving a response. The above actions are executed by client-side scripts in the web browser.

Web applications typically execute within a single web page. A web page is represented internally by the browser as its DOM [8], which as mentioned before, consists of the page's elements organized in a hierarchical format. Figure 1 shows an example of a DOM for a web page. In the figure, the web page consists of multiple HTML div elements, which are represented in the top-level nodes of the tree. The "div" elements are logical partitions of the page, each of which consists of nodes representing text and link elements. Further, the web page has a head node with two client-side scripts as its child nodes. JavaScript code executing in the web browser can read and write to the DOM through special APIs. Any changes made to the DOM cause the web page to be re-rendered by the web browser. User actions are converted into events by the browser and sent to the nodes of the DOM on which they are trigerred. If the node has an installed event handler for the event, then the event-handler is executed.

### B. Scenario

Consider an application developer who wants to test the robustness of a web application to faults. She would interact with the application on the client, inject a fault into it and check if the application behaves as expected. However, this approach is time consuming and requires the user to repeat the same interactions with the application for each injected fault. Further, the user needs to rely on visual perception to determine whether an injected fault affects the application[3]. Finally, web applications exhibit variations in their behavior from one execution to another (as we show later in this paper).

[3]While it can be argued that faults that are not perceived by the user do not matter, it may be that a different user perceives the fault.

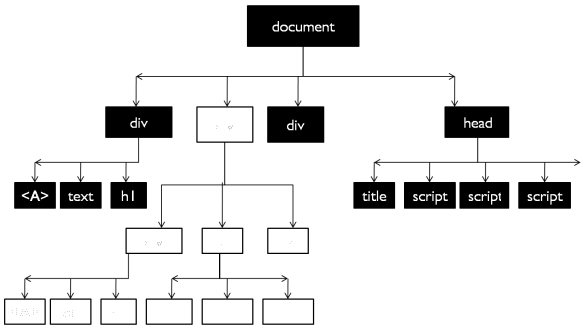



Fig. 2. Proposed solution in the context of the example DOM.

In the face of such variations, it becomes challenging to identify whether a perceived difference is the result of a fault or if it is due to the natural behavior of the application. Further, the variations can occur in the middle of the web application's execution which makes them difficult to detect.

This paper proposes a systematic method to characterize the correct behavior of a web application for robustness testing. We assume that the web developer has one or more user interaction sequences (i.e., sequences of user actions) under which she wants to test the application's robustness. Our solution involves extracting an invariant characteristic of the web application's DOM from multiple executions of the application. We characterize the expected behavior of the application based on the invariants. We then inject faults into the application and consider significant deviations from the invariant behavior of the application as erroneous executions.

### C. Dynamic Invariant Extraction

This section illustrates our proposed solution for the problem illustrated in Section II-B. The crux of the solution is in characterizing the invariant portions of the DOM for the web application under a given sequence of user interactions. Specifically, we characterize the common portions of the DOM of the application's web page over multiple executions, and the changes made to the DOM by the application in response to various events (i.e., user actions, timeouts, and network messages). After each event executes, we check the conformance of the resulting DOM to the invariant portions. A deviation indicates an error. We first illustrate using the example in Figure 1 and then present the general case.

Figure 2 shows the invariant portion of the DOM for the example considered in Figure 1. In the figure, the darkly-shaded nodes represent the invariant portions of the tree (also called the web page's backbone) while the lightly-shaded nodes represent the non-invariant portions. We consider two example faults to illustrate the error-detection process. First, consider the case where the user clicks on a specific DOM node which in turn triggers an event handler on the node. The event handler is supposed to update the left most element (A) in the invariant DOM but fails to do so because of an error (e.g., the handler throws an exception). This error will be detected as the resulting DOM would deviate from the invariant DOM for the event (in this case, the event is the mouse click).

| Executions | Event 1 | Event 2 | ... | Event n |
|---|---|---|---|---|
| Execution 1 | $T_{1_1}$ | $T_{2_1}$ | ... | $T_{n_1}$ |
| Execution 2 | $T_{1_2}$ | $T_{2_2}$ | ... | $T_{n_2}$ |
| ... | ... | ... | ... | ... |
| Execution M | $T_{1_M}$ | $T_{2_M}$ | ... | $T_{n_M}$ |
| Invariants | $T_{1_I}$ | $T_{2_I}$ | ... | $T_{n_I}$ |

TABLE I
INVARIANT DOMS.

Second, assume that the web application is supposed to import a script from a domain but fails to do so because of the domain being unavailable. Assume that the script is supposed to modify the 'div' element in the far left branch of the tree. This element is part of the invariant DOM and hence the lack of modification will be detected as an error. *Our hypothesis is that the majority of the DOM structure is invariant in a web application and hence the invariant DOM can be used to detect the majority of errors in a web application.*

In the above example, the invariant DOM in Figure 2 represents a snapshot of the DOM during the course of its evolution in response to events. In reality, the technique will capture the entire sequence of invariant DOMs and use the invariant sequence to check for deviations as we show below.

