UQAC
Université du Québec
à Chicoutimi

THÈSE

PRÉSENTÉE À

L'UNIVERSITÉ DU QUÉBEC À CHICOUTIMI

COMME EXIGENCE PARTIELLE

DU DOCTORAT EN INFORMATIQUE

PAR

OUSSAMA BEROUAL

DETECTION ET CORRECTION AUTOMATIQUE DES BUGS D'INTERFACE DANS LES APPLICATIONS WEB

AOÛT 2018

RÉSUMÉ

Les problèmes de mise en page dans les interfaces des applications web, appelés « bugs d'interface », croissent de jour en jour avec la popularité grandissante de ces applications et donnent lieu à des ennuis d'affichage embarrassants et des difficultés allant jusqu'à l'entrave de fonctionnalité de l'interface utilisateur. Le nombre considérable d'utilisateurs des applications web incapables de pallier à ces contraintes rend l'enjeu très grand. Malheureusement, les solutions efficaces apportées à ce sujet sont très rares.

Nous proposons dans ce travail de thèse, une nouvelle approche permettant la spécification du contenu attendu d'une interface, la vérification automatique du respect de la spécification et, plus particulièrement, l'octroi d'un verdict détaillé et utile lors d'une violation.

L'approche proposée est une technique générique de localisation de défauts, basée sur le concept de « réparation » ou « correction » et applicable avec différents langages de spécification, y compris la logique propositionnelle. Nous portons une attention particulière à son utilisation dans la détection des défauts de mise en page dans les applications Web.

TABLE DES MATIÈRES

TABLE DES FIGURES

LISTE DES TABLEAUX

INTRODUCTION

Le mot *bug*, qui signifie en anglais « insecte » et qui a été francisé pour devenir « bogue », a vu le jour dans les années quarante à la suite de la panne qu'a connu le dernier cri des ordinateurs de l'époque, le Mark II. Comme son nom l'indique, la cause était un insecte qui s'était introduit dans un relais électromécanique de celui-ci. La panne fut découverte par la brillante mathématicienne et pionnière de la programmation, Grace Hopper. Il s'agit du tout premier vrai bug *informatique*, car on utilisait déjà le mot pour désigner des problèmes dans les appareils électriques [17]. Depuis, le mot bug est devenu synonyme de tout dysfonctionnement ou anomalie d'un programme.

Le monde d'aujourd'hui est piloté par des ordinateurs dans tous les domaines : énergétique, judiciaire, sanitaire, militaire, etc. De ce fait, la manifestation d'un bug peut entraîner des désordres, des perturbations, voire des catastrophes. Les trois dernières décennies ont connu une multitude de bugs de grande envergure.

Parmi les plus importants, mentionnons une interruption en 2003 de plusieurs jours dans l'alimentation en électricité d'une cinquantaine de millions d'américains. Lors de cet incident, une douzaine de personnes ont même trouvé la mort, empoisonnées au monoxyde de carbone en tentant de remédier à ce problème par des générateurs au diesel. On a compté plus de

six milliards de dollars de dégâts matériels. A l'origine de cette panne, une paralysie totale provoquée par deux logiciels chargés du contrôle de la production, qui essayaient de modifier le même fichier simultanément [25, 3, 16, 29].

Mentionnons également des centaines d'accidents de la route meurtriers survenus entre 2009 et 2011, et dont les victimes étaient les propriétaires de la Lexus ES350 du constructeur Toyota. Les conducteurs perdaient la maîtrise du véhicule, une fois qu'il atteignait la vitesse de 150 km/h, puisque ce seuil entraînait la désactivation de la pédale de frein. Durant deux années, les chauffeurs sont accusés par Toyota de confondre les pédales de frein et d'accélération, alors qu'une expertise finit par révéler une défaillance dans l'ordinateur de bord. Les pertes sont estimées dans ce cas à 2,4 milliards de dollars [32, 28].

Parmi les bugs informatiques les plus meurtriers, on compte également un dysfonctionnement en 1987 de la machine de radiothérapie Therac 25 chargée d'administrer aux patients la quantité de radioactivité qui leur est prescrite. À certaines occasions, la machine leur donne vingt fois la dose mortelle, occasionnant de ce fait la blessure d'un patient et la mort de cinq autres. Le logiciel destiné à veiller au bon positionnement et à la présence de la plaque métallique censée recevoir le rayon pour le filtrer et le concentrer a malheureusement failli à son rôle [14].

Tous ces exemples sont des bugs aux conséquences catastrophiques. Heureusement, tous les bugs ne sont pas aussi dévastateurs, mais ils peuvent néanmoins s'avérer nuisibles et répandus. C'est le cas d'un type de bug auquel les informaticiens font face à chaque instant, à savoir les bugs de mise en page dans les interfaces des applications web, qui se trouvent à l'origine des problèmes d'affichage rencontrés quotidiennement dans les interfaces web. À cet égard, des chiffres datant de 2015, liés à Internet donnent matière à réflexion. Le réseau compte plus de trois milliards d'internautes, dont deux milliards sont inscrits sur les réseaux sociaux. Plus de

huit cent mille nouveaux sites Internet sont mis en ligne chaque jour [15], [4]. Par conséquent, le nombre d'utilisateurs des applications web est considérable, et le nombre de personnes touchées par les bugs d'interfaces résultant de ces applications est énorme. L'enjeu de ce fait est très grand sur tous les plans.

La bonne conduite d'une application web exige une bonne apparence visuelle des pages web sans aucune défaillance d'affichage, facilitant ainsi l'utilisation de l'application et offrant un meilleur service à l'utilisateur. Trois types de problèmes d'affichage sont relevés : un premier type qualifié de non gênant tel qu'un débordement d'un paragraphe de sa bordure ou un problème d'alignement ou même une sorte de caractère spécial mal affiché (mojibake). Un deuxième type qualifié de gênant telle qu'une image déplacée qui couvre un paragraphe ou une autre partie de la page :

Par contre le troisième type est plus grave et peut conduire à un blocage de l'application dont les effets risquent d'être sérieux compromettant la fonctionnalité de l'interface utilisateur. Un

exemple de bug qui affecte la fonctionnalité de l'application : Le mauvais fonctionnement des boutons : de nombreuses applications montrent des éléments superposés qui se comportent comme des « pop-ups » dans la fenêtre. Cependant, dans un certain nombre de cas, les boutons de ces fenêtres sont inopérants : cliquer dessus plusieurs fois ne produit aucun effet. Ce problème a été observé dans des sites aussi variés qu'American Airlines, Dropbox et BitBucket. Dans certains cas, l'utilisateur est effectivement bloqué dans la fenêtre contextuelle ou la page qui contient le bouton, et ne peut pas continuer sans forcer un rechargement complet du document.

Cependant les travaux visant à résoudre les problèmes d'interface se font très rares. Hallé *et al.* [51] sont parmi les premiers à s'intéresser à ce genre de problème. Mahajan et Halfond [58] ont abordé la problématique en proposant une approche basée sur la vision par ordinateur et le traitement d'images. Walsh *et al.* [74, 72] ont également traité ce genre de bugs dans les applications dites responsives (*responsive web design*).

Deux catégories d'utilisateurs sont confrontées à ces types de problèmes : l'une représente les spécialistes du domaine (les informaticiens) chez lesquels ces types de problèmes peuvent trouver leurs solutions après un travail laborieux. L'autre représente le grand public pour lequel le deuxième et le troisième type de problème constituent des contraintes pour l'exploitation de la page car la solution dans ce cas exige certaines connaissances (l'outil, le langage, etc), qui échappent généralement au grand public. Ce dernier forme la majorité des utilisateurs. De ce fait, il est nécessaire de lui trouver une solution lui permettant d'utiliser les sites web dans les meilleures conditions possibles. Il doit avoir tout simplement sur son écran une interface correcte sans aucun bug pour qu'il ne soit pas obliger à recourir à une technique de correction qui le dépasse. D'autant plus que les applications web connaissent une évolution rapide et commencent à conquérir plusieurs domaines (commerce électronique, éducation, loisir, etc.) et même à être utilisées dans des opérations sensibles telles que les transactions monétaires

via internet. Ce qui fait d'eux des programmes très complexes, dynamiques et interactifs. En plus de la rareté des méthodes de détection de ces bugs, pire encore : presque rien n'a été fait pour donner un feedback à l'utilisateur. Lorsqu'un bug est trouvé, les solutions actuelles ne font que retourner « vrai/faux ». Dans ce travail, nous proposons une nouvelle approche permettant, dans une première étape de détecter automatiquement les différents types de bugs d'interfaces et de les corriger automatiquement dans une deuxième étape. Il s'agit d'une approche générique de localisation de défauts, basée sur le concept de réparation.

Les objectifs et contributions de cette thèse sont :

1. Proposer un langage pour spécifier le contenu attendu d'une interface web.
2. Décrire un algorithme permettant de vérifier automatiquement qu'une spécification est respectée.
3. Décrire une méthode permettant de fournir un verdict détaillé et utile lors d'une violation.

Cette thèse comporte six chapitres. Le chapitre 1 est dédié à une vue globale sur les notions de base relatives au web. Les différents types de bugs ainsi que des exemples réels de chacun de ces types sont présentés au chapitre 2. Le chapitre 3 est consacré aux travaux connexes sur la détection des bugs d'interfaces dans les applications web, à quelques exemples d'approches et d'outils de détection pour lesquels certains points faibles sont identifiés.

Dans le chapitre 4, on retrouve l'ensemble des informations spécifiques à l'outil Cornipickle conçu et utilisé dans la détection des bugs. Ceci inclut la syntaxe du langage et son utilisation pour exprimer des propriétés, ainsi que les détails de son implémentation (objectifs de conception, architecture et scénario typique). Le chapitre 5 est réservé à l'utilisation de Cornipickle, en combinaison avec un robot d'exploration dynamique (crawler) pour détecter efficacement les bugs comportementaux dans les RIA (Rich Internet Applications). Le dernier

chapitre, quant à lui, présente deux mécanismes par lesquels un verdict vrai/faux peut être enrichi d'information supplémentaire pour le développeur. Une conclusion générale mettant en évidence l'intérêt du travail réalisé et l'importance des résultats obtenus clôture la thèse.

Il est à signaler que les contributions présentées dans cette thèse ont fait l'objet de publications dont je suis coauteur :

Un premier article dont la contribution consiste en la formalisation de la sémantique de l'interpréteur Cornipickle (section 3.3 de l'article) et la présentation du concept des témoins (section 4.3). Cette contribution correspond au chapitre 4 de la thèse :

1. Sylvain Hallé, Nicolas Bergeron, Francis Guérin, Gabriel Le Breton, Oussama Beroual : Declarative layout constraints for testing web applications. *J. Log. Algebr. Meth. Program.*, Elsevier. 85 (5) : 737-758 (2016). [51]

Un deuxième article présentant le mécanisme de transformations (feedback enrichi pour l'utilisateur), ce qui correspond au chapitre 6 :

1. Sylvain Hallé, Oussama Beroual : Fault Localization in Web Applications via Model Finding. CREST@ETAPS 2016 : 55-67. Elsevier, Electronic Notes in Computer Science. (2016) [52]

Un troisième article sur l'automatisation des tests avec intégration à un crawler, ce qui correspond au chapitre 5 :

1. Oussama Beroual, Francis Guérin, Sylvain Hallé : Searching for Behavioural Bugs with Stateful Test Oracles in Web Crawlers. SBST@ICSE 2017. ACM. 7-13.(2017) [38]

CHAPITRE 1

NOTIONS GÉNÉRALES SUR LE WEB

Deux termes sont d'usage confondus par le public non averti, le terme « web » et le terme « Internet ». Ce chapitre est destiné à lever cette confusion, puis à mettre l'accent sur la majorité des aspects d'une application web, pièce maîtresse de notre recherche.

1.1 LE FONCTIONNEMENT DU WEB

Une différence de taille entre « Internet » et « web » est à noter. *Internet* est un réseau composé d'une multitude de réseaux connectés entre eux. Chacun de ces réseaux est composé d'un ensemble d'équipements (fibres optiques, etc.) constituant un support physique d'information. Le *web*, quant à lui, est un système de fichiers véhiculés par des serveurs. Il représente le contenu principal d'Internet à côté d'autres contenus tels que le courrier électronique, la messagerie, etc. Il n'est donc qu'une des applications d'Internet.

Un site web est un contenu sur Internet. Ce contenu n'est rien d'autre qu'un ensemble de fichiers (HTML, CSS, JavaScript, etc.) hébergés sur un serveur. Selon leur mécanisme de fonctionnement, deux types de site sont distingués : les sites *statiques*, dont le contenu est invariable, et les sites *dynamiques*, dont le contenu est variable. Les premiers ne sont modifiables

que par leurs propriétaires, alors que les seconds sont modifiables par des utilisateurs autres que leurs propriétaires. La réalisation de chacun de ces sites fait appel à des technologies bien déterminées telles que HTML, CSS et JavaScript.

Les acteurs principaux dans le fonctionnement du web sont : les *clients*, représentés par des navigateurs tels que Firefox, Internet Explorer, Chrome et Safari, et les *serveurs*, représentés par les machines abritant les sites web où les fichiers sont stockés. Chaque serveur est identifié sur un réseau par son adresse IP (Internet Protocol) et chaque page web est associée à une adresse URL (Uniform Resource Locator ou « localisateur uniforme de ressource ») caractérisée par un contenu. Une demande d'une page web à un serveur correspond donc à une demande d'un contenu. La communication entre les clients et les serveurs est assurée par un protocole appelé « HTTP » (HyperText Transfer Protocol ou « protocole de transfert hypertexte »), via lequel les requêtes sont formulées par les navigateurs aux serveurs [37].

HTTP est donc la langue de conversation entre le navigateur et le serveur. La conversation se fait selon le principe de « requête-réponse ». La formulation d'une requête HTTP par le navigateur est suivie par une réponse HTTP du serveur après l'avoir décodée et étudiée. En plus de la ligne de requête, définissant le document demandé, la méthode appliquée, et le protocole utilisé, une requête est composée de deux ensembles de lignes : des lignes facultatives et des lignes optionnelles. Les premières sont les champs d'en-tête de la requête, et sont chargées de fournir des informations supplémentaires sur la requête ou le client (type de navigateur, système d'exploitation, etc). Les secondes forment le corps de la requête et sont chargées, lors de l'envoi de données au serveur, de permettre un envoi de données par une méthode spécifique (l'envoi de données au serveur par un formulaire par une méthode POST par exemple).

Une requête de type GET nomme l'URL à récupérer : `http://uqac.ca` par exemple dans la figure 1.1. Le navigateur s'identifie dans l'en-tête `User-Agent` et indique les types de réponses

```
GET / HTTP/1.1
Host: www.uqac.ca
Connection: keep-alive
User-Agent: Mozilla/5.0 (Windows NT 6.1; Win64; x64)
  AppleWebKit/537.36 (KHTML, like Gecko) Chrome/61.0.3163.100
  Safari/537.36
Upgrade-Insecure-Requests: 1
Accept: text/html,application/xml;q=0.9
Accept-Encoding: gzip, deflate
Accept-Language: fr-FR,fr;q=0.8,en-US;q=0.6,en;q=0.4
Cookie: PHPSESSID=tphmk53fee883355e4eq24dmb5

Upgrade-Insecure-Requests: 1
Connection: keep-alive
Host: www.uqac.ca
```

Figure 1.1 – Exemple d'une requête HTTP

qu'il accepte dans l'en-tête `Accept` et `Accept-Encoding`. L'en-tête `Connection` demande au serveur de garder la connexion TCP ouverte pour d'autres requêtes. La requête contient également les *cookies* que le navigateur conserve pour ce domaine. Les cookies sont des paires clé-valeur qui suivent l'état d'un site web entre différentes demandes de pages. Ainsi, les cookies stockent le nom de l'utilisateur connecté, un numéro secret attribué à l'utilisateur par le serveur, certains paramètres de l'utilisateur, etc. Les cookies sont stockés dans un fichier texte sur le client, et envoyés au serveur à chaque demande.

La réponse du serveur sur la requête de la figure 1.1 générée et renvoyée est montrée dans figure 1.2.

L'en-tête qui définit `Content-Type` en `text/html` indique au navigateur de rendre le contenu de la réponse [13] au format HTML au lieu de le télécharger en tant que fichier. Le navigateur utilise l'en-tête pour décider comment interpréter la réponse, mais tient également compte d'autres facteurs, tels que l'extension de l'URL.

```
HTTP/1.0 200 OK
Date: Sat, 11 Nov 2017 19:03:23 GMT
Server: Apache/2.2.22 (Unix) mod_ssl/2.2.22 OpenSSL/1.0.1e-fips
X-Powered-By: PHP/5.2.17
Access-Control-Allow-Origin: https://wprodl.uqac.ca
Expires: Thu, 19 Nov 1981 08:52:00 GMT
Cache-Control: no-store, no-cache, must-revalidate
Pragma: no-cache
Content-Type: text/html
X-Cache: MISS from w3rep.uqac.ca
X-Cache-Lookup: MISS from w3rep.uqac.ca:80
Via: 1.0 w3rep.uqac.ca (squid/3.1.23)
Connection: close
```

Figure 1.2 – Exemple d'une réponse HTTP

1.2 LE LANGAGE HTML

Nous allons nous limiter dans ce qui suit à la présentation de l'essentiel des éléments de base relatifs à ces trois langages, à savoir les balises pour le HTML, les sélecteurs pour le CSS et quelques notions de base sur JavaScript.

Le langage HTML « HyperText Markup Language » en anglais, ou « langage de balisage hypertexte » en français ou encore « langage de marquage hypertexte » dont la création revient à 1991, est un langage de description de document, pouvant être écrit avec un simple éditeur de texte sans une application spécifique et servant à produire (sur Internet), des pages Web d'une grande variété de contenus, de mise en forme ou d'animations, et à insérer différents types d'éléments(texte, des liens, des images, etc). Il permet aussi de désigner au navigateur certaines spécificités telle que la considération d'un texte comme un paragraphe ou un titre.

1.2.1 STRUCTURE D'UN DOCUMENT

La structure d'un document HTML doit satisfaire un schéma précis et comprendre un ensemble de balises spéciales essentielles pour tout document HTML. Les éléments composants la structure d'un document HTML sont les suivants.

Doctype Un document HTML débute toujours par le soulignement de la nature du document c'est à dire le langage utilisé (HTML) en faisant appel à la balise < *!DOCTYPE html*>.

L'élément *<html>* L'élément *html* comprend deux balises <html> et </html>. Il définit l'unique racine du document HTML. Tous les autres éléments doivent y être placés.

L'élément *<head>* Cet élément représente l'en-tête du document et renferme exclusivement des méta-données (titre de la page, type d'encodage utilisé, etc), exploitées par les navigateurs pour améliorer l'ergonomie de la page.

L'élément *<body>* L'élément body représente le corps du document. Il est toujours placé après l'en-tête et contient tout le contenu « visible » de la page : les textes, images, liens, vidéos, etc.

L'élément *<title>* La structure s'achève par une des méta-données utilisées par le navigateur qui constitue le seul élément HTML obligatoire : le titre du document placé à l'intérieur de l'élément head.

Voici le document HTML formel le plus simple qu'on puisse écrire :

```
<!DOCTYPE html>
<html>
  <head>
   <title>Titre de la page</title>
  </head>
  <body>

     <h1>Un grand titre</h1>
     <p>Un paragraph.</p>

  </body>
</html>
```

Figure 1.3 – Un exemple simple de page HTML contenant un grand titre et un paragraphe.

```
<!DOCTYPE html>
<html>
<head>
<title>Un document HTML</title>
</head>
<body>
<!-- Du contenu pour l'utilisateur ici -->
</body>
</html>
```

Le code HTML dans la figure 1.3 par exemple indique que l'on souhaite créer un grand titre (grâce à l'élément h1) et un paragraphe (grâce à l'élément p).

1.2.2 VERSIONS DU HTML

Le Web a connu ces deux dernières décennies une évolution extraordinaire. En effet, l'avancée technologique a donné lieu à une amélioration des performances des composants physiques et une augmentation de la vitesse de connexion, entraînant par conséquent une évolution

remarquable des sites Web. A leur tour, les langages tels que le HTML et le CSS ont également connu une évolution considérable (modifications, enrichissements, etc) d'où l'apparition de nouvelles versions de ces langages bien que l'évolution n'a pas été linéaire ni continue pour l'utilisateur final. Les nouvelles versions sont dotées chacune de nouvelles fonctionnalités et change parfois totalement la syntaxe du langage. La première version de HTML, l'HTML1 a vu le jour en 1991. De nombreuses améliorations apportées par la suite par le créateur du HTML, jugées intéressantes et importantes l'ont conduit en 1994 à partager publiquement la nouvelle "version" de son langage, le HTML2. D'autres versions se sont succédées : HTML2 (1995), HTML3.2, (janvier 1997), HTML4 modifiée à plusieurs reprises (1997, 1998, 1999, 2000), HTML5 (2014), HTML5.1 (2016). Parmi toutes ces versions, la plus stable, celle qui offre de nouvelles fonctionnalités et ouvre de nouvelles possibilités intéressantes est la plus récente à savoir HTML5.1.

1.2.3 ÉLÉMENTS DE BASE EN HTML

Le fonctionnement du HTML s'appuie sur la notion d'éléments. Ces derniers ont pour rôle de structurer du contenu, pour donner du sens aux différents objets de ce contenu. Ils sont constitués de balises renfermant des attributs et du contenu entre elles.

Les balises en HTML Le nombre de balises constituant le HTML est environ 140. Elles servent à fournir au navigateur des indications sur le sens d'un élément, son interprétation ou son affichage, en plus d'autres indications telles que : la gestion des formulaires en ligne, l'ajout des fichiers multimédias, etc. On distingue plusieurs types de balises, chargée chacune d'une fonction bien déterminée, bien que certaines n'ont de fonction que de séparer des sections d'un document. Parmi ces fonctions : définir des titres, créer des paragraphes, créer des liens vers des ressources externes, intégrer une image dans un document HTML, créer des listes. Les balises

suivantes sont données à titre d'exemple : `<p>`, `<h1>`, `<img>`, `<li>`, `<a>`. Les éléments sont constitués généralement d'une paire de balises (ouvrante et fermante) et d'un contenu entre les balises et exceptionnellement d'une balise unique dite dans ce cas « orpheline ». Une balise fermante doit avoir le même nom que la balise ouvrante correspondante (notez la présence du slash (barre oblique)).

Les balises « auto fermantes » (ou balises vides) Certaines balises sont dépourvues de texte du fait qu'elles sont ouvrantes et fermantes en même temps. les balises AUTO FERMANTES sont soit : des balises de type BLOC (exemple : `<hr />`) ; des balises EN LIGNE (exemple : `<img />`).

Les attributs en HTML Les attributs se placent dans la balise ouvrante d'un élément et possèdent toujours une valeur (parfois implicite) ; Ils viennent apporter plus de détails sur les éléments. Par exemple, l'attribut `href` (hypertexte référence) va offrir l'adresse (la valeur) de la page du lien à l'élément <a> (pour anchor) chargé de la création des liens vers d'autres sites ou d'autres pages.

L'élément `<img>` constitué d'une seule balise orpheline, chargé d'insérer une image dans une page HTML, a besoin de deux attributs : `src` et `alt`. Le premier src assure le nom et l'emplacement de l'image (la valeur) alors que l'attribut alt se charge de l'affichage d'un texte alternatif à une indisponibilité de l'image.

Notez bien que les balises et les attributs ne seront jamais affichés par le navigateur : il va s'en servir d'indication pour justement savoir ce qu'il doit afficher (un paragraphe, un titre, un lien, une image, etc.).

1.3 LES CASCADING STYLESHEETS (CSS)

Le CSS « Cascading StyleSheets » en anglais ou « feuilles de styles en cascade » en français est un langage informatique apparu en 1996, voué à la mise en forme du contenu des documents HTML et XML moyennant des styles pour définir la taille, la couleur ou l'alignement du texte afin d'agrémenter le résultat visuel final de ce contenu. Il est utilisé dans la conception de sites web. Le code ci-dessous par exemple indique que les titres `h1` écrits en HTML doivent avoir une couleur verte et une taille de 20px tandis que les paragraphes doivent être noir et soulignés.

```
h1 {
    color: green;
    font-size: 20px;
}

p {
    color: black;
    text-decoration: underline;
}
```

Le CSS est donc un autre langage du web venant compléter le HTML. Il permet la mise en page d'un contenu et le changement de son apparence en lui appliquant des styles (choisir la couleur du texte, sélectionner la police utilisée, définir la taille du texte, les bordures, le fond...). L'apparition du HTML a devancé l'apparition du CSS de cinq années, période dans laquelle la mise en page était effectuée par le HTML qui consacrait des balises à cette fin, telle que la balise : `<font color="#aab1c3">` définissant la couleur du texte par exemple. Le langage CSS est venu répondre à la complexité qu'ont connu les pages HTML avec une augmentation remarquable des balises entre autres, conduisant à une mise à jour des pages web de plus en plus complexe. À l'instar du HTML, le CSS est passé par plusieurs versions ;

les plus importantes sont : CSS1, CSS2.0, CSS2.1 et CSS3.

Écriture du code CSS Le code CSS peut être écrit à trois endroits distincts :

1. Dans un L'élément HTML `<style>`.
2. Dans la balise ouvrante des éléments HTML(méthode la moins recommandée).
3. Dans un fichier CSS séparé (méthode la plus recommandée).

1.3.1 PROPRIÉTÉS ET SÉLECTEURS CSS

Les propriétés CSS permettent de choisir quel(s) aspect(s) (ou « styles ») d'un élément HTML que l'on souhaite modifier.

L'application d'un style à un ou plusieurs éléments HTML, signifie leurs sélections au préalable pour les soumettre à un style particulier. L'intervention d'un sélecteur est donc nécessaire. Le CSS est fondé sur un ensemble de règles, les sélecteurs sont la première partie d'une règle CSS. C'est une syntaxe chargée d'identifier les éléments du document auxquels la règle est dédiée (appliquée). Depuis son arrivée (1996), le CSS a spécifié un certain nombre de sélecteurs, très acceptés de nos jours par les différents navigateurs. les plus fréquemment utilisés sont :

Les sélecteurs de type Ce sélecteur cible l'élément du document à styler en se basant sur le nom de l'élément. Il doit correspondre dans ce cas au nom de l'élément. Exemple : pour mettre en bleu le texte des titres de niveau 1, le sélecteur sera `h1` :

```
h1 {
    color: blue;
}
```

Les sélecteurs de classe Ce sélecteur permet de cibler les éléments en fonction de la valeur de leur attribut class. Il permet donc de sélectionner tous les éléments ayant une certaine classe. Précéder le nom de la classe par un point (`.`), suffit pour obtenir le sélecteur correspondant à cette classe. Exemple : la classe `<important>` est attribuée à tous les éléments jugés importants. Il suffit de définir dans le fichier CSS une règle stipulant que le texte de tous les éléments possédant la classe `Important` soit écrit en rouge :

```
<p class="Important">Coucou</p>

...

.Important
{
    color: red;
}
```

Les sélecteurs d'identifiant Ce sélecteur permet de cibler un élément d'un document en fonction de la valeur de son attribut `<id>`. Dans un document, il ne doit y avoir qu'un seul identifiant donné à l'élément. Précédé le nom de l'identifiant par un dièse `(#)`, suffit pour obtenir le sélecteur correspondant à cet identifiant. Exemple : Un identifiant *Menu* est utilisé pour repérer notre menu dans le document. Il suffit de définir dans le fichier CSS une règle indiquant que notre menu ne soit pas affiché :

```
<div id="Menu">
...
</div>

...

#Menu
{
    display: none;
}
```

Le sélecteur universel « * » Ce sélecteur permet de cibler tous les éléments d'un document, d'où l'appellation de sélecteur universel. Il existe également une variante pour ne cibler qu'un seul élément.

Regroupements des sélecteurs Une autre manière de procéder consiste en la réduction de taille du fichier CSS en appliquant une même règle à plusieurs sélecteurs. Ces derniers sont dans ce cas séparés par une virgule (`,`). Exemple : les éléments possédant la classe `Important` et les titres `<h2>`, sont écrits en rouge :

```
.Important, h2
{
    color: red;
}
```

1.4 JAVASCRIPT

JavaScript est un langage de programmation de scripts utilisé surtout dans les pages web interactives ainsi que pour les serveurs. Il a été crée par Brendan Eich en 1995 en l'espace de dix jours, suite à une demande formulée par la Netscape Communications Corporation. Les bases du langage et ses principales interfaces sont fournies par des objets ; ce qui fait de lui un langage orienté objet à *prototype*. Les objets en question ne sont pas des instances de classes. Ils sont équipés chacun de constructeurs permettant de créer leurs propriétés, et notamment le prototypage qui sert à créer des objets héritiers personnalisés. JavaScript dont les fonctions sont des objets de première classe, supporte le paradigme objet impératif et fonctionnel.

Depuis sa création, JavaScript a connu d'innombrables modifications. Il a été standardisé en 1997 par « Ecma International » donnant naissance à la première édition du standard

« ECMA-262 ». la deuxième édition du standard « ECMA-262 » a vu le jour en 1998 suite à des changements rédactionnels apportés au standard « ECMA-262 » pour le conformer au standard international « ISO/CEI 16262 ». Des améliorations(dans la manipulation des chaînes de caractères, dans les instructions de contrôle, etc), introduites dans la deuxième édition ont donné lieu en 1999 à la publication de la troisième édition du standard ECMA-262. Depuis la troisième édition, les éditions se sont succédées pour arriver actuellement à la huitième édition. Les différentes éditions apparues ont chacune participé avec un plus dans le développement des performances du langage.

La plupart des langages de programmation ont des fonctionnalités de base communes. L'utilisation de JavaScript nécessite la connaissance de ces bases pour mieux comprendre son fonctionnement.

Variables Les variables sont des conteneurs servant à stocker temporairement des informations. Une variable a le pouvoir de varier, autrement dit de pouvoir stocker différentes valeurs dans le temps en « écrasant » sa valeur précédente. Une variable est déclarée au commencement avec le mot-clé `let` suivi du nom qu'on souhaite utiliser pour la variable.

Un certain nombre de règles sont à considérer en JavaScript :

1. les lignes de code doivent terminer par un point-virgule pour indiquer que c'est la fin de la ligne. L'omission de ces points-virgules peut conduire à des comportements inattendus, voire des erreurs ;
2. N'importe quel nom peut être (quasiment) utilisé pour nommer une variable. Il y a cependant quelques restrictions sur ces noms ;
3. Éviter la casse, JavaScript y est sensible, cela veut dire que maVariable sera considéré comme un nom différent de mavariable. L'apparition des problèmes sur les noms de

variables dans le code, implique la vérification de la casse utilisée ;

4. Avec les versions récentes de JavaScript, il est conseillé d'utiliser le mot-clé let. Cependant des variables déclarées avec le mot-clé var sont parfois rencontrées. Ce dernier est utilisé lorsque notre code doit supporter des navigateurs anciens (IE < 11), let n'étant pas supporté dans ce cas. Une fois une variable est déclarée, on lui donne une valeur : `maVariable = 'ouss';`.

Pour utiliser la valeur plus loin dans le code, il suffit de faire appel à la variable en utilisant son nom : `maVariable;`. Lorsque on crée une variable et qu'on lui donne une valeur ; cela peut se faire sur une seule ligne : `let maVariable = 'ouss';`. Une fois qu'on a donné une valeur à une variable, on peut la changer plus loin :

```
let maVariable = 'ouss';
maVariable = 'Sylvain';
```

Les variables sont réparties en différents types de données et ont chacune une fonction. Parmi ces variables on a : la chaîne de caractères, le nombre, le Booléen, le tableau, l'objet..etc. Les variables sont indispensables à la programmation. Si les valeurs sont statiques, rien ne pourrait se faire. Par exemple, on ne pourrait pas personnaliser un message de bienvenue ou changer l'image affichée dans une galerie.

Les opérateurs Un opérateur est un symbole mathématique qui produit un résultat en fonction de plusieurs valeurs (la plupart du temps on utilise deux valeurs et un opérateur). Parmi les opérateurs les plus simples, on a : l'opérateur d'affectation, l'opérateur d'identité, l'opérateur de négation, l'opérateur d'inégalité...

Il y a plein d'autres opérateurs mais on s'en tiendra à ceux-là.

Il est à noter que l'utilisation de différents types de données avec un même opérateur faussera

le résultat obtenu. le résultat obtenu par"70" + "12" n'est pas le même que celui obtenu par 70 + 12, le deuxième résultat est le bon car les nombres entourés de guillemets sont donc considérés comme des chaînes de caractères.

Les structures conditionnelles Les structures conditionnelles sont des éléments du code qui permettent de tester si une expression est vraie ou non et d'exécuter des instructions différentes selon le résultat. La structure conditionnelle utilisée la plus fréquemment est `if ... else`. Par exemple :

```
let parfumGlace = 'sorbet de fraise';
if (parfumGlace === 'sorbet de fraise') {
  alert("J'adore le sorbet de fraise !");
} else {
  alert("J'aurai préféré un sorbet de fraise.");
}
```

Le test à réalisé est contenue dans `if ( ... )`. Il consiste en une comparaison de la variable parfumGlace avec la chaîne de caractères « sorbet de fraise » via l'opérateur d'identité pour vérifier leur égalité. Si cette comparaison renvoie true, le premier bloc de code sera exécuté. Sinon, c'est le code du second bloc, celui présent après `else`, qui sera exécuté.

Les fonctions Les fonctions sont chargées d'organiser les fonctionnalités à réutiliser. Par exemple, au lieu d'exécuter deux fois la même action, plutôt que de recopier le code, la fonction est écrite une fois puis utiliser aux deux endroits souhaités.

```
let maVariable = document.querySelector('h1');
alert('Salut !');
```

Ces fonctions (querySelector et alert) sont disponibles dans le navigateur. Elles ressemblent à un nom de variable suivi de parenthèses et utilisent des arguments dans leurs calculs. Les

arguments sont placés entre les parenthèses, séparés par des virgules s'il y en a plusieurs. Par exemple, la fonction alert() fait apparaître une fenêtre de pop-up dans la fenêtre du navigateur. Un argument est utilisé pour indiquer à la fonction le contenu qu'on désire écrire dans cette fenêtre. En plus des fonctions déjà existantes, d'autres fonctions peuvent être définies par nous-mêmes. Par exemple, fonction toute simple qui considère deux arguments et renvoie le résultat de la multiplication :

```
function multiplier(num1,num2) {
  let résultat = num1 * num2;
  return résultat;
}
```

Avant une utilisation répétée de cette fonction, elle doit être déclarée dans la console :

```
multiplier(4,7);
multiplier(20,20);
multiplier(0.5,3);
```

L'instruction return indique au navigateur qu'il faut renvoyer la variable résultat en dehors de la fonction afin qu'elle puisse être réutilisée par ailleurs. Cette instruction est nécessaire car les variables définies à l'intérieur des fonctions sont uniquement disponibles à l'intérieur de ces fonctions. C'est ce qu'on appelle une portée.

Les événements Un site web vraiment interactif est caractérisé par des événements. Les événements sont des structures de code à l'écoute du navigateur permettant de déclencher des actions au moindre problème. L'exemple concret est l'événement de clic qui est déclenché une fois l'utilisateur clique sur quelque chose dans le navigateur. L'exemple ci-dessous peut donner une idée sur ce que ça donne en pratique, il suffit de saisir ces quelques lignes dans la console puis cliquez sur la page :

```
document.querySelector('html').onclick = function() {
    alert('arrêtez de cliquer');
}
```

Les méthodes pour « attacher » un événement à un élément sont multiples. Dans cet exemple, deux paramètres sont définis : l'élément HTML concerné et un gestionnaire `onclick` qui est une propriété égale à une fonction anonyme (elle n'a pas de nom) qui contient le code à exécuter quand l'utilisateur clique.

On pourra noter que `document.querySelector('html').onclick = function() {};` est équivalent à :

```
let maHTML = document.querySelector('html');
maHTML.onclick = function() {};
```

La première syntaxe est simplement plus courte. D'autres fonctionnalités peuvent s'ajouter aux bases en JavaScript exposées.

1.5 LE FONCTIONNEMENT INTERNE DES NAVIGATEURS WEB

Pour la période allant de mars 2015 à mars 2016, la part de marché des navigateurs, d'après les statistiques de StatCounter [19], est de près de 96% entre Internet Explorer, Firefox, Chrome et Opera. Le tableau 1.1 montre en pourcentage la part de chaque navigateur.

Navigateur	Part d'utilisation en %
Chrome	52,32%
Internet Explorer	16,2%
Firefox	15,58%
Safari	9,78%
Opera	1,81%
Autres	4,31%

Tableau 1.1 – Statistiques des parts d'utilisation des navigateurs dans le monde entre mars 2015 et mars 2016.

Une ressource web étant sélectionnée en faisant appel au serveur est ensuite affichée par le navigateur. Celle-ci peut être un document HTML (qui est le cas général) comme elle peut être un autre type de fichier. Une URL est mobilisée par l'utilisateur pour la récupération de la ressource. Les spécifications HTML et CSS, qui précisent au navigateur la manière d'interpréter et d'afficher les fichiers HTML, sont maintenues par l'organisation W3C (World Wide Web Consortium) [12].

La structure d'un navigateur comprend plusieurs composants tels qu'une interface utilisateur, le moteur du navigateur, un composant réseau, une interface d'arrière-plan (backend UI), un interpréteur JavaScript, un analyseur XML (XML parser), un composant de stockage de données, et finalement le moteur de rendu, qui est la pièce la plus importante dans cet ensemble [49], [48]. La figure 1.4 montre le flux entre les composants du navigateur.

Avant son affichage à l'utilisateur, une page web parcourt le chemin suivant :

1. La requête est envoyée vers le serveur en utilisant l'interface utilisateur qui enferme une barre d'adresse, des boutons "précédent" et "suivant", un menu de marque-page,

des boutons d'actualisation et d'arrêt, etc. La requête est transportée via le composant réseau qui assure les appels réseau telles que les requêtes HTTP. Il est doté d'une interface indépendante de la plateforme et en dessous des implémentations pour chaque plateforme.

2. La réponse est renvoyée par le serveur après plusieurs traitements au niveau du serveur d'application afin de générer la page demandée en HTML (les détails sur ces deux étapes ont été exposés dans la section 1.1).
3. Le composant réseau passe les données brutes récupérées à un autre composant qui a comme rôle d'enregistrer toutes sortes de données sur le disque dur, par exemple, des cookies. Il s'agit d'une couche de persistance. La nouvelle spécification HTML (HTML5) définit le terme « base de données Web », qui est un système complet (bien que léger) de base de données dans le navigateur.
4. Les octets bruts sont transportés au moteur de rendu qui est responsable de l'affichage du contenu demandé à l'écran en suivant plusieurs processus d'analyse sur le code HTML et CSS. Les détails de ces processus d'analyse sont abordés dans le reste du chapitre.
5. Un interpréteur JavaScript est appelé par le moteur de rendu pour l'analyse et l'exécution du code JavaScript. Pareillement pour l'analyseur XML (XML parser) qui est utilisé pour l'analyse des documents XML dans l'arbre du DOM (Document Object Model). C'est l'un des sous systèmes les plus réutilisables dans l'architecture. En fait, presque toutes les implémentations des navigateurs exploitent un analyseur XML existant plutôt que de créer leur propre analyseur à partir de zéro.

6–7. L'interface d'arrière-plan (backend UI) est destiné à dessiner des widgets de base, du genre : listes déroulantes et fenêtres. Une interface générique non spécifique à la plateforme est présentée par le navigateur qui utilise d'autre part en dessous, l'interface utilisateur du système d'exploitation.

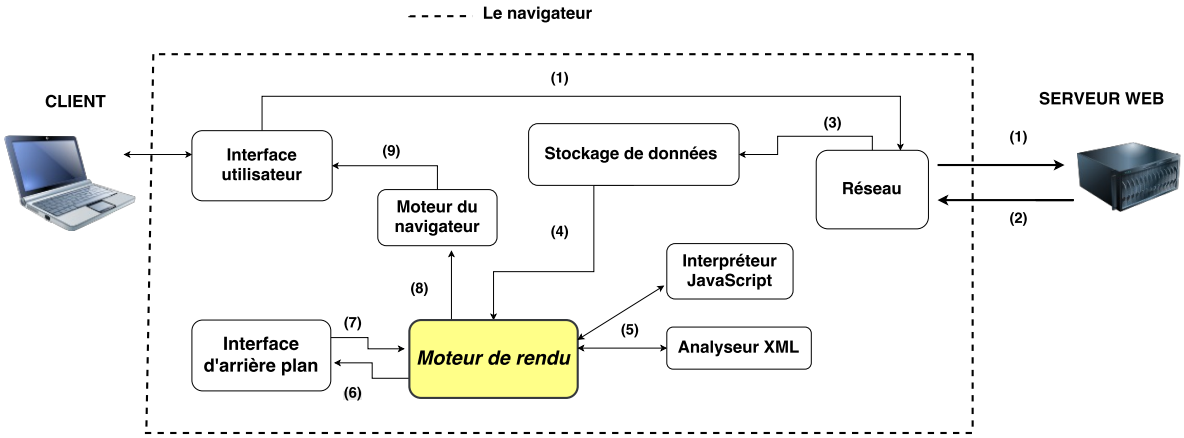


Figure 1.4 – Le flux entre les composants du navigateur pour la récupération et le traitement d'une page web dans le navigateur

8–9. Ce dernier composant est relié au moteur de rendu qui envoie le rendu final à l'utilisateur dans la dernière étape de la figure.

L'affichage d'une page web par le navigateur, moyennant le code HTML, CSS et JavaScript, n'est pas aussi simple. Pour ce faire, le navigateur fait particulièrement appel à l'un de ses composants : le moteur de rendu.

Les opérations qui se manifestent à l'intérieur du navigateur et en particulier le fonctionnement des moteurs de rendu des navigateurs les plus utilisés ont été expliqués par Garsiel et Irish [11]. L'affichage d'une page web comporte de fait plusieurs phases. La première consiste en la construction des arborescences DOM (modèle d'objet de document) et CSSOM (modèle d'objet CSS). Cette dernière phase est suivie par la transformation des balisages HTML et CSS en DOM et CSSOM respectivement, dont la combinaison forme une arborescence d'affichage (arbre de rendu). Cette arborescence, à son tour, est chargée de la mise en page de chaque élément visible et de l'introduction des données exigées pour l'affichage des pixels à l'écran [5].

```
<html>
  <head>
    <meta name="viewport" content="width=device-width,initial-scale=1">
    <link href="design.css" rel="stylesheet">
    <title>Exemple</title>
  </head>
  <body>
    <p>Salut <span>Oussama</span> Beroual </p>
    <div><img src="ma-photo.jpg"></div>
  </body>
</html>
```

Figure 1.5 – Un exemple simple de page web illustrant le rendu dans un navigateur.

Construction du modèle d'objet DOM

Le processus de construction du modèle d'objet DOM est exposé ci-dessous à travers l'exemple d'une page web simple en HTML brut avec du texte et une seule image. Le code HTML de la page à étudier est donné à la figure 1.5.

Le traitement de cette page impose au moteur de rendu du navigateur d'exécuter quatre processus de transformation, tel qu'illustré dans la figure 1.6.

1. Le premier processus est la **conversion** : le moteur de rendu lit les octets bruts du HTML sur le disque ou le réseau, et les traduit en caractères individuels en fonction de l'encodage spécifique du fichier (UTF-8, par exemple).
2. Le deuxième processus est la **création de jetons** : le moteur de rendu convertit les chaînes de caractères en différents jetons spécifiés par la norme HTML5 du W3C, telles que `<html>` et `<body>`. Chaque jeton possède une signification particulière et un ensemble de règles.
3. Le troisième processus est l'**analyse lexicale** : les jetons émis sont convertis en *objets* qui définissent leurs propriétés et leurs règles.

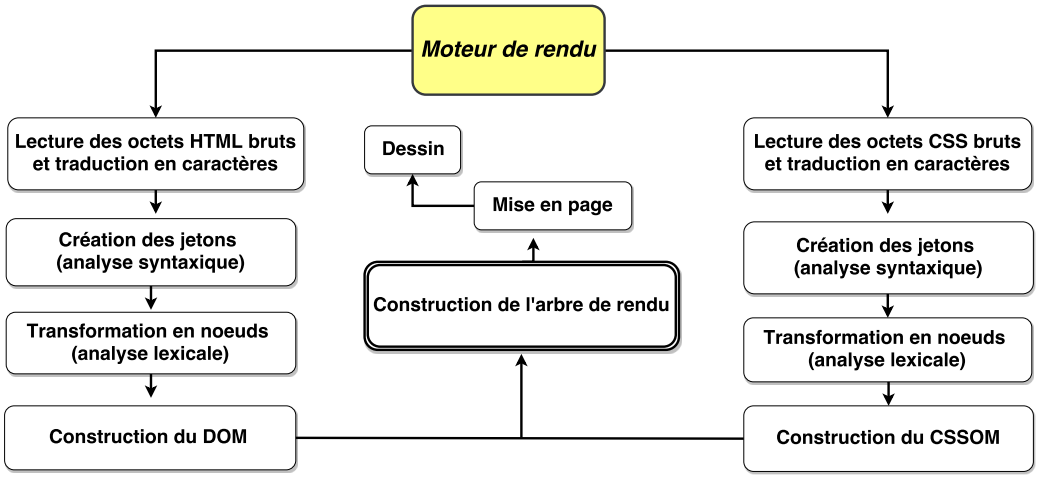


Figure 1.6 – Les processus de traitement de HTML et CSS au niveau du moteur de rendu du navigateur.

4. Le quatrième processus est la **construction du DOM** : puisque le balisage HTML définit les relations entre les différentes balises (certaines balises sont contenues dans d'autres), les objets créés sont associés selon une arborescence, qui capture également la relation parent-enfant définie dans le balisage d'origine (l'objet HTML est un parent de l'objet `body`, `body` est un parent de l'objet `p`, etc.).

Le résultat final de l'ensemble de ce processus est le modèle d'objet de document (ou DOM) de notre page simple, que le navigateur utilise pour tout traitement supplémentaire de la page. L'arbre DOM résultant est illustré dans la figure 1.7.

Modèle d'objet CSS (CSSOM)

L'information sur l'apparence d'un élément lors de son affichage est offerte par une autre construction : le CSSOM. Durant la construction du DOM de notre page, il s'est avéré que le navigateur a décelé une balise de lien `link` dans la section `head` du document, signalant une feuille de style CSS externe : `style.css`. Du fait qu'il a besoin de cette ressource pour

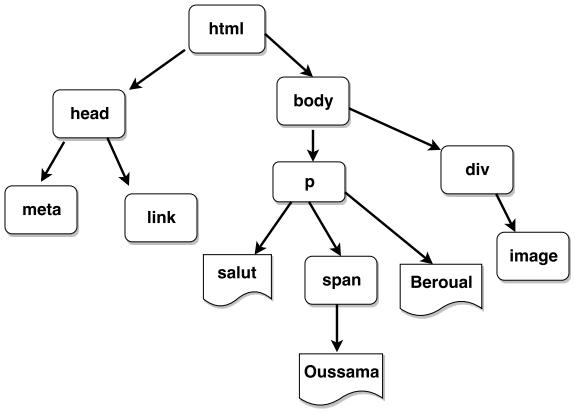


Figure 1.7 – Un arbre du modèle d'objet DOM.

```
body { font-size: 24px }
p { font-weight: bold }
span { color: blue }
p span { display: none }
img { float: right }
```

Figure 1.8 – Un exemple de fichier CSS simple.

l'affichage de la page, le navigateur anticipe l'envoi immédiat d'une demande pour cette ressource, et répond avec le contenu du fichier illustré à la figure 1.8. Ces styles auraient pu être déclarés directement dans le balisage HTML (intégré) ; cependant, il est recommandé de séparer les codes CSS et HTML, puisque les graphistes travaillent sur le code CSS, les développeurs sur le code HTML, etc.

Les règles CSS reçues sont converties en un contenu que le navigateur peut comprendre et traiter, de même que pour le code HTML. Le processus HTML est répété, mais dans ce cas pour le code CSS (voir figure 1.6). Les octets CSS sont convertis en caractères, puis en jetons et en nœuds, pour finalement se relier au sein d'une arborescence appelée *CSS Object Model (CSSOM)* ou « modèle d'objet CSS ».

Le calcul de l'ensemble final de styles d'un objet de la page, tel qu'effectué par le navigateur,

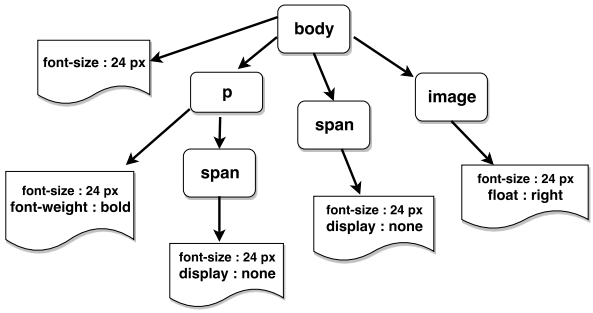


Figure 1.9 – Arbre du modèle d'objet CSSOM

comprend deux étapes. Dans une première étape, la règle la plus générale pour ce nœud est appliquée (par exemple, s'il s'agit d'un élément enfant du corps, tous les styles du corps s'appliquent). Dans une deuxième étape, des règles plus spécifiques, c'est-à-dire en descendant la cascade, sont appliquées afin d'affiner de manière récursive les styles calculés.

La concrétisation de cette démarche est faite à partir de l'observation de l'arborescence CSSOM dans la figure 1.9. On peut y lire que tout texte est écrit en bleu, et que sa taille de police est de 24 pixels. Il est placé dans la balise `span` contenue dans l'élément `body`. Ce dernier est chargé de transmettre la taille de police à l'élément `span`. Toutefois, si une balise `span` est un enfant d'une balise paragraphe (`p`), le contenu de cette balise n'est pas affiché.

Il faut se rappeler que l'arborescence CSSOM ci-dessus n'est pas complète. Elle ne montre que les styles qui remplacent ceux de la feuille de style. Dans l'absence de styles proposés, tout navigateur fournit un ensemble de styles par défaut, également appelés styles *user-agent*.

Construction de l'arbre de rendu, la mise en page et le dessin

Les données HTML et CSS ont servi jusqu'ici à la création des arborescences des modèles DOM et CSSOM, qui sont des objets indépendants chargés chacun de capturer un aspect du document : l'un décrit le contenu, et l'autre les règles de style appliquées au document.

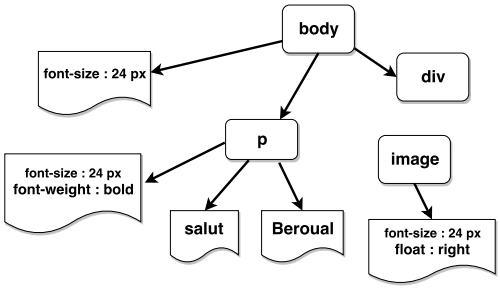


Figure 1.10 – Arbre de rendu

Les arborescences des deux modèles DOM et CSSOM sont combinées pour former une arborescence d'affichage responsable de l'affichage de la page et ne contenant que des nœuds nécessaires à l'affichage de la page (voir figure 1.10).

Une arborescence d'affichage comporte plusieurs types de nœuds : des nœuds Visibles ; des nœuds invisibles tels que les balises de *script*, les balises *Meta*, etc et des nœuds masqués par le code CSS. Seul le premier type est considéré dans l'arborescence d'affichage. Les deux autres sont omis. Le procédé d'omission est effectué en mobilisant deux propriétés : la propriété *visibility : hidden* rendant l'élément invisible tout en occupant de l'espace dans la mise en page et la propriété *display : none* supprimant totalement l'élément de l'arborescence d'affichage. La figure 1.11 donne une vue globale des étapes et processus de la construction des arbres abordées ci-dessous.

Tous les nœuds visibles sont soumis aux règles CSSOM et émis avec leurs contenus et leurs styles calculés. Néanmoins leurs positions et leurs géométries (tailles) restent indéfinies. La détermination ou le calcul de ces deux paramètres constitue la phase de *mise en page*, appelée parfois ajustement de la mise en page. Cette dernière est traduite par l'exemple simple donné à la figure 1.12. Le corps de la page ci-dessus contient deux éléments `div` imbriqués : le premier élément `div` (parent) définit la taille d'affichage du nœud à 50% de la largeur de la fenêtre, et le second élément `div` contenu par le parent définit sa largeur à 50% de celui du parent, soit

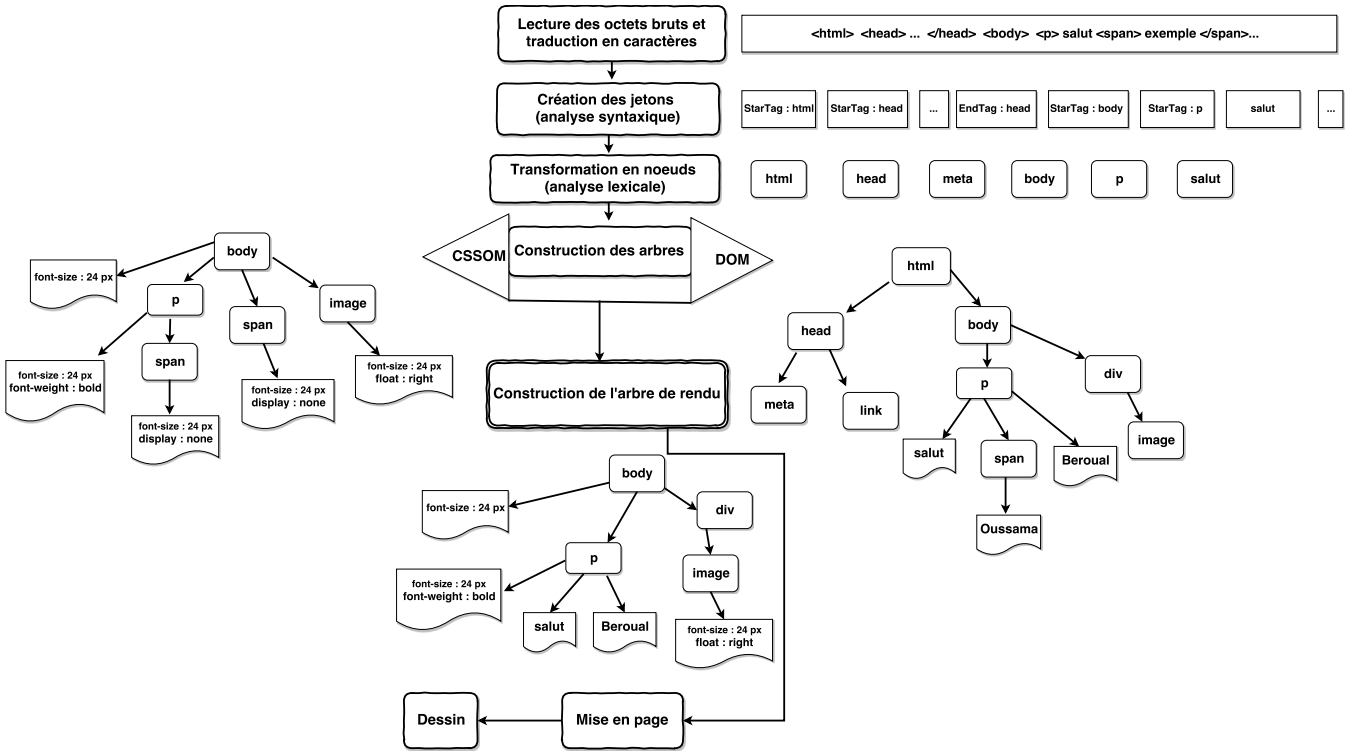


Figure 1.11 – Les étapes et processus de la construction des arbres DOM, CSSOM et de l'arbre de rendu.

```
<html>
  <head>
    <meta name="viewport" content="width=device-width,initial-scale=1">
    <title>Exemple de mise en page</title>
  </head>
  <body>
    <div style="width: 50\%">
      <div style="width: 50\%">Hello world!</div>
    </div>
  </body>
</html>
```

Figure 1.12 – Un second exemple simple illustrant le processus de mise en page par le navigateur.

25% de la largeur de la fenêtre.

Maintenant que toutes les informations relatives aux nœuds sont réunies, ceux-ci sont peints, convertis en pixels réels et affichés à l'écran à partir de l'arborescence d'affichage finale. La construction est achevée et la page est enfin visible dans la fenêtre. La figure 1.13 résume toutes les étapes précédemment décrites dans cette section.

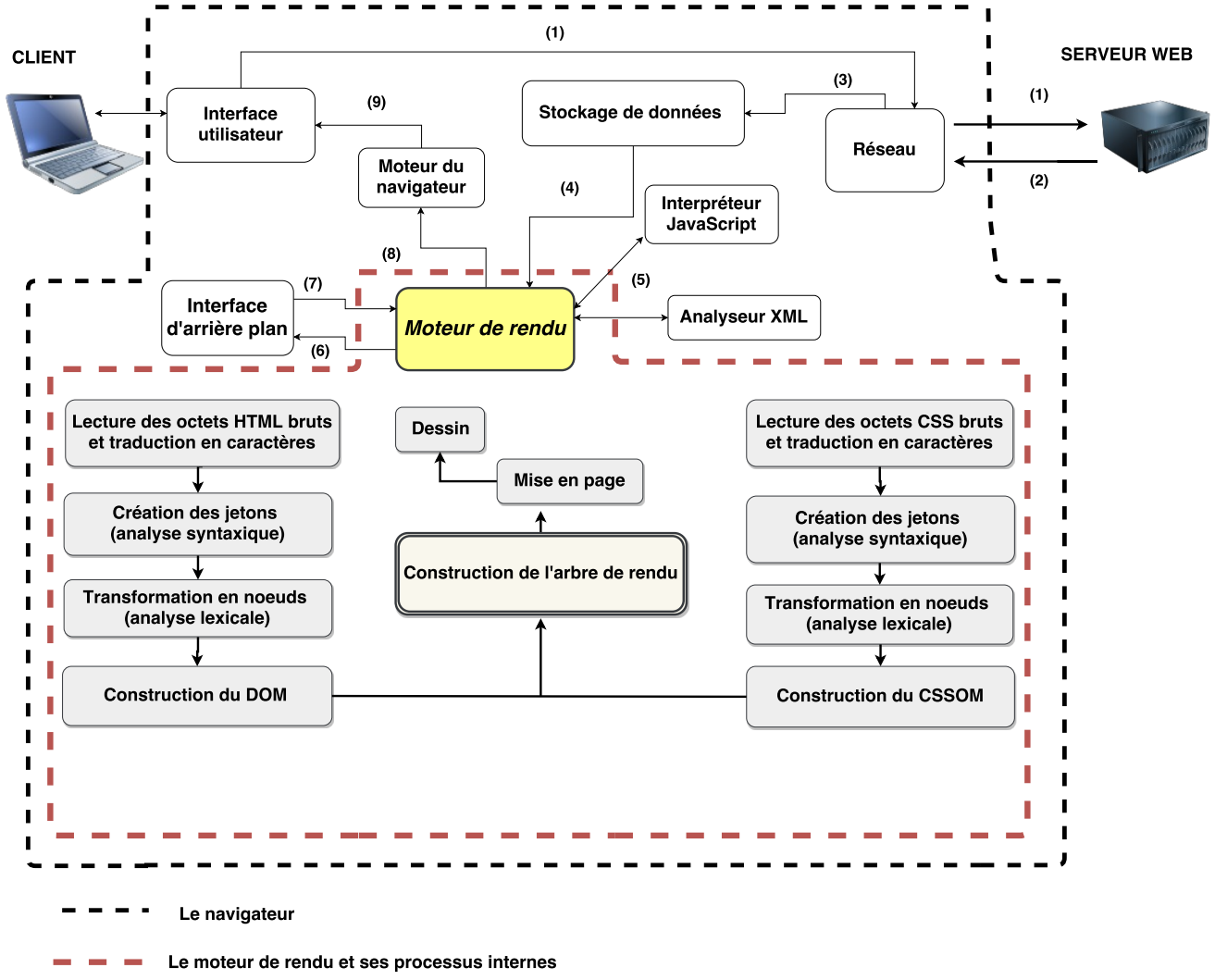


Figure 1.13 – Vue globale du flux de récupération, de traitement et d'affichage d'une page web.

CHAPITRE 2

LES BUGS D'INTERFACE DANS LES APPLICATIONS WEB

Les utilisateurs du web sont souvent confrontés lors d'une application web sur internet à des problèmes très ennuyeux appelés « bugs d'interfaces ». Dans ce chapitre, nous exposons les différents types d'application web et les bugs qui les affectent en montrant que les bugs d'interface sont largement présents dans un grand nombre de sites et applications web du monde réel et comment des bugs de ce genre ne sont pas limités à des problèmes de présentation simples, statiques, et qu'ils révèlent dans de nombreux cas des défauts dans le comportement de l'application.

2.1 TYPES D'APPLICATIONS WEB

Le web est maintenant peuplé par un nombre considérable d'applications ; par conséquent, le nombre de personnes affectées par les bugs qu'elles peuvent contenir est énorme.

2.1.1 APPLICATIONS STATIQUES

Les sites web traditionnels, qu'on appelle également applications « Web 1.0 », sont statiques : le contenu d'une page ne change pas après le chargement, et chaque page de l'application peut

être chargée indépendamment de toute interaction préalable avec le site. Le test automatique d'une telle application peut être effectué à l'aide d'un robot d'indexation en chargeant une page de démarrage, puis en explorant automatiquement les différentes interactions possibles sur la page pour obtenir de nouvelles pages à visiter. Comme chaque page de ces applications est indépendante, les bugs ne peuvent se produire que sur une seule page, d'où vient la possibilité d'écrire des oracles de test simples qui analyseraient le contenu de ces pages prises isolément.

Le *Responsive Web Design* est un moyen de concevoir un site web de sorte que son contenu s'adapte automatiquement à la résolution d'écran du terminal utilisé pour le visualiser. Une application RWD est donc un site web adaptatif. La notion de web adaptatif repense la conception ergonomique des sites web, puisqu'il ne s'agit plus de concevoir autant de sites que de terminaux, mais de concevoir une seule interface auto-adaptative. Ainsi, les informations et les structures techniques ne sont pas dupliquées ; ce qui génère des économies d'échelle dans la conception et la maintenance des sites web bénéficiant de ce mode de conception.

Un site web conçu avec un design responsive est donc optimisé pour tous les appareils disponibles sur le marché : ordinateurs, tablettes, smartphones. Cette façon de concevoir des applications web est recommandée en raison de quelques avantages majeurs : le développement et la maintenance seront plus rapides et plus simples (un seul fichier gérant tous les affichages). Un référencement naturel optimal est utilisé (pas de sous-domaines ou de redirections vers des annuaires mobiles) puisqu'il n'y a qu'un seul site regroupant toutes les versions et uniquement une adresse Web (URL). Plus de détails sur le concept de RWD sont présentés dans la section 2.2.3.

2.1.2 APPLICATIONS WEB 2.0

Les applications web récentes, également appelées RIA (*Rich Internet Applications*), utilisent des scripts côté client, des états côté serveur et d'autres fonctionnalités pour offrir une expérience utilisateur améliorée. Les modifications apportées à l'état de la page peuvent venir à la suite de requêtes HTTP asynchrones chargeant un nouveau contenu. Par conséquent, ces applications sont maintenant *stateful* : la même action, ou la même requête HTTP, peut renvoyer des résultats différents en fonction des interactions passées de l'utilisateur. Ces applications avec état viennent avec de nouveaux types de bugs, appelés bugs comportementaux. Contrairement à un bug statique, qui s'évalue en analysant le contenu d'une seule page indépendamment des autres, un bug comportemental relie les données et l'ordre de consultation de plusieurs pages de l'application.

2.2 TYPES DE BUGS D'INTERFACE

Un bug d'interface est un défaut dans un système web qui a des effets visibles sur le contenu des pages servi à l'utilisateur. Nous avons effectué une étude sur de plus de 35 applications web [51] en tous genres. Un relevé des bugs ayant un impact sur la présentation ou le contenu de l'interface utilisateur a été fait. Le tableau 2.1 donne la liste des sites web et des applications pour lesquelles au moins un bug de mise en page a été trouvé. Dans ce qui suit, nous présentons brièvement les bugs découverts dans cette étude. Nous soulignons qu'aucun des bugs décrits ici représente un problème de compatibilité entre les navigateurs. Les bugs sont présents peu importe le navigateur utilisé pour afficher la page, et ils ne sont pas causés par une interprétation divergente des normes par deux navigateurs différents.

- Academia.edu
- Acer
- Adagio Hotels
- Air Canada
- Air France
- AllMusic
- American Airlines
- Boingo
- Canadian Mathematical Society
- Customize.org
- Digital Ocean
- Dropbox
- EasyChair
- Elsevier
- Evous France
- Facebook
- IEEE
- Just for Laughs
- LinkedIn
- Liveshout
- Microsoft TechNet
- Monoprix
- Moodle
- Naymz
- NSERC
- OngerNeige
- ProQuest
- Rail Europe
- ResearchGate
- St-Hubert
- SpringerOpen
- TD Canada Trust
- Time Magazine
- Uniform Server
- YouTube

Tableau 2.1 – Sites et applications web pour lesquelles au moins un bug de mise en page a été trouvé.

2.2.1 BUGS STATIQUES

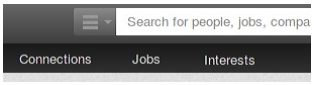
Un premier type de bugs, appelés bugs statiques, collectés par une étude empirique dans des sites web et applications réelles. On y retrouve des problèmes tels que le chevauchement des éléments, les éléments qui s'étendent en dehors de leur conteneur ou le bug d'éléments mal empilés. Ce genre de bugs peut être divisé en deux catégories : des bugs liés à la mise en page, et d'autres non liés à la mise en page.

Bugs de mise en page

Une première catégorie de bug correspond à des perturbations dans la mise en page ou la présentation de la page elle-même, c'est-à-dire tout résultat inattendu du concepteur de cette page en termes de contenu ou de propriétés (position, taille, nombre d'éléments).

Éléments mal alignés Un problème de mise en page banal mais fréquent est le désalignement ou le décalage d'éléments qui doivent être alignés. La figure 2.1a montre un exemple de

(a) LiveShout



(b) LinkedIn

Figure 2.1 – Exemple d'éléments mal alignés : le cadre blanc est horizontalement mal aligné (LiveShout) ; le menu *Interests* est décalé d'un pixel par apport aux autres menus (LinkedIn).

ce bug courant, mais parfois, le désalignement est subtil : un élément peut être décalé par un seul pixel (figure 2.1b).

Éléments qui se chevauchent Le problème de chevauchement des éléments qui devraient être disjoints dans la présentation d'une page est aussi l'un des bugs les plus répandus dans cette étude ; les problèmes de cette nature ont été trouvés dans des sites aussi variés que le Time Magazine, Air Canada, Moodle et Rail Europe. La figure 2.2 montre un exemple de ce problème.

Dans certains cas, le chevauchement se produit par une interruption de la mise en page causée par le redimensionnement du document principal en dessous des dimensions minimales, comme dans l'exemple de la figure 2.2a. Les conteneurs qui doivent être côte à côte sont affichés les uns sur les autres par le moteur de rendu du navigateur. Cependant, nous avons également constaté que, dans de nombreux cas, ce chevauchement est causé par le fait que les

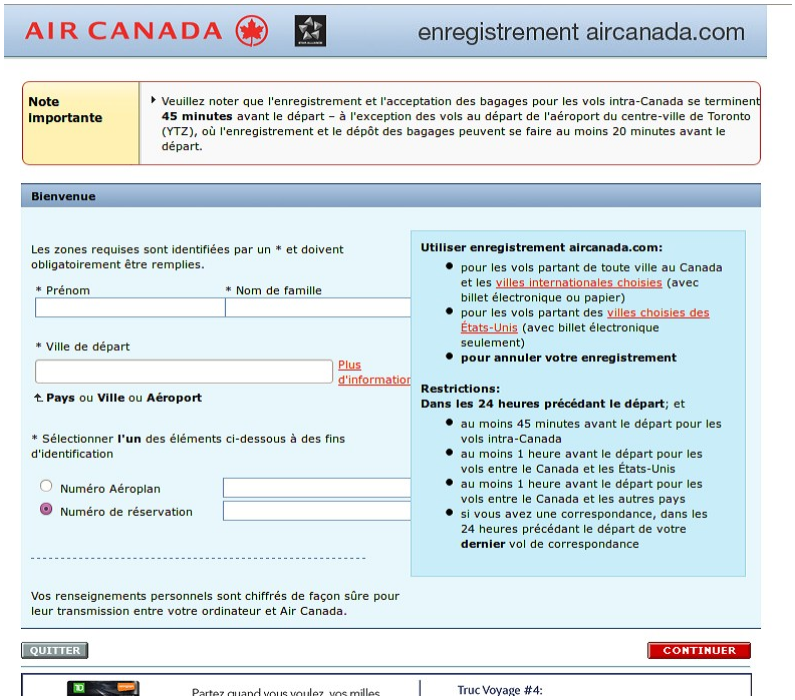
éléments sont absolument positionnés dans une page par rapport à leurs dimensions lorsqu'ils contiennent du texte en anglais et lors de l'affichage du site web dans une autre langue (comme le français), il peut arriver que le texte correspondant soit plus long que la version anglaise, ce qui cause le chevauchement de deux éléments qui devraient être disjoints. Cela a été observé, par exemple, sur la figure 2.2b pour le site web RailEurope.

Éléments qui s'étendent à l'extérieur de leurs conteneurs Un autre problème répandu est la présence d'éléments qui dépassent les limites de leurs conteneurs parent, ce qui provoque le chevauchement indésirable avec des éléments environnants. Ce bug est l'inverse du précédent : plutôt que d'étendre leur conteneur en dehors de ses dimensions nominales, cette fois, les éléments qui s'élargissent à l'extérieur de leur conteneur sont simplement coupés de l'affichage.

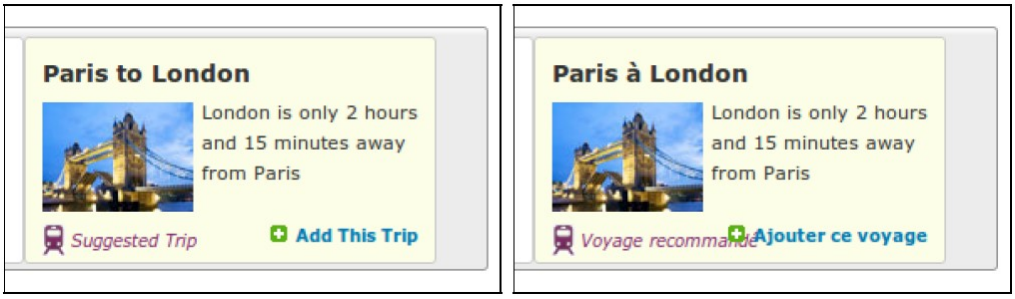
La figure 2.3 montre un exemple de ce dernier type dans la plateforme d'enseignement en ligne Moodle (la figure 2.3a) : des éléments de la table sont coupés de leur côté droit. Lors de l'analyse du code source de la page, il se trouve que quatre autres colonnes doivent effectivement être visibles et sont tout simplement inaccessibles puisque ils ont été coupé. Cela rend la page plus ou moins inutilisable.

Un cas particulier survient lorsque le conteneur soit la fenêtre du navigateur entière et la fenêtre est redimensionnée en dessous d'un certain seuil. Cela peut être montré dans la Figure 2.3b : le contenu du menu en haut disparaît lorsque la fenêtre est redimensionnée. Le navigateur fournit une barre de défilement horizontale, mais cela ne fait défiler que la partie inférieure de la page et pas le menu du haut. En conséquence, certains boutons du menu principal du site ne peuvent plus être cliqués.

Nous plaçons aussi dans cette catégorie un bug rencontré en utilisant la fenêtre de messagerie

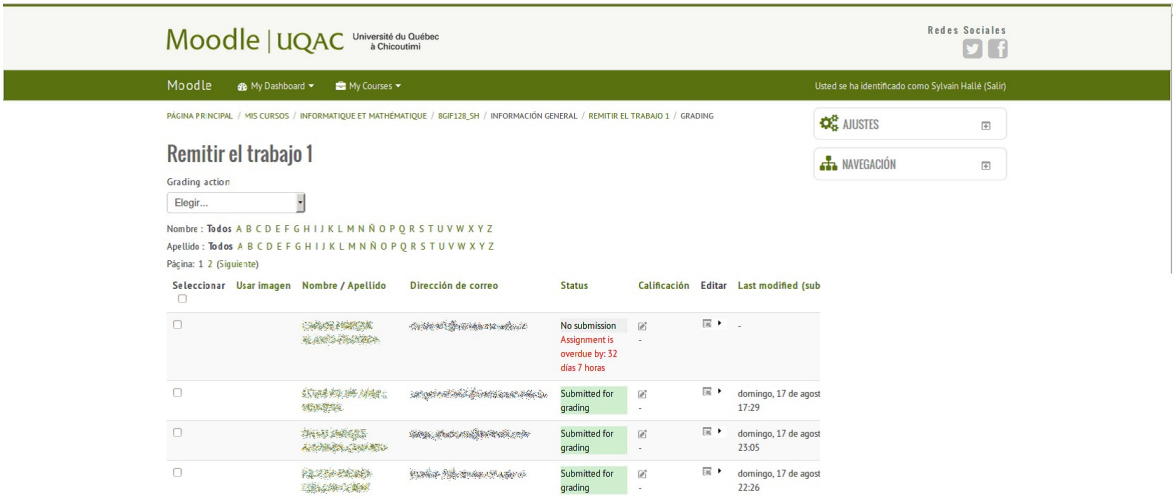


(a) Air Canada

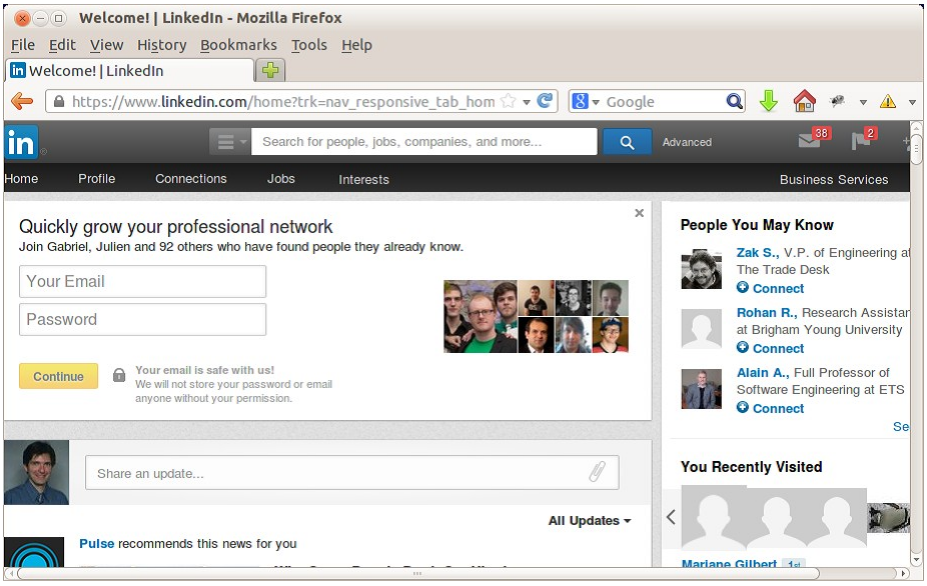


(b) Rail Europe

Figure 2.2 – Exemple d'éléments qui se chevauchent



(a) Moodle



(b) LinkedIn

Figure 2.3 – Le contenu de la page est dissimulé de l'interface en raison de son prolongement au-delà des dimensions du conteneur parent.