An invariant DOM is a subset of the web application's DOM that is shared by multiple executions of the application. We derive an invariant DOM for each event in the application (an event refers to a user action, network message or timeout). Table I shows the DOMs obtained during multiple executions of the web application under a given sequence of events. The rows of the table indicate different executions of the web application, while the columns indicate the events in the application. The DOMs are indicated by $T_{i_j}$, where $i$ is the event after which the DOM is obtained and $j$ is the corresponding execution. The invariant DOMs $T_{i_I}$ are derived from the DOMs of individual executions $T_{i_j}$ corresponding to the event $i$. As can be seen from the table, the invariant DOMs generalize across multiple executions to incorporate only the common features of each DOM. However, they are specific to a given sequence of events in order to ensure high error-detection coverage. We consider the implication of this trade-off in Section VI.

### D. Fault Model

This section discusses the errors injected in this study to evaluate the derived invariants. Note that we inject errors (i.e., manifestations of faults) rather than the underlying faults. However, in keeping with convention, we refer to these experiments as *fault injections*.

**Event errors**: These correspond to errors encountered when processing events in the client-side code of the application. These can be caused by exceptions in the corresponding event handlers or the events not being triggered correctly. In some cases, the web browser may abort an event handler if it executes for too long.

**Domain failures**: These can be caused by a network failure or by the unavailability of the domains' servers. They can also be caused by client-side plugins or administrative proxies which may block scripts from certain domains.

While the above errors may seem somewhat simplistic, it is a first step towards characterizing faults in Web 2.0 applications. Prior work has characterized fault-models for Web 1.0 applications [9], which comprise only server-side code. However, to the best of our knowledge, there has been no similar effort to characterize common failure modes of Web 2.0 applications which involve significant amounts of client-side processing. We also note that our method to extract invariant DOMs is independent of the fault model, which is only used to evaluate the invariants. Future work will attempt to extend the scope of the injections and consider more realistic faults.

## III. Approach

The overall approach for extracting and learning invariants over the DOMs of web applications consists of the following steps: First, we record a sequence of user interactions and events on a page (trace). We then replay this trace over multiple executions and capture the sequence of DOMs generated after each event in the trace. Finally, we extract invariants over the set of all DOM sequences using an offline learning process. We built a tool, DoDOM, to automate the process of recording a user-interaction sequence with a web application, replaying it, and extracting invariants from the observed DOM sequences.

In this section, we first describe the challenges encountered in the above process and then discuss how DoDOM address these challenges. We also discuss the design choices and the trade-offs made in DoDOM.

### A. Challenges

We identify three main challenges in extracting invariants using DoDOM. First, we need to record the sequence of user-interactions with the web application in an unobtrusive manner because we want to obtain realistic user-interaction sequences. Second, we need to replay the user-interaction events on the web application on the same DOM nodes on which they occurred when recording the sequence. In particular, the web page rendered by the application may undergo minor changes from one execution to another (because of server-side changes[4]) and hence the replay should be robust to such changes. Finally, it is desirable that the method be independent of the web browser and that it not require any changes to the same.

DoDOM addresses the above challenges as follows. First, during recording, it passively records the interaction of a user with the web application without requiring the user to perform any additional steps. Second, in order to find nodes during replay, it uses both the contents of a node in the DOM and its relative position to other nodes. Further, the comparison is not exact, but is based on heuristic measures, thereby ensuring that events are played back on the original nodes on which they occurred even if the web page has undergone small changes during replay. Finally, the DoDOM tool is predominantly implemented using JavaScript (with a small portion implemented as a proxy server), and hence does not require any modifications to the web browser.

[4]For ease of deployment, we do not require the execution of the server code to be controlled. This is especially important when testing real web sites.

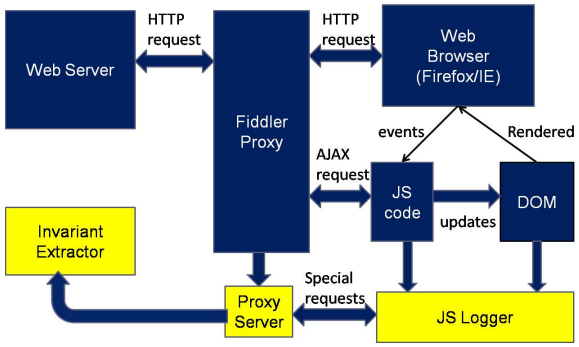



Fig. 3. Architecture of DoDOM system: The components we added are lightly shaded.

### B. DoDOM Operation

Figure 3 shows the architecture of *DoDOM*. DoDOM consists of three components as follows.

(1) The **proxy** is a client-side proxy server written as a plugin in the Fiddler web application testing framework [10]. The proxy's main purpose is to inject the JS logger code into the web page(s) of the application, collect the events and responses sent by the JS logger and record them.

(2) The **JS logger** is a piece of JavaScript code that is executed on the client's browser for each iFrame in the page. The JS logger can read/write to the web page's DOM, install event handlers that trap the page's handlers and log changes to the DOM. It can also intercept messages sent by the client through the XMLHttp interface and the corresponding response (i.e., AJAX messages).

(3) The **invariant extractor** performs offline analysis of multiple executions recorded by the proxy and extracts the invariants. It runs outside the web browser.