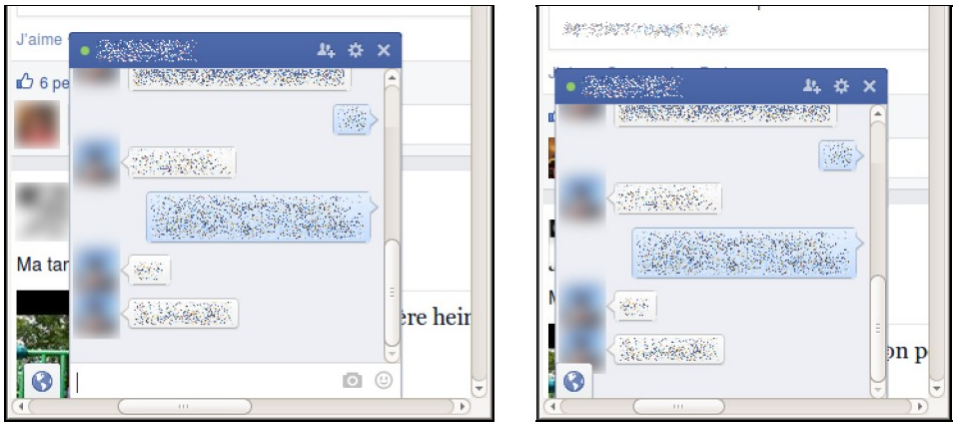
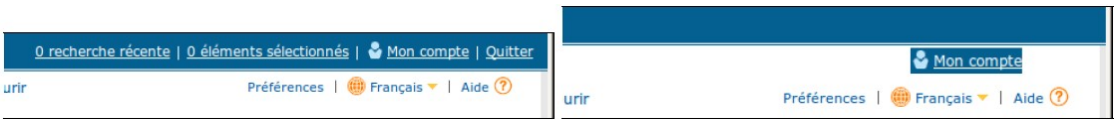


Figure 2.4 – Le bug Facebook. La zone de texte permettant au utilisateur de taper un message instantané (à gauche) disparaît soudainement (à droite).

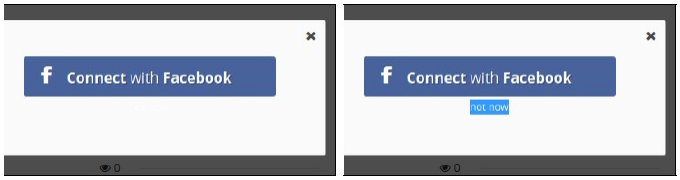
instantanée de Facebook. Lorsque la fenêtre de parent est redimensionnée au-dessous d'une largeur spécifique, la zone de texte permettant à un utilisateur d'entrer un nouveau message disparaît soudainement de l'interface, comme le montre la figure 2.4. Tous les autres éléments de la fenêtre restent identiques, mais il devient impossible de taper un message.

Ton sur ton Ce bug est causé par un problème de mise en page où le texte d'un élément est de la même couleur que le fond de la page à cet endroit, ce qui rend le texte invisible dans l'affichage. Bien que ce type de comportement peut être fait exprès, afin de dissimuler un élément, nous croyons qu'il est très peu probable que cela soit l'intention du développeur, puisqu'il existe des moyens beaucoup plus élégants d'atteindre le même résultat en utilisant les propriétés CSS (en définissant la propriété d'affichage à `none`, ou la propriété `opacity` à 0).

La figure 2.5 montre un exemple d'un tel problème de mise en page sur Academia.edu (figure 2.5b). Dans la figure 2.5a, le redimensionnement de la fenêtre du navigateur en dessous d'une largeur spécifique a pour effet imprévu de pousser les éléments du menu supérieur en dehors de leur conteneur prévu (celui qui a un fond bleu). Par conséquent, ces éléments qui ont du



(a) ProQuest



(b) Academia

Figure 2.5 – Le bug « ton sur ton ». Les éléments de menu sont poussés en dehors de leurs conteneur et disparaissent (a) ; le texte du lien est de la même couleur que le fond, révélé en le sélectionnant avec la souris (b).

texte blanc, se placent dans une zone blanche et deviennent invisibles (à l'exception d'un seul élément du menu qui a un fond bleu pour une raison inconnue).

Éléments mal empilés Ce problème se produit quand un élément qui devrait être rendu au-dessus d'un autre est placé en dessous de ce dernier, comme le montre la figure 2.6. Dans cet exemple, le bouton orange en haut de l'image est censé apporter un menu (liste déroulante). Toutefois, le contenu de ce menu apparaissent sous le reste du contenu de la page, plutôt que sur eux, ce qui rend certains de ses éléments inutilisables.

L'ordre du processus du dessin est défini dans le standard de CSS : c'est en fait l'ordre dans lequel les éléments sont empilés dans la pile des contextes. Cet ordre affecte le dessin puisque les piles sont dessinées de l'arrière vers l'avant. L'ordre d'empilement d'un bloc de rendu est :

1. Couleur d'arrière-plan
2. Image d'arrière-plan
3. Bordures

Figure 2.6 – Un élément est placé incorrectement sur un autre.

4. Enfants
5. Contour

Ainsi, une violation de cet ordre pour une raison inconnue peut causer ce genre de problème. La plupart des problèmes de cette nature peuvent être corrigées en attribuant correctement la propriété `z-index` de l'élément défectueux.

Disposition brisée Une rupture majeure dans la structure attendue d'une page se produit quand un certain nombre de ressources importantes, telles que les fichiers CSS, le code JavaScript ou les images, ne parvient pas à être récupéré par le navigateur. En conséquence, de nombreux éléments de la page n'ont pas leurs déclarations de style et sont positionnés et dimensionnés en fonction du style par défaut fourni par le moteur de rendu.

La figure 2.7 montre un cas assez grave d'un tel problème, où essentiellement toutes les

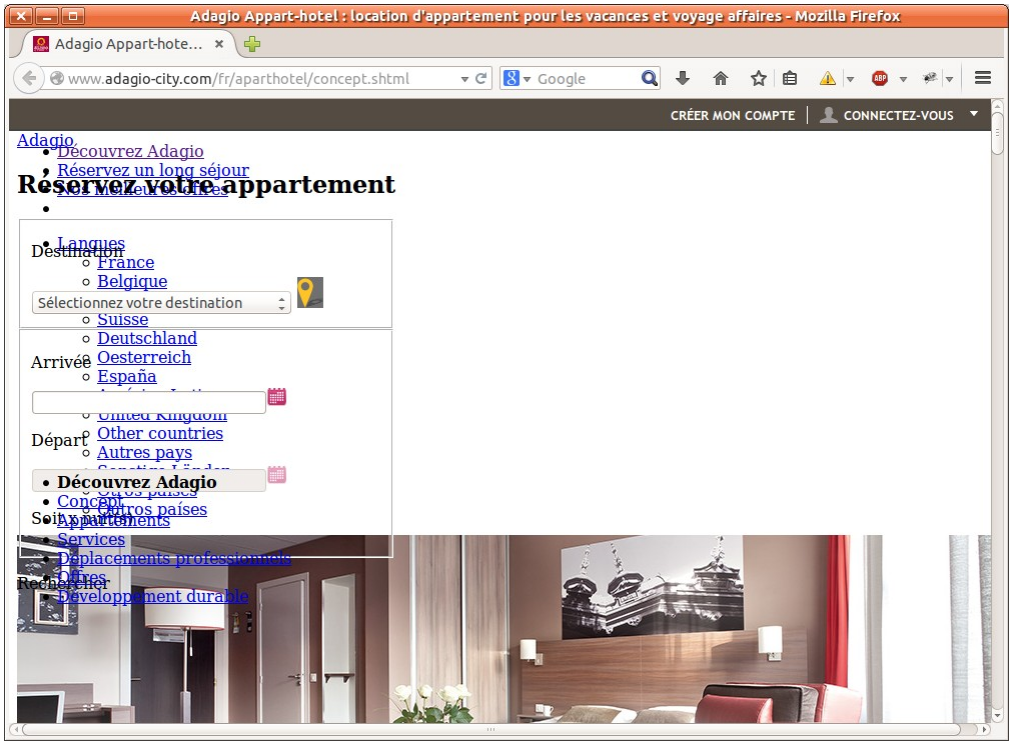


Figure 2.7 – Un exemple d'interface brisée en raison de l'échec du chargement de certaines ressources.

déclarations CSS sont manquantes. Ceci est causé par le fait que le fichier `aparthotel.css` n'a pas réussi à être chargé pour une raison inconnue. Le même problème a été observé momentanément sur le portail web Digital Ocean 2.8.

Sur le site web de Digital Ocean, la page de connexion (login) est brisée et toutes les autres pages aussi comme le montre la figure 2.8a. Cela est dû au fait que le navigateur a abandonné le chargement de nombreuses ressources dans la page (y compris toutes les images). Il est à noter que malgré ces problèmes, le site Web reste fonctionnel. La figure 2.8b représente une capture d'écran de ce à quoi la page devrait ressembler.

HTML malformé Dans la figure 2.9 l'HTML malformé rend impossible de voir le fond d'écran.

(a) Digital Ocean sans CSS. (b) Digital Ocean avec CSS.

Figure 2.8 – Exemple d'interface brisé dans le site Digital Ocean.

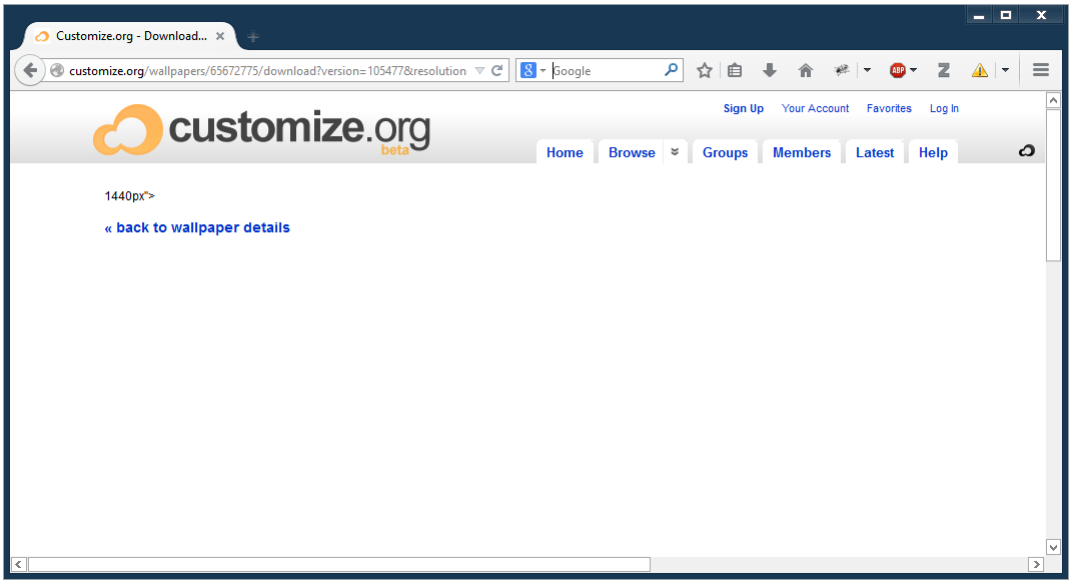


Figure 2.9 – Exemple d'HTML mal formé.

Éléments inaccessibles Dans la figure 2.10a, la page affiche correctement une mise en page «écran large» lorsque la fenêtre du client est assez large. Par contre, si la largeur de la fenêtre est moyenne comme le montre la figure 2.10b on observe que le bouton de connexion (sign in) en haut à droite s'est déplacé à l'extérieur de la bande supérieure.

Incohérence dans les valeurs possibles d'un champ de formulaire Dans une page du site CallingCards, le champ du code postal dans le formulaire est pré-rempli avec 6 caractères de données antérieures mais en essayant de modifier le code postal, il n'est pas possible de taper plus de 5 caractères dans ce champ (figure 2.11).

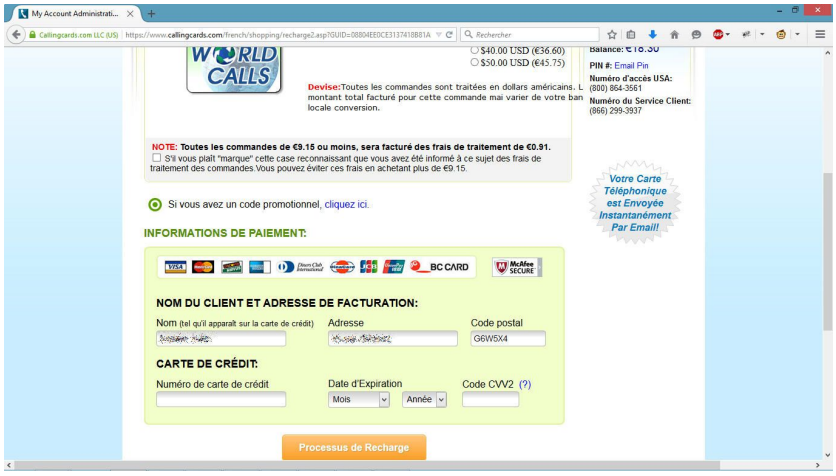


Figure 2.11 – Incohérence dans le champ de formulaire du site CallingCards.

Bugs non liés à la mise en page

Plusieurs autres bugs peuvent être détectés en analysant le contenu et la présentation d'une page. Nous verrons dans la suite que certains d'entre eux se rapportent même aux comportements attendu ou aux fonctionnalités de l'application, mais peuvent quand même être détectés par des règles exprimées sur un seul instantané statique de la fenêtre de l'application.

Mojibake et problèmes d'encodage Plusieurs sites et applications gèrent mal les données de chaîne en dehors de l'ASCII 7-bits. Divers encodages de caractères, tels que Cp1252 ou UTF-8, peuvent être utilisés pour représenter les caractères « accentués » ou « étrangers ». Cependant, le même caractère peut être représenté par une valeur binaire différente selon le schéma de codage utilisé ; pire, certains codages, comme UTF-8, peuvent utiliser plusieurs octets pour représenter un seul caractère.

Les problèmes surgissent quand un système n'interprète pas correctement le contenu d'une chaîne de caractères, croyant qu'un document est dans un certain encodage alors qu'il en utilise réellement un autre. Il en résulte souvent un caractère incorrect, superflu, ou pas affiché du tout ; ce phénomène est appelé *mojibake*. [1] Par exemple, l'encodage UTF-8 du caractère « é », lorsqu'il est interprété comme une chaîne Cp1252, produit « Ã© », comme le montre la figure 2.12a. À l'inverse, le rendu Cp1252 du caractère « é » donne « » quand il est interprété comme du UTF-8 (figure 2.12b).

Bien que ces caractères puissent être des données légitimes, leur présence dans le contenu d'un élément indique très probablement une manipulation incorrecte de l'encodage des chaînes par l'application étudiée.

Un autre exemple de ce genre de bug est montré dans la figure 2.13 :
2.13a donne l'exemple lors de l'invitation de quelqu'un de votre carnet d'adresses dont le nom contient un accent, Doodle affiche son nom incorrectement et cela donne une adresse invalide.
2.13b montre qu'en cliquant sur la suggestion, on observe que l'ajout est fait mais que le nom est coupé en deux parties à l'endroit même du caractère échappé.

1. Terme japonais signifiant « transformation de caractère ».

Problèmes d'échappement Les problèmes d'échappement surviennent lorsque les chaînes avec des caractères spéciaux ne parviennent pas à être correctement codées ou décodées entre deux applications. La manifestation la plus fréquente de ce problème est lors de la lecture et de l'écriture des chaînes de caractères dans une base de données. Certains caractères, comme l'apostrophe, nécessitent d'être doublés afin de ne pas être confondus avec un séparateur de chaîne. Les problèmes surviennent quand un système ne parvient pas à remplacer les apostrophes doubles par des apostrophes simples lors de l'affichage des données dans un formulaire.

Lors de l'enregistrement du contenu du formulaire, au retour à la base de données, chaque apostrophe sera de nouveau doublée, entraînant un quadruplet d'apostrophes lors du chargement de la page la prochaine fois, et ainsi de suite ; ceci est illustré dans la figure 2.14. Ainsi la recherche de plusieurs apostrophes dans un formulaire peut être utilisée pour détecter l'échappement incorrect à l'intérieur du code source.

Un problème de nature similaire se produit lorsque des caractères spéciaux dans le code source de la page (tels que HTML ou JavaScript) sont échappés incorrectement. Par exemple, une séquence comme `<p>` peut être transformée en `&lt;p&gt;`, résultant en la chaîne littérale `<p>` affichée comme du texte, plutôt que d'être interprétée comme une balise de paragraphe.

Nous avons découvert des exemples de ce problème sur EasyChair, qui affiche du code HTML brut, ou sur YouTube qui affiche du code JavaScript, comme on peut le voir à la figure 2.15.

2.2.2 BUGS COMPORTEMENTAUX

D'autres bugs peuvent être appelés « comportementaux » : ils révèlent un problème dans la fonctionnalité du site, ou alors ils peuvent être exprimés uniquement en termes d'une séquence

de plusieurs pages dans l'ordre.

Contrairement à une application web traditionnelle, une application Internet riche (RIA) utilise les technologies Web modernes émergentes telles que AJAX [47], Flash et Silverlight. Par conséquent, de nouveaux problèmes de tests web sont ajoutés aux problèmes existants. Une caractéristique importante de ces applications est qu'elles sont *stateful* : leur code peut stocker des données persistantes sur le client (en utilisant WebStorage, les propriétés CSS, les variables JavaScript et les objets) et sur le serveur (à l'aide de cookies, de stockage de session et de bases de données). De plus, l'état de l'application est dispersé sur un certain nombre d'éléments et ne peut pas simplement être assimilé à l'URL de la page en cours (affichée dans la barre d'adresse du navigateur). Des fonctionnalités telles que la communication asynchrone, la manipulation DOM côté client et les gestionnaires d'événements permettent de changer l'état de l'application sans un rechargement complet de la page et sans modifier son URL [35].

Une conséquence positive de ces fonctionnalités est qu'une telle application peut fournir une expérience utilisateur plus riche (d'où son nom) ; sans les cookies et JavaScript, les opérations simples telles que les manipulations de panier, les sessions utilisateur (connexion / déconnexion) et autres fonctionnalités ne seraient pas possibles. Cependant, la présence d'un état dans l'application introduit également la possibilité d'incohérences dans l'état affiché d'une page à une autre. Ces problèmes sont appelés bugs comportementaux, car ils sont la conséquence de l'interaction de plusieurs pages entre elles et dans un certain ordre.

Éléments mobiles Certains éléments peuvent changer leur position involontairement lors de l'interaction avec un utilisateur. La figure 2.16 montre un exemple d'un tel problème sur le site web du CRSNG, où écrire un texte dans une zone de texte vide auparavant réduit sa largeur de 4 pixels et décale légèrement les zones de texte restantes vers la gauche.

Ce problème s'est avéré plus répandu que nous nous attendions ; les variations comprennent certains styles d'éléments (la bordure ou la taille) qui ont été changés sans aucune raison apparente.

Le dysfonctionnement des boutons Beaucoup d'applications montrent à l'utilisateur des éléments qui se comportent comme une fenêtre pop-up (une fenêtre secondaire qui s'affiche, sans avoir été sollicitée par l'utilisateur, devant la fenêtre de navigation principale lorsqu'on navigue sur Internet). Ce moyen est communément utilisé pour afficher des messages publicitaires ou un avertissement comme la réponse à un message privé dans un forum.

Toutefois dans un certain nombre de cas, les boutons dans ces fenêtres sont inopérants : cliquer sur eux un certain nombre de fois ne produit aucun effet. Ce problème a été témoin dans des sites aussi variés qu'American Airlines, Dropbox et BitBucket. Dans certains cas, l'utilisateur est effectivement coincé dans le pop-up ou la page qui contient le bouton, et ne peut pas sortir sans forcer un rechargement complet du document.

Confusion dans le statut de connexion Deux sites web dans notre étude présentent des incohérences dans l'état d'une page, mélangeant des éléments de la fenêtre d'une page normalement réservés aux utilisateurs enregistrés (tels que les menus personnalisés) ainsi que des éléments réservés aux utilisateurs qui sont déconnectés (comme un formulaire de connexion).

Nous avons assisté à un tel comportement dans le site web de l'IEEE, pour le formulaire de demande de membre Senior. Après l'expiration d'un certain délai, l'utilisateur doit nécessairement se connecter à nouveau ; toutefois, la page présentée à l'utilisateur est celle représentée sur la figure 2.17. On remarque que le nom de l'utilisateur est toujours présent en haut à droite de la page, même s'il est censé être déconnecté (les identifiants de connexion sont demandés).

Cliquer sur ce nom apporte le menu déroulant qui est normalement accessible uniquement lorsque l'utilisateur est connecté. Le même problème a été trouvé sur le site de réservation d'Air Canada.

Incohérences dans le résultat de la recherche Un autre problème de comportement est celui de l'incompatibilité entre une requête faite par un utilisateur, et les résultats affichés par l'application en réponse à cette requête.

La figure 2.18 montre un exemple de ce problème sur le site web des épiceries Monoprix. Lors de la recherche des magasins à proximité, un utilisateur peut taper un code postal dans un champ de formulaire, puis cliquer sur « Valider ». Cependant, les résultats de recherche affichés dans la page suivante ne montrent presque jamais les magasins avec le code postal souhaité.

Comme avec tous les bugs dans la présente catégorie, cela peut être observé seulement par la corrélation de plusieurs éléments dans deux états différents de la page : le texte dans la zone de texte, un clic sur un bouton spécifique, suivi par le texte dans la liste des éléments qui apparaissent dans la page résultante.

2.2.3 BUGS DE RESPONSIVE WEB DESIGN

Il y a quelques années, les développeurs Web pouvaient faire des hypothèses sur la taille de l'écran des appareils des utilisateurs. Les ordinateurs de bureau ont été conçus pour accéder aux sites web en assumant une taille de fenêtre minimale ; il était donc « normal » que, pour une largeur de fenêtre plus petite, le site semble brisé. Il s'agissait d'une approche valide dans une époque où les tablettes et les smartphones étaient inouïs (pas à la portée de tout le monde : peu utilisés). Aujourd'hui, une autre approche est nécessaire pour s'assurer que les

sites fonctionnent correctement dans une gamme de différents appareils et tailles de viewport ("le viewport désigne la partie d'une page web qui est visible dans la fenêtre du navigateur d'un ordinateur, smartphone, tablette ou d'un autre dispositif. Le viewport est variable et mouvant en fonction de la taille de la fenêtre du navigateur et en fonction de l'utilisation des fonctions de défilement ou scrolling (souris ou ascenseur))"[6].

Par conséquent, la conception des sites doit désormais tenir compte de plusieurs catégories d'appareil selon la taille de l'écran. Cependant, le changement rapide des propriétés de l'appareil n'a pas pu être suivi par les développeurs web. Pour remédier à ce problème, ces derniers ont eu recours aux mêmes hypothèses en fonction de la demande du serveur. La demande d'une ressource par un navigateur est suivie par une chaîne d'agent utilisateur (*user agent string*), généralement envoyée avec la demande pour identifier le type de navigateur utilisé. La lecture de cette dernière du côté du serveur entraîne la diffusion de deux versions : une version mobile conçue pour une largeur maximale si l'utilisateur envoyait des chaînes d'agents d'utilisateurs mobiles et une version bureau conçue pour une largeur minimale.

Les développeurs ont pu, suite à l'introduction des *media queries* de CSS, écrire des déclarations de style conditionnelles par des propriétés multimédia telle que la taille de la fenêtre (voir figure 2.19). L'adaptation d'un site pour un support spécifique ou une taille de fenêtre au moment de l'exécution est donc devenue donc possible. Ce concept, qui sert à faire en sorte qu'un seul site réponde à différentes propriétés multimédias (principalement la taille de la fenêtre) au moment de l'exécution afin d'améliorer l'expérience de l'utilisateur est appelé *Responsive Web Design* [59].

Une étude récente, menée par Walsh *et al.*, identifie cinq types de bugs survenant plus particulièrement dans les sites web RWD [73]. Pour la plupart, il s'agit de bugs déjà observés dans d'autres sites au cours de notre propre étude.

Collision d'éléments C'est un bug dans lequel les éléments se chevauchent en raison de la modification de la fenêtre (viewport). Ce bug peut également influencer le bon fonctionnement des sites web dans le cas où certains éléments fonctionnels dans les pages sont masqués.

Dépassement d'éléments Les éléments ont besoin de se redimensionner, ils manquent d'espace mais ils doivent aussi être assez grands pour contenir tous leurs enfants. Cela arrive dans le cas où un élément dépasse à l'extérieur de son parent en raison d'un manque d'espace. L'élément peut alors être inaccessible, masqué par un autre élément ou par-dessus d'autres éléments.

Ce bug sur le site `https://www.thelily.com/`. On peut le voir dans la figure 2.20 où le *div* avec les boutons du menu se termine en dehors de la barre de menu et hors de vue.

Dépassement du viewport Ce bug se produit lorsque les éléments sont poussés hors de la fenêtre (viewport) et deviennent inaccessibles ou cachés. Sur le site web `https://www.slaveryfootprint.org`, ce bug a été trouvé. La figure 2.21 montre comment les bugs non observables peuvent créer des problèmes à des largeurs inférieures. Ici, le *div* du milieu est un peu en dehors de la fenêtre, mais il ne montre aucun problème observable. Cependant, c'est à 440px que nous pouvons voir clairement le contenu déborder de la fenêtre.

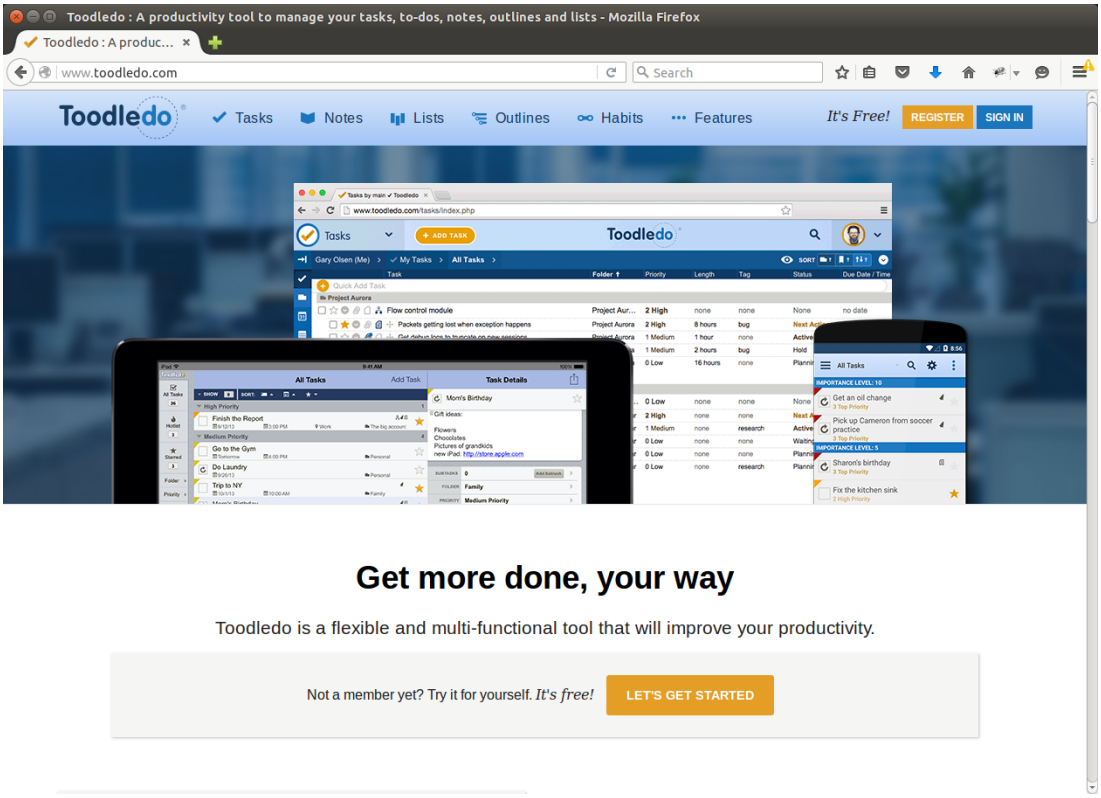
Disposition à petite échelle En fonction de l'implémentation, certaines mises en page peuvent être affichées correctement que dans un petit intervalle de largeur. Par exemple, un affichage pourrait être seulement correct entre 320 et 325 pixels de largeur.

Couverture d'éléments Ce bug survient lorsqu'un conteneur n'est pas assez large pour contenir tous les éléments et qu'un ou plusieurs éléments sont poussés sur une ligne supplé-

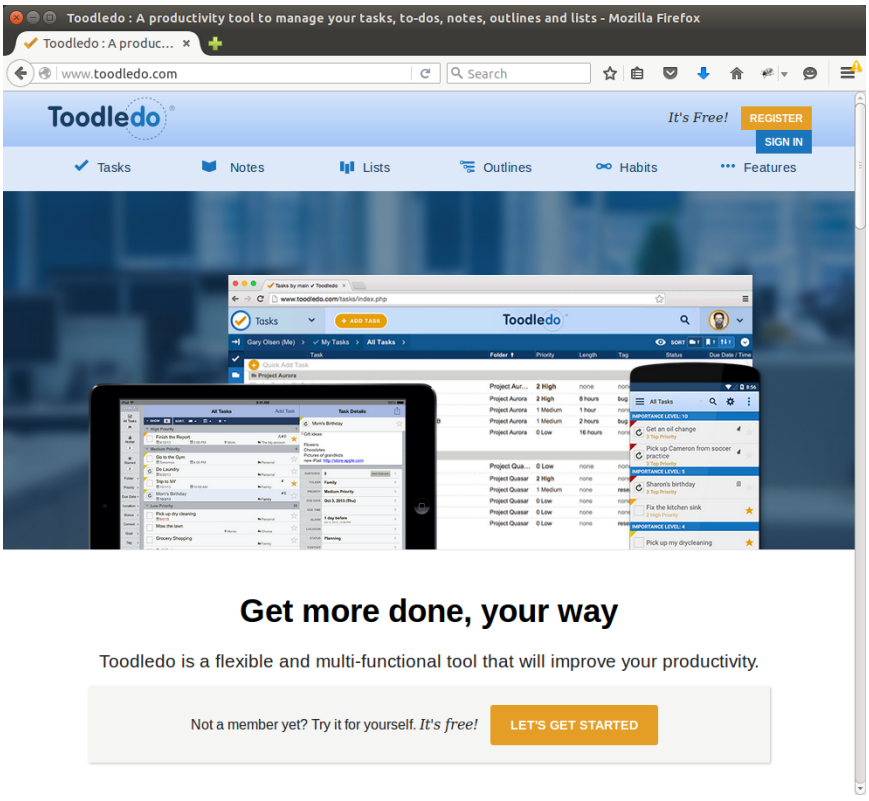
mentaire.

Un exemple d'élément enveloppé peut être vu dans la figure 2.22. On pourrait faire valoir que ce n'est pas un bug, cependant, à des largeurs inférieures, la liste est de nouveau alignée en haut. Cela montre qu'ayant cette liste alignée en haut est la disposition désirée.

Nous donnerons des exemples sur la détection de ce genre de bugs dans la section 5.4.2.



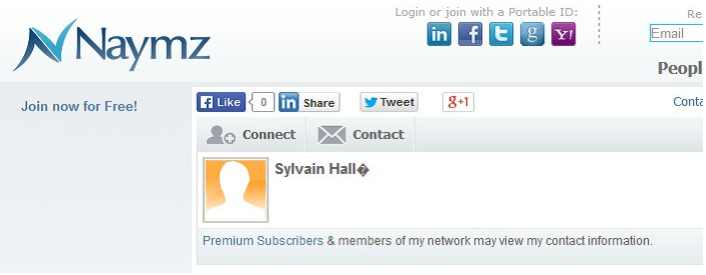
(a) Fenêtre large



(b) Fenêtre moyenne.

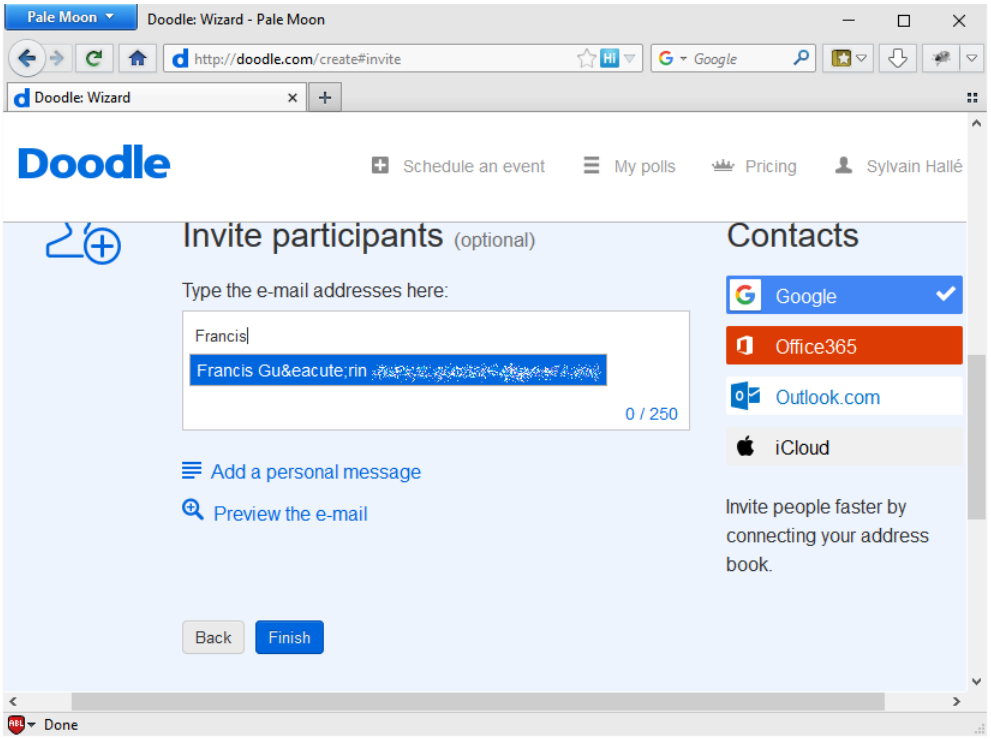
Figure 2.10 – Exemple d'éléments inaccessibles

(a) SpringerOpen

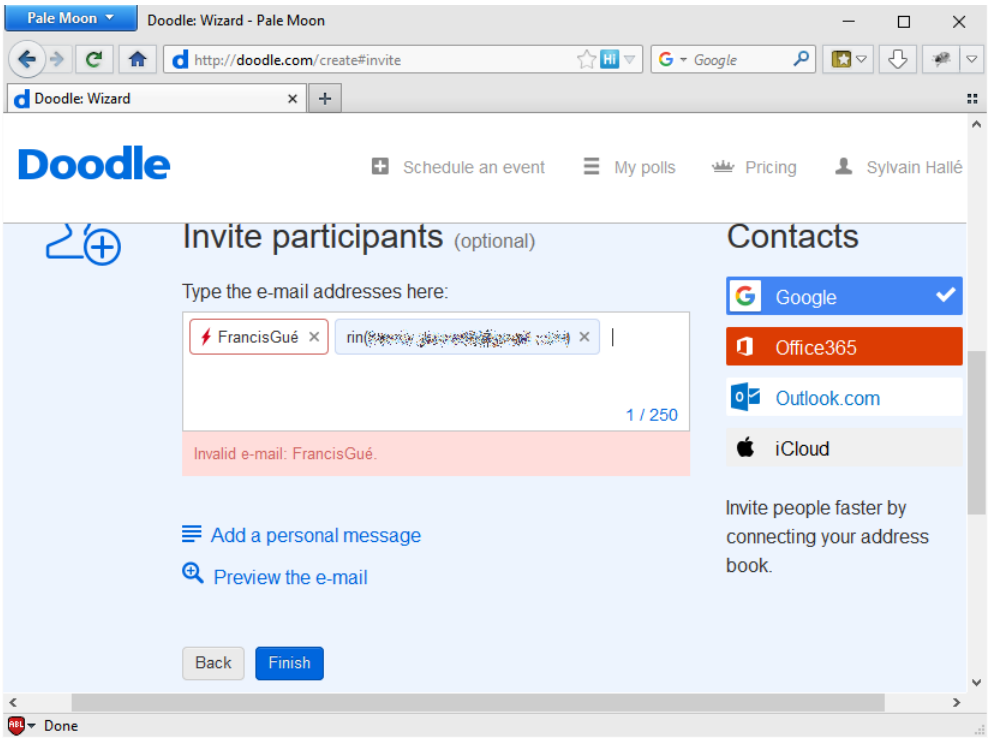


(b) Naymz

Figure 2.12 – Exemple de mojibake : (a) données UTF-8 affichées comme Cp1252 ; (b) données Cp1252 affichées comme UTF-8.



(a) Mojibake : adresse invalide



(b) Mojibake : nom coupé en deux partie

Figure 2.13 – Exemples de Mojibake dans Doodle.

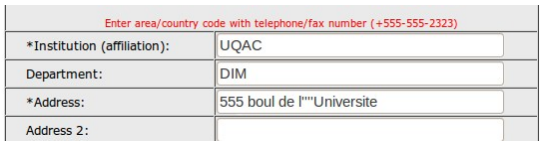


Figure 2.14 – Des données extraites de la base de données sont incorrectement échappées dans IEEE PDF eXpress : on constate la présence de plusieurs apostrophes.

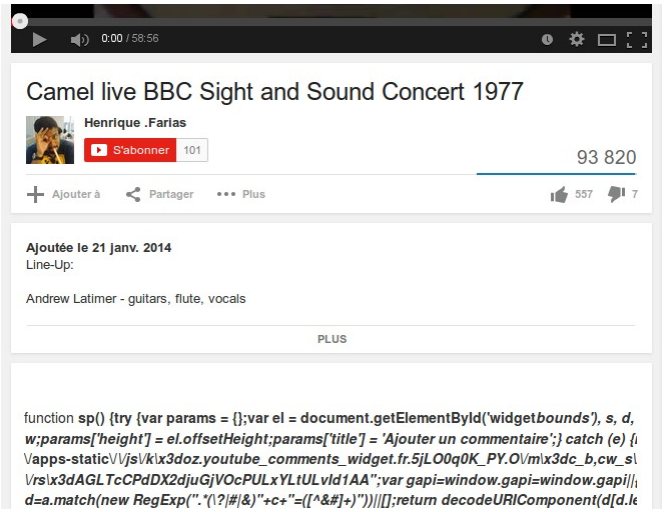


Figure 2.15 – Sur cette page YouTube, du code JavaScript est affiché au lieu d'être interprété.

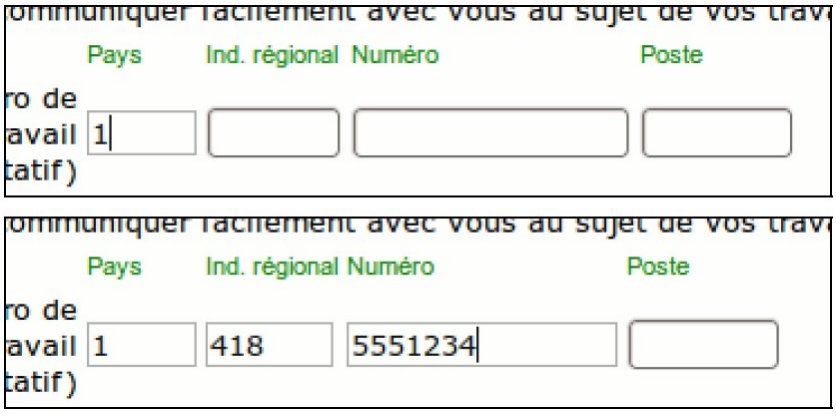


Figure 2.16 – Éléments mobiles :la position et la bordure de la zone du texte changent lors de la saisie du texte (NSERC)

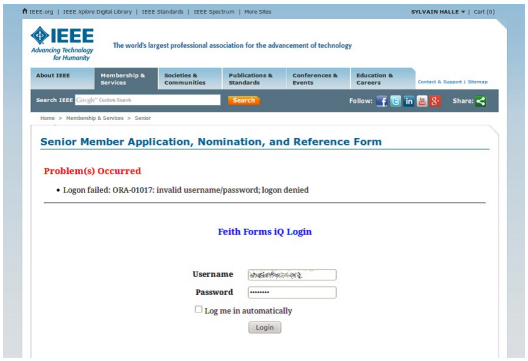


Figure 2.17 – Confusion connexion/déconnexion dans une page web : le contenu pour un utilisateur connecté (en haut de la page a droite) coexiste avec le formulaire de connexion réservé aux utilisateurs qui ne sont pas en principe connectés.

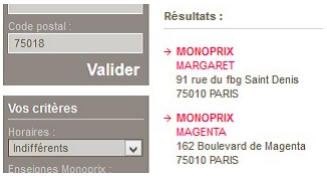


Figure 2.18 – Incohérences dans le résultat de la recherche.

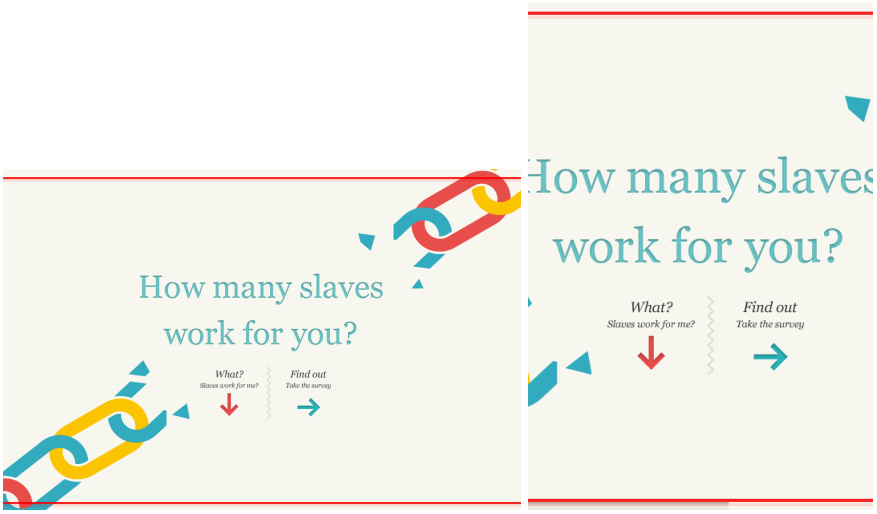
```
@media (max-width: 420px) {
  body {
    background-color: white ;
  }
}
@media (min-width: 421px) {
  body {
    background-color: blue ;
  }
}
```

Figure 2.19 – Code CSS avec des conditions sur la largeur de l'appareil. Il s'agit d'un exemple de *media queries*.

(a) Tous les boutons sont correctement dans la barre de menu.

(b) Le bouton « About » en surbrillance fait saillie en dehors de la barre de menu, son parent.

Figure 2.20 – Le bug de dépassement d'éléments sur le site `https://www.thelily.com/`.



(a) Le *div* est en surbrillance mais il n'y a pas de bug observable.

(b) En 440px, nous voyons le même *div* avec un bug observable où le texte déborde en dehors de la fenêtre.

Figure 2.21 – Le bug de dépassement du viewport sur le site `https://www.slaveryfootprint.org/`.

(a) La liste est en haut (top-aligned).

(b) À une largeur inférieure, l'élément « CGV » est poussé sur une ligne supplémentaire.

Figure 2.22 – Le bug de couverture d'éléments sur le site `https://www.anthedesign.fr/`.

CHAPITRE 3

ÉTAT DE L'ART

Il existe un certain nombre d'outils servant à tester les applications web. Dans ce chapitre, nous classons les approches existantes en quatre grandes familles qu'on nomme successivement : outils de tests, approches visuelles, approches déclaratives et outils RWD. Nous allons les exposer en donnant un peu plus de détails sur quelques outils relatifs à chaque famille d'approches. Enfin, nous citons leurs points faibles en montrant leurs limites dans certains cas.

3.1 OUTILS DE TEST

La détection de bugs d'interface peut d'abord être abordée comme un problème de test logiciel classique. Sous cet angle, il se veut une généralisation des fonctionnalités offertes par des logiciels d'analyse de sites web comme Google Analytics [1] ou Piwik [2]. Ces outils suivent les requêtes HTTP et fournissent un tableau de bord pour analyser des données de base : pays d'origine, durée d'une session, etc. Cependant, ces outils ne gèrent généralement pas Ajax, ne peuvent pas être utilisés comme outils de test (ils n'évaluent pas les conditions sur le contenu de la page par exemple) et ne signalent rien sur le contenu de la page, sur les actions de

1. http://www.google.com/analytics
2. http://www.piwik.org

```
WebDriver driver = new FirefoxDriver();
driver.get("http://...");
List<WebElement> items = driver.findElements(By.cssSelector("li"));
int left = -1;
for (WebElement item: items) {
  if (left == -1) {
    left = item.getLocation().getX();
    continue;
  }
  assert(left != item.getLocation().getX());
}
```

Figure 3.1 – Exemple de code pour Selenium WebDriver

l'utilisateur ou les requêtes Ajax.

Les logiciels de test en ligne tels que Capybara, Selenium WebDriver, Watij ou Sahi sont plus proches de nos objectifs. Ces outils fournissent différents langages pour décrire les tests et écrire des assertions sur l'application. Les scripts WebDriver sont écrits en Java ; Capybara a sa propre API [3] dans le langage Ruby ; Watij utilise WebSpec, [4] qui est une API définie par-dessus Java (3.5 donne un exemple de test webspec-watij) ; et Sahi utilise SahiScript, une extension de JavaScript. [5] Tous ces langages sont *impératifs* (c'est-à-dire procéduraux) et visent à piloter une application en effectuant des actions. La partie test se réduit à l'insertion d'instructions similaires à `assert` dans le code du script. A titre d'exemple, la figure 3.1 donne un extrait de code Java pour Selenium WebDriver pour vérifier que tous les éléments satisfaisant un sélecteur CSS sont à gauche.

3. http://makandracards.com/makandra/1422-capybara-the-missing-api
4. http://watij.com/webspec-api/
5. http://sahi.co.in

3.1.1 CAPYBARA

Capybara est un framework d'automatisation Web utilisé pour créer des tests fonctionnels qui simulent l'interaction des utilisateurs à une application. Il constitue une bibliothèque conçue pour être utilisée sur un pilote Web sous-jacent (underlying web driver) tels que *selenium-web driver*, *rack test driver* ou *capybara-webkit*. Il fournit un DSL (Domain Specific Language) servant à décrire les actions exécutées par ces pilotes Web [30] Une fois la page est chargée via le DSL et le pilote Web sous-jacent, Capybara essayera de localiser l'élément pertinent dans le DOM (Document Object Model) et exécutera une action, du genre : clics de boutons, de liens, etc. Son DSL déployé pour exprimer les actions à exécuter est très intuitif. la figure 3.2 montre quelques unes de ses commandes de base. Dans le but de trouver un élément[30], Capybara, par l'intermédiaire du DSL, cherchera dans le DOM, les attributs suivants :

— Champ de texte (`fill_in`) : id, name, élément d'étiquette associé (label).

— Cliquez sur un lien/bouton (`click_link/ click_button`) : id, titre, texte dans la balise, valeur.

— Case à cocher/bouton radio/liste déroulante (check/ uncheck/ choose/ select) : id, nom, élément d'étiquette associé (label).

— Télécharger le fichier (`attach_file`) : id, nom.

3.1.2 WATIJ

Watij ou Web Application Testing en Java est un projet de test, implémenté en Java, destiné à automatiser les tests fonctionnels des applications Web au-dessus de IE (Internet Explorer). Il est basé sur la conception de Watir [33] et possède une capacité de recherche d'éléments lui permettant de trouver, d'accéder et de contrôler facilement n'importe quel élément HTML à

```
visit('page_url') # navigate to page
click_link('id_of_link') # click link by id
click_link('link_text') # click link by link text
click_button('button_name') # fill text field
fill_in('First Name', :with => 'John') # choose radio button
choose('radio_button') # choose radio button
check('checkbox') # check in checkbox
uncheck('checkbox') # uncheck in checkbox
select('option', :from=>'select_box') # select from dropdown
attach_file('image', 'path_to_image') # upload file
```

Figure 3.2 – Commandes de base de Capybara [30]

travers le DOM en mobilisant l'interface COM. Il prend en charge, d'autre part, les expressions XPath pour trouver les éléments HTML sur une page et gère les pop-up (fenêtres contextuel du navigateur) en les interceptant et les rendant disponibles pour l'interaction. Watij peut détecter la fin du chargement d'une page en cours de chargement. Watij dispose d'un ensemble relativement riche d'API pour l'écriture de scripts de simulation. La connexion à un site Web exige à chaque fois, des informations tel que le nom de l'utilisateur et le mot de passe (page de connexion (login) de Yahoo figure 3.4). Les tâches manuelles demandées, dans ce cas, à l'utilisateur sont les suivantes :

1. Afficher une fenêtre de navigateur ;
2. Mettre l'URL correcte pour ouvrir cette page ;
3. Attendre que la page se charge et se stabilise ;
4. Taper le nom d'utilisateur dans le champ ID ;
5. Taper le mot de passe ;
6. Cliquer sur le bouton Connexion.

Le code approprié pour automatiser les tests fonctionnels de ces étapes, en utilisant Watij, est donné dans la figure 3.3.

```
IE ie = new IE();
ie.start("www.mail.yahoo.com");
ie.textField(SymbolFactory.id,"username").set("ouss");
ie.textField(SymbolFactory.id,"passwd").set("ber2017");
ie.button(SymbolFactory.id,".save").click()
```

Figure 3.3 – Exemple d'utilisation de Watij [71]

Figure 3.4 – Page de connexion (login) de Yahoo [71]

```
public class WebspecDemoTest {

  WebSpec spec;

  @Before
  public void setup(){
    spec = new WebSpec().safari();
  }

  @After
  public void tearDown(){
    spec.browser().close();
  }

  @Test
  public void testSearchWikipedia() throws Exception {
    spec.open("http://de.wikipedia.org/wiki/Wikipedia:Hauptseite");

    spec.find.input().with.id("searchInput").set.value("Softwaretest");
    spec.find.button().with.id("searchButton").click();
    assertEquals("Softwaretest", spec.find.h1().with.id("firstHeading").get.innerText());

    spec.find.a().with.innerText("Quality").click();
    assertEquals("Softwarequalitat", spec.find.h1().with.id("firstHeading").get.innerText()
  }

  @Test
  public void testOpenGoogle() throws Exception {
    spec.open("http://www.google.de/");
    spec.jquery("input[name='q']").set.value("Testing");
    spec.find.input().with.name("btnG").click();
    assertTrue(spec.find.div().with.id("res").find.a().get.innerText()
                .startsWith("Software␣testing"));
  }
}
```

Figure 3.5 – Exemple simple d'un test JUnit Watij WebSpec [7]

3.1.3 SAHI

Sahi est un outil open source d'automatisation et de test d'applications web. En tant qu'outil d'automatisation, Sahi offre la possibilité d'enregistrer et de lire des scripts. Il prend en charge Java et JavaScript. Même si SahiScript ressemble à JavaScript, il n'est pas exécuté comme le JavaScript normal dans le navigateur. L'idée de base du fonctionnement de Sahi est décrite ci-dessous :

les parties centrales de Sahi montrés dans la figure 3.7 incluent *le serveur proxy Sahi* et le moteur JavaScript. Les réponses HTML qui transitent par le proxy sont modifiées de telle sorte que JavaScript soit injecté au début et à la fin de la réponse. Cela permettra au navigateur d'enregistrer et de lire les scripts et de communiquer avec le proxy en cas de besoin. En plus de la gestion des demandes de pages du navigateur, Sahi gère également les commandes personnalisées liées à l'enregistrement, à la lecture, etc ; envoyées par celui-ci. Les fonctionnalités propres de Sahi, font de lui un bon support des fichiers de base de données, supportant JavaScript, AJAX ainsi que les API simples, Outre ses fonctionnalités normales, à l'égard de la prise en charge de « ant »(l'outil logiciel). Du fait que ses API ne dépendent pas beaucoup de la structure HTML, le contrôleur Sahi (IDE) peut être utilisé dans différents navigateurs. Sahi qui n'utilise pas XPath, renferme des API tels que : `_ near`, `_in`, etc, l'aidant à trouver un élément par rapport à un autre. « SahiScript » est fondé sur JavaScript. Il est analysé par Sahi et exécuté par le moteur JavaScript *rhino*. La figure 3.6 illustre un exemple de constructions (exemple d'écriture de conditions) dans Sahi. Elles sont identiques à JavaScript hormis le $ obligatoire utilisé dans les variables.

De plus, tous les outils mentionnés ci-dessus (à l'exception de Sahi) nécessitent un plugin spécifique au navigateur pour fonctionner, et ne supportent donc qu'une poignée de navigateurs, en général les « Big Five » (Firefox, Safari, IE, Opera et Chrome). Cependant, la part de marché

```
// Comparer des valeurs normales
if ($username == "PartnerUser"){
  _click(_link("Partner Login"));
}

// Comparer avec les attributs du navigateur exposés par Sahi
if (_getText(_div("page_type")) == "Partner Page"){
  _click(_link("Partner Login"));
}

// Comparaison avec les attributs du navigateur
// NON exposés par les fonctions intégrées de Sahi
// en utilisant _fetch
if (_fetch(_link(0).href) == "http://sahi.co.in/"){
  _click(_link("Partner Login"));
}

// Comparaison avec les attributs du navigateur
// NON exposés par les fonctions intégrées de Sahi
// en utilisant _condition
if (_condition(_link(0).href == "http://sahi.co.in/")){
  _click(_link("Partner Login"));
}
```

Figure 3.6 – Exemple de Sahi script [18]

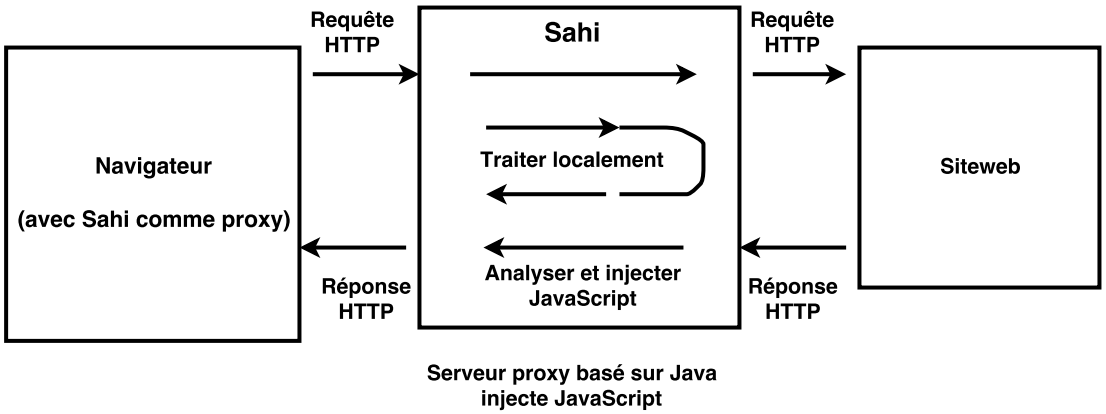


Figure 3.7 – Architecture de Sahi (figure tirée de [21])

des navigateurs autres que ceux-ci équivaut à 20%, et s'élève à 63% pour les appareils autres que les ordinateurs de bureau (tablettes, consoles et mobiles). [6]. En dehors de Sahi, ces outils de test n'atteignent pas plus des trois quarts du marché, et pour certains, seulement 25% pour les appareils autres que les ordinateurs de bureau. Par conséquent, l'affirmation selon laquelle « la plupart des utilisateurs » sont pris en charge par eux est à peu près non fondée.

3.2 APPROCHE VISUELLE

Pour déceler les défauts dans la mise en page, l'outil principal déployé par ce genre de techniques est généralement la vision par ordinateur. Ces dernières consistent entre autres, en la délimitation des bordures des éléments par la détection des contours et en la découverte des changements par le calcul de la différence entre deux captures d'écran dont les couleurs du texte des arrière-plans seront comparées pour repérer le texte illisible. Au lieu d'agir sur des informations spécifiques à la mise en page, telles que la taille et la position des éléments, ces techniques sont basées sur la comparaison des captures d'écran, pixel par pixel. Dans ce cas, les erreurs sous la forme d'une capture d'écran sont signalées et clairement marquées.

3.2.1 WEBSEE

Certains travaux ont également été réalisés sur l'utilisation des techniques d'analyse d'image pour identifier les problèmes de mise en page [69]. WebSee [58] est en particulier, un outil implémenté en Java qui utilise plusieurs bibliothèques tierces pour implémenter certains des algorithmes spécialisés. Il applique des techniques du domaine de la vision par ordinateur pour analyser la représentation visuelle des pages Web pour détecter automatiquement et localiser les échecs de présentation. WebSee identifie les échecs de présentation, puis détermine les

6. En date de février 2014, tel que récupéré sur StatCounter : `http://gs.statcounter.com`

éléments dans la source HTML de la page qui pourraient être responsables des échecs observés en comparant la représentation visuelle de la page Web rendue sous test et son apparence d'origine (oracle).

A cette fin, WebSee prend en entrée le code HTML/CSS de la page à analyser et un oracle (une image) du rendu attendu de la page. Un ensemble de différences entre la page rendue et l'image de référence est calculé, et ces différences sont ensuite regroupées en groupes susceptibles de représenter différents défauts sous-jacents dans la page. Une deuxième phase de traitement tente d'identifier les éléments HTML correspondant aux pixels de différence, qui sont ensuite ordonnés par une métrique de priorité et envoyés à l'utilisateur. La figure 3.8 montre les différentes phases de l'approche : La première entrée est la page web (P) qui devrait être analysé pour déterminer les défaillances de présentation. La forme de P est une URL qui pointe vers soit un emplacement sur le réseau ou d'un système de fichiers où tous les fichiers HTML, CSS, JavaScript, et les fichiers médias de P sont accessibles.

La deuxième entrée est l'oracle (O) qui spécifie les propriétés d'exactitude visuels de P. La forme de O est une image qui peut être soit une maquette ou une capture d'écran d'une version correcte de P.

La troisième entrée est un ensemble de régions spéciales (SR) définissant des zones de O qui contiendront du texte dynamique, des annonces, etc., qui définissent les régions dynamiques communes dans les applications web modernes. Les régions spéciales fournissent un mécanisme pour permettre aux développeurs de spécifier de telles régions qui devraient être traitées spécialement.

L'approche comporte trois phases. La première phase est la phase de la détection : elle compare les représentations visuelles de P et O pour détecter un ensemble de différences soit dans les régions spéciales ou dans les autres parties de la page web. L'ensemble des différences

identifié est regroupées ensuite en groupes qui sont susceptibles de représenter des défauts sous-jacents dans P. Pour ce faire, WebSee exploite Selenium WebDriver pour prendre des captures d'écran et il exploite la bibliothèque « pdiff » qui est une bibliothèque perceptuelle de différenciation d'image pour comparer les images et calculer les différences. L'algorithme de regroupement (clustering) : *DBSCAN* (Density Based Spatial Clustering of Applications with Noise) est utilisé pour regrouper les pixels de différences (dbscan est un algorithme implémenté dans la bibliothèque Apache Commons Math3).

La deuxième phase est la phase de localisation : elle analyse une carte de rendu de P pour identifier l'ensemble des éléments HTML qui définissent les pixels de chaque ensemble de différences en cluster. Pour cette étape, WebSee tire parti de la mise en œuvre de la bibliothèque R-tree [50] et Selenium WebDriver pour extraire des informations de délimitation de rectangles.

Enfin, la troisième phase est la phase du traitement de l'ensemble du résultat, elle priorise l'ensemble des éléments identifiés pour chaque grappe et fournit cela comme une sortie pour le développeur 3.9. La capacité de recherche de sous-image pour l'heuristique en cascade est fournie par OpenCV.

3.2.2 FLB (FIGHTING LAYOUT BUGS)

Une combinaison de l'injection CSS avec des techniques de différentiation d'image est présentée par Tamm dans une présentation *Tech Talk* de Google [69] en tant que moyen pour détecter quelle partie d'un site web est du texte et si elle chevauche d'autres éléments frontières. L'approche colore tout le texte sur une page Web en noir, puis en blanc, tout en prenant des captures d'écran entre les deux comme le montre la figure 3.10. Les parties de l'image qui sont du texte sont localisées en différant les deux images. Les lignes verticales et horizontales des

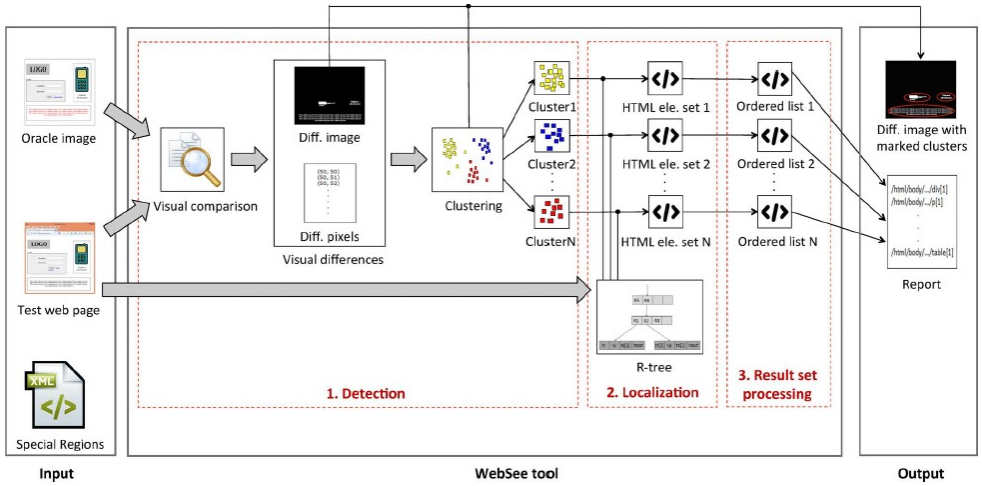


Figure 3.8 – Le fonctionnement général de l'outil d'analyse WebSee.

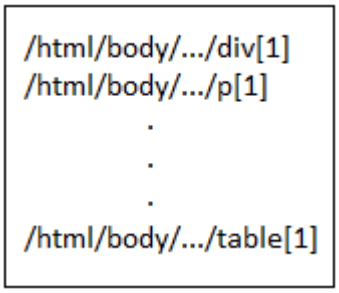


Figure 3.9 – Outputs de WebSee : éléments HTML défectueux

éléments dans l'image résultante sont découvertes via des techniques basées sur le traitement d'image. Une comparaison de ces lignes avec les éléments de texte découverts à l'étape précédente permet de déterminer les cas où le texte chevauche les bordures des éléments. La même technique permet aussi de déterminer si les éléments de texte ont le contraste approprié à leur arrière-plan.

Cependant, cette technique ne fonctionne que pour le texte et d'autres éléments qui peuvent facilement être identifiés en utilisant des techniques basées sur le traitement d'image pour la reconnaissance d'objets. Le champ d'application est petit, car il se concentre exclusivement sur le texte pour le chevauchement des éléments.

3.2.3 PHANTOMCSS

PhantomCSS [20] est un framework de test d'interface graphique open source qui possède la capacité de situer les changements d'une itération à une autre par des algorithmes différant sur deux images. D'autre part, il permet d'exclure certaines parties de l'interface graphique du test. Les pages Web susceptibles d'afficher des données non contrôlées par le développeur et des éléments tels que des annonces Web, des données utilisateur, des bannières animées, des images et du texte, trouvent dans ces caractéristiques un instrument bénéfique. Le développeur dans ce cas est obligé de spécifier manuellement les parties de l'interface graphique non concernées par les tests en nommant l'élément en question, ou le spécifiant ses coordonnées x et y.

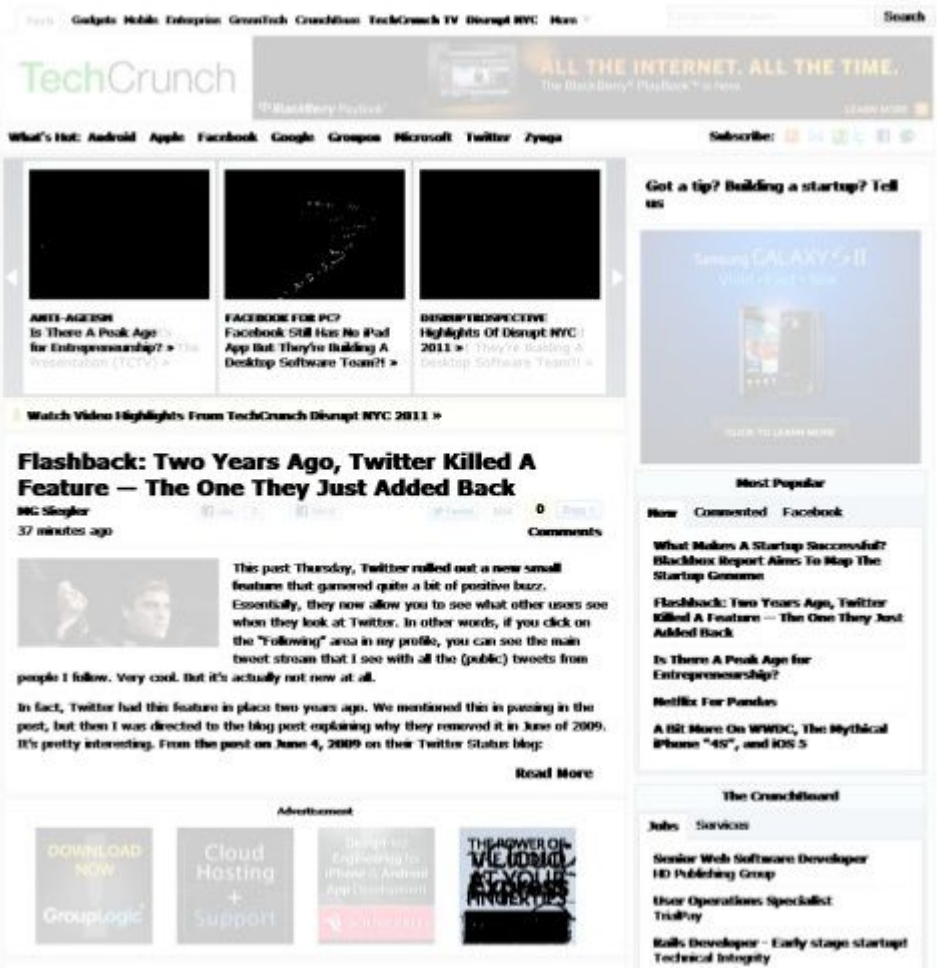


Figure 3.10 – Exemple de traitement d'image avec FLB (figure tirée de [63])

3.2.4 SIKULI

Un autre framework d'automatisation est Sikuli [39], qui identifie et manipule les contrôles de l'interface graphique dans une page web en utilisant la recherche par image (*sub-image searching*). Les captures d'écran constituent la base de cette approche visuelle pour la recherche et l'automatisation des interfaces utilisateurs. Elle permet aux utilisateurs :

— de prendre une capture d'écran d'un élément de l'interface graphique (comme un bouton de la barre d'outils, une icône ou une boîte de dialogue) ;

— d'interroger un système d'aide en faisant appel à la capture d'écran au lieu du nom de l'élément ;

— de fournir également une API de script visuel pour automatiser les interactions de l'interface graphique, par l'intermédiaire des modèles de capture d'écran pour diriger les événements de la souris et du clavier.

Dans l'exemple montré dans la figure 3.11, le bouton de fermeture doit effacer le contenu de la zone de texte ainsi que lui-même. Supposons que l'interface graphique soit déjà dans un état qui contient un « 5 », au début nous trouvons la zone de texte bleue sur l'écran et stockons la région correspondante qui a la plus grande similarité dans la zone bleue. Ensuite, après avoir cliqué sur le bouton de fermeture, deux assertNotExist sont utilisés pour vérifier la disparition dans la zone bleue [39].

3.2.5 APPLITOOLS

La segmentation pure de l'image des pages Web et la comparaison visuelle pixel par pixel sont à l'origine de l'outil commercialisé AppliTools Eyes [2], qui offre une interaction des scripts de test crées par l'utilisateur et son application. Dans cet outil, le serveur Web est

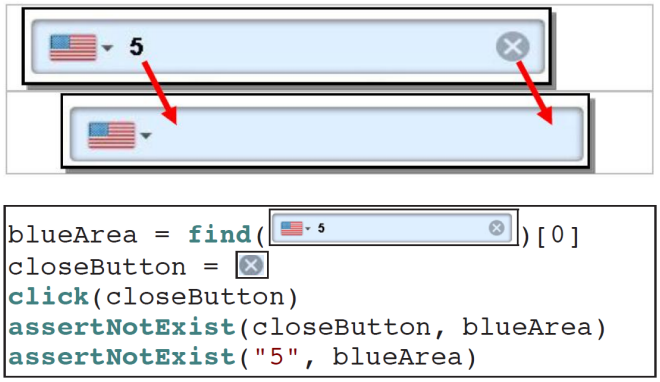


Figure 3.11 – Exemple d'utilisation de Sikuli (figure tirée de [39])

chargé de télécharger les captures d'écran en appliquant un algorithme de différence d'image entre lui et une capture d'écran précédente. La différence entre les deux images est traduite par AppliTools par une détection des régressions dans une mise en page GUI. Ces changements dans une interface Web sont exploités par le développeur pour actualiser l'image de base dans le cas où le changement était intentionnel.

Cependant, cette approche est orientée vers la détection de bugs de type statique de chevauchement ou de débordement des éléments dans un document, et actuellement ne supporte pas la vérification de modèles temporels à travers plusieurs instantanés de la même page. De plus, l'approche génère beaucoup de faux positifs si la page rendue contient du texte légèrement différent de l'image de référence. C'est le cas lorsque, par exemple, la fenêtre du navigateur a des dimensions différentes et que le texte s'écoule différemment (mais pas nécessairement incorrectement) par rapport à l'image. Pour résoudre ce problème, l'utilisateur doit définir manuellement, pour chaque oracle, ce que l'on appelle des *régions dynamiques* qui devraient être ignorées par le système lors de l'analyse de la page.

3.3 APPROCHE DÉCLARATIVE

Les techniques dans cette dernière famille fonctionnent directement sur des informations sur la mise en page. Elles peuvent obtenir des informations sur les éléments (position, largeur et hauteur) impliqués dans l'interface graphique, que ce soit par analyse d'image ou par « siphonnage » (*scraping*) de l'interface graphique. C'est d'ailleurs ce que ces techniques ont en commun. La manière de relier les différents éléments de mise en page les uns aux autres ainsi que les valeurs de leurs paramètres de mise en page sont indiqués par les entrées de ces approches, qui ne sont pas tant un script que des documents déclaratifs.

Les assertions opérées sur l'interface graphique peuvent par exemple être de la forme « l'élément 1 est-il situé à gauche de l'élément 2 ? » ou « l'élément 1 a-t-il une largeur inférieure à l'élément 2 ? ». Certaines de ces techniques ont des langages spécialisés décrivant des assertions comme celles-ci.

Les spécifications déclaratives des interfaces utilisateurs ont fait l'objet de beaucoup de recherches dans le passé. Les premières tentatives incluent le système MASTERMIND, qui utilise un langage de modélisation basé sur CORBA [67] ; d'autres approches incluent le modèle de mise en page d'Auckland [57], Adobe Adam et Eve [66] et les modèles de propriétés [57].

3.3.1 MASTERMIND

Dans MASTERMIND, le développeur de l'interface utilisateur doit créer des modèles de tâche (task model), d'application (domaine) et de présentation. Le modèle d'application est spécifié à l'aide du langage de définition d'interface CORBA (IDL). Le modèle de tâche présente les tâches de l'utilisateur final dans une structure hiérarchique et comporte les informations de

commande nécessaires pour contrôler l'interface utilisateur lors de l'exécution. Le modèle de présentation décrit la disposition de l'interface utilisateur, y compris les affichages statiques et dynamiques. Il permet la spécification des mises à jour automatiques de présentation lorsque les données d'application ou le contexte de présentation changent. En outre, il intègre des principes de conception graphique afin de donner un soutien complet au concepteur de dialogue.

3.3.2 AUCKLAND LAYOUT MODEL (ALM)

Le modèle de mise en page d'Auckland (ALM) est une technique implémenté pour « .NET », Java et Haiku, permettant de spécifier une mise en page 2D. Elle est utilisée pour organiser les contrôles dans une interface graphique. Le modèle permet la spécification de contraintes basées sur l'algèbre linéaire. Il calcule une disposition optimale en utilisant la programmation linéaire. Les égalités et les inégalités linéaires peuvent être spécifiées sur les tabulations horizontales et verticales, qui sont des lignes virtuelles formant une grille dans laquelle tous les éléments de l'interface graphique sont alignés [57]

L'exemple dans la figure 3.12 montre, la manière de disposer trois boutons en mobilisant trois zones. Les boutons ont déjà été ajoutés à la fenêtre, mais ils n'ont pas été arrangés. Leur emplacement et leur taille sont déterminés par la spécification ALM (figure 3.13). Quelque soit le redimensionnement de la fenêtre, les deux colonnes auront toujours la même largeur, et les deux lignes la même hauteur en raison de la linéarité des deux contraintes.