The proxy injects the JS logger script into every page loaded by the browser (a page is defined as any entity that consists of a head tag). The proxy also assigns to each JS logger script a unique tag called the pageID. The pageID is used to distinguish individual iframes in a multi-frame web application. The JS logger script is instantiated at the client after the page completes loading (after the *onLoad* event), upon which it performs the following actions (in sequence): (1) Creates a compact representation of the web page's DOM, and sends it back to the proxy. (2) Installs a new replacement handler for all DOM elements that have event handlers and stores the old handler as part of the element. (3) Replaces the *setTimeout* and *setInterval* API calls in the window object with custom versions (after storing the old handler) to intercept timeouts. (4) Replaces the *XMLHttpRequest* object with a

custom version that intercepts all messages sent to the server using the AJAX interface and their corresponding responses. (5) Installs change handlers on each element of the DOM tree to track any additions, modifications, and removals of the sub-tree rooted at the element. The above operations are performed using the DOM API calls in JavaScript.

The JS logger operates in two modes: record and replay. During *record mode*, the user interacts normally with the page by moving the mouse, clicking on objects, etc. The browser translates the user's actions into user-events and invokes the replacement event handlers installed by the JS logger on the corresponding DOM nodes. The handlers create a snapshot of each event and send it to the proxy, which in turn adds the events to a global queue for the page. The JS logger periodically polls the proxy for outstanding events, upon which the enqueued events are sent to the client (in order). The JS logger then invokes event's original handers on the node on which they occured.

During *replay*, the proxy reads in the list of events from the event log and populates the queue with the events. It injects the JS logger into the web page as before. However, when the JS logger polls for events, the proxy retrieves the events from the queue and sends them to the client one at a time along with the corresponding node on which the event occurred[5]. The JS logger attempts to identify the node using the node's contents sent by the proxy and its relative position in the DOM tree (i.e., pre-order traversal index). If an exact match is not found, it searches the DOM for the closest match starting from the node with the same pre-order index as the original node[6].

The proxy records the changes made to the web page's DOM tree after every event. The invariant extractor post-processes these traces to obtain a sequence of invariant DOM trees for each event. Each tree in the invariant sequence is learned independently based on the corresponding trees in the individual executions.

The algorithm for learning each of the invariant DOM trees from a set of execution traces is as follows (the pseudo-code for the complete algorithm can be found in [11]). We set the initial invariant tree to the tree obtained from the first execution trace. For each event in the execution trace, we compare its tree with the corresponding invariant tree recursively starting from the root nodes and traversing the tree in post order. When any of the following three conditions are met, the node is removed and a dummy node substituted in its place to hold the tree together. (1) the contents of a node of the invariant tree do not match the corresponding nodes in the execution tree, (2) a node in the invariant tree has more children than its corresponding node in the execution tree, (3) the invariant tree has nodes that are not present in the execution tree. The comparison among nodes' contents is based on the fraction of its field-value pairs that match each other. If this fraction is higher than a value known as the *match threshold*, then the nodes are considered to match each other. The comparison does not take into account the order of the field-value pairs in each node.

The match threshold thus determines how much of the invariant tree is pruned away because of differences among the DOMs of individual executions. A high match threshold means that only nodes that match closely across executions will be retained in the invariant tree. On the other hand, a low match threshold indicates that that the invariant tree may contain nodes that exhibit high variation in their contents among executions. Therefore an invariant tree with a high match threshold has high content similarity with individual executions.

### C. Trade-offs and Limitations

For portability and ease of deployment, DoDOM is implemented predominantly in JS with a small part implemented as a client-side proxy. However, the use of JS incurs certain limitations. First, the JS logger is executed only after the *onLoad* event in a web page. Therefore, it cannot trap events that occur before the firing of the onLoad event and would hence ignore them. Second, DoDOM traps events by hooking into the event handlers of DOM nodes and replacing the existing functions with a custom wrapper. This behavior requires the web application to use the DOM 1.0 event model because the DOM 2.0 event model does not provide any way to remove a handler from the chain of event-listeners on a DOM node, or to ensure that the event handlers are invoked in a specific order. However, most applications are written using the DOM 1.0 model for compatibility reasons[7]. Finally, DoDOM's reliability is limited by the reliability of the web browser's JavaScript Virtual Machine (JS VM). In our experiments, we did not find cases where the JavaScript VM crashed or hung. Nonetheless, we provide a heartbeat service to detect crashes or hangs of the JS VM and reload the page.

Currently, DoDOM only supports single web page applications, i.e., web applications where the user interacts with the application on a single web page. However, even single web page applications often consist of multiple iframes each of which contains its own DOM. The user may interact with multiple iframes, and therefore it is necessary to record the interactions on a per-frame basis. This is done by assigning a unique tag to each frame (pageID) and using the tag to disambiguate interactions for different iframes. A similar mechanism can be applied for multi-page web applications which consist of multiple pages displayed in sequence.

Finally, DoDOM assumes that the web application uses standard mechanisms such as AJAX for communication between the client and the server. However, some frameworks such as Dojo use their own custom mechanisms for handling communications with the server (e.g., hidden iframes). DoDOM cannot currently support such mechanisms.

[5]It also introduces a time delay corresponding to the occurrence of the event in the recording mode.

[6]The closest match is the node(s) with the highest fraction of matching field-value pairs.

[7]The DOM 3.0 specification allows for removing event handlers and controlling their order, but unfortunately, is not implemented by most browsers.

## IV. Experimental Setup

This section describes the experiments performed and the benchmarks used to evaluate DoDOM. We used an Intel Core2 Duo Intel dual-core processor (running at 3 GigaHertz) with 4 GigaBytes RAM. We used Firefox version 3.5 on Windows Vista as the platform for evaluation.

The main research questions are:

*Q1. How many executions do we need to learn the invariant DOMs for a web application?*

*Q2. How many event errors impact the web application's DOM and how effective are the invariants at detecting these errors?*

*Q3. How many domain failures impact the web application's DOM and how effective are the invariants at detecting these errors?*

**Invariant Extraction**: The goal of this experiment is to answer Q1. We first record the sequence of user interactions and create a log of events. We then replay the user events multiple times using DoDOM. From the set of all executions (i.e., replay sequences), we randomly choose a subset of executions used to learn the invariants, called the *training set*. In these experiments, we vary the size of the training set in order to understand how quickly the invariants converge to a stable value. We also vary the match threshold described in Section III to understand how much variation is present among individual executions.

We measure the following characteristics of the DOM tree in order to measure its convergence. (1) number of nodes in the DOM, (2) average number of children per node, i.e., its fanout, (3) maximum number of levels from each node, i.e., the height of the sub tree, and (4) average number of total descendants per node.

We also compare the invariant DOM sequence learned from the training set with the DOM sequences from all executions. If any of the DOMs in its sequence exhibits a mismatch with the corresponding DOM in the invariant sequence, we consider the execution a false positive.

**Event Errors**: The goal of this experiment is to answer Q2. We measure the error-detection coverage of the invariant sequences for event errors corresponding to those in Section II-D. Table II shows the types of faults introduced and the injection method. Each run injects at most one fault to ensure that the fault's effects can be uniquely determined. After a fault is injected, the sequence of DOMs corresponding to the execution is compared with the sequence of invariant DOMs. We classify the execution as a successful detection if any of the DOM trees in the sequence exhibits a mismatch with the corresponding DOM in the invariant sequence. The coverage obtained by DoDOM for a fault is calculated as the percentage of successful detections among the total number of executions corresponding to the fault.

**Domain Failures**: The goal of this experiment is to answer Q3. It emulates the effect of domain failures as described in Section II-D using the NoScript plugin in Firefox[8]. First, the invariant DOM sequence is obtained from multiple executions. Then, each domain in the web page is blocked one at a time and the corresponding DOM sequences are obtained. The DOM sequence for a blocked domain is compared to the invariant DOM sequence, and a mismatch indicates that the domain's failure is detected by the invariants.

| Fault Type | Injection Method |
|---|---|
| User-Event Error | Do not replay the event at the client |
| Message Error | Do not forward the message to the server |
| Timeout Error | Do not replay the timeout at the client |

TABLE II
Faults injected and their characterization.

| Website | Lines of JS code | No. of domains | No. of events | No. of DOM nodes |
|---|---|---|---|---|
| Java Petstore | 499 | 1 | 211 | 398 |
| Slashdot | 9647 | 5 | 13 | 5162 |
| CNN | 15603 | 9 | 9 | 2417 |

TABLE III
Characteristics of the web applications.

**Benchmarks**: We demonstrate DoDOM on three representative Web 2.0 applications: Slashdot, CNN and JavaPetStore. Slashdot aggregates technology-related news from different web sites and allows users to comment on a news story. Java Petstore is a freely available Web 2.0 application that mimics an e-commerce web site for buying pets [12]. CNN is a popular news web site that delivers customized content to its readers.

Table III summarizes the characteristics of the web applications which include the number of domains in the application, the total number of lines of JS code (obtained with Firefox's Phoenix plugin[9]), and the number of events in the recorded trace for the application.

We choose the Slashdot application as the primary source of measurements as it represents a middle ground among the applications in terms of lines of code and number of domains. We interact normally with a Slashdot news story and replay the interactions with DoDOM. The results reported are for a specific news story on Slashdot with close to 300 comments. We obtain a total of 13 events for the story including user interactions and timeout, and perform a total of 58 replays. We also repeated the measurements with a different sequence of interactions, but the results were similar and are not reported.

For the other two applications, we measured only the invariant extraction capabilities of DoDOM, using 50 replays each. Java Petstore allows the user to browse through pet listings and choose a pet corresponding to the user's preferences. We interact with the first page of the application by moving over and clicking on different elements of the page. For CNN, we interact with the main page of the application, which displays the daily news, by moving the mouse over various elements of the page and clicking on them as a normal user would.

[8]Available at *http://noscript.net/*.

[9]Available at *https://addons.mozilla.org/en-US/firefox/addon/11708/*.

## V. RESULTS

In this section, we present the results corresponding to the research questions in Section IV. We first summarize the main results and then present the details. The results presented pertain to the Slashdot application. We present the results for the other applications at the end of this section.

**R1: corresponds to Q1**: We show that the invariant DOM sequences converge with a training set size of 6 executions, which corresponds to 10% of the total executions. We further show that the invariant DOM converges to 99% of the original DOM size, suggesting that most of the DOM is invariant.

**R2: corresponds to Q2**: The invariants detect 100% of the injected event errors that affect the DOM.

**R3: corresponds to Q3**: Only one the 5 domains included by Slashdot has an effect on the DOM. DoDOM provides 100% coverage for failures of this domain.

**Invariant Extraction**: In this experiment, we vary the training set size[10] from 1 to 10 over 58 executions and obtain the invariant DOM sequence. The characteristics of the invariant DOM corresponding to the metrics listed in Section IV are shown in Figure 4 (a) to (d). In each graph in the figure, the X-axis represents the event number and the Y-axis represents a metric corresponding to the event. The lines in each graph represent the invariants obtained with a training set of a specific size. Note that the Y-axis in each graph does not start from 0, and there is very little variation among the different training set sizes.