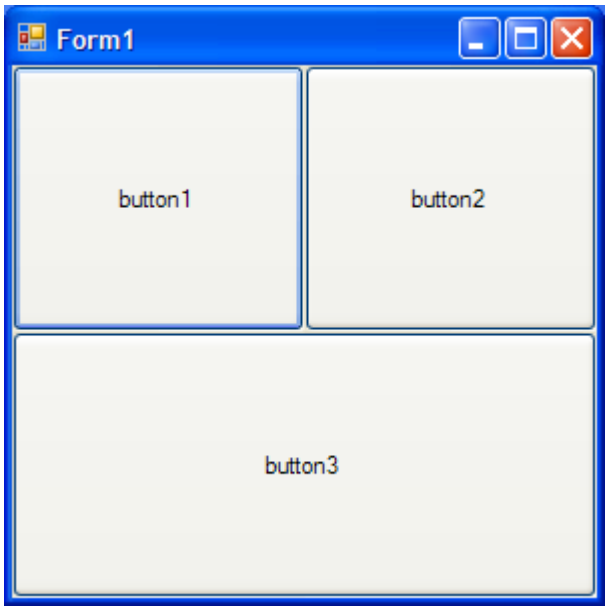


Figure 3.12 – Exemple avec Auckland (figure tirée de [1])

```
LayoutSpec ls = new LayoutSpec ();
XTab x1 = ls.AddXTab ();
YTab y1 = ls.AddYTab ();

ls.AddArea (ls.Left, ls.Top, x1, y1, bouton1);
ls.AddArea (x1, ls.Top, l.Right, y1, bouton2);
ls.AddArea (ls.Left, y1, ls.Right, ls.Bottom, bouton3);

// donne la même largeur aux deux colonnes et la même hauteur aux deux rangées
ls.AddConstraint (nouveau double [] {2, -1}, nouvelle variable [] {x1, ls.Right},
        OperatorType.EQ, 0);
ls.AddConstraint (nouveau double [] {2, -1}, nouvelle variable [] {y1, ls.Bottom},
        OperatorType.EQ, 0);
```

Figure 3.13 – Une simple spécification avec Auckland [1]

3.3.3 ADOBE ADAM ET EVE

ASL (Adobe Source Libraries) est un projet au sein du Adobe Software Technology Lab (STLab). C'est un ensemble de bibliothèques de composants logiciels, rendu disponible sous licence Open Source par Adobe Systems, permettant de définir des algorithmes sous forme déclarative. Les deux premières bibliothèques significatives dans ASL sont : la bibliothèque de modèle de propriétés (Adam) et la bibliothèque de modèle de mise en page (Eve) dont les composants permettent de modéliser l'apparence et le comportement d'une interface dans une application logicielle. Adam consiste en un solveur et un langage déclaratif pour décrire les contraintes et les relations sur une collection de valeurs qui sont généralement les paramètres d'une commande d'application (une fonction). Adam fournit la logique qui contrôle le comportement d'une interface Humaine (IH). Il est similaire dans son concept à une feuille de calcul ou à un gestionnaire de formulaires. Les valeurs sont définies et les valeurs dépendantes sont recalculées. Adam procure des fonctionnalités pour résoudre les valeurs interdépendantes, mais il ne constitue pas un système de contrainte général. Eve consiste en un solveur et un langage déclaratif pour la construction d'une IH. Le solveur de mise en page prend en compte une description riche des éléments de 14 interfaces pour obtenir une disposition de haute qualité rivalisant avec ce qui peut être réalisé avec le placement manuel. Une seule description suffit pour plusieurs plateformes et langages OS. Eve a été développée pour fonctionner avec Adam ; il peut cependant aussi, être utilisée seule. Adam et Eve peuvent être intégrées dans un certain nombre d'environnements. Ils ont la faculté de fonctionner ensemble, ou indépendamment, mais, ils doivent être combinés avec d'autres installations pour construire une application. Parmi les exemples typiques d'interfaces utilisateur effectuant la synthèse de paramètres de commande : la boîte de dialogue « Enregistrer sous » dans le cas d'enregistrement d'un fichier image (figure 3.14). Elle se compose d'un champ de texte pour entrer le nom du fichier, un menu de types de fichiers et des curseurs offrant deux possibilités

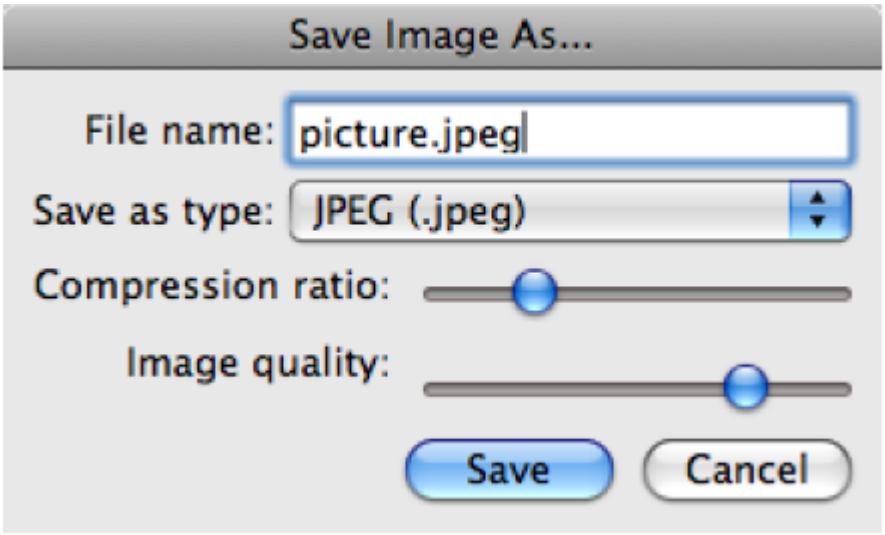


Figure 3.14 – Exemple d'une boîte de dialogue pour enregistrer un fichier image.

pour configurer la compression lors de l'enregistrement dans un format qui le prend en charge. Les valeurs des curseurs sont liées par une relation exprimant le compromis entre le taux de compression et la qualité de l'image.

La boîte de dialogue aide l'utilisateur à synthétiser les paramètres d'une commande d'enregistrement de fichier. La figure 3.15 montre une spécification pour cette tâche de synthèse de paramètres de commande écrite dans le langage déclaratif spécifique de « Adam ». Dans 3.15 les sections « interface », « output » et « invariant » déclarent les variables, ou les propriétés du modèle de propriétés. Les variables d'interface peuvent être mises à jour par un client du modèle de propriété, par exemple, à la suite de l'interaction d'un utilisateur avec un widget d'interface utilisateur. Les valeurs des variables de sortie (output) constituent le résultat de la synthèse des paramètres de commande. La valeur (booléenne) des variables invariantes indique si un ensemble de variables satisfait une condition donnée. La section « logique » dans 3.15, définit toujours les dépendances et les règles de calcul entre les variables. Le langage pour ces calculs est le langage d'expression ASL, qui peut faire des appels aux fonctions externes

```
sheet save image file {
 interface:
   file name : " ";
   file type : ''bmp'';
   compression ratio : 100;
   image quality : 100;
 logic:
  relate {
   compression ratio <== 100 - 4 x (100 - image quality);
   image quality <== 100 - (100 - compression ratio) / 4;
}
 output:
  result <== (file type == ''jpeg'') ?
   { type: file type, name: file name, ratio: compression ratio }:
   { type: file type, name: file name };
 invariant: check name <== file name 6= '' '';
}
```

Figure 3.15 – Une simple spécification déclarative dans le langage Adam du modèle de propriété pour le dialogue dans Figure3.14 [56]

enregistrées (C ++), et peut ainsi effectuer des actions arbitraires. La tâche du programmeur de l'application est de définir ces calculs appelés méthodes et qui sont exécutées est contrôlées par le modèle de propriétés.

Le langage « Eve », peut être utilisé pour spécifier la disposition et la qualité de présentation des éléments d'interface, ainsi que les liaisons entre les widgets dans l'interface utilisateur et les valeurs dans le modèle de propriété. La spécification de disposition pour le dialogue de la figure 3.14 apparaît dans la figure 3.16.

Nous allons cependant les rejeter, ainsi que beaucoup d'autres langages de balisage GUI (QML, XAML), car ils visent à *générer* des interfaces basées sur des contraintes et ne peuvent pas être utilisés comme des assertions à tester ; de plus, ils se concentrent principalement sur la résolution de contraintes linéaires relatives à la taille et au placement des éléments dans un formulaire.

```
layout save image file {
   view dialog(name: ''Save Image As...'',
   placement: place column,
   spacing: 6,
   child horizontal: align fill) {
   edit text(bind: @file name, name: ''File name:'');
   popup(bind: @file type, name: ''Save as type:'', items: [
     { name: ''Windows bitmap (.bmp)'', value: ''bmp'' },
     { name: ''JPEG (.jpeg)'', value: ''jpeg'' }
   ]);
   row() {
     column(child horizontal: align right) {
      label(name: ''Compression ratio:'');
      label(name: ''Image quality:'');
     }
     column(horizontal: align fill, child horizontal: align fill) {
       slider(bind: @compression ratio,
        format: {first: 1, last: 100, interval: 1} );
       slider(bind: @image quality,
        format: {first: 1, last: 100, interval: 1} );
     }
}
   row(horizontal: align right) {
     button(name: ''Save'', action: @ok, bind: @result,
       default: true);
     button(name: ''Cancel'', action: @cancel);
}}}
```

Figure 3.16 – Une simple spécification avec Eve [56]

Pour montrer pourquoi les langages de génération de GUI ne sont pas appropriés, considérons l'exemple simple de CSS. Comme nous l'avons vu plus haut, CSS exprime à quoi les éléments *devraient* ressembler (suggestions), mais pas comment ils vont l'être. Considérez les déclarations de style suivantes, dans le cas où un élément `div2` est imbriqué dans `div1` :

```
#div1 { ...width: 200px; }
#div2 { ...width: 300px; }
```

Il n'est pas possible avec CSS d'affirmer que le contenu de `div1` ne devrait jamais être plus large que sa propre largeur déclarée. Si tel est le cas, le navigateur peut soit développer la boîte de l'élément (sans tenir compte de sa déclaration), soit afficher les éléments en dehors de son conteneur (un problème dont nous avons beaucoup parlé dans la section 2.2). La seule solution consiste pour le concepteur du document CSS à s'assurer que la largeur déclarée de tous les éléments qui seront contenus dans `div1` est inférieure à 200 pixels. En revanche, une propriété déclarative pourrait facilement imposer à l'exécution que la largeur de `div1` est toujours de 200 pixels.

On pourrait argumenter que cela peut aussi être vérifié en analysant statiquement le document HTML avec le fichier CSS (bien qu'aucun travail apparenté ne semble répondre à cette question). Il y a cependant des cas où ce n'est même pas possible. Si nous supprimons la déclaration de `div2` dans l'exemple ci-dessus, mais que `div2` contient une image de 300 pixels de large, il est impossible de découvrir que la propriété désirée n'est pas satisfaite en regardant simplement le CSS.

En guise d'exemple final, considérons le code suivant, qui stipule que les éléments de certains menus doivent être placés verticalement et que leur taille est fixe :

```
#menu li {
  float: left;
```

```
  width: 200px;
  height: 50px;
}
```

Rien ne permet au concepteur de spécifier que tous les éléments doivent toujours être sur la même ligne (c.-à-d. Avoir la même position `top`) (même si la valeur de `top` est définie). Ce ne sera pas le cas si la fenêtre contenant est redimensionnée suffisamment petite pour que les éléments puissent flotter en dessous. Dans un tel cas, il faut effectivement rendre la page dans un navigateur, avec ses dimensions réelles, pour découvrir la présence du problème.

3.3.4 GALEN FRAMEWORK

Galen Framework [10] est une bibliothèque de tests d'interfaces graphiques pour tester les mises en page d'applications web. Le comportement mutuel entre les différents éléments d'une interface graphique et le type de paramètres de mise en page valides sont décrits par des descriptions de mise en page crées par l'utilisateur. La vérification de la validité de l'emplacement et la taille des éléments est réalisée par un test oracle en utilisant ces descriptions.

Pour déclarer divers aspects de la position et de la taille d'un élément par rapport à d'autres éléments ou en valeurs absolues, le développeur de test utilise un langage de description de test personnalisé fourni par le framework Galen. Galen est basé sur un outil d'automatisation de navigateur web appelé Selenium, qui permet de créer des scripts automatisés simulant les entrées de l'utilisateur. Il permet également de spécifier des plages de valeurs acceptables. La figure 3.17 montre un exemple de ce à quoi ressemble une spécification Galen [9].

Le test de régression des mises en page d'applications web est entièrement automatiser par le framework Galen en combinant ces deux technologies. Des informations précises sur les

valeurs valides pour les éléments de l'interface étant nécessaires au fichier de spécification requis par le framework Galen, cette opération peut être coûteuse et rendre la modification d'une interface graphique plus difficile.

Un regard sur le langage de spécification de Galen indique que le travail requis par le développeur de test est similaire à la spécification de l'interface graphique elle-même. Les spécifications de Galen et la conception GUI elle-même sont effectuées avec à peu près la même fréquence. L'inconvénient est la flexibilité réduite et l'augmentation des coûts. Cependant l'approche permet une grande précision lors de la réalisation des tests de régression et la vérification de la fonctionnalité de la conception sur différentes tailles d'écran.

3.3.5 ITARRAY AUTOMOTION

Le framework Automotion d'ITArray [70] est similaire au framework Galen. La différence par rapport à beaucoup d'autres frameworks est que la bibliothèque d'assertions permet réellement d'affirmer si la position et l'alignement des éléments sont liés les uns aux autres avec des appels tels que : `is ElementInside (otherElement)` ou `areElementsAligned (List)`. La façon dont ces tests sont spécifiés est encore très verbeuse. Un exemple de script Automotion est illustré à la Figure 3.18.

3.4 OUTILS RWD

Il est toujours judicieux de tester la conception d'un site web sur divers appareils. Mais cette méthode classique prend beaucoup de temps et en raison de la variété des dispositifs disponibles aujourd'hui, le développeur peut ne pas avoir accès à tous ces dispositifs. Plusieurs outils ont été développés pour fournir le service permettant d'afficher une page dans une

```
#Declaring objects with css and xpath locators
@objects
  header               css     div.fusion-secondary-header
  sw-logo              css     .fusion-logo-link

  navigation-links-*   xpath   //ul[@id='menu-main']/
  li[not((contains(@id,'mobile-menu-item')))]
  nav-menu             css     div.fusion-secondary-main-menu
  follow-us-box        xpath   //div[contains(@class,'fusion-alert')][1]

#Home Page Tag
= Home Page =
  #This is for Desktop
   @on desktop
       #Header properties
       header:
           inside screen 0px top
           inside screen 0px left
           inside screen 0px right

       #Logo properties
       sw-logo:
           #31 px below the header
           below header 31px
           #Image comparison with %2 precision ratio
           image file sw-logo.png, error %5

       #Each navigation links must be alligned horizontally to each other
       #itemName -> Current item and nextItem -> Next item
       @forEach [navigation-links-*] as itemName, next as nextItem
           ${itemName}:
               aligned horizontally all ${nextItem}

       #Navigation menu must be 20px above to the follow us allert box
       nav-menu:
           above follow-us-box 20px
```

Figure 3.17 – Exemple d'une spécification Galen [9]

```
boolean result = uiValidator.init("Scenario name")
.findElement({rootElement}, "Name of élément we validate")
.sameOffsetLeftAs({élément}, "Panel 1")
.sameOffsetLeftAs({élément}, "Button 1")
.sameOffsetRightAs({élément}, "Button 2")
.sameOffsetRightAs({élément}, "Button 3")
.withCssValue("border", "2px", "solid", "#FBDCDC")
.withCssValue("border-.radius", "4px")
.withoutCssValue("color", "#FFFFFF")
.sameSizeAs({list_éléments})
.insideOf({élément}, "Container")
.notOverlapWith({élément}, "Other élément")
.withTopElement({élément}, 10, 15)
.changeMetricsUnitsTo(ResponsiveUIValidator.Units.PERCENT)
.widthBetween(50, 55)
.heightBetween(90, 95)
.drawMap()
.validate();
```

Figure 3.18 – Exemple d'un script Automotion [70]

fenêtre personnalisée de tailles variables à l'aide d'un navigateur Web. Cependant, ces outils ne fournissent aucune information autre que le rendu des pages sur différentes tailles ce qui oblige le développeur à évaluer la réactivité (responsiveness) du site web manuellement. Mentionnons ici quelques outils spécifiques à la détection de problèmes liés au RWD.

Avec des fonctionnalités de recherche intelligente et de révision rapide, WebSiteResponsiveTest [34] prend en charge tous les principaux navigateurs pour fournir l'aperçu exact du site web sur un périphérique spécifique. Il faut entrer l'URL d'une page Web pour évaluer rapidement la réactivité. En fournissant des résultats précis en quelques secondes, l'outil économise beaucoup de temps. La convivialité et la compatibilité avec les navigateurs sont d'autres fonctionnalités qui le rendent attractif par rapport aux autres outils disponibles.

Respondr [22] permet de vérifier la réactivité en entrant l'URL d'un site web. En outre, l'appareil pour lequel le site web ou la page Web est testé peut également être choisi dans la

liste donnée. Une fois les sélections effectuées, un simple clic sur « Go » est nécessaire pour recevoir une analyse complète du site ou de la page web en donnant l'affichage sur l'appareil sélectionné. La page peut être facilement prévisualisée à une largeur appropriée.

Screenfly [27] est un outil de test de compatibilité multi-périphérique qui permet de prévisualiser les pages Web telles qu'elles apparaissent sur différents appareils. Les plus populaires comprennent les tablettes et autres appareils intelligents tels que Galaxy Tab, Apple iPad, Motorola Xoom. En outre, il prend en charge différentes tailles d'écran et résolutions. Le site détecte automatiquement si l'URL saisie comporte un site mobile et vous redirige vers celui-ci. Pour basculer entre les résolutions d'écran, tout ce qu'il faut faire est de cliquer sur l'icône du type d'appareil ou choisir l'appareil qui a la résolution d'écran la plus proche.

Responsive Web Design Bookmarklet [26] affiche n'importe quelle page Web dans plusieurs tailles d'écran pour la prévisualisation, simulant la fenêtre d'affichage de différents périphériques. Il s'agit d'un outil de conception Web rapide qui peut être consulté à partir d'un ordinateur de bureau pour tester la réactivité de tout site web. Il suffit de faire glisser le bookmarklet au-dessus de la barre des signets (*bookmarks*) pour obtenir une barre d'outils en haut avec des boutons pour différentes résolutions d'écran. Il ne reste plus qu'à choisir l'aperçu de la page en cours sur différentes largeurs d'écran de tablettes et de *smartphones*.

ViewPortResizer [31] est également un bookmarklet de navigateur qui peut être utilisé avec n'importe quel navigateur Web moderne. Un outil de navigation convivial, ViewPortResizer est entièrement configurable. Il permet la sélection d'une plage initiale de tailles de résolution d'écran et la construction d'un bookmarklet personnalisé.

Responsinator [23] aide les propriétaires de sites à avoir une idée de la façon dont leur site fonctionnera sur les appareils les plus populaires. Juste en tapant l'URL du site, le site s'affichera rapidement sur des écrans de différent(e)s tailles.

Le processus de ResponsivePX [24] consiste à entrer l'URL du site et utiliser des boutons pour ajuster la largeur et la hauteur de la fenêtre d'affichage afin de trouver la largeur exacte du point d'arrêt en pixels. Extrêmement simple à utiliser et à améliorer les fonctionnalités, cet outil de conception Web aide les concepteurs à créer des sites utilisables et réactifs.

Froont [8] rend les tests de conception Web réactifs accessibles sans nécessités de compétences de codage. Les conceptions peuvent être crées dans le navigateur avec cet outil. En testant chaque URL spécifiquement, il teste les conceptions sur de vrais appareils tout de suite.

De son côté, ReDeCheck [74] [73] est un outil de test de conception web réactif. Il est inspiré du graphe d'alignement utilisé dans X-PERT, un concept proposé et développé par Choudhary *et al.* [42]. ReDeCheck se concentre spécifiquement sur les bugs de mise en page causés par des conceptions réactives ; il utilise un graphique de mise en page adaptable (RLG), qui tient compte de l'alignement des éléments de la page Web, des changements de visibilité et d'autres aspects de la page lorsque la largeur de la fenêtre varie. En tant que tel, ReDeCheck peut seulement vérifier un ensemble fixe de problèmes de mise en page prédéfinis, et ne fournit pas un langage à usage général pour exprimer des assertions.

3.5 DISCUSSION SUR LES APPROCHES EXISANTES

Nous allons maintenant discuter quelques limites des approches précédemment présentées :

3.5.1 INCONVÉNIENTS DE L'APPROCHE DÉCLARATIVE

Le développeur de test doit se soumettre aux exigences des descriptions/scripts de test assez verbeux en décrivant les règles de son interface graphique (comment les éléments sont-ils placés les uns par rapport aux autres, comment ils devraient se comporter quand la taille de

la fenêtre change, etc.) de façon plus détaillée, au point où le script de test devient presque aussi descriptif que le code de l'interface graphique qu'il teste. A l'instar des scripts de test, ce problème nécessitera des mises à jour à peu près au même rythme que les modifications apportées à la conception de l'interface graphique.

3.5.2 LIMITES DES TECHNIQUES VISUELLES

Impossible de comparer des images de différentes tailles d'écran Le fait d'assurer l'égalité et la validité entre une image de référence (oracle) et l'image à tester par la différence de couleur de pixel, implique que celles-ci ne peuvent pas être des captures d'écran de tailles d'écran différentes ; ce qui exclut l'utilisation de l'image de référence à partir d'une taille d'écran pour vérifier la validité de la mise en page sur une taille d'écran différente. La validité de la mise en page d'une application censée fonctionner sur de nombreuses tailles d'écran différentes par ces techniques exige d'établir des références pour toutes les tailles d'écran.

Ne fonctionne pas bien avec les données dynamiques Ces techniques basées sur l'image et travaillant sur l'information de pixel au lieu du contenu ne donnent des résultats satisfaisants qu'avec des données statiques (parce qu'on utilise une image de référence pour faire la comparaison). Par conséquent, elles posent un problème majeur. Chaque exécution d'une application correspond souvent à un affichage du contenu différent puisque ces données sont généralement récupérées sur un serveur Web ou crées par l'utilisateur —ce qui diminue fortement l'utilité de ces techniques.

Signalent beaucoup de faux positifs Une différence dans les valeurs de pixels de deux captures d'écran consécutives peut entre constatée, notamment si ces captures sont prises sur des machines différentes avec des configurations différentes, d'où l'inconvénient des

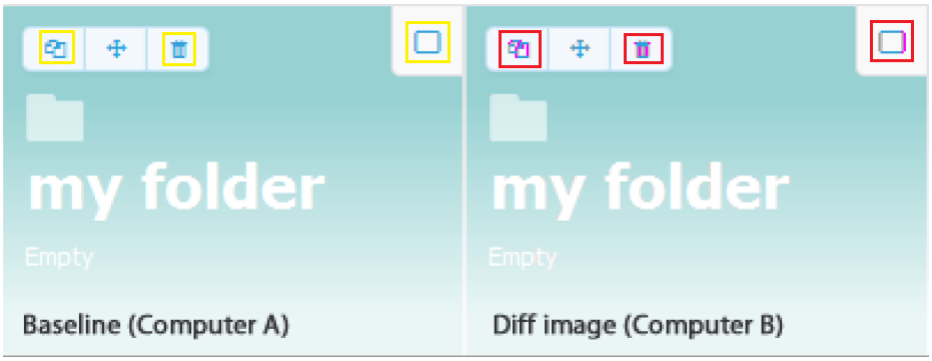


Figure 3.19 – Exemple de faux positif avec PhantomCSS

techniques basées sur les différences d'images qui sont sujettes à la déclaration de faux positifs, comme indiqué sur le site web du référentiel PhantomCSS [20]. Ces petites différences de pixels sont parfois faussement signalées par l'algorithme de test comme des erreurs de mise en page ; cependant, elles ne le sont pas réellement. La figure 3.19 montre un exemple de ce cas où nous pouvons observer de petites zones de pixels violets (nous avons marquées ces petites zones avec des rectangles rouges), ces images ressemblent aux images de base qui sont entourées de rectangles jaunes. En réalité, ce sont pas des erreurs, mais des faux négatifs rapportés en raison de petites différences de pixels dans des rendus différents (cela pourrait être dû à l'utilisation de différents navigateurs).

En conclusion, le tableau 3.1 donne les grandes lignes de différences entre trois outils majeurs des approches existantes :

Sahi	Selinium	Websee
Pas de plugin (solution proxy web)	Plugin spécifique pour chaque navigateur	Configuration spécifique et bibliothèques tierces pour implémenter certains des algorithmes spécialisés.
Tests fonctionels des éléments de la page basé sur le trafique HTTP enregistré au niveau du proxy (playback)	Tests fontionels	Tests de mise en page en se basant sur une version de reference (juste pour des pages statiques)

Tableau 3.1 – Limites et différences entre trois outils majeurs des approches existantes

CHAPITRE 4

DÉTECTION DE BUGS D'INTERFACE

Comme nous avons pu le voir, une analyse statique du contenu HTML et des déclarations CSS d'une page web n'est pas suffisante pour détecter les bugs d'interface cités dans le deuxième chapitre de cette thèse, car CSS n'est pas un langage qui peut exprimer des propriétés normatives pour la mise en page d'un document. Les déclarations CSS sont juste un ensemble d'instructions traitées par un moteur de rendu. Donner à CSS une expressivité prescriptive impliquerait la possibilité de spécifier à quoi un élément ne doit pas ressembler, ou qu'une déclaration de style particulier ne peut pas être remplacée par d'autres constructions –ce que le CSS ne fournit pas. En outre, certains des bugs décrits précédemment impliquent la comparaison du contenu d'un document à plusieurs moments dans le temps ; une chose que CSS n'est pas évidemment conçu pour faire.

Par conséquent, afin d'exprimer des propriétés normatives pour le contenu et la mise en page d'une page web, un langage complémentaire à CSS est nécessaire. Ce langage doit permettre aux utilisateurs d'écrire des propriétés déclaratives sur les styles des éléments, et de traiter des événements, quelles que soient les déclarations CSS ou le code côté client qui ont pu être déclarées.

Pour combler cette nécessité, dans ce chapitre nous présentons Cornipickle, un langage

déclaratif permettant d'exprimer des propriétés à propos du document et des propriétés CSS d'une page. On discutera également d'une implémentation d'un algorithme permettant de vérifier automatiquement si des énoncés du langage sont vrais pour un site web particulier.

4.1 UN INTERPRÉTEUR POUR LES PROPRIÉTÉS CORNIPICKLE

Nous décrivons maintenant la mise en œuvre de l'interpréteur pour l'évaluation automatisée des spécifications Cornipickle sur les applications web. Cette implémentation est composée d'environ 7 000 lignes de code Java et JavaScript et est disponible sous licence publique générale GNU. [1] Une vidéo de l'outil en action sur des exemples simples peut également être consultée en ligne. [2]

4.1.1 OBJECTIFS DE CONCEPTION

Outre la fonctionnalité principale à mettre en œuvre, le développement de l'outil a été motivé par un certain nombre d'objectifs de conception importants.

Pas de plugins spécifiques au navigateur

Tout d'abord, la solution doit fonctionner sur autant de combinaisons de navigateurs et de systèmes d'exploitation que possible. Ceci exclut explicitement les plugins spécifiques au navigateur (ou limités par le navigateur), tels que les plugins Chrome, les plugins Firefox ou l'utilisation d'outils tels que Selenium WebDriver. Pour la même raison, la solution ne doit pas reposer sur la présence de frameworks JavaScript (jQuery, Prototype, etc.) et être

1. `https://github.com/liflab/cornipickle`
2. `http://youtube/90YitGRRx2w`

autonome. Cela implique que notre outil peut fonctionner dans des combinaisons inhabituelles (navigateurs/systèmes d'exploitation), comme BoatBrowser sur un téléphone Android, ou Qupzilla sous Haiku OS.

Collecte d'informations côté client

Deuxièmement, l'évaluation des spécifications doit être faite en fonction des informations recueillies sur le client ; cela écarte la possibilité d'effectuer une évaluation statique de HTML et CSS du côté serveur. Ceci est obligatoire pour plusieurs raisons. Il faut tenir compte du fait que le standard CSS n'est pas interprété de la même manière par tous les navigateurs. Par exemple, CSS stipule que la largeur d'un élément n'inclut pas le remplissage, mais certaines versions d'Internet Explorer incluent le remplissage dans la largeur et rendent le même élément avec des dimensions différentes.

Dans une large mesure, la vérification des contraintes en examinant uniquement le code HTML + CSS impliquerait d'émuler le moteur de rendu de chaque navigateur, complété par ses « bizarreries » spécifiques, pour parvenir à un verdict fidèle.

En plus des problèmes susmentionnés, toutes les applications que nous avons étudiées contiennent du code côté client qui peut modifier la disposition d'une page après que le moteur de mise en page a traité les déclarations statiques trouvées dans le document HTML initial et les fichiers CSS traités au moment du chargement. Ce code, programmé pour être exécuté lors du chargement de la page, remplace complètement les déclarations de style que les fichiers CSS d'origine peuvent initialement définir. Par conséquent, dans tous les cas, il ne suffit pas d'analyser l'ensemble des fichiers HTML et CSS définis par l'application, car tout ce contenu peut être modifié à la volée grâce à des interactions avec l'utilisateur une fois la page chargée.

Pas d'interprétation côté client

Troisièmement, l'interprétation des spécifications de Cornipickle ne devrait pas être faite du côté des clients. Ceci est fait de manière à ne pas imposer une charge de calcul excessive dans le navigateur, et permet l'utilisation d'un autre langage que JavaScript pour l'implémentation de cette fonctionnalité. Plus important encore, il permet de gérer les propriétés comportementales impliquant plus d'un instantané de page par l'outil. En utilisant du code strictement client, un problème survient lorsqu'un rechargement de page complet se produit, car cela réinitialise l'état de tout objet JavaScript associé à cette page. Étant donné que les spécifications comportementales requièrent la sauvegarde des informations du passé, certains moyens de préserver ces informations dans le client, à travers les rechargements de pages, doivent être conçus. HTML5 fournit des fonctionnalités de stockage, mais leur utilisation limiterait la prise en charge des navigateurs (par exemple, uniquement pour Opera 11.5, Safari 4 et IE 9 et plus récents [3]).

Interprétation de l'exécution

Enfin, il devrait être possible pour un utilisateur d'ajouter, de supprimer ou de modifier les spécifications évaluées par l'outil. Cela pose un défi en raison de la construction spéciale `We say that`, qui permet d'ajouter de nouvelles constructions grammaticales dans le langage de base. Ceci est différent des définitions de fonction ou de prédicat habituelles disponibles dans la plupart des langages, où la syntaxe des appels de fonctions est fixe et seuls de nouveaux identificateurs de *fonctions* peuvent être ajoutés au moment de l'analyse. Cela a nécessité le développement d'un analyseur BNF, appelé Bullwinkle [4] qui peut accepter de nouvelles règles

3. `http://www.html5rocks.com/en/features/storage`
4. `https://github.com/sylvainhalle/Bullwinkle`

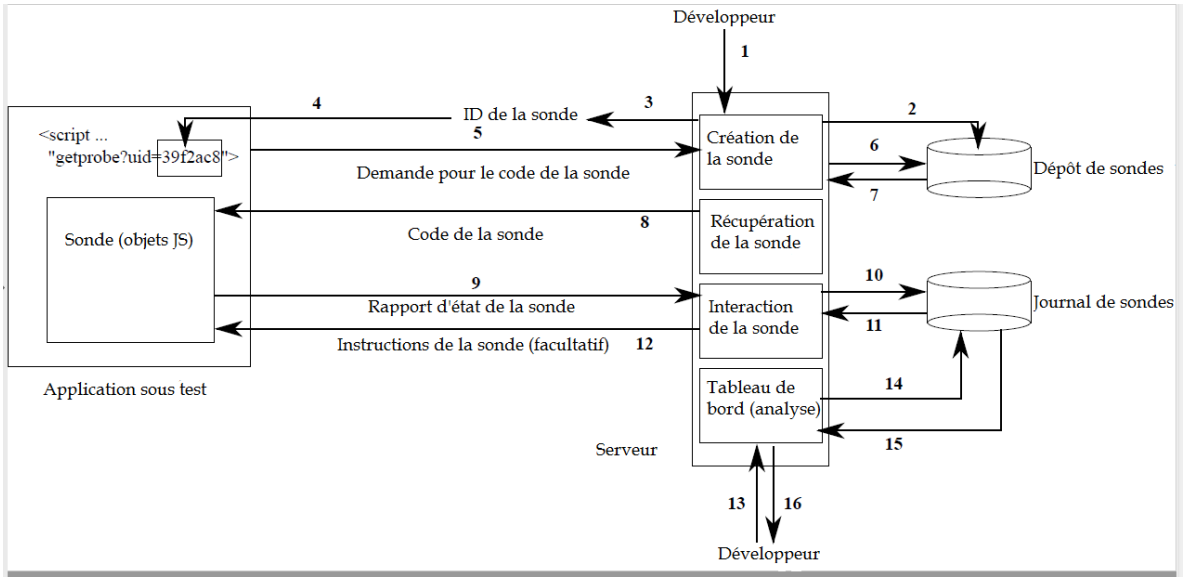


Figure 4.1 – Architecture et interactions de Cornipickle

d'analyse à l'exécution —contrairement à la plupart des autres analyseurs qui nécessitent une étape de compilation chaque fois que la grammaire change.

4.1.2 ARCHITECTURE ET SCÉNARIO D'UTILISATION TYPIQUE

La combinaison de toutes les exigences dans [51] impose plus ou moins une architecture pour l'outil Cornipickle, où le code côté serveur prend en charge la collecte et l'évaluation des spécifications (caractéristiques), tandis que le code côté client agit uniquement comme une « sonde ». L'interrogation des informations pertinentes sur l'état actuel de la page est relayée vers le serveur pour un traitement ultérieur. Cette interaction client-serveur a un avantage, cependant : le code côté client peut être relativement léger et sans état (être remis à zéro à chaque fois que la page se recharge), comme tout traitement à état qui peut être fait par le serveur.

La figure 4.1 montre les interactions avec l'outil Cornipickle. Un développeur écrit d'abord un jeu d'états déclaratifs (1), qui sont stockés dans la mémoire de Cornipickle (2). En donnant

```
{
  "tagname": "window",
  "width": 956,
  "height": 165,
  "children": [
    {
      "tagname": "p",
      "id": "go",
      "width": 90,
      "children": [
        {
          "tagname": "CDATA",
          "text": "Hello␣World!"
        }
      ]
    }
  ]
}
```

Figure 4.2 – Une page simple sérialisée en JSON

un identifiant unique à ce jeu d'états, qui peut être appelé dans le code JavaScript à être inséré dans l'application afin que la sonde peut être chargée dans chaque page (3-4) ; cette addition est générique et ne diffère que dans la chaîne d'identification. Quand une page de l'application doit être chargée (5), Cornipickle crée dynamiquement la sonde JavaScript correspondant à l'ensemble d'assertions pour les évaluer et les envoyer au client (6-8). Cette sonde est conçue pour signaler un instantané des données DOM et CSS pertinents sur chaque événement déclenché par l'utilisateur. Quand un tel événement se produit, la sonde recueille toutes les informations pertinentes sur le contenu de la page (figure 4.2) et les relaie au serveur Cornipickle (9), qui les enregistre dans un journal (10-11).

En option, des informations sur l'état actuel des assertions en cours d'évaluation (vrai / faux) peuvent être relayées à la sonde (12). Un tableau de bord d'analyse peut récupérer le journal enregistré qui peut être consulté par le développeur, pour interroger l'état de toutes les

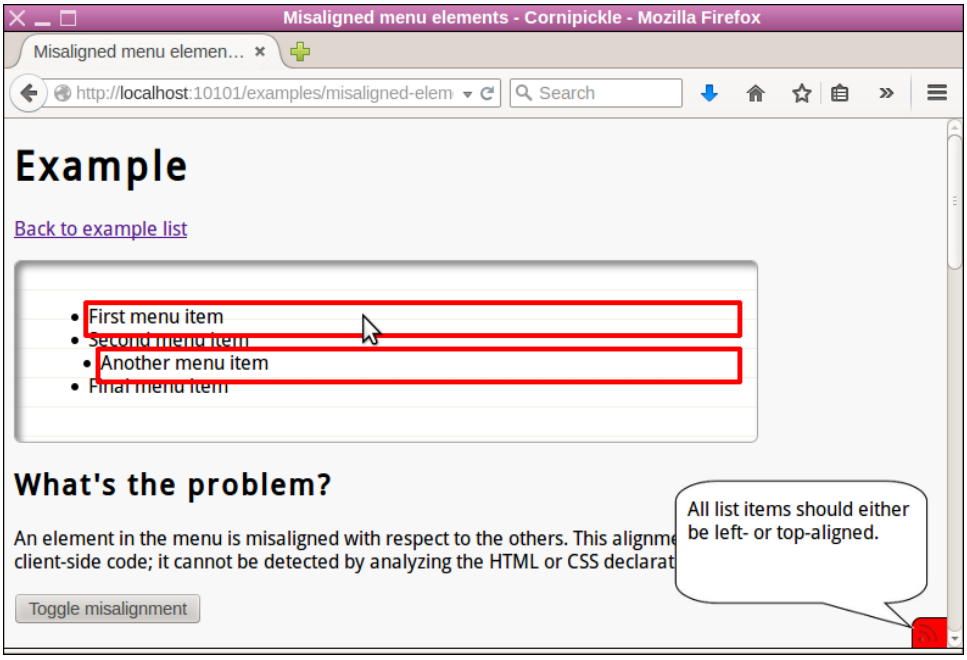


Figure 4.3 – Une capture d'écran de Cornipickle en action.

propriétés d'entrée au début du processus (13-16).