We observe that the number of nodes monotonically increases with the event number, while the maximum number of levels in the DOM monotonically decreases with the event number. The other two metrics, namely the number of children and the number of descendants, show no consistent trend.

Figure 4 (a) shows that as the training set size increases, the number of nodes in the DOM decreases because more and more nodes are eliminated from the invariant DOMs. However, the number stops decreasing once the training set reaches a size of 6 (roughly 10% of the 58 executions). Nonetheless, the converged values are within 1% of their original values (i.e., the number of DOM nodes for any given execution). This shows that the amount of variation among executions, while non-trivial, is within 1% of each other. Similarly, the other three metrics, namely the maximum number of levels, the average number of children, and the average number of descendants, steadily increase with increase in the training set size, but also stabilize at a training set size of 6. This shows that the invariant DOMs can be learned using only 10% of the executions.

**False Positives**: To further confirm the convergence of the invariants, we measure the false-positive rate for the executions[11]. Table IV shows the false-positive rate as a function of the size of the training set for different values of the match threshold (introduced in Section III). As can be seen in the table, the false-positive rate initially starts out high when the training set is very small (2 executions), but quickly decreases with increase in the training set size. For a training set size of 6 or more, the false-positive rate is nearly zero. This confirms the earlier observation that a training set size of 6 is the point at which the invariant DOMs stabilize.

| Training Set Size | Match Threshold | | |
|---|---|---|---|
| | 0.95 | 0.50 | 0.05 |
| 2 | 53 | 53 | 53 |
| 4 | 29 | 29 | 29 |
| 6 | 1 | 1 | 0 |
| 8 | 1 | 1 | 0 |
| 10 | 1 | 1 | 0 |

TABLE IV
FALSE POSITIVES VERSUS TRAINING SET SIZE OF A TOTAL OF 58 EXECUTIONS.

Interestingly, the false positives do not drop to 0 when the match thresholds are 0.95 and 0.50, but remain stable at 1 up to a total of 10 executions. This suggests that one of executions exhibits significant content differences from the invariant DOM. Nonetheless, the false-positive curve drops to 0 when the match threshold is decreased to 0.05, suggesting that the execution conforms to the structural characteristics of the invariant DOM. We cannot fully explain this deviation as we are using a live web-site whose server-side code is not available to us.

**Coverage for Event Errors**: We measure the error-detection coverage of the invariants for event errors through fault-injection experiments shown in Table II. For each event in the trace (13 in all), we inject a fault corresponding to the event and compare the resulting DOM sequence with the invariant DOM sequence. A mismatch among the sequences indicates that the fault was successfully detected. For each fault, the application is executed five times, so there are a total of 65 executions (= 5 * 13) performed in this experiment. Table V shows the error-detection coverage for each fault (event-number) that was injected. Based on the previous results for false positives, we focus on the invariants derived with training sets of 6 or more. As before, the match threshold was set to 0.95 for these experiments.

Table V shows that the detection rate of the invariants is either 0% (0 detections) or 100% (5 detections) depending on the injected fault. The reason for the differences among events is as follows. Either an event-handler affects the DOM or it does not. The events that have 0 detection rates (namely 1, 3, 7, 9, 11 and 12) are timeout events, and their handlers do not update the DOM, an observation confirmed by examining the handler's source code). The other events are mouse-clicks and message-handling events, and the corresponding handlers either add or remove nodes from the DOM. Thus, the invariants detect all event errors that affect the DOM.

**Domain Failures**: The goal of this study is to measure the error-detection coverage of the invariant DOMs for failures of

[10]We also varied the inputs included in the training set, but the results were similar and are hence not reported.

[11]Recall that a false positive is an execution that deviates from the invariant DOM sequence derived from the training set.

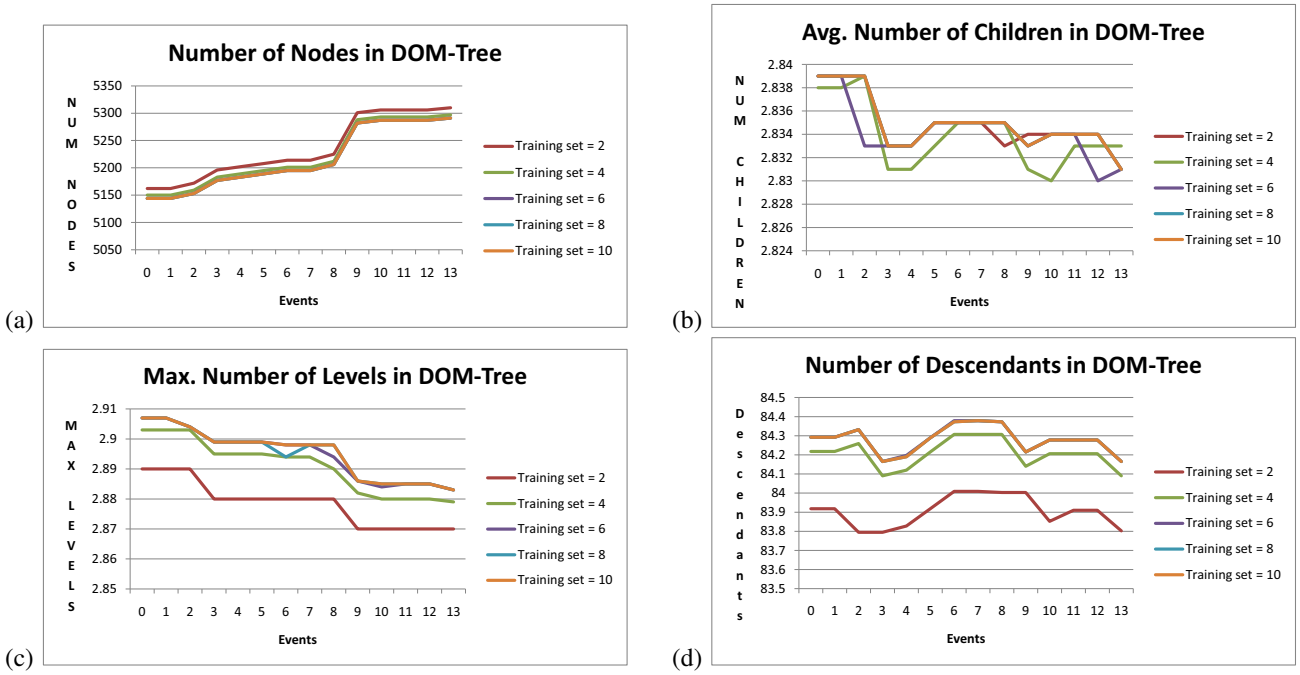



Fig. 4. Invariant Characteristics of the DOM.

| Event No. | Affects DOM? | Injected | Detected |
|---|---|---|---|
| 1 | No | 5 | 0 |
| 2 | Yes | 5 | 5 |
| 3 | No | 5 | 0 |
| 4 | Yes | 5 | 5 |
| 5 | No | 5 | 0 |
| 6 | Yes | 5 | 5 |
| 7 | No | 5 | 0 |
| 8 | Yes | 5 | 5 |
| 9 | No | 5 | 0 |
| 10 | Yes | 5 | 5 |
| 11 | No | 5 | 0 |
| 12 | No | 5 | 0 |
| 13 | Yes | 5 | 5 |

TABLE V
ERROR-DETECTION COVERAGE OF THE DOM INVARIANTS.

| Domain Name | Affects DOM? | Total executions | No. of detections |
|---|---|---|---|
| No domain | No | 90 | 0 |
| doubleClick.net | No | 16 | 0 |
| fsdn.com | Yes | 27 | 27 |
| Google Ads | No | 31 | 0 |
| mediaplex.com | Maybe | 81 | 2 |
| 2mdn.com | No | 25 | 0 |

TABLE VI
DOMAIN FAILURES FOR SLASHDOT: THE LEFT-MOST COLUMN SHOWS THE BLOCKED DOMAIN.

the domains in Slashdot. The experiment involves blocking each domain in the application using the NoScript plugin and replaying the events with DoDOM. Table VI shows the results. In Table VI, the first column shows the blocked domain while the second and third column respectively show the number of executions performed for that domain and the number of executions that resulted in a mismatch between the invariant DOM and the observed DOM[12]. The *no domain* case in which no domain was blocked, yields 0 mismatches, showing that the invariants incur no false positives for this experiment.

We consider two questions with Table VI. First, how many domains affect the DOM and second, how many of these are detected using the invariants. From the table, one can observe that only *Fsdn.com* and *mediaplex.com* exhibit mismatches between the invariant and observed DOMs. Of the two domains, *mediaplex.com* differs from the invariant DOM in only 2 executions out of over 80 executions. Hence, these two executions are likely false-positives. On the other hand, Fsdn.com exhibits mismatches in 27 of 27 executions. Therefore, this domain likely has an influence on the DOM, and its failure is detected by the derived invariants. Hence, the invariants detect failures of all domains that affect the DOM.

**Other applications**: We run DoDOM on two other web applications, namely *CNN* and *Java PetStore*, to test its invariant extraction capabilities. As before, we measure the convergence of the invariant DOMs for the two applications as a function of the training set size. Table VII shows the results for both applications. The table focuses on the final events in the applications' traces. As can be seen in the table, the invariant DOMs stabilize with a training set size of 6 for both applications. Similar results were obtained for the other events, but are not presented due to space constraints.

[12]The number of executions is different for different blocked domains, because we capped the total time for each experiment to 30 minutes.

| *Training Set* | **Java PetStore** | | | | **CNN** | | | |
|---|---|---|---|---|---|---|---|---|
| *Size* | *NumNodes* | *NumChildren* | *MaxLevels* | *Descendants* | *NumNodes* | *NumChildren* | *MaxLevels* | *Descendants* |
| 2 | 397 | 3.04 | 2.81 | 44.2 | 2413 | 2.47 | 2.63 | 48.4 |
| 4 | 397 | 3.04 | 2.94 | 44.2 | 2408 | 2.47 | 2.63 | 48.5 |
| 6 | 387 | 3.10 | 2.94 | 45.5 | 2407 | 2.47 | 2.63 | 48.5 |
| 8 | 387 | 3.10 | 2.94 | 45.5 | 2407 | 2.47 | 2.63 | 48.5 |
| 10 | 387 | 3.10 | 2.94 | 45.5 | 2407 | 2.47 | 2.63 | 48.5 |

TABLE VII
RESULTS FOR JAVA PETSTORE AND CNN APPLICATIONS.