La figure 4.3 montre un exemple de Cornipickle en action. Dans ce cas, la sonde injectée a inséré une petite icône dans le coin inférieur droit de la fenêtre, qui devient rouge chaque fois qu'une propriété est violée. Pour contourner le fait que de nombreux navigateurs désactivent les requêtes HTTP inter-site, Ajax n'est pas utilisé pour la communication entre le client et le serveur. La sonde envoie plutôt ses données en modifiant l'attribut `src` d'une image de zéro pixel injectée dans la page, et en passant les données collectées en tant que paramètres GET de cette requête. En sens inverse, le serveur relaie les informations via un cookie spécialement encodé que la sonde peut interroger périodiquement. De cette façon, le serveur Cornipickle peut résider sur un serveur différent de celui de l'application testée, et avoir toujours une communication bidirectionnelle ponctuelle avec sa sonde.

4.2 LE LANGAGE CORNIPICKLE

Le langage Cornipickle comporte des constructions issues de la logique du premier ordre et de la logique temporelle linéaire tels que les quantificateurs et les opérateurs temporels et qui permettent à un utilisateur de spécifier des relations complexes sur les différents éléments du document à plusieurs moments dans le temps, caractéristique qui est absente dans beaucoup de langages de script.

Cornipickle n'est pas un convertisseur basé sur des expressions régulières entre des fichiers texte et des commandes de script, mais sa grammaire lutte pour le même genre de lisibilité. En particulier, pour améliorer la lisibilité, la grammaire de Cornipickle permet d'insérer différents mots à l'intérieur des différentes constructions. Ces mots n'ont aucun effet sur l'analyse et l'interprétation des expressions.

4.2.1 SYNTAXE DU LANGAGE

Dans Cornipickle, les propriétés sont exprimées sous forme d'assertions sur le contenu et les attributs d'une *capture* (snapshot) d'une page prise à un instant donné. La manière précise par laquelle ces captures sont prises à partir d'une application web donnée sera détaillée plus loin.

Nous commençons par une illustration des différentes constructions de la grammaire dans le tableau 4.1.

Sélection d'éléments Les *éléments* de la page sont l'unité principale dans Cornipickle, ils sont sélectionnés afin d'exprimer quelques-unes de leurs propriétés. Ces propriétés peuvent être appliquées à tous les éléments sélectionnés, ou au moins à un.

⟨*S*⟩ : := ⟨*predicate*⟩ | ⟨*def-set*⟩ | ⟨*statement*⟩

Énoncés Cornipickle

⟨*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*⟩

Logique du premier ordre

⟨*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*⟩)

Expressions temporelles

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

Tableau 4.1 – La grammaire BNF pour Cornipickle (Partie I)

Par conséquent, la sélection de l'élément se fait par le biais de la quantification du premier ordre classique, en utilisant l'anglais pour la syntaxe comme : `For each $x in S` ou `There exists $x in S`(pour dire chaque `$x` dans `S` ou Il existe `$x` dans `S`). Dans ces expressions, `S` désigne soit un sélecteur CSS [5], ou un autre ensemble précédemment défini par l'utilisateur. Les sélecteurs CSS sont exprimés en utilisant la syntaxe de jQuery `$(...)`. Un sélecteur spécial appelé `CDATA` peut être utilisé pour désigner le contenu du texte des nœuds feuilles dans un arbre DOM (les parties qui composent la page en texte clair). Si `$x` est une variable quantifiée en utilisant le mécanisme décrit ci-dessus, on peut accéder au DOM ou aux attributs CSS de cet élément en utilisant `$x's property` (où `property` est l'attribut CSS recherché). Par exemple, la largeur de l'élément s'écrirait : `$x's width`.

Les attributs de l'élément (qui sont soit des chaînes de caractères ou de chiffres) peuvent alors être comparés en utilisant des connectifs tels que : `is greater than` ou `equals` ; le matching d'expressions régulières est fait à travers le prédicat `match`, et l'inclusion de chaîne est affirmée par l'assertion `contains`. Des assertions de base sur les éléments peuvent également être combinées en utilisant des connecteurs booléens classiques : `and`, `or`, `if...then`, `not`.

Événements et opérateurs temporels Dans Cornipickle, les événements déclenchés par l'utilisateur tels que les touches et les clics de souris sont représentés comme des attributs sur l'élément qui est la cible de l'événement. Par exemple, un élément qui a été cliqué par l'utilisateur possédera momentanément un attribut `event` avec une valeur `click`. Par conséquent affirmer qu'un élément `$x` a été cliqué peut être écrit : `$x's event is 'click'`.

L'inclusion d'événements dans le langage conduit naturellement à la notion de *séquences* de documents de captures instantanées. Par conséquent, Cornipickle fournit des opérateurs

5. Un sélecteur CSS est une expression de chemin (*path expression*) qui définit les éléments d'un document qui font l'objet d'un ensemble donné de règles. Ces expressions sont définies par une grammaire régulière, comme stipulé dans la norme CSS.

Opérateurs

⟨*equality*⟩ : := ⟨*property-or-const*⟩ equals ⟨*property-or-const*⟩
| ⟨*property-or-const*⟩ is ⟨*property-or-const*⟩
⟨*gt*⟩ : := ⟨*property-or-const*⟩ is greater than ⟨*property-or-const*⟩
⟨*lt*⟩ : := ⟨*property-or-const*⟩ is less than ⟨*property-or-const*⟩
⟨*regex-match*⟩ : := ⟨*property-or-const*⟩ matches ⟨*property-or-const*⟩
⟨*constant*⟩ : := ⟨*number*⟩ | ⟨*string*⟩
⟨*property-or-const*⟩ : := ⟨*elem-property*⟩ | ⟨*constant*⟩
⟨*number*⟩ : := ^\d+
⟨*string*⟩ : := ^"[^"]*"

Constructions auxiliaires

⟨*el-or-not*⟩ : := élément | ε
⟨*set-name*⟩ : := ⟨*css-selector*⟩ | ⟨*userdef-set*⟩ | ⟨*regex-capture*⟩
⟨*userdef-set*⟩ : := ⟨*string*⟩
⟨*var-name*⟩ : := ^\$[\w\d]+

Sélecteur CSS

⟨*css-selector*⟩ : := $(⟨*css-sel-contents*⟩)
⟨*css-sel-contents*⟩ : := ⟨*css-sel-part*⟩ ⟨*css-sel-contents*⟩ | ⟨*css-sel-part*⟩
⟨*css-sel-part*⟩ : := ^[\w\d\u0023\.*]+ ;

Attributs CSS

⟨*css-attribute*⟩ : := value | height | width | top | left | right
| bottom | color | id | text | border | event

Propriétés des éléments

⟨*elem-property*⟩ : := ⟨*elem-property-pos*⟩ | ⟨*elem-property-com*⟩
⟨*elem-property-pos*⟩ : := ⟨*var-name*⟩ 's ⟨*css-attribute*⟩
⟨*elem-property-com*⟩ : := the ⟨*css-attribute*⟩ of ⟨*var-name*⟩

Expressions régulières

⟨*regex-capture*⟩ : := match ⟨*elem-property*⟩ with ⟨*string*⟩

Tableau 4.2 – La grammaire BNF pour Cornipickle (Partie II)

empruntés à la logique temporelle Linéaire (LTL) pour exprimer des assertions sur l'évolution du contenu d'un document au fil du temps.

La construction `Always` φ, nous permet de faire l'assertion suivante : quelle que soit l'expression de φ, elle doit être vraie (True) dans tous les snapshots du document. De même, on utilise `Eventually` φ pour dire que φ sera vraie dans certains futurs snapshots du document, et `Next` φ est utilisé pour dire que φ est vrai dans la capture suivante.

Une construction spéciale appelée `The next time` φ `then` ψ affirme que ψ doit être vraie, mais seulement une fois que φ est Vraie. Par exemple, on peut utiliser cette construction pour exprimer que quelque chose doit être observé après qu'un élément ait été cliqué ; l'assertion ne lie pas jusqu'à ce moment. Ceci est une légère réécriture de l'opérateur *until* de LTL.

Un but particulier des opérateurs temporels est de comparer l'état du même élément sur plusieurs snapshots. Cela peut être fait dans Cornipickle avec la construction `When $x is now $y` φ. Si `$x` se réfère à l'état d'un élément capturé dans un snapshot antérieur, alors `$y` contiendra l'état du même élément dans la capture (snapshot) actuelle.

Toutes ces constructions peuvent être librement combinées. Par exemple, la propriété suivante affirme que chaque élément de la liste se déplacera vers le bas de la page, à un certain moment :

```
For each $x in $(li) (
  Eventually (
    When $x is now $y (
      $y's top is greater than $x's top ))).
```

Extension de la grammaire Une caractéristique très importante et exceptionnelle qui contribue à la lisibilité des spécifications Cornipickle est la possibilité d'étendre le vocabulaire de base du langage. Ce dernier donne aux utilisateurs cette possibilité en utilisant leurs propres définitions. Ces nouvelles définitions seront lues par l'interpréteur, et pourront ensuite être

Ensemble défini en extension

⟨*def-set*⟩ : := A ⟨*def-set-name*⟩ is any of ⟨*def-set-éléments*⟩
⟨*def-set-name*⟩ : := ^.* ?(?=is)
⟨*def-set-éléments*⟩ : := ⟨*def-set-élément*⟩ , ⟨*def-set-éléments*⟩ | ⟨*def-set-élément*⟩
⟨*def-set-élément*⟩ : := ⟨*constant*⟩

Prédicats définis par l'utilisateur

⟨*predicate*⟩ : := We say that ⟨*pred-pattern*⟩ when (⟨*statement*⟩)
⟨*pred-pattern*⟩ : := ^.* ?(?=when)

Énoncés définis par l'utilisateur

⟨*userdef-stmt*⟩ : := empty

Tableau 4.3 – Extensions de la grammaire BNF pour Cornipickle

utilisées librement comme tout élément de base du langage.

Les prédicats peuvent être définis avec la construction `We say that...when`. Le texte entre `that` et `when` est interprété comme une chaîne de caractères qui peut contenir des variables. Puis le texte après `when` décrit comment cette expression doit être évaluée en termes du vocabulaire existant. Par exemple, on peut définir l'expression « left-aligned » comme suit :

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

La construction `$x and $y are left-aligned` ($x et $y sont alignés à gauche) peut ensuite être réutilisée (éventuellement avec différents noms de variables) dans des assertions ultérieures. Les utilisateurs peuvent également définir des ensembles, soient des ensembles de chaînes de caractères, des chiffres ou des ensembles d'éléments à partir d'une page en les énumérant en utilisant la construction `A...is any of` :

```
A Mojibake is any of "Ã©", "Ã'", "Ã¨".
```

Notez que le nom de l'ensemble ne doit pas nécessairement être un seul mot ; l'analyseur interprète tout ce qui est entre `A` et `is any of` comme un nom.

La quantité de données pouvant être relayée de cette manière étant limitée, Cornipickle se charge d'envoyer une sonde qui ne récupère que les informations nécessaires à l'évaluation des spécifications fournies par l'utilisateur. Par conséquent, la sonde reçoit des instructions sur les éléments de la page qui sont intéressants et sur les attributs DOM et CSS nécessaires pour ces éléments. Ceci est fait en récupérant l'ensemble de tous les noms d'attributs apparaissant dans une expression, et l'ensemble de tous les sélecteurs CSS utilisés dans les quantificateurs.

La sonde parcourt la structure DOM d'une manière en profondeur et produit un nœud de sortie pour chaque nœud DOM visité. Par défaut, le nœud de sortie est vide : il agit uniquement comme un espace réservé vide, afin de préserver la relation parent-enfant entre les nœuds de sortie. Ce n'est que si l'emplacement du nœud actuel correspond à l'un des sélecteurs CSS que les attributs et les valeurs seront ajoutées au nœud, et seulement pour les attributs présents dans l'expression à évaluer. Des réductions supplémentaires peuvent être réalisées en réduisant tous les sous-arbres qui contiennent uniquement des nœuds vides. Ainsi, la structure DOM produite par la sonde peut être vue comme une version « évidée » du document original, ne contenant que des nœuds et des attributs importants pour l'évaluation d'une propriété.

Incidemment, il faut noter que ce filtrage est relativement grossier. Considérons par exemple l'expression suivante :

```
For each $x in $(p)
  If $x's height equals 400 Then
    For each $y in $(h1)
      $x's width is greater than $y's width.
```

Cornipickle sera chargé d'aller chercher la largeur et la hauteur de tous les paragraphes et rubriques ; pourtant, on ne peut voir que les paragraphes de 400 pixels de hauteur qui sont

réellement nécessaires pour décider de la vrai valeur de la propriété. De plus, les informations sur les titres n'ont d'importance que si de tels paragraphes existent dans le document, sinon la propriété est vide. Par conséquent, les conditions de filtrage pourraient être affinées ; un compromis doit être atteint entre les économies de bande passante d'un tel filtrage et la puissance de calcul nécessaire du côté client pour évaluer les conditions.

4.2.2 SÉMANTIQUE FORMELLE

Nous allons maintenant présenter la sémantique formelle de Cornipickle. La première étape consiste à formaliser la structure, le contenu et les propriétés de style d'une page affichée.

Nous définissons d'abord un ensemble A de *noms d'attributs*. Cet ensemble comprend tous les attributs du DOM (Document Object Model Level 2) [55] et toutes les propriétés de feuilles de style (CSS) qui peuvent être associées à un élément. Un *nœud DOM* est une fonction $\nu : A \to V$, qui associe à chaque nom d'attribut une valeur prise à partir d'un ensemble V. Nous utilisons la valeur spéciale « ? » pour indiquer qu'un attribut est indéfini pour un nœud donné. Nous distinguons un sous-ensemble $E \subset V$ qui désigne les *noms* d'éléments correspondants au nom de la balise HTML réelle qui représente l'élément (par exemple : `a`, `h1`, `img`, etc.).

Nous indiquerons par N l'ensemble de tous les nœuds DOM. L'ensemble T de *documents* DOM comprend tous les arbres dont les nœuds sont des nœuds DOM. Conformément à la convention adoptée par la plupart des navigateurs Web, les éléments de texte ne peuvent apparaître que comme feuilles et reçoivent le nom d'élément spécial `#TEXT`. La figure 4.4 représente un tel document. Si nous laissons ν se référer au deuxième paragraphe du document `body`, nous avons par exemple $\nu(\text{elementName}) = \text{“p”}$, $\nu(\text{style.color}) = \text{“red”}$, etc. Nous étendons ν aux valeurs en définissant $\nu(v) = v$ pour tout $v \in V$.

```
<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>
```

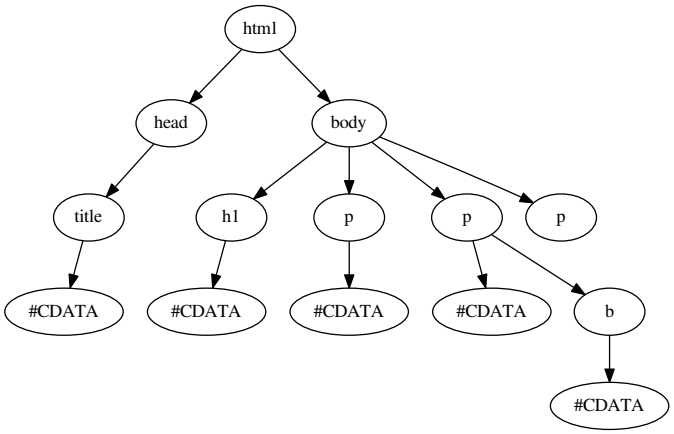


Figure 4.4 – Un document DOM simple. (a) Structure HTML (b) Représentation en arbre DOM ; Seuls les *noms* des éléments sont affichés : les attributs et valeurs restants sont omis pour plus de clarté.

Soit $\kappa : T \times N \to 2^N$ la fonction qui, étant donné un document $t \in T$ et un nœud $v \in N$, produit l'ensemble $\kappa(t, v)$ de tous les enfants de v dans T. Soit C l'ensemble de tous les sélecteurs CSS. La fonction $\chi : T \times S \to 2^N$ renverra l'ensemble des nœuds en t correspondant à un sélecteur CSS $c \in C$. Les événements déclenchés par l'utilisateur sont pris en compte en supposant que certains éléments portent un attribut avec le nom spécial « event », dont la valeur décrit l'événement auquel cet élément est lié. Par exemple, un utilisateur qui clique sur un bouton ferait en sorte que l'attribut « event » de ce bouton porterait « click » comme valeur. De cette façon, il est possible qu'un instantané d'un document contienne des informations sur les événements dynamiques survenant dans l'application.

La sémantique de Cornipickle est définie sur des *traces* des documents DOM ; une trace est une séquence finie d'éléments de T, que nous désignerons par $\bar{t} = t_0, t_1, \ldots, t_k$. Étant donné que toutes les expressions impliquant des constructions définies avec `We say that` peuvent facilement être converties en expressions qui utilisent uniquement des constructions à partir du langage de base, il suffit de définir la sémantique pour ce langage de base. On dira qu'une trace $\bar{t}$ *satisfait* une expression Cornipickle φ, notée $\bar{t} \models \varphi$, lorsque son évaluation renvoie la valeur Vrai ($\top$). La notation $\bar{t}^i$ indique le suffixe de $\bar{t}$ à partir de son i-ème événement.

La sémantique complète est définie récursivement dans le tableau 4.4. En termes formels, l'expressivité du langage Cornipickle correspond à une extension du premier ordre de la logique temporelle linéaire où les événements sont des structures arborescentes des paires nom-valeur, semblables à celles utilisées par le moniteur d'exécution BeepBeep [53] ; cependant, BeepBeep n'a pas la possibilité de créer des constructions grammaticales définies par l'utilisateur. En outre, le langage a été étendu à des constructions qui, même si elles n'accroissent pas l'expressivité, améliorent la lisibilité des spécifications, tel que `The next time`.[6]

6. Formellement, $\bar{t} \models$ `The next time` φ `then` ψ si et seulement si $\bar{t} \models$ `Not` φ `Until (`φ `And` ψ`)`.

$$
\begin{array}{rcl}
\bar{t} \models v\texttt{'s } a \texttt{ equals } v'\texttt{'s } a' & \Leftrightarrow & v(a) = v'(a') \\
\bar{t} \models v\texttt{'s } a \texttt{ equals } v & \Leftrightarrow & v(a) = v \\
\bar{t} \models \texttt{Not}\,\varphi & \Leftrightarrow & \bar{t} \not\models \varphi \\
\bar{t} \models \varphi \texttt{ And } \psi & \Leftrightarrow & \bar{t} \models \varphi \text{ and } \bar{t} \models \psi \\
\bar{t} \models \varphi \texttt{ Ou } \psi & \Leftrightarrow & \bar{t} \models \varphi \text{ or } \bar{t} \models \psi \\
\bar{t} \models \texttt{If } \varphi \texttt{ Then } \psi & \Leftrightarrow & \bar{t} \not\models \varphi \text{ ou } \bar{t} \models \psi \\
\bar{t} \models \texttt{There exists } \xi \texttt{ in \$(}c\texttt{) such that } \varphi & \Leftrightarrow & \bar{t} \models \varphi[\xi/v] \text{ pour certains } v \in \chi(\bar{t}_0, c) \\
\bar{t} \models \texttt{For each } \xi \texttt{ in \$(}c\texttt{) } \varphi & \Leftrightarrow & \bar{t} \models \varphi[\xi/v] \text{ pour tout } v \in \chi(\bar{t}_0, c) \\
\bar{t} \models \texttt{Always}\,\varphi & \Leftrightarrow & \bar{t} \models \varphi \text{ et } \bar{t}^1 \models \texttt{Always}\,\varphi \\
\bar{t} \models \texttt{Eventually}\,\varphi & \Leftrightarrow & \bar{t} \models \varphi \text{ ou } \bar{t}^1 \models \texttt{Eventually}\,\varphi \\
\bar{t} \models \texttt{Next}\,\varphi & \Leftrightarrow & \bar{t}^1 \models \varphi \\
\bar{t} \models \varphi \texttt{ Until } \psi & \Leftrightarrow & \text{Il existe } i \geq 0 \text{ tel que} \\
 & & \bar{t}^i \models \psi \text{ and } \bar{t}^j \models \varphi \text{ for } j < i \\
\text{When } \xi \text{ is now } \xi' \ \varphi & \Leftrightarrow & \text{Il existe } v' \in t_0 \text{ tel que} \\
 & & v(id) = v'(id) \text{ and } \bar{t} \models \varphi[\xi/v']
\end{array}
$$

Tableau 4.4 – La sémantique formelle de Cornipickle ; $a, a' \in A$ sont les noms d'attributs DOM, $v \in V$ est une valeur d'attribut, $c \in C$ est un sélecteur CSS, ξ et ξ' sont des noms de variables $v, v' \in N$ sont les nœuds DOM, et φ et ψ sont des énoncés Cornipickle quelconques. Lorsque $\bar{t}$ est vide, *Always* s'évalue à *Vrai* et *Eventually* et *Next* s'evaluent à *Faux*.

Le cas de l'expression `When $x is now $y` justifie une explication, cependant. Cette construction est utilisée pour désigner le même élément d'un document à deux moments différents dans le temps. En raison de la nature dynamique des applications web, il ne suffit pas d'utiliser simplement `For each $x in $(s)` suivi par `For each $y in $(s)`, avec le même sélecteur CSS `s`. Les éléments d'une page peuvent être déplacés arbitrairement vers n'importe quelle partie d'un document et, par conséquent, la récupération d'éléments avec le même sélecteur ne garantit pas qu'ils seront répartis sur le même domaine deux fois. Cornipickle résout ce problème en donnant à chaque élément un attribut unique, appelé `cornipickleid` (raccourci à `id` dans le tableau). Cet identifiant ne change jamais, quelles que soient les manipulations de l'application sur un élément. L'expression `When $x is now $y` évalue la variable `$y` avec l'élément ayant la même `cornipickleid` comme cela a été donné à l'évaluation de la variable `$x`, permettant de comparer les attributs du même élément dans deux instantanés distincts de la page.

4.3 EXPRIMER DES PROPRIÉTÉS AVEC CORNIPICKLE

Au moyen d'une tel langage, il est possible de donner des exemples de propriétés correspondant à certains des bugs cités précédemment. Par exemple, en prenant pour `Mojibake` l'ensemble défini précédemment, la présence de problèmes de codage de caractères dans une page peut être détectée avec :

```
We say that $x doesn't contain $y when (
  Not ($x's text matches $y's value )).
For each $text in $(CDATA) (
  For each $mojibake in "Mojibake" (
   $text doesn't contain $mojibake )).
```

Nous ajoutons la construction `doesn't contain` à la grammaire, simplement pour améliorer

la lisibilité de la déclaration qui suit.

Similairement, pour préciser qu'une classe spécifique d'éléments ne devrait jamais se déplacer, on peut écrire ce qui suit :

```
We say that $x is immobile when (
  Always (
    When $x is now $y (
     ($x's left equals $y's left)
    And
      ($x's top equals $y's top )))).
For each $item in $(li) ( $item is immobile ).
```

L'intuitivité de spécifications peut encore être mise en évidence dans ce dernier exemple, qui stipule qu'au moins un élément de la liste a la valeur d'un autre élément de liste la dernière fois que l'utilisateur a cliqué sur un bouton appelé « Go » :

```
We say that I click on Go when (
  There exists $b in $(button) such that (
    ($b's text is "Go")
  And
   ($b's event is "mouseup"))).
Always (
  If (I click on Go) Then (
    There exists $x in $(.value) such that (
     The next time (I click on Go)
  Then (
    There exists $y in $(.value) such that (
      $x's text equals $y's text )))))).
```

La lisibilité de cette spécification devrait être mise en contraste avec le code procédural qu'on aurait besoin d'écrire pour détecter le même problème, qui est objectivement plus long, et beaucoup moins clair. Par exemple, dans jQuery, on obtiendrait l'équivalent de la figure 4.5.

Il est maintenant possible de reprendre certains exemples de bugs mentionnés en début de thèse, et de montrer comment ceux-ci peuvent être détectés par des expressions Cornipickle

```
$(document).mouseUp(function(event) {
   var e = arguments.callee;
   if ($(event.target).text() === "Go") {
   var current_values = [];
    $(".value").each(
     current_values.push($(this).text());
    );
   if (e.lastValues !== undefined) {
     var found = false;
     for (var v in current_values) {
       if ($.inArray(v, e.lastValues)) {
         found = true;
         break;
       }
     }
   }
   If (!found)
     console.log("Error");
   e.lastValues = current_values;
 }
};);
```

Figure 4.5 – Exemple de code jQuery vérifiant que deux éléments ont le même texte.

appropriées. Prenons d'abord le cas d'un élément qui se déplace dans une page : cliquer sur un élément change sa classe CSS ; cela entraîne la modification de la zone de délimitation de l'élément, éventuellement le déplacement d'autres éléments qui ne doivent pas bouger.

On doit d'abord définir ce que signifie être immobile. La construction `When $x is now $y` nous permet de comparer les propriétés d'un même élément dans deux snapshots différents de la page, même si le positionnement relatif de l'élément dans le DOM a changé :

```
We say that $x is immobile when (
  Always (
    When $x is now $y (
      ($x's left equals $y's left)
      And
      ($x's top equals $y's top)
    )
  )
).
```

Avec cette expression, il devient facile de spécifier, par exemple, que chaque élément d'une liste doit demeurer à sa position :

```
"""
  @name Immobile items
  @description List items should never change position
  @severity Error
"""
For each $item in $(li) (
  $item is immobile
).
```

De la même manière, on peut spécifier que des éléments doivent toujours être alignés les uns par rapport aux autres.

Nous définissons d'abord quelques prédicats en utilisant la construction `We say that` :

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

We say that the page is big when (
  The media query "(min-width: 200px)" applies
).
The following rules apply when (
  the page is big
).
```

Ces prédicats nous permettent de simplifier l'expression recherchée, qui devient une double quantification sur les éléments d'une même liste :

```
"""
  @name Menus aligned
  @description All list items should either be left- or top-aligned.
  @severity Warning
"""
For each $z in $(.menu li) (
  For each $t in $(.menu li) (
    ($z and $t are left-aligned)
    Or
    ($z and $t are top-aligned)
  )
).
```

Nous ne faisons aucune réclamation formelle concernant l'exhaustivité du langage ou son expressivité. Cependant, des preuves anecdotiques révèlent que tous les bogues de mise en page dans notre enquête peuvent être exprimés par une expression appropriée, suggérant qu'il est bien adapté à la tâche à accomplir.

CHAPITRE 5

DÉTECTION AVANCÉE : BUGS COMPORTEMENTAUX ET RWD

Nous avons présenté au chapitre précédent Cornipickle et la façon d'évaluer des bugs de présentation en analysant le contenu d'une seule page relativement indépendamment des autres. Dans ce chapitre, nous nous intéressons davantage aux bugs dits *comportementaux*. Dans ces bugs, ce n'est pas la présentation graphique des pages qui est défectueuse, mais bien la fonctionnalité même de l'application. Malgré tout, nous pouvons exprimer et détecter ces bugs à partir de propriétés d'éléments de l'interface.

Nous donnerons d'abord des exemples de bugs comportementaux illustrés dans une application appelée *Beep Store*. Nous citons en suite les solutions actuelles, et décrivons ensuite notre approche qui constitue une technique automatisée fournissant des oracles de test dans le but de détecter les bugs comportementaux qui lient les données à l'ordre des consultations de plusieurs pages de l'application ; cela est fait en combinant Cornipickle avec un robot d'exploration (RIA Crawler). Cette technique est aussi capables de vérifier la cohérence d'une mise en page réactive (responsive) sur une large gamme de largeurs de la fenêtre. Pour cela, nous avons intégré une petite application dans le but de changer simultanément la taille de la fenêtre du navigateur afin de détecter des bugs RWD.

5.1 BUGS COMPORTEMENTAUX DANS LE *BEEP STORE*

Afin d'étudier les problèmes de bugs comportementaux dans les applications RIA (Web 2.0) définis dans 2.1.2, nous montrons quelques exemples de bugs illustrés sur une application appelée le *Beep Store* [53].

Le Beep Store est une application web client-serveur autonome, implémentée en PHP et JavaScript, qui permet aux utilisateurs enregistrés de parcourir une collection fictive de livres et de musique et de gérer un panier virtuel composé de ces éléments. Cette application, dont les caractéristiques ont été extraites d'une étude exhaustive des applications web du monde réel, est une RIA au sens propre du terme : les interactions utilisateurs sont complètement asynchrones, ne nécessitent jamais le rechargement de la page, dépendent des actions passées des utilisateurs, et consistent en un seul document dont les différentes parties sont montrées ou cachées en fonction de l'état actuel de l'application.

Connexions multiples Un des bugs qui peuvent être basculés dans le Beep Store permet à l'utilisateur d'accéder à la page de connexion tout en étant déjà connecté. Ceci est détecté par le fait que le lien « S'identifier » (Login) apparaît dans la barre d'action supérieure même après que l'utilisateur s'est connecté avec succès, comme le montre la figure 5.1 [1]. Évidemment, une application bien construite ne fournirait pas un bouton de connexion après qu'un utilisateur se soit déjà connecté ; cette propriété est à état, dans le sens où l'état valide d'une page dépend de la séquence des actions passées qui sont effectuées par l'utilisateur (dans ce cas, le fait qu'une connexion réussie ait eu lieu).

1. On a vu au chapitre 2 comment des sites web réels présentent exactement ce problème.

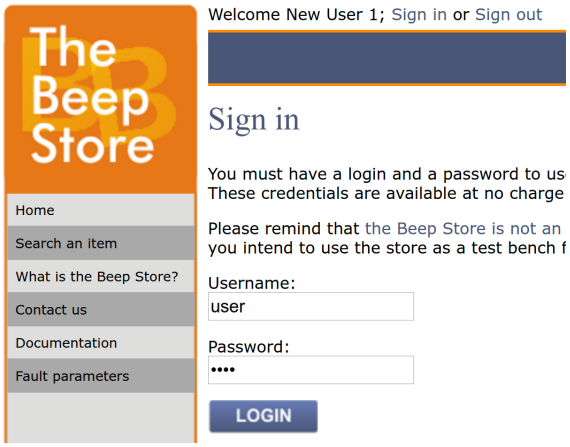


Figure 5.1 – Le bug de connexions multiples. Un utilisateur déjà connecté se voit proposer l'option de se reconnecter.

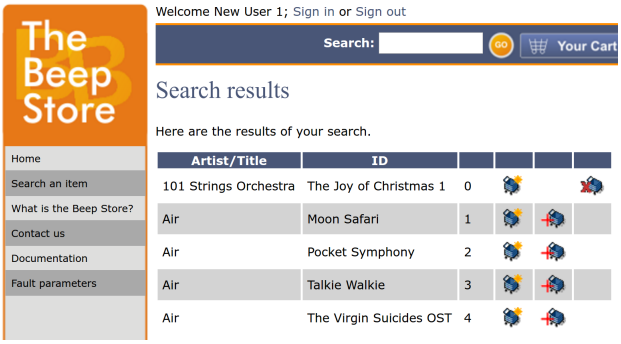


Figure 5.2 – Le bug des paniers multiples. Des boutons pour créer un panier et ajouter des articles au panier coexistent sur la même page.

Paniers multiples Un autre bug permet à l'utilisateur de créer plusieurs paniers d'achat, même après en avoir créé un premier. La figure 5.2 montre un exemple de ce bug : un panier a déjà été créé, puisque l'interface affiche des boutons permettant à l'utilisateur d'ajouter des articles au panier. Pourtant, les boutons pour créer un nouveau panier sont également affichés.

Supprimer d'un panier inexistant Ce bug est l'inverse du précédent : parfois, le Beep Store permet à l'utilisateur de retirer un objet de son panier avant même de le créer. Nous pouvons voir dans la figure 5.3 que les boutons pour créer un panier sont présents avec les boutons pour en retirer les éléments.

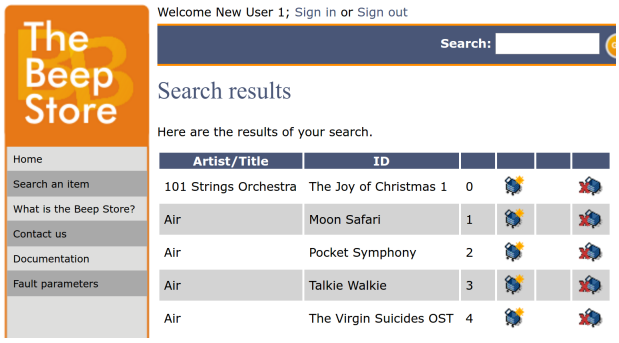


Figure 5.3 – Le bug de la suppression d'un panier inexistant. Des boutons pour retirer du panier et créer un panier coexistent sur la même page.

Ces bugs sont clairement comportementaux, car ils sont causés par des actions antérieures de l'utilisateur ou des actions qui n'ont jamais eu lieu. Il convient également de noter que, selon l'implémentation du serveur, ces bugs ne déclenchent pas nécessairement des messages d'erreur dans les logs. Nous pouvons facilement imaginer un cas où une action est rejetée et ne crée pas d'autres problèmes, mais le client ne devrait jamais avoir été présenté avec l'option.

5.2 SOLUTIONS ACTUELLES

Les travaux connexes sur le test d'applications web pour de tels types de bugs se concentrent sur les moyens de trouver des erreurs dans les applications, en effectuant une recherche exhaustive de leur espace d'état.

Les robots d'exploration (« crawlers ») font partie intégrante des moteurs de recherche web et sont dédiés à la collecte et à l'indexation de documents web. Ils ont été développés à l'origine pour l'archivage, la récupération d'informations (trouver des adresses e-mail ou des pages de produits par exemple). Cependant, on a rapidement découvert que les crawlers pouvaient être utilisés à d'autres fins : en particulier, un crawler peut être considéré comme un outil d'exploration d'espace d'état et à ce titre, être utilisé pour effectuer des tests automatisés.

Un processus d'exploration traditionnel commence à partir des URL de démarrage. Les pages web correspondant à ces URL sont téléchargées et les hyperliens présents sur celles-ci sont extraits et ajoutés à un ensemble d'URL à visiter, également appelé le *crawl frontier*. Comme le nombre d'URL qui peuplent la frontière d'exploration augmente très rapidement, un critère de priorisation du téléchargement de certaines pages est généralement appliqué. À leur tour, les URL les mieux classées dans la limite de l'exploration sont téléchargées et de nouveaux liens sont extraits. Cette opération est répétée jusqu'à ce que tous les liens accessibles soient visités [62, 65].

Certaines fonctionnalités de base des applications web traditionnelles sont modifiées dans les RIA ; ce qui rend l'exploration des RIA plus difficile que celle des applications web traditionnelles. Dans l'analyse RIA, l'ordre d'analyse respecte la séquence de pages possible, comme si un internaute l'utilisait. Comme nous l'avons vu, contrairement aux applications web traditionnelles, une URL n'identifie pas de façon unique l'état de l'application, et ne peut donc pas être simplement demandée au serveur à tout moment.

Dans une application avec une gestion du panier comme le Beep Store, il serait possible pour un robot d'exploration traditionnel de trouver des bugs comportementaux là où il n'y en a pas ; l'ordre de visite est crucial. Par exemple, dans un scénario où l'utilisateur peut créer un panier, supprimer un panier, ajouter un article à un panier et modifier un panier pour modifier la quantité, un bug qui permet à l'utilisateur d'éditer un article dans son panier sans avoir de panier pourrait être découvert. Après la création d'un panier, l'ajout d'un élément et la suppression du panier, l'ensemble d'URL du crawler traditionnel contient l'URL à modifier. Ensuite, lorsque vous essayez d'atteindre cette URL, un bug survient car le panier a été supprimé. Cependant, il peut s'agir d'un faux positif car cette séquence d'actions n'est probablement pas possible pour un utilisateur. Un robot d'exploration traditionnel n'est donc pas adapté à la recherche de bugs comportementaux.

Dans un crawler web pour RIA, la page associée à une seed URL (un crawler commence par une liste d'URL à visiter, appelée les seeds) est extraite et chargée dans un navigateur virtuel. Un module est requis pour vérifier si c'est la première fois que la page construite est rencontrée. Un extracteur d'événements récupère les événements JavaScript de la page ; un de ces événements est sélectionné et exécuté sur la page. La page résultante est ensuite collectée et le processus se poursuit jusqu'à épuisement de toutes les actions découvertes [41]. Sur la base de ce modèle, différentes stratégies d'exploration (telles que la recherche en profondeur (depth-first search), Greedy, Model-Based et Component-Based) ont été suggérées [61, 36, 45, 40, 46, 44].

Certains outils ont déjà été conçus pour tester les RIA. Par exemple, WebMate [43] peut extraire l'arborescence d'état d'une application web. Cet arbre est ensuite comparé aux arbres d'état d'autres navigateurs pour trouver les différences de mise en page. Cependant, il se concentre sur la compatibilité inter-navigateurs (cross-browser compatibility) et ne semble pas prendre en charge les tests externes définis par l'utilisateur.

WebMole est un autre crawler automatisé qui généralise les approches existantes. Il élimine tout backtracking arbitraire en interceptant les requêtes HTTP et fait des sauts de page [54]. Cependant, l'objectif de WebMole est simplement d'extraire les graphes de navigation d'une application ; il ne permet pas à un utilisateur d'écrire des oracles de test à évaluer pendant l'exploration de l'application.

De son côté, Crawljax [60] utilise une stratégie en profondeur (depth-first strategy) pour explorer et produire une machine à états finis du comportement de l'application. Il est possible grâce à son architecture de plugin de tester chaque état pendant qu'ils sont visités. Cependant, les tests en question doivent être écrits par l'utilisateur en code Java pur ; cela rend l'écriture des oracles de test dynamiques difficile, pour des raisons mentionnées plus haut.

5.3 SOLUTION PROPOSÉE

Pour remédier à ces problèmes, nous proposons dans cette section une architecture qui combine un robot d'indexation RIA (dans ce cas, Crawljax) avec notre interpréteur de langage de haut niveau pour les oracles de test web : Cornipickle. Crawljax est responsable de l'exploration d'une application web en tenant compte de son état, tandis que nous utilisons les opérateurs empruntés de la logique temporelle linéaire fournie par Cornipickle pour exprimer des assertions sur l'évolution du contenu d'un document au fil du temps. Cette architecture a été codée dans un plugin open source pour Crawljax [2].

5.3.1 INTERACTION AVEC CRAWLJAX

Crawljax est un outil pour explorer automatiquement l'état dynamique des applications web modernes. Via des interfaces de programmation, il a la capacité d'interagir avec le code côté client de l'application. Nous l'utilisons pour explorer le comportement de l'application web à tester. Pour détecter les clics, Crawljax analyse une page Web et l'utilise systématiquement pour explorer le comportement dynamique de l'application [60, 70].

Les modifications détectées dans l'arbre DOM dynamique sont validées en tant que nouveaux états du comportement. De nombreuses options sont disponibles avec Crawljax pour configurer le comportement d'analyse. Nous pouvons par exemple spécifier les liens ou les widgets à cliquer ou non au cours de l'exploration. Dans une variante, Crawljax effectue une recherche en profondeur en premier (depth-first search), stocke l'historique des exécutions d'événements et n'exécute un événement que si l'événement n'a pas été exécuté auparavant, quel que soit l'état de l'application [68].

2. http ://github.com/liflab/crawljax-cornipickle-plugin

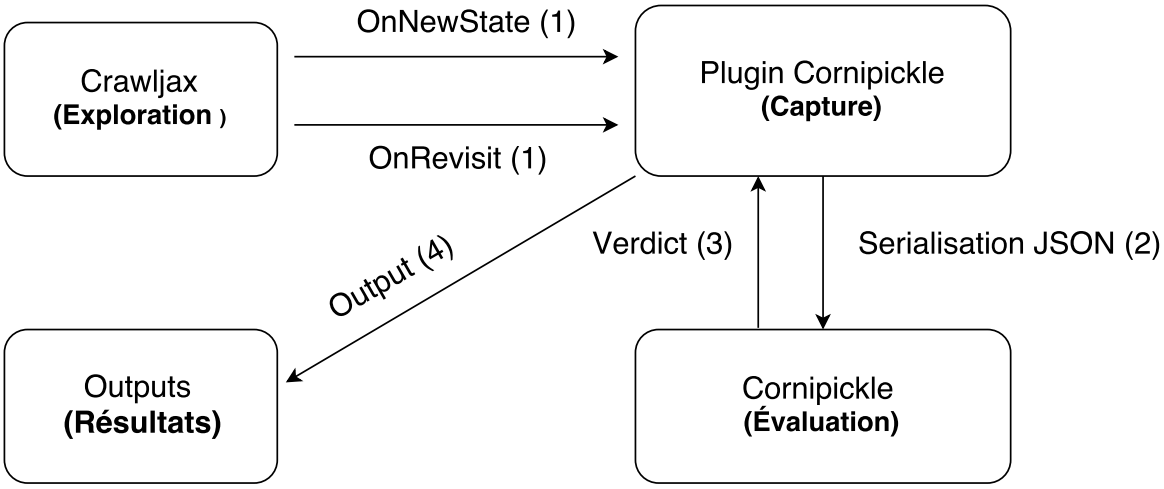


Figure 5.4 – Flux d'interaction et de sérialisation (Crawljax-Cornipickle)

La figure 5.4 montre le workflow du système combiné pour détecter les bugs comportementaux de l'application testée. Crawljax explore le comportement de l'application web sous test (Exploration). Il interagit avec Cornipickle à travers son architecture de plugin (Plugin Cornipickle (capture)). (1) Chaque fois qu'un état est créé (OnNewState) ou visité (OnRevisit), Crawljax sérialise la page (sérialisation JSON) et l'envoie à l'interpréteur (2) pour l'évaluer de la même manière que la sonde envoie la page au serveur Cornipickle dans l'architecture traditionnelle (Cornipickle évaluation). Après que la page ait été évaluée par Cornipickle, le verdict (3) est retourné et le plugin affiche le résultat (Outputs)(4).

Il est important de se rappeler que chaque état de l'application est visité par Crawljax dans la même séquence qu'un utilisateur. Même quand il revient à un état antérieur, il recommence au début de l'exploration et prend le même chemin jusqu'à ce que l'état désiré soit atteint.

5.3.2 REDIMENSIONNMENT DU NAVIGATEUR

Ce même principe peut être facilement modifié pour également détecter des bugs RWD. Pour ce faire, nous avons créé un autre plugin qui, cette fois, redimensionne le navigateur d'une

largeur donnée à une autre largeur.

Étant donné que l'utilisation d'une barre de défilement verticale n'est pas un problème dans le responsive design, seul le redimensionnement horizontal est l'approche correcte pour détecter les bugs RWD. Puisque nous voulons explicitement trouver des bugs liés au RWD, le plugin diminue lentement la largeur du navigateur ; ces bugs apparaissent sur des largeurs inférieures où l'espace disponible devient de plus en plus rare en référence à des largeurs plus grandes. Il est possible de fournir au plugin la borne supérieure, la borne inférieure et la quantité de pixels pour la décrémentation. Le plugin met également en évidence les bugs qu'il trouve et prend une capture d'écran de la page. Nous obtenons ensuite des captures d'écran où les éléments responsables du bug ont des bordures rouges.

5.4 EXPÉRIENCES ET RÉSULTATS

Comme on peut le voir, la détection de bugs comportementaux et de bugs RWD, dans les deux cas, peut se résumer à la vérification de propriétés exprimées sur des séquences de pages. Dans le premier cas, elles sont fournies par un robot d'exploration, et dans le second cas, par le redimensionnement répétitif de la fenêtre du navigateur sur une même page.

Nous détaillons maintenant quelques exemples de propriétés Cornipickle permettant de détecter des bugs comportementaux et des bugs RWD.

5.4.1 DÉTECTION DE BUGS COMPORTEMENTAUX DANS BEEPSTORE

Nous expliquons d'abord les bugs comportementaux du Beep Store décrits précédemment, et nous montrons comment ils peuvent être capturés par Crawljax en évaluant les assertions de Cornipickle lors de l'exploration d'une application.

Connexions multiples Le premier bug est celui des connexions multiples. Ce bug peut facilement être détecté par les expressions suivantes :

```
We say that we are signed in when (
  There exists $p in $(#action-band) such that (
    $p's text matches "^Welcome.*"
)).

We say that we are in the login page when (
  There exists $div in $(#sign-in) such that (
    Not ( $div's display is "none" )
)).

Always (
  If ( we are signed in ) Then (
    Not ( we are in the login page )
)).
```

Les deux définitions `We say that` expliquent comment on définit le fait d'être connecté et d'être dans la page de connexion dans le Beep Store. L'expression *There exists x in y such that (z)* est utilisée pour affecter à la variable *x* un élément de l'ensemble *y* où *z* est vrai. Nous pouvons voir que l'ensemble *y* dans le second prédicat est composé de tous les éléments avec l'identifiant « sign-in » et il s'assure que *z* est vrai avec au moins un d'entre eux. La construction *x matches y*, quant à elle, vérifie si *x* correspond à l'expression régulière *y* et la construction *x is y* vérifie si *x* est égal à *y*. Enfin, l'instruction *Always (x)* vérifie que *x* est vrai dans chaque instantané. En un mot, il ne devrait jamais arriver que la bande d'action dise « Welcome » pendant que le *div* avec l'identifiant « sign-in » est affiché.

À titre de comparaison, la figure 5.5 montre comment on pourrait attraper le même bug uniquement avec Crawljax et son architecture de plugin. La lisibilité est beaucoup plus faible et avec des propriétés plus complexes, nous pouvons voir donc comment le code peut devenir complexe et long.

```
private enum Verdict {TRUE, FALSE, INCONCLUSIVE};
private Verdict m_verdict;

@Override
public void onNewState(CrawlerContext context, StateVertexnewState) {
if(m_verdict == Verdict.INCONCLUSIVE) {
EmbeddedBrowser browser = context.getBrowser();

    Identification identificationActionBand =
        new Identification(Identification.How.id, "action-band");
booleansignedIn = false;

    Identification identificationSignInDiv =
        new Identification(Identification.How.id, "sign-in");
booleancurrentlyInLoginPage = false;

if(browser.elementExists(identificationActionBand)) {
WebElementactionBand = browser.getWebElement(identificationActionBand);
if(Pattern.matches("^Welcome.*", actionBand.getText())) {
signedIn = true;
     }
    }

if(browser.elementExists(identificationSignInDiv)) {
WebElementsignInDiv = browser.getWebElement(identificationSignInDiv);
if(signInDiv.isDisplayed()) {
currentlyInLoginPage = true;
     }
    }

if(signedIn) {
if(currentlyInLoginPage) {
m_verdict = Verdict.FALSE;
     }
    }
  }
output(context, newState);
}
```

Figure 5.5 – Le code nécessaire pour attraper le bug des connexions multiples, en utilisant Crawljax sans Cornipickle

Paniers multiples Le bug des paniers multiples peut être détecté par cette propriété :

```
We say that we are signed in when (
  There exists $p in $(#action-band) such that (
    $p's text matches "^Welcome.*"
)).

We say that we create a cart when (
  There exists $button in $(.button-create-cart)
such that (
    $button's event is "click"
)).

The next time ( we are signed in ) Then (
  The next time ( we create a cart ) Then (
    Always (
      Not ( we create a cart )
))).
```

Les déclarations temporelles `The next time x Then (y)` évaluent à vrai si *y* évalue à vrai mais seulement après que *x* le soit. Donc, après que nous nous soyons connectés et après que nous ayions cliqué sur le bouton « créer un panier », nous ne devrions plus jamais cliquer sur « créer un panier ».

Le bug de suppression d'un panier existant se gère de manière similaire ; nous ne le décrirons pas en détail ici.

Il est à noter que l'évaluation d'un état avec ces propriétés Cornipickle prend entre 36 et 74 millisecondes par page avec un processeur Intel Core i5-3470. Gardez à l'esprit que, bien que les propriétés soient assez simples, le Beep Store est une très grande application à sérialiser car même les blocs non affichés doivent être inclus.

5.4.2 DÉTECTION DE BUGS RWD DANS DE VRAIS SITES WEB

Nous montrons maintenant quelques exemples de propriétés Cornipickle pour la détection de bugs RWD. Les comportements d'un site web sont uniques pour chaque site. Pour cette raison, la détection des bugs comportementaux nécessite des propriétés spécifiques. D'autre part, le Responsive Web Design est une approche générale de la conception Web, similaire aux modèles de conception (*design patterns*) dans les langages traditionnels. Les échecs dans l'implémentation de cette conception doivent être détectables avec des propriétés générales. Pour cette raison, les propriétés décrites dans cette section ne constituent que des avertissements : une violation ne devrait pas signifier qu'il s'agit d'un bug dans tous les cas.

Présence de barres de défilement L'une des premières indications d'un site web pas responsive est la présence d'une barre de défilement horizontale. Pour détecter ce bug, une simple propriété Cornipickle peut être définie :

```
We say that there is an horizontal scrollbar when (
  the page's width is less than
    the page's scroll-width
).

"""
  @name No horizontal scrollbar
  @description There should never be an horizontal scrollbar
  @severity Error
"""
Always (
  Not ( there is an horizontal scrollbar )
).
```

Dans cette propriété, l'interception d'une barre de défilement horizontale peut être obtenue en comparant la largeur de la fenêtre (viewportwidth) avec la largeur de défilement (scroll-width).

Cela ne devrait jamais arriver si elle est toujours entourée avec la construction *Always... Not....*

Collision d'éléments C'est le bug où les éléments se chevauchent. Cette propriété commence par certaines définitions du langage pour simplifier le cœur de la propriété à la fin. Elle décrit les intersections horizontales et verticales, un élément visible, deux éléments identiques et des chevauchements.

```
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 is visible when (
  Not ( $x's display is "none" )
).

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 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)
).
```

La première définition utilise « right - 1 » car les éléments qui se croisent doivent se croiser d'au moins 2 pixels. Il surmonte un problème où nous recevons des dimensions et des coordonnées en entiers (pixels) mais le navigateur peut travailler avec des floats dans le cas d'éléments ayant des dimensions en ratios. Ces floats sont arrondis et peuvent provoquer des différences de 1 pixel entre ce qui est affiché et ce qui est sérialisé. Il est vrai que nous pouvons manquer des bugs qui sont légitimement à 1 pixel, mais il est important de ne pas punir les bonnes pratiques.

La définition d'un élément visible vérifie uniquement si la propriété *display* est « none » car ces éléments ne provoquent aucun changement de disposition. En outre, cette valeur est affectée consciemment par le développeur afin que leur position sur la page soit correcte. Le troisième ? décrit deux éléments qui sont identiques en utilisant la propriété « cornipickleid ». Cette propriété est une valeur unique donnée à chaque élément important dans la page pendant la phase de sérialisation. Comme il est unique, il peut être utilisé pour identifier si deux éléments sont identiques.

La dernière construction définit deux éléments qui se chevauchent. Si elles sont à la fois visibles et qu'elles se croisent verticalement et horizontalement, elles sont considérées dans une collision.

```
"""
  @name Element Collision
  @description All items that aren't
    overlapping initially shouldn't ever overlap
```

```
  @severity Warning
"""
Always (
  For each $x in $(body *) (
    For each $y in $($x > *) (
      For each $z in $($x > *) (
        If ( ($y and $z are not the same) And
          ($y and $z do not overlap) ) Then (
          Next (
            When $y is now $a (
              When $z is now $b (
                $a and $b don't overlap
)))))))).
```

Enfin, les trois constructions `For each` rassemblent tous les éléments et leurs enfants directs. Elle permet de tester des paires de frères et sœurs pour leur propriété de recouvrement. Notez qu'elle ne vérifie pas si un élément chevauche un « cousin » car ce cousin aurait besoin de violer la propriété Protrusion (qui n'est pas détaillée dans cette thèse). La propriété pourrait être faite en testant chaque élément avec tous les autres éléments, mais il est coûteux en performance. Les deux constructions `When $x is now $y` rassemblent la paire dans la capture d'écran suivante afin de la comparer à elle-même dans la capture d'écran précédente. Dans l'ensemble, la propriété dit que si deux frères et sœurs ne se chevauchent pas à un moment donné, ces deux frères et sœurs ne devraient pas se chevaucher au moment suivant.

Les autres bugs RWD (dépassement des éléments, etc.) décrits au chapitre 2 peuvent être traités de manière similaires ; nous ne les détaillerons pas ici.

À titre évaluatif, nous avons évalué une propriété en fonction du nombre d'éléments. Le temps de calcul de l'interpréteur en fonction du nombre d'éléments dans la page est montré dans la figure 5.6. Le temps de calcul en incluant la sérialisation de la page par la sonde JavaScript et l'interprétation de la propriété, en fonction du nombre d'éléments dans la page est présenté dans la figure 5.7. Nous pouvons constater que l'interpréteur s'exécute très rapidement ; c'est

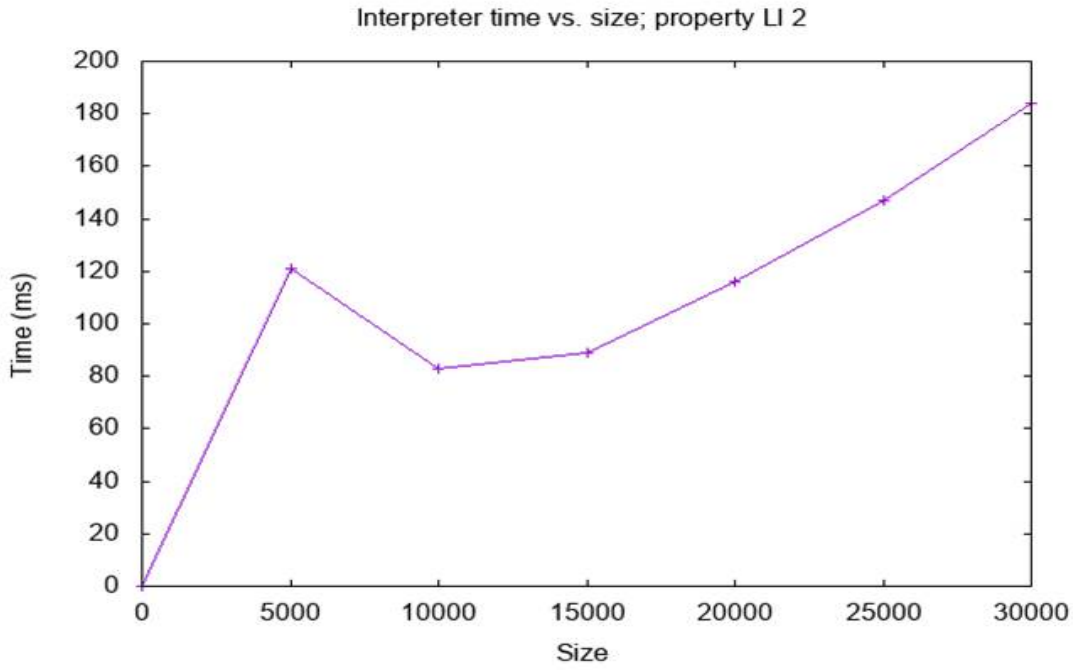


Figure 5.6 – Temps de calcul de l'interpréteur en fonction du nombre d'éléments dans la page.

bien la sérialisation de la page par la sonde qui prend le plus de temps dans le processus global.

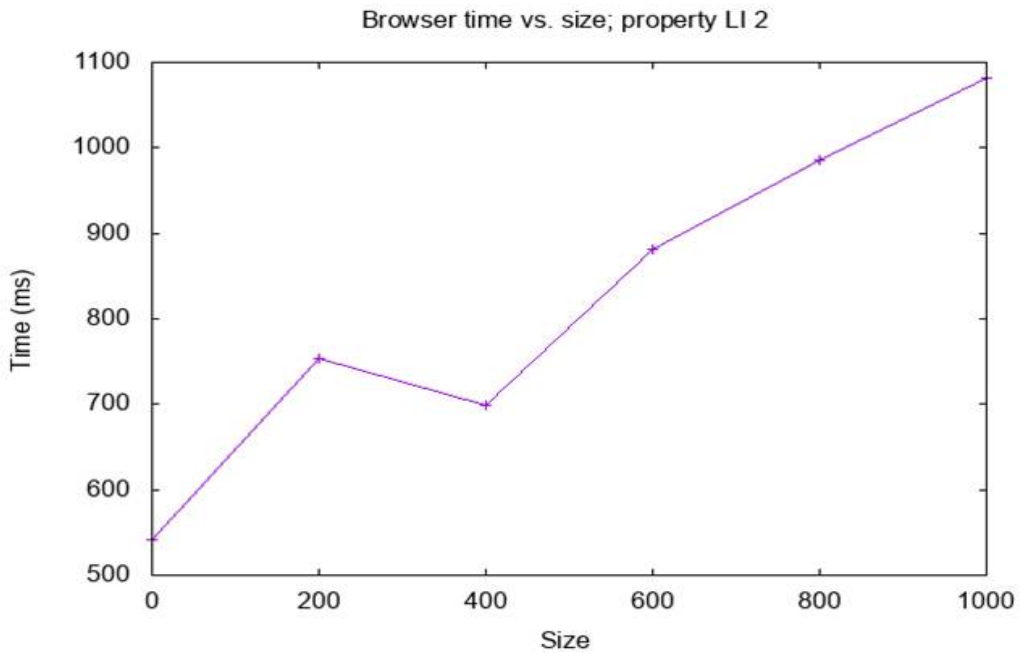


Figure 5.7 – Temps de calcul incluant la sérialisation de la page par la sonde JavaScript et l'interprétation de la propriété, en fonction du nombre d'éléments dans la page.

CHAPITRE 6

VERS UN MEILLEUR FEEDBACK POUR L'UTILISATEUR

Jusqu'ici, nous avons présenté un outil automatisé pour la détection des bugs d'interface permettant d'évaluer les expressions dans un langage déclaratif de haut niveau basé sur la logique temporelle linéaire et de premier ordre. Or, les pages web sont composées de centaines d'éléments avec des dizaines d'assertions de propriétés qui doivent tenir ; de plus, les défauts de mise en page sont parfois trop subtils pour être visibles à l'œil nu (comme les éléments d'un seul pixel). Par conséquent l'évaluation de base de ces propriétés, renvoyant un simple verdict vrai/faux, ne serait pas très utile pour un concepteur. Le fait de simplement dire que « quelque chose ne va pas » apporte peu de valeur ajoutée lorsque quelqu'un doit rechercher le problème dans une page aussi complexe. Pour fournir une réelle évaluation aux praticiens, un outil d'analyse de la mise en page devrait donc être capable de repérer des éléments spécifiques de la page qui sont responsables de certains bugs.

À cette fin, Cornipickle a été équipé d'un mécanisme pour essayer de circonscrire des parties d'une page qui expliquent la faute découverte. Notre travail sur la détection de bugs de mise en page est donc devenu une forme de *localisation de défaut* (fault localization). Nous allons dans ce chapitre exposer deux tentatives afin de fournir à l'utilisateur un verdict plus riche qu'un simple vrai/faux. La première conduit à une construction appelée « témoin », basée sur une

fonction appliquée récursivement sur la *formule* qui est falsifiée. Un témoin met en évidence un ensemble d'éléments dans la page qui sont liés d'une certaine manière à la violation d'une propriété.

Cela s'est révélé insuffisant dans la pratique ; par la suite, nous reprenons le travail sur une nouvelle base formelle, fondée cette fois sur le concept de « réparation ». Intuitivement, une réparation est un ensemble minimal de transformations qui, lorsqu'elles sont appliquées à l'objet original, rétablissent sa satisfiabilité par rapport à la spécification. L'avantage de ce concept est qu'il est indépendant de la nature de l'objet et du langage de spécification utilisé pour déclarer ses propriétés attendues. Il pourrait donc être appliqué à une variété d'autres scénarios, en plus des applications web.

6.1 GÉNÉRATION DE CONTRE-EXEMPLE : LES TÉMOINS

Grâce à la nature des spécifications du langage Cornipickle, basées sur la logique, il est possible d'analyser systématiquement une assertion quand elle est évaluée à *faux*, et d'identifier les éléments qui sont « responsables » de la fausseté de cette assertion.

Verdicts et témoins

Nous appelons *témoin* un arbre d'éléments DOM ; soit $W \subseteq T$ l'ensemble de tous les témoins (W pour *witness*). L'ensemble des *verdicts* est défini comme $V : \mathbb{B} \cup \{\top, \bot, ?\} \times W \times W$; un verdict est composé d'une valeur de vérité et de deux témoins : l'un correspondant à la valeur de vérité $\top$, l'autre à la valeur de vérité $\bot$.

La conjonction de verdicts est une fonction $\otimes : V \times N \times V \to V$ définie comme suit :

$$\otimes(\langle b, w_\top, w_\bot \rangle, v, \langle b', w'_\top, w'_\bot \rangle) = \begin{cases} \langle \bot, w_\top, w_\bot \cup \{(v, w'_\bot)\} \rangle & \text{si } b' = \bot \\ \langle b \wedge b', w_\top \cup \{(v, w'_\top)\}, w_\bot \rangle & \text{si } b \neq \bot \\ \langle b, w_\top, w_\bot \rangle & \text{sinon} \end{cases}$$

Nous interprétons $b \wedge b'$ comme la conjonction classique à trois valeurs. La notation (v, w) désigne la création d'un nouveau témoin (witness) dont la racine est le nœud DOM v, avec le témoin w comme son enfant. La notation $w \cup w'$ désigne l'addition de w' aux enfants du témoin w. Nous allons abuser de la notation et accepter que le deuxième argument de $\otimes$ pourrait être un élément « vide » de N nous désignerons comme $v_\emptyset$. Cet élément est nécessaire, de sorte que chaque opération sur les verdicts peut surclasser un témoin existant avec un nouveau nœud racine, même s'il n'y a rien à « témoigner ».

La conjonction de verdicts met à jour le contenu d'un verdict existant v, et donne un autre verdict v' et un élément DOM v. Si v' est faux, il porte un témoignage de cette fausseté, à savoir $w'_\bot$; ce témoin est attaché comme un enfant d'un nouvel arbre dont la racine est v, et cet arbre est ajouté au témoignage de la fausseté de v, $w_\bot$. De plus, la valeur de vérité de v est définie comme étant $\bot$. Autrement dit, l'explication de v' pour être fausse est ajoutée à l'explication de v pour être fausse. Dans le cas contraire, si ni le premier élément de v, ni celui de v' est faux, alors le témoin v' associé à $\top$ est ajouté au témoin $\top$ de v, et sa valeur de vérité est mise à jour en conséquence. Dans tous les autres cas, v est laissé inchangé.

$$
\begin{array}{rcl}
\omega(\bar{t}, v\texttt{'s } a \texttt{ equals } v'\texttt{'s } a') & = & \begin{cases} \langle \top, \{v, v'\}, \emptyset \rangle & \text{if } v(a) = v'(a') \\ \langle \top, \emptyset, \{v, v'\} \rangle & \text{otherwise} \end{cases} \\
\omega(\bar{t}, v\texttt{'s } a \texttt{ equals } v) & = & \begin{cases} \langle \top, \{v\}, \emptyset \rangle & \text{if } v(a) = v \\ \langle \bot, \emptyset, \{v\} \rangle & \text{otherwise} \end{cases} \\
\omega(\bar{t}, \texttt{Not } \varphi) & = & \ominus(\omega(\bar{t}, \varphi), v_\emptyset) \\
\omega(\bar{t}, \varphi \texttt{ And } \psi) & = & \otimes(\otimes(\langle \top, \emptyset, \emptyset \rangle, v_\emptyset, \omega(\bar{t}, \varphi)), v_\emptyset, \omega(\bar{t}, \psi)) \\
\omega(\bar{t}, \varphi \texttt{ Or } \psi) & = & \oplus(\oplus(\langle \bot, \emptyset, \emptyset \rangle, v_\emptyset, \omega(\bar{t}, \varphi)), v_\emptyset, \omega(\bar{t}, \psi)) \\
\omega(\bar{t}, \texttt{If } \varphi \texttt{ Then } \psi) & = & \oplus(\oplus(\langle \bot, \emptyset, \emptyset \rangle, v_\emptyset, \ominus(\omega(\bar{t}, \varphi), v_\emptyset)), v_\emptyset, \omega(\bar{t}, \psi)) \\
\omega(\bar{t}, \texttt{There exists } \xi \texttt{ in } \texttt{\$}(c) \texttt{ such that } \varphi) & = & \bigoplus_{v \in \chi(\bar{t}_0, c)}^{\langle \bot, \emptyset, \emptyset \rangle} \omega(\bar{t}, \varphi[\xi / v]) \\
\omega(\bar{t}, \texttt{For each } \xi \texttt{ in } \texttt{\$}(c) \texttt{ } \varphi) & = & \bigotimes_{v \in \chi(\bar{t}_0, c)}^{\langle \top, \emptyset, \emptyset \rangle} \omega(\bar{t}, \varphi[\xi / v])
\end{array}
$$

Tableau 6.1 – La définition récursive de la fonction de calcul du verdict ω

La disjonction de verdict $\oplus : V \times N \times V \to V$ est définie comme le dual de la conjonction :

$$
\oplus(\langle b, w_\top, w_\bot \rangle, v, \langle b', w'_\top, w'_\bot \rangle) = \begin{cases} \langle \top, w_\top \cup \{(v, w'_\top)\}, w_\bot \rangle & \text{si } b' = \top \\ \langle b \vee b', w_\top, w_\bot \cup \{(v, w'_\bot)\} \rangle & \text{si } b \neq \top \\ \langle b, w_\top, w_\bot \rangle & \text{sinon} \end{cases}
$$

Enfin, la négation du verdict est la fonction $\ominus : V \times N \to V$ définie comme suit :

$$
\ominus(\langle b, w_\top, w_\bot \rangle, v) = \begin{cases} \langle \neg b, w_\bot \cup \{(v, w_\top)\}, w_\top \cup \{(v, w_\bot)\} \rangle & \text{si } b \in \{\top, \bot\} \\ \langle b, w_\top, w_\bot \rangle & \text{sinon} \end{cases}
$$

Intuitivement, $\ominus$ inverse les témoins associés à $\top$ et $\bot$, et les surmonte d'une nouvelle racine avec le nœud DOM v. Cela n'a aucun effet lorsque le verdict est « ? ».

En utilisant ces opérateurs, la sémantique formelle de Cornipickle décrite dans le tableau 4.4 peut alors être étendue à une fonction $\omega : T^* \times \Phi \to V$, qui, sur une expression $\varphi \in \Phi$ et une trace $\bar{t} \in T^*$, calcule un verdict.

La notation $\bigotimes_{v \in \chi(\bar{t}_0, c)}^{\langle \top, \emptyset, \emptyset \rangle} \omega(\bar{t}, \varphi[\xi / v])$ est un raccourci pour :

$$\otimes(\langle \top, \emptyset, \emptyset \rangle, v_\emptyset, \otimes(\omega(\bar{t}, \varphi[\xi / v_0]), v_0, \otimes(\omega(\bar{t}, \varphi[\xi / v_1]), v_1, \cdots)))$$

En d'autres termes, elle désigne la conjonction répétée du verdict de $\omega(\bar{t}, \varphi[\xi / v])$ pour chaque $v \in \chi(\bar{t}_0, c)$, à partir du verdict vide $\langle \top, \emptyset, \emptyset \rangle$. Une notation similaire est utilisée pour $\oplus$.

Cette définition est difficile à lire, en termes de notation ; cependant, le lecteur peut réaliser en l'examinant que la définition de chaque cas correspond à l'intuition. Par exemple, construire le verdict de « φ `And` ψ » revient à démarrer du verdict « vide » $\langle \top, \emptyset, \emptyset \rangle$, et lui joindre successivement le verdict de φ et ψ.

De même, construire le verdict pour une expression quantifiée existentiellement $\varphi(x)$ est obtenu en calculant successivement la disjonction du verdict de $\varphi(k)$ pour chaque k du verdict initial $\langle \bot, \emptyset, \emptyset \rangle$. Ceci est compatible avec le fait que $\exists x \in S : \varphi(x)$ est équivalent à $\bigvee_{k \in S} \varphi(k)$. Enfin, un raisonnement similaire s'applique aux opérateurs temporels linéaires. Par exemple, $\bar{t} \models \texttt{Always}\, \varphi$ peut être défini comme $\bar{t} \models \varphi$ et $\bar{t}^1 \models \texttt{Always}\, \varphi$; d'où les verdicts $\omega(\bar{t}, \varphi)$ et $\omega(\bar{t}^1, \texttt{Always}\ \varphi)$ sont combinés en utilisant la conjonction verdict.

A titre d'illustration de cette procédure, nous montrerons comment un verdict peut être calculé pour l'expression suivante, en considérant l'arbre de la figure 4.4 : `For each $x in $(p) For each $y in $(p) $x's width equals $y's width`. Le document contient trois paragraphes que nous appellerons p_1, p_2 et p_3 ; le premier et le troisième ont une largeur (width) de 400, tandis que le second a une largeur de 200. La déclaration intérieure `$x's width equals $y's width` sera donc évaluée neuf fois, une fois pour chaque combinaison de nœuds DOM pour `$x` et `$y`.

Selon la définition de la fonction ω dans le tableau 6.1, chaque évaluation produira un verdict

de la forme $\langle \top, \{p_i, p_j\}, \emptyset \rangle$ quand p_i et p_j ont la même largeur, et $\langle \bot, \emptyset, \{p_i, p_j\} \rangle$ quand ils sont définis autrement. Dans le premier cas, la déclaration s'évalue à $\top$, et les nœuds DOM p_i et p_j sont ajoutés comme $\top$-témoins de ce fait. L'inverse s'applique lorsque l'instruction est fausse.

Ces verdicts sont ensuite réunis dans le quantificateur universel le plus profond. Un verdict vide $\langle \top, \emptyset, \emptyset \rangle$ est d'abord créé, et tous les verdicts pour l'expression intérieure sont ensuite combinés en utilisant la conjonction verdict. Par exemple, quand `$x` se réfère à p_1, trois verdicts sont joints : $\langle \top, \{p_1, p_1\}, \emptyset \rangle$, $\langle \bot, \emptyset, \{p_1, p_2\} \rangle$ et $\langle \top, \{p_1, p_3\}, \emptyset \rangle$. Selon la définition de conjonction de verdict, le verdict résultant sera :

$$\langle \bot, \{(v_\emptyset, \langle \top, \{p_1, p_1\}, \emptyset \rangle), (v_\emptyset, \langle \top, \{p_1, p_3\}, \emptyset \rangle)\}, \{(v_\emptyset, \langle \bot, \emptyset, \{p_1, p_2\} \rangle)\} \rangle$$

Les deux verdicts internes s'évaluant à $\top$ sont attachés au témoin associé à $\top$, et le verdict évaluant $\bot$ est attaché au témoin associé à $\bot$. Trois de ces verdicts seront produits par le quantificateur le plus interne, un pour chaque valeur de `$x`, qui seront ensuite combinés en utilisant à nouveau la conjonction par le quantificateur universel le plus externe, ce qui donnera le verdict final :

$$\begin{aligned}
&\langle \bot, \emptyset, \{ \\
&\quad (v_\emptyset, \langle \bot, \{(v_\emptyset, \langle \top, \{p_1, p_1\}, \emptyset \rangle), (v_\emptyset, \langle \top, \{p_1, p_3\}, \emptyset \rangle)\}, \{(v_\emptyset, \langle \bot, \emptyset, \{p_1, p_2\} \rangle)\} \rangle), \\
&\quad (v_\emptyset, \langle \bot, \{(v_\emptyset, \langle \top, \{p_2, p_2\}, \emptyset \rangle)\}, \{(v_\emptyset, \langle \bot, \emptyset, \{p_1, p_2\} \rangle), (v_\emptyset, \langle \bot, \emptyset, \{p_2, p_3\} \rangle)\} \rangle), \\
&\qquad (v_\emptyset, \langle \bot, \{(v_\emptyset, \langle \top, \{p_1, p_3\}, \emptyset \rangle), (v_\emptyset, \langle \top, \{p_3, p_3\}, \emptyset \rangle)\}, \{(v_\emptyset, \langle \bot, \emptyset, \{p_2, p_3\} \rangle)\} \rangle) \ \} \rangle
\end{aligned}$$

L'implémentation actuelle de Cornipickle peut calculer ces verdicts et les renvoyer à la sonde JavaScript. Les verdicts sont envoyés à la sonde sous forme d'une liste Cornipickle ID. Chaque

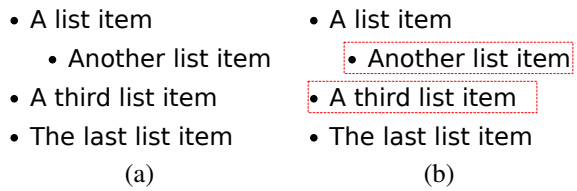


Figure 6.1 – Exemple d'une erreur de mise en page Web simple : (a) l'un des éléments de la liste est incorrectement aligné avec les autres ; (b) un témoin (witness) produit par l'outil Cornipickle.

ID est unique correspondant un élément spécifique dans la page ; ce qui permet d'entourer l'élément en question dans la fenêtre du navigateur.

6.2 LOCALISATION DES ERREURS DANS LES APPLICATIONS WEB

À notre connaissance, le principe de calcul du verdict décrit précédemment fait de Cornipickle un des tout premiers systèmes à *expliquer* graphiquement en quoi une propriété est violée. Malheureusement, nous avons découvert que ce principe laisse tout de même un peu à désirer. Par exemple, considérons la propriété voulant que tous les éléments d'une liste avec l'ID « menu » doivent être alignés verticalement :