**Threats to Validity**: An internal threat to validity is the limited number of applications we examined in the study. We chose popular Web 2.0 applications without apriori knowledge of their behavior. However, it is possible that there are web applications that do not exhibit any invariants over their DOMs (see Section VI). An external threat to validity is that our fault-injection is limited to event errors and domain failures. While DOM invariants are effective at detecting the injected errors, it is possible that they may not detect more subtle bugs. Future work will consider more extensive fault models.

**Performance overhead**: We also evaluate the performance overhead of the DoDOM tool. The JS logger consists of about 1500 lines of JavaScript code. When compressed, this code occupies 16.5 Kilobytes, which constitutes less than 10% of the code loaded by typical Web 2.0 applications [13]. Further, the JS logger incurs an overhead of 3.5 seconds to traverse the entire DOM and install event handlers after the page is loaded. The time taken for Slashdot to finish loading is approximately 10 seconds, so DoDOM adds an overhead of 35% to the initial load time. However, once the page has been loaded, DoDOM incurs negligible overhead in capturing and replaying the events in the trace. Finally, the invariant extraction procedure is performed offline and hence does not contribute to the performance overhead of DoDOM.

## VI. DISCUSSION

In this paper, we consider a single kind of DOM invariants, namely those that are specific to a given user interaction sequence. We showed that such invariants are highly effective at detecting errors in the application and can be used in robustness testing of the application. However, it may be possible to generalize the invariants across multiple user-interaction sequences, in order to capture the most typical behavior of the web application under common usage scenarios. Such invariants may have broad uses beyond error detection. We consider some of the use cases below.

**Dependability Benchmarking**: One of the main challenges in benchmarking the dependability of web applications is in ensuring the repeatability of the benchmarking experiments [14]. This is because web applications exhibit a high degree of variation from one execution to another, and it is challenging to obtain a characterization of the common aspects of the application across different executions. Such a characterization can be provided by the invariants.

**Security Enforcement**: The invariants can also be used to check if a web page has been tampered with either during transmission or rendering at the client. This is similar to the Web Tripwire project [15], with the difference that we can apply it to arbitrary web applications that execute client-side code. Further, the invariants can also help identify if a web page has been permanently defaced or its contents have been significantly modified (for example, through a Type 2 XSS attack [16]).

**Better Domain Filtering**: Section V shows that failures of the majority of domains do not impact the invariant DOM for Slashdot. We believe this is also likely to be so for many web applications that include multiple domains. We could filter such domains at the client and prevent them from being loaded in the first place, for advertisement blocking or performance optimization. The NoScript plugin already allows domain filtering but leaves it to the user to decide which domains to block. With an approach such as DoDOM, we can automate the decision making process based on which domains impact the invariant DOM.

**Criteria for choosing Applications**: An interesting question to ask is "What kinds of web applications are likely to exhibit invariants across multiple user-interaction sequences?" We believe any application that has a fixed static structure (i.e., its backbone) in addition to dynamically generated content will fall into this category. Examples of such applications are news websites, e-commerce sites and online forums. However, certain applications such as office applications or productivity tasks may not satisfy this requirement because their content is highly dependent on the user and the specific task performed and hence may not exhibit generalizable invariants.

Standard frameworks such as Dojo and AJAX.Net are being increasingly used to construct web applications [17]. We hypothesize that applications written using these frameworks are more likely to exhibit invariants over their DOM structures by virtue of following programming patterns that are specific to the framework. We will explore this hypothesis in future work.

## VII. RELATED WORK

A number of approaches have been developed to test web applications that execute primarily at the server, i.e., Web 1.0 applications [18], [19]. An example of this approach is Veriweb [18], which systematically explores a web site by navigating to each of its pages. However, Veriweb cannot be

applied to Web 2.0 applications which often execute within a single page. Marchetto et al. [20] propose an approach to test Web 2.0 applications using an abstract state machine model provided by the developer. Mesbah and Deursen [7] extend this work to infer the state machine model automatically by finding clickable elements in the application and emulating clicks on them using an automated tool called ATUSA. Similar to our work, they use invariants on the DOM tree to check the validity of a state. Unlike DoDOM however, the invariants used in ATUSA correspond to generic invariants on the validity of the DOM, and are not specific to the web application being tested. Further, while ATUSA allows the programmer to specify other invariants, it leaves open the question of how to derive them.

There has been substantial work on regression testing of web applications [2], [21], [22]. These papers also capture a user's interaction with the application, replay their executions, and automatically characterize the invariant properties of its output. However, they differ from DoDOM in two ways. First, they consider only the server-side state of the application during replay, and hence do not apply to Web 2.0 applications. Secondly, the techniques assume that the web page does not change once it is loaded, which does not hold for Web 2.0 applications that continue to change even after they are loaded.

Concurrent to our work, Roest et al. [23] propose a method for regression testing of Web 2.0 applications by specifying oracle comparators. This work requires developers to manually specify the comparators based on generic templates. In contrast, our approach is fully automatic.

Finally, Swaddler [24] derives dynamic invariants on web applications written using the PHP language and uses the inferred invariants to detect security attacks that attempt to bypass the application's workflow and force the application into an inconsistent state. However, Swaddler's analysis and enforcement is implemented on the server side and hence cannot be used for Web 2.0 applications.

## VIII. Conclusion

This paper presents an automated approach to test the robustness of Web 2.0 applications and compare their outputs using DOM-based invariants. The approach automatically derives invariants on web pages' DOMs through dynamic execution and uses the invariants to detect errors. We present *DoDOM*, an automated tool to extract DOM invariants over multiple executions of the application. We show that DOM invariants (1) exist in real web applications, (2) can be learned using DoDOM within a small number of executions (six in our experiments), and (3) can be used to detect event errors and domain failures that affect the DOM with high accuracy.

As future work, we plan to (1) consider more realistic fault models, (2) extract invariants across multiple user-interaction sequences, and (3) implement DoDOM in the web browser.

## Acknowledgements

We thank Nikhil Swamy, Benjamin Livshits, Emre Kiciman and Trishul Chilimbi for their insightful comments about this work. We also thank Suzanne Zorn for help with editing this paper and the anonymous reviewers for helpful feedback.

## References

[1] T. OReilly, "What is Web 2.0: Design patterns and business models for the next generation of software," 2005.

[2] S. Sprenkle, E. Gibson, S. Sampath, and L. Pollock, "Automated replay and failure detection for web applications," in *Intl. Conference of Automated Software Engineering (ASE)*, 2005, pp. 253–262.

[3] A. Guha, S. Krishnamurthi, and T. Jim, "Using static analysis for AJAX intrusion detection," in *Intl. conference on World Wide Web (WWW)*, 2009, pp. 561–570.

[4] T. Chilimbi and V. Ganapathy, "HeapMD: Identifying heap-based bugs using anomaly detection," pp. 219–228, 2006.

[5] M. D. Ernst, A. Czeisler, W. G. Griswold, and D. Notkin, "Quickly detecting relevant program invariants," in *International Conference on Software Engineering (ICSE)*, 2000, pp. 449–458.

[6] S. Hangal and M. Lam, "Tracking down software bugs using automatic anomaly detection," in *International Conference on Software Engineering (ICSE)*, vol. 24, 2002, pp. 291–301.

[7] A. Mesbah and A. van Deursen, "Invariant-based automatic testing of AJAX user interfaces," in *International Conference on Software Engineering (ICSE)*, 2009, pp. 210–220.

[8] A. e. a. Le Hors, "Document Object Model (DOM) level 3 core specification," *W3C Recommendation*, 2004.

[9] S. Pertet and P. Narasimhan, "Causes of failure in web applications," *Carnegie Mellon University Tech Report, CMU-PDL-05-109*, 2005.

[10] Microsoft, "Fiddler: Web debugging proxy." [Online]. Available: http://www.fiddler2.com/Fiddler/help/WebTest.asp

[11] K. Pattabiraman and B. Zorn, "DoDOM: Leveraging DOM invariants for Web 2.0 applications' reliability," *Microsoft Research Technical Report (MSR-TR-2009-176)*, December 2009.

[12] Sun-Microsystems. Java Petstore 2.0. [Online]. Available: http://java.sun.com/developer/technicalArticles/J2EE/petstore/

[13] B. Livshits and E. Kiciman, "Doloto: code splitting for network-bound Web 2.0 applications," in *Intl. Symposium on Foundations of Software Engineering (FSE)*, 2008, pp. 350–360.

[14] J. Durães, M. Vieira, and H. Madeira, "Dependability benchmarking of web-servers," *Lecture notes in computer science*, pp. 297–310, 2004.

[15] C. Reis, S. Gribble, T. Kohno, and N. Weaver, "Detecting in-flight page changes with web tripwires," in *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2008, pp. 31–44.

[16] S. Di Paola and G. Fedon, "Subverting AJAX," in *23rd Chaos Communication Congress*, 2006.

[17] B. Livshits and U. Erlingsson, "Using web application construction frameworks to protect against code injection attacks," in *Workshop on Programming Languages and Analysis for Security (PLAS)*, 2007, pp. 95–104.

[18] M. Benedikt, J. Freire, and P. Godefroid, "VeriWeb: Automatically testing dynamic web sites," in *Proceedings of 11th International World Wide Web Conference (WWW)*, 2002.

[19] F. Ricca and P. Tonella, "Analysis and testing of web applications," in *Intl. Conference on Software Engineering (ICSE)*, 2001, pp. 25–36.

[20] A. Marchetto, P. Tonella, and F. Ricca, "State-based testing of AJAX web applications," in *Intl. Conference on Software Testing Verification and Validation (ICST)*, 2008, pp. 121–130.

[21] K. Dobolyi and W. Weimer, "Harnessing web-based application similarities to aid in regression testing," in *International Symposium on Software Reliability Engineering (ISSRE)*, 2009, pp. 71–80.

[22] S. Sprenkle, L. Pollock, H. Esquivel, B. Hazelwood, and S. Ecott, "Automated oracle comparators for testing web applications," *In the International Symposium on Software Reliability Engineering (ISSRE)*, pp. 117–126, 2007.

[23] D. Roest, A. Mesbah, and A. van Duersen, "Regression Testing AJAX Applications: Coping with Dynamism," in *Intl. Conference on Software Testing, Verification and Validation (ICST)*, 2010, pp. 127–136.

[24] M. Cova, D. Balzarotti, V. Felmetsger, and G. Vigna, "Swaddler: An approach for the anomaly-based detection of state violations in web applications," *Lecture Notes in Computer Science (LNCS)*, vol. 4637, p. 63, 2007.