```
For each $x in $(#menu li) (
  For each $y in $(#menu li) (
    $x's left equals $y's left
)).
```

Pour cet exemple particulier, la figure 6.1a montre une page simple pour laquelle la propriété serait violée. Nous pouvons voir le résultat de l'application de ω, définie dans la section précédente, sur l'arbre DOM de la figure 6.1a. La fonction retourne un arbre contenant des pointeurs sur deux des éléments de la page, surlignés en rouge dans la figure 6.1b. (En fait, la fonction renvoie plusieurs ensembles, chacun contenant le second élément de liste et l'un des éléments restants.)

Intuitivement, un tel résultat est logique pour un concepteur de sites web ; en effet, ces deux éléments doivent être alignés, alors qu'ils ne le sont pas. Cependant, cette information ne peut être déduite que par la connaissance de la propriété violée ; le témoin pointe simplement ces deux éléments, sans fournir d'informations sur « ce qui ne va pas » à leur sujet. Alors que la génération de contre-exemple récursif présente dans la version actuelle de Cornipickle fournit plus d'informations qu'un simple verdict vrai/faux, dans de nombreux cas, elle peut donc s'avérer trop vague pour être utile.

Nous introduisons la notion de *réparation*, qui peut être définie intuitivement comme un ensemble de modifications nécessaires à un objet pour le rendre conforme à une propriété. La notion de réparation peut être vue comme une localisation de défaut, exprimée en sens inverse : indiquer comment un objet doit être réparé indirectement pointe vers des aspects de sa structure qui sont responsables du fait que la propriété n'est pas actuellement remplie. Nous verrons que, contrairement au concept de témoin, qui est une technique linéaire en fonction de la taille de la formule et fortement associé au langage de spécification et aux objets de domaine utilisés, les réparations sont définies à un niveau d'abstraction qui ne dépend pas des propriétés de l'un ou de l'autre.

6.2.1 DÉFINITIONS

Soit Σ un ensemble de *structures*, et T_Σ un ensemble d'endomorphismes sur Σ ; c'est-à-dire que chaque $\tau \in T_\Sigma$ est une fonction $\tau : \Sigma \to \Sigma$. Soit 2^{T_Σ} l'ensemble de tous les sous-ensembles de T_Σ. Un ensemble d'endomorphismes $T = \{\tau_1, \ldots, \tau_n\} \in 2^{T_\Sigma}$ est dit être *bien défini* si deux éléments τ_i, τ_j sont tels que $\tau_i \circ \tau_j \equiv \tau_j \circ \tau_j$. Un tel ensemble bien défini sera appelé *transformation*. Lorsque le contexte est clair, nous allons abuser de la notation et considérer T comme l'endomorphisme (défini de façon unique) $\tau_1 \circ \cdots \circ \tau_n$. L'inclusion d'ensembles induit

un ordre partiel sur les transformations.

Soit Φ un ensemble *d'expressions de langage* équipées d'une relation de satisfaction $\models : \Sigma \times \Phi \to \{\top, \bot\}$. Pour une expression $\varphi \in \Phi$ et une structure $\sigma \in \Sigma$, nous écrirons $\sigma \models \varphi$ si et seulement si $\models (\sigma, \varphi) = \top$. Dans un tel cas, nous dirons que σ « vérifie » φ, ou alternativement que σ est un *modèle* de φ.

Soit $\sigma \in \Sigma$ une structure telle que $\sigma \not\models \varphi$ pour une expression $\varphi \in \Phi$. Une *réparation* est définie comme une transformation $T \in 2^{T_\Sigma}$ telle que $T(\sigma) \models \varphi$. Une réparation est dite *prime* si aucun sous-ensemble $T' \subseteq T$ est tel que T' est aussi une réparation. Intuitivement, une réparation principale est un ensemble de « changements » sur une structure σ qui satisfait φ, de sorte qu'aucune modification « plus petite » ne restaure aussi la satisfiabilité. Comme $\subseteq$ est une commande partielle, il peut y avoir plusieurs réparations principales mutuellement incomparables.

La figure 6.2 illustre ce concept. L'image représente toutes les transformations qui peuvent être appliquées à une structure, dans le cas simple où seulement quatre morphismes existent. La transformation vide est en bas et chaque flèche dans le graphe représente l'ajout d'un morphisme supplémentaire à une transformation existante. Les nœuds rouges indiquent les transformations qui ne sont pas réparées, tandis que les nœuds jaunes et verts indiquent les réparations. Parmi ceux-ci, les réparations principales sont colorées en vert ; on peut voir que tous les antécédents des nœuds verts sont rouges. L'inverse, cependant, n'est pas vrai : tous les descendants d'une réparation ne sont pas eux-mêmes réparés.

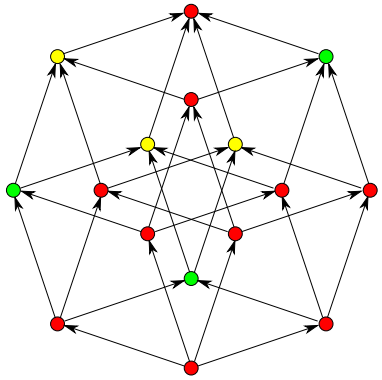


Figure 6.2 – Illustration du concept de réparation principale.

6.2.2 EXEMPLES

Cette définition simple peut ensuite être appliquée à une variété de langages de spécification, comme nous allons l'illustrer à travers les exemples qui suivent.

Logique propositionnelle

Comme premier exemple, soit Φ l'ensemble des formules de logiques propositionnelles avec les variables $X = \{x_1, \ldots, x_n\}$ pour certains $n \geq 1$. Soit Σ l'ensemble des fonctions $X \to \{\top, \bot\}$, que nous appellerons *des évaluations*. La relation de satisfaction $\models$ est définie comme $\sigma \models \varphi = \top$ si et seulement si φ vaut vrai lorsque ses variables sont remplacées par la valeur de vérité correspondante spécifiée par σ, et sinon par $\bot$.

Soit $b \in \{\top, \bot\}$ et $i \in [1, n]$. Nous noterons $\tau_{x_i \mapsto b}$ l'endomorphisme défini comme :

$$(\tau_{x_i \mapsto b}(\sigma))(x) = \begin{cases} b & \text{si } x = x_i \\ \sigma(x) & \text{sinon} \end{cases}$$

Ce morphisme met x_i à la place de b et laisse inchangé le reste de l'évaluation initiale.

L'ensemble des endomorphismes T_Σ est alors défini comme :

$$T_\Sigma = \bigcup_{i\in[1,n]} \bigcup_{b\in\{\top,\bot\}} \tau_{x_i\mapsto b}$$

Deux transformations $\tau_{x\mapsto b}$, $\tau'_{y\mapsto b'}$ commutent si $x \neq y$. Ainsi, un ensemble de transformations $T \in 2^{T_\Sigma}$ est bien défini si et seulement si chaque endomorphisme qu'il contient change la valeur d'une variable différente.

Soit $X = \{a,b,c\}$, soit σ l'évaluation $\{a \mapsto \top, b \mapsto \bot, c \mapsto \bot\}$ et φ la formule propositionnelle $a \wedge b$. On peut facilement observer que $\sigma \not\models \varphi$.Une réparation est la transformation $T = \{\tau_{b\mapsto\top}\}$; qui est, $T(\sigma) \models \varphi$. Cela correspond à l'intuition que l'explication de la fausseté de φ est que b est faux alors qu'il devrait être vrai. Notons que bien que $T' = \{\tau_{b\mapsto\top}, \tau_{c\mapsto\top}\}$ rendrait aussi φ vrai, ce n'est pas une réparation *primaire*, puisque $T \subseteq T'$. Cela correspond à l'intuition que la valeur de vérité de c est pas pertinente à la fausseté de φ.

Soit σ l'évaluation $\{a \mapsto \top, b \mapsto \bot, c \mapsto \bot\}$ et φ la formule propositionnelle $a \to b$. Cette fois, deux réparations primaires existent : $T = \{\tau_{b\mapsto\top}\}$ et $T' = \{\tau_{a\mapsto\bot}\}$. Il est possible de vérifier que les deux fixent la valeur de vérité de l'évaluation initiale. Informellement, la première transformation représente la fausseté de φ sur le fait que a est vrai, tandis que l'autre l'explique plutôt par le fait que b est faux — ce qui correspond bien à l'intuition. Puisque les deux réparations sont incomparables, aucune de ces explications n'est « préférée ». Nous reviendrons sur ce concept plus tard.

Logique du premier ordre

Le concept de réparation peut facilement être levé à l'ensemble Φ de la formule logique de premier ordre sur les domaines finis. Soit A un ensemble d'éléments ; un prédicat n-aire est

défini comme une fonction $p : A^n \to \{\top, \bot\}$; Soit P^i l'ensemble des prédicats de l'arité i. Une signature est un ensemble de prédicats $P = \{p_1, \ldots, p_m\}$, respectivement d'arité $a_1, \ldots, a_m$. Pour une signature donnée, l'ensemble des éléments de domaine est défini comme :

$$\Sigma = P^{a_1} \times \cdots \times P^{a_m}$$

La relation de satisfaction $\models$ est définie comme $\models (d, \varphi) = \top$ si φ est évalué à vrai lors de l'évaluation de prédicats tels que définis dans σ, et $\bot$ définie autrement.

Dans ce contexte, un endomorphisme représentera le changement de la valeur de vérité pour une entrée d'un prédicat. Soit p_k un prédicat de l'arité i, $(a_1, \ldots, a_k) \in A^n$ un k-tuple d'éléments de A, et $b \in \{\top, \bot\}$. La transformation $\tau_{p_k(a_1,\ldots,a_k)\mapsto b}$ est défini comme le prédicat p'_k tel que :

$$p'_k(x_1, \ldots, x_k) = \begin{cases} b & \text{si } x_1 = a_1, \ldots, x_n = a_n \\ p_k(x_1, \ldots, x_k) & \text{autrement} \end{cases}$$

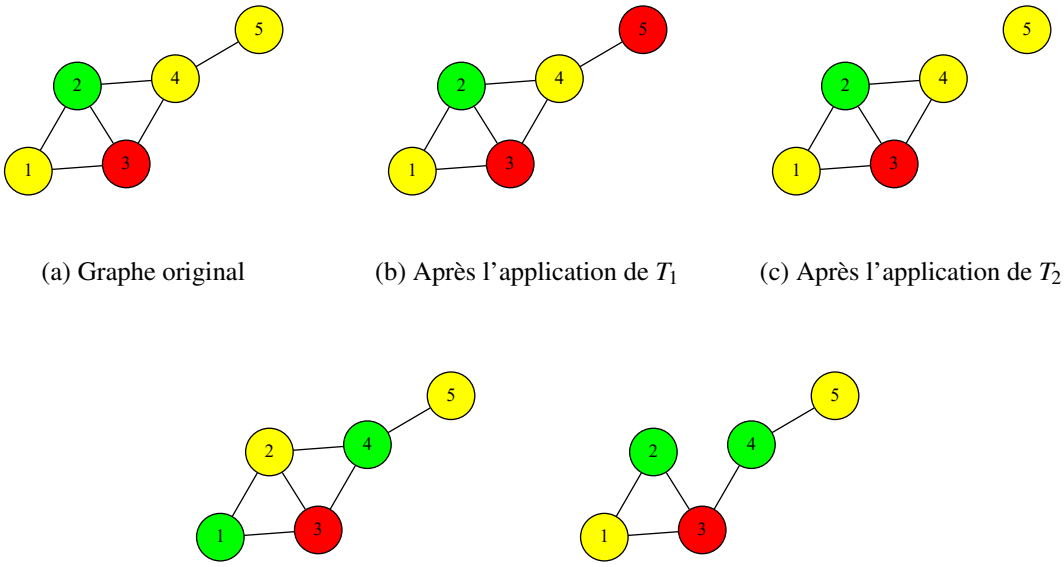
L'ensemble des transformations pour p_k, noté T_{p_k}, est définie comme suit :

$$T_{p_k} \triangleq \bigcup_{(a_1,\ldots,a_k)\in A^n} \left(\bigcup_{b\in\{\top,\bot\}} \{\tau_{p_k,(a_1,\ldots,a_k),b}\} \right)$$

L'ensemble global des transformations est alors :

$$T_\Sigma \triangleq \bigcup_{p\in P} T_p$$

De manière similaire à la logique du premier ordre, on peut vérifier que deux endomorphismes



(a) Graphe original (b) Après l'application de T_1 (c) Après l'application de T_2

(d) Après l'application de T_3 (e) Après l'application de T_4

Figure 6.3 – Quelques réparations possibles pour un coloriage de graphe défectueux

commutent s'ils opèrent sur un prédicat différent, ou changent la valeur d'une entrée différente sur le même prédicat.

Soit $A = \{0,1,2\}$, φ la formule du premier ordre $\forall x : \exists y : x \neq y \wedge p(x,y)$, et le prédicat binaire p défini comme $\{(0,0),(0,1),(1,1)\}$. Il y a deux réparations primaires pour restaurer la vérité de φ : $T_1 = \{\tau_{p(2,0)\mapsto\top}\}$, $T_2 = \{\tau_{p(2,1)\mapsto\top}\}$. Cela correspond à l'intuition que la valeur 2 manque au moins un « partenaire » dans p, et que 0 ou 1 pourraient chacun correspondre à ce but.

Soit $A = [1,5]$ un ensemble de sommets de graphe, p un prédicat binaire codant la relation d'adjacence des arêtes de graphe, et q_1,q_2,q_3 un ensemble de prédicats unaires tel que $q_i(x)$ contient si et seulement si le sommet x ayant la couleur i. Supposons que les prédicats p et q soient définis en fonction de la représentation graphique montrée dans la figure 6.3a.

Une solution au problème de coloriage de graphe peut être représentée par trois expressions de premier ordre :

$$\varphi_1 \triangleq \forall x : (q_1(x) \wedge \neg q_2(x) \wedge \neg q_3(x)) \vee (\neg q_1(x) \wedge q_2(x) \wedge \neg q_3(x)) \vee (\neg q_1(x) \wedge \neg q_2(x) \wedge q_3(x))$$

$$\varphi_2 \triangleq \forall x : \forall y : p(x,y) \rightarrow p(y,x)$$

$$\varphi_3 \triangleq \forall x : \forall y : p(x,y) \rightarrow ((q_1(x) \rightarrow \neg q_1(y)) \wedge (q_2(x) \rightarrow \neg q_2(y)) \wedge (q_3(x) \rightarrow \neg q_3(y)))$$

La première stipule que chaque sommet a une couleur exacte ; la seconde indique que la relation d'adjacence est symétrique et l'expression finale stipule qu'aucun sommet adjacent ne peut avoir la même couleur. On peut voir que le graphe original ne satisfait pas $\varphi_1 \wedge \varphi_2 \wedge \varphi_3$. Il existe plusieurs réparations principales, dont certaines sont indiquées ici :

$$\begin{aligned}
T_1 &= \{\tau_{q_1(5)\mapsto\bot}, \tau_{q_2(5)\mapsto\top}\} \\
T_2 &= \{\tau_{p(4,5)\mapsto\bot}, \tau_{p(5,4)\mapsto\bot}\} \\
T_3 &= \{\tau_{q_1(1)\mapsto\bot}, \tau_{q_3(1)\mapsto\top}, \tau_{q_1(4)\mapsto\bot}, \tau_{q_3(4)\mapsto\top}\} \\
T_4 &= \{\tau_{p(2,4)\mapsto\bot}, \tau_{p(4,2)\mapsto\bot}, \tau_{q_1(4)\mapsto\bot}, \tau_{q_3(4)\mapsto\top}\}
\end{aligned}$$

La réparation T_1 corrige le graphe en changeant la couleur du sommet 5 en rouge. Notons que cela nécessite non seulement de mettre $q_2(5)$ à $\top$, mais aussi $q_1(5)$ à $\bot$; sinon, la structure résultante violerait φ_1. Une autre réparation (non représentée), change le sommet 5 en vert. La réparation T_3 modifie plutôt la relation d'adjacence et coupe le sommet 5 du reste du graphe, de sorte que le conflit de couleur soit résolu.

Ceux-ci correspondent aux moyens « intuitifs » de fixer le coloriage du graphe. Cependant, il existe plusieurs autres réparations primaires qui répondent à la définition. Par exemple, la transformation T_4 échange les couleurs des sommets 1, 2 et 4. Notons qu'il s'agit bien d'une réparation primaire, en ce sens qu'aucun sous-ensemble de ces endomorphismes ne restaure la satisfiabilité de la formule d'origine. De la même façon, T_5 coupe le bord entre les sommets 2 et 4 et passe au vert. Au total, il y a 17 réparations primaires distinctes dans cet exemple particulier.

Encore une fois, il convient de noter que sans contexte supplémentaire, aucune de ces réparations n'est une explication possible de la fausseté de $\varphi_1 \wedge \varphi_2 \wedge \varphi_3$ sur le graphe original.

Logique de premier ordre étendue

L'exemple précédent montre la nécessité d'étendre la sémantique de la logique du premier ordre à des fonctions arbitraires au lieu de prédicats strictement booléens. Cela peut facilement être fait comme suit. Soit $A_1, \ldots, A_n$ et B des ensembles finis. Nous désignerons par $F^{A_1,\ldots A_n \to B}$ l'ensemble de toutes les fonctions $(\prod_i A_i) \to B$. Une signature est un tuple de la forme :

$$\langle (A_{1,1}, \ldots A_{1,n_1}) \to B_1, \ldots, (A_{m,1}, \ldots A_{m,n_m}) \to B_m \rangle$$

tel que f_i est une fonction de l'arité n_i avec le domaine $A_{1,1}, \ldots A_{1,n_i}$ et image B_i. La logique des prédicats est le cas particulier où $B_1 = \cdots = B_{n_m} = \{\top, \bot\}$, dans ce cas, l'image peut être omise, et où $A_{i,j}$ sont tous les mêmes, de sorte que seule l'arité doit être connue. Si f est une fonction $A \to B$ et x désigne un élément de A, nous écrirons $x.f$ pour désigner $f(x)$, permettant ainsi une certaine forme de notation « objet » pour les fonctions.

Dans ce contexte, les quantificateurs de premier ordre doivent préciser sur quel $A_{i,j}$ ils s'ap-

pliquent, de sorte que les expressions deviennent de la forme $\forall x \in A_{i,j} : \varphi$ et $\exists \in A_{i,j} : \varphi$. Les termes de base peuvent maintenant comparer les valeurs de deux termes de fonction, en utilisant n'importe quel opérateur binaire approprié.

Les endomorphismes sont toujours définis de la même manière que pour la logique classique du premier ordre, à condition qu'ils se réfèrent aux valeurs appropriées dans le domaine et l'image de la fonction soumise au changement.

Notons que ce formalisme étendu n'ajoute aucune expressivité à la logique du premier ordre si tous les ensembles sont maintenus finis. Il doit cependant simplifier l'expression de nombreuses propriétés.

Avec ce formalisme modifié, nous sommes prêts à envisager des réparations dans les propriétés de mise en page Web. Soit E un ensemble d'éléments de page, P un ensemble de valeurs de pixels et C un ensemble de couleurs CSS. Sur ces trois ensembles, nous définissons les fonctions $E \to P$ appelées `left`, `right`, `top`, et `bottom`, correspondants respectivement aux coordonnées x et y des coins : supérieur gauche(top-left) et inférieur droit (bottom-right) d'un élément. De plus, nous définissons un ensemble S de *sélecteurs* CSS ; l'évaluation d'un sélecteur CSS sur un document peut être formalisée comme une fonction $\$: S \to 2^E$ qui, pour une expression de filtre donnée, retourne le sous-ensemble de E correspondant au sélecteur.

Les endomorphismes peuvent être définis pour chacune de ces fonctions et doivent être écrits en utilisant la notation introduite précédemment. Par exemple, $\tau_{\text{width}(e)\mapsto k}$ correspond à l'endomorphisme définissant la valeur de la fonction `width (largeur)` pour l'élément $e \in E$ à k, et laissant tout le reste tel qu'il est.

On peut alors exprimer la propriété que tous les éléments dans une liste avec l'ID « menu »

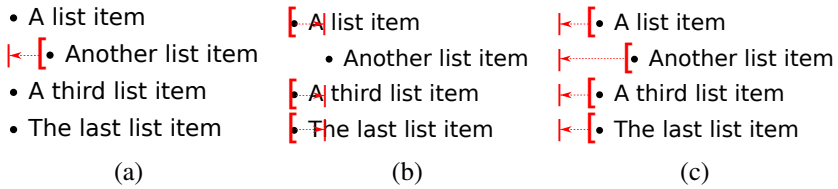


Figure 6.4 – Trois réparations pour l'exemple web

devraient être alignés à gauche comme l'expression de premier ordre suivante :

$$\forall x \in \$(\texttt{\#menu li}) : \forall y \in \$(\texttt{\#menu li}) : x.\texttt{left} = y.\texttt{left}$$

Notons que cette expression correspond directement à la traduction du premier ordre de l'expression de Cornipickle montrée dans la section 6.2.

Trouver les réparations principales pour cette expression et le fragment de page montré dans la figure 6.1a produit un certain nombre de solutions, dont trois sont montrées dans la figure 6.4. Les deux premières sont assez intuitifs. La figure 6.4a corrige la page en déplaçant l'élément désaligné de la liste avec les autres, alors que la Figure 6.4b fait le contraire, en alignant les trois éléments de liste les plus à gauche avec le second. La figure 6.4c donne un exemple de l'une des nombreuses solutions restantes ; dans ce cas, tous les éléments de liste sont déplacés vers une nouvelle position commune x, qui s'avère être une coordonnée qu'aucun élément n'avait dans la page d'origine.

Ce dernier exemple fournit une illustration graphique de la différence entre le concept original de témoin et celui de réparation. Alors qu'un témoin dans ce cas met en évidence une paire choisie au hasard d'éléments mal alignés (comme montré dans la figure 6.1b)), une réparation choisit des éléments spécifiques et, en plus, décrit ce qui doit être fait avec eux afin de réparer la violation de la propriété. Ceci est sans doute plus révélateur pour un utilisateur, et constitue

à notre avis l'un des principaux avantages de cette technique.

6.3 CALCUL DE LA RÉPARATION

Le concept de base de réparation présenté dans la section précédente se prête à quelques points de discussion. En particulier, le nombre de réparations principales possibles est potentiellement élevé, et la tâche de générer ces réparations peut donc s'avérer très coûteuse.

6.3.1 ALGORITHME DE BASE ET COMPLEXITÉ

L'algorithme montré dans 1 est un algorithme pour itérer toutes les réparations possibles d'une structure. Il énumère simplement toutes les transformations possibles : $T \in 2^{T_\Sigma}$. Il vérifie d'abord si T est bien défini (c'est-à-dire que toute paire d'endomorphismes commute), et si une réparation générée précédemment (stockée dans l'ensemble T_S) est un sous-ensemble de l'actuelle. Il vérifie enfin si l'application de cette transformation corrige la structure d'origine. Il passe à la prochaine transformation candidate, si l'une de ces trois situations se produit. Sinon, il ajoute cette transformation à son ensemble et le renvoie comme son prochain élément.

Theorem 1 *L'algorithme 1 est cohérent et complet.*

Soit T une sortie de transformation par l'algorithme. Par construction, T est une réparation, puisqu'elle est bien définie et elle corrige la valeur de vérité de σ dans φ. De plus, au moment où T est sorti, aucun des éléments de T_S n'est un sous-ensemble de T. Puisque T_S contient toutes les réparations de cardinalité inférieure à T, et que, par construction, toutes les transformations de cardinalité similaires ne peuvent pas être des sous-ensembles, il s'ensuit que T n'est incluse par aucune réparation existante et est donc principale. Cela prouve la

Algorithm 1 Algorithme générique pour l'itération des réparations primaires

```
procedure ComputeRepairs(φ, σ, 2^{T_Σ})
    T_S = ∅
    for all T ∈ 2^{T_Σ} do                          ▷ Énumérés en cardinalité croissante
        if ¬WellDefined(T) then
            skip
        end if
        if Subsumed(T, T_S) then
            skip
        end if
        if T(σ') ⊭ φ then
            skip
        end if
        T_S ← T_S ∪ {T}
        yield T
    end for
end procedure
```

cohérence de l'algorithme.

Le fait que toutes ces réparations majeures soient finalement énumérées est garanti par le fait que tous les sous-ensembles de T_Σ sont générés à un moment donné, ce qui prouve la complétude.

Cet algorithme a été implémenté en Java et est disponible publiquement [1]. En raison de sa simplicité et de sa généricité, l'implémentation des expressions, des structures et des itérations de réparation ne représente que 325 lignes de code. L'énumération des réparations est exposée à l'utilisateur sous la forme d'une classe classique Java `Iterator`, qui peut être utilisée par les méthodes traditionnelles `hasNext()` et `next()` pour passer à travers l'ensemble complet de réparations principales, dans l'ordre croissant de cardinalité. Les classes spécifiques au domaine définissants les constructions logiques propositionnelles et de premier ordre sont constituées d'environ 500 lignes de code supplémentaires. Il est facile de voir que le temps

1. `https://github.com/liflab/fault-finder`

d'exécution de cet algorithme est exponentiel en fonction de la taille de T_Σ, qui peut elle-même être exponentielle dans un autre facteur (dépendant du problème modélisé). Dans la logique du premier ordre (telle qu'utilisée par un fragment de Cornipickle), si $a_1, \ldots, a_n$ est l'arité respective de chaque prédicat dans la signature, le nombre d'endomorphismes est $\sum_i 2|A|^{a_i}$ pour un domaine donné A.

Malgré cela, il est possible de montrer que cet algorithme est limité par une borne inférieure théorique. Un ensemble d'endomorphismes T_Σ est dit *complet* si pour tout $\sigma, \sigma' \in \Sigma$, il existe une transformation bien définie $T \subseteq T_\Sigma$ tel que $T(\sigma) = \sigma'$.

Theorem 2 *Étant donné un ensemble de structures* Σ*, un ensemble d'expressions de langage* Φ *et un ensemble complet de transformations* T_Σ*, le problème du calcul des réparations principales est aussi difficile que le problème de satisfiabilité pour* Φ*.*

Soit $\varphi \in \Phi$ une expression du langage. Si φ est satisfaisable, alors il existe une structure $\sigma \in \Sigma$ telle que $\sigma \models \varphi$. Prenons une structure arbitraire $\sigma' \in \Sigma$. Puisque T_Σ est complet, il existe au moins une transformation $T \subseteq T_\Sigma$ telle que $T(\sigma') = \sigma$. Prenons le plus petit ensemble de ce genre ; par définition, il s'agit d'une réparation principale et sera finalement énumérée par l'algorithme 1. Puisque l'algorithme est complet, au contraire, aucune réparation ne sera trouvée si φ n'est pas satisfaisable.

6.3.2 RÉDUCTION DU NOMBRE DE SOLUTIONS CANDIDATES

Ces résultats de complexité de base justifient une discussion sur la réduction du nombre de réparations potentielles qui doivent être explorées.

Suppression des endomorphismes

Le nombre de transformations potentielles peut d'abord être réduit en supprimant les endomorphismes dont on sait qu'ils sont impossibles, en fonction du contexte. Par exemple, supposons que les symboles propositionnels a et b dans l'exemple 6.2.2 correspondent respectivement aux assertions « le client paie pour un objet » et « le client reçoit l'article ». On pourrait supposer qu'une évaluation où a est vraie ne peut pas être modifiée en la rendant fausse ; cela correspondrait au fait qu'une action effectuée par un acteur ne peut être annulée. Dans un tel contexte, seuls les endomorphismes réglant les fausses variables à vrai seraient considérés.

Dans le cas des graphes, comme dans l'exemple 6.2.2, on pourrait imposer des restrictions sur les changements qui lui sont autorisés ; par exemple, on pourrait dire que les arêtes existantes doivent rester inchangées, ou que seuls des sommets spécifiques peuvent être coloriés différemment. Ceci, encore une fois, a pour effet de préférer certaines transformations aux autres, et réduit globalement le nombre de réparations disponibles.

Transformations en groupes

La granularité des endomorphismes disponibles peut également être modifiée. Dans le cas de l'exemple de coloriage de graphe, il est évident qu'aucune réparation ne consistera jamais en un seul endomorphisme $\{\tau_{q_i(x)\mapsto\top}\}$. La raison est que l'expression φ_1 requiert que chaque sommet ait exactement une couleur ; assigner q_i à $\top$ pour un sommet implique que le q_j restant pour $j \neq i$ soit mis à $\bot$. On peut donc définir un nouvel ensemble de transformations

appropriées au contexte, représentant les *changements* de couleur :

$$T_C = \bigcup_{x \in A} \bigcup_{\substack{i \in [1,3] \\ j \neq i \\ k \neq j \neq i}} \{\{\tau_{q_i(x) \mapsto \top}, \tau_{q_j(x) \mapsto \bot}, \tau_{q_k(x) \mapsto \bot}\}\}$$

De même, comme la relation d'adjacence est symétrique, mettre $p(x,y)$ à $\top$ (resp. $\bot$) ne peut pas être fait sans mettre $p(y,x)$ à $\top$ (resp. $\bot$). Au lieu de considérer les changements individuels aux seules entrées de p, on peut définir un ensemble de *changements de bord* :

$$T_E = \bigcup_{x \in A} \bigcup_{y \in A} \bigcup_{b \in \{\top, \bot\}} \{\{\tau_{p(x,y) \mapsto b}, \tau_{p(y,x) \mapsto b}\}\}$$

On pourrait alors utiliser $T_C \cup T_E$ comme l'ensemble des transformations, au lieu de T_Σ. Bien que cela ne change rien à la théorie sur les solutions actuelles, le fait que $T_C \cup T_E$ soit plus petit que T_Σ a un effet positif sur la performance d'un algorithme d'énumération dans la pratique.

La même chose peut être dite des endomorphismes de l'exemple 6.2.2. Plutôt que de considérer tous les changements individuels des coordonnées (x,y) des quatre coins de chaque élément, on pourrait définir des sous-ensembles correspondants à des modifications plus intuitives ; par exemple, l'ensemble des *déplacements horizontaux* pourrait être défini comme :

$$T_H = \bigcup_{e \in E} \bigcup_{p \in P} \{\{\tau_{\text{left}}(e) \mapsto p, \tau_{\text{right}}(e) \mapsto (\tau_{\text{right}}(e) - p)\}\}$$

On peut alors limiter la recherche pour les réparations à celles qui sont faites uniquement des déplacements (horizontaux ou verticaux), ou des *redimensionnements* (horizontaux ou verticaux) d'éléments, etc.

(a) Le résultat attendu

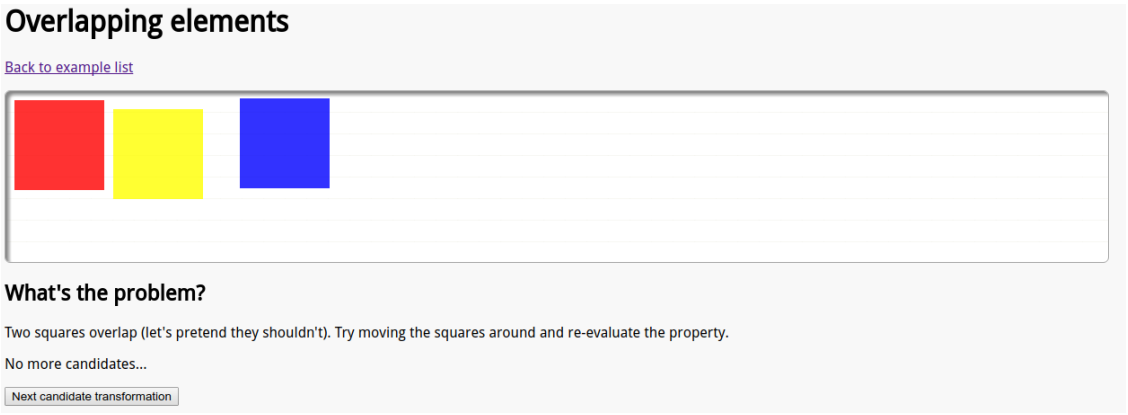
(b) Problème d'alignement.

Figure 6.5 – Éléments mal alignés : capture et suggestion de correction

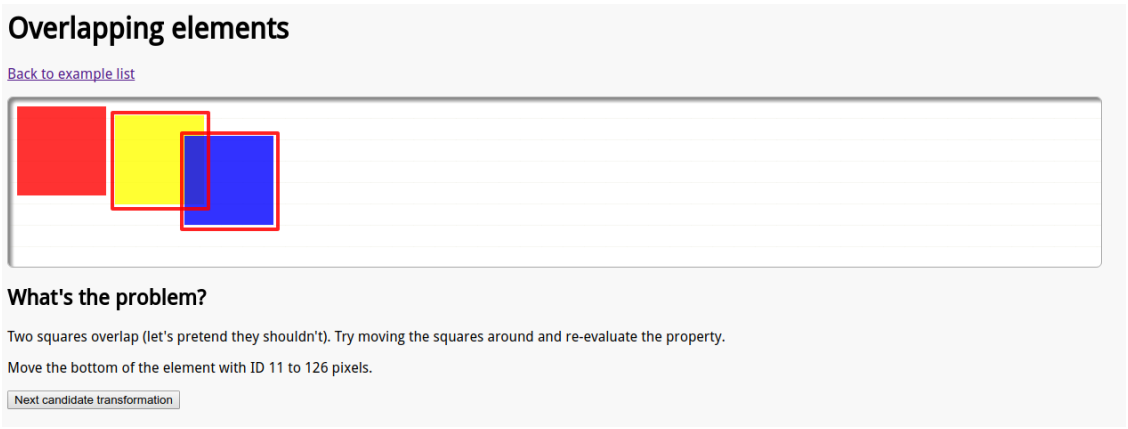
6.4 EXEMPLES

Les trois figures suivantes montrent des exemples simples de bugs montrant la capacité de l'outil à rechercher des candidats de correction en se basant sur l'approche proposée. Les figures montrent les verdicts qui sont des suggestions données par l'outil pour chacun des cas. Il est a noté que le processus prend entre 2 et 20 millisecondes pour trouver un candidat (sans prendre compte du temps d'évaluation).

Exemple 1 : éléments mal alignés Dans ce cas 6.5, la suggestion est de déplacer 69 pixels vers la gauche l'élément qui a l'identifiant ID=2.



(a) Le résultat attendu

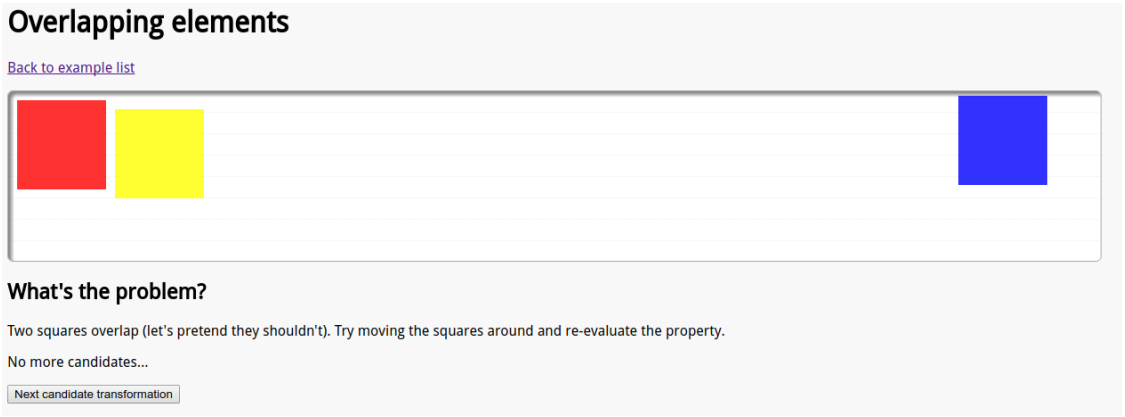


(b) Problème de chevauchement.

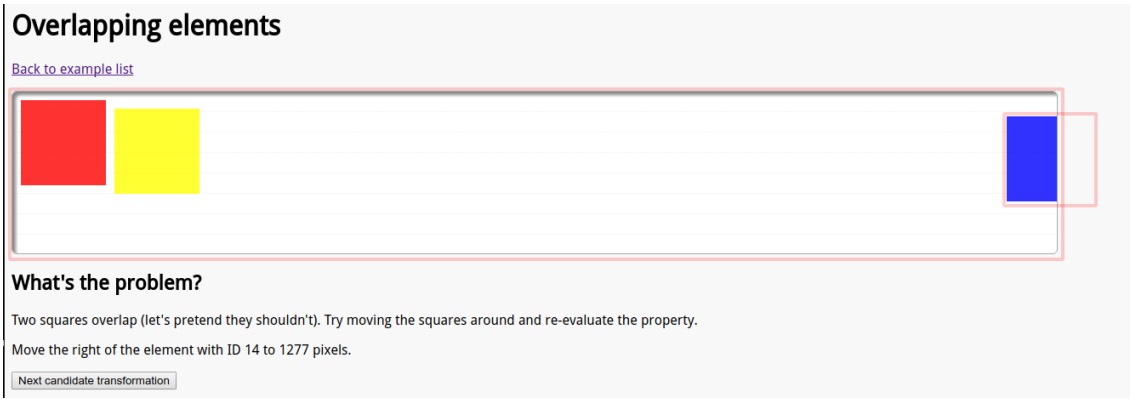
Figure 6.6 – Éléments qui se chevauchent : capture et suggestion de correction.

Exemple 2 : éléments qui se chevauchent La suggestion est de déplacer le bat de l'élément avec l'ID 11, à 126 pixels 6.6.

Exemple 3 : élément qui déborde de son conteneur. La suggestion est de déplacer la droite de l'élément avec l'ID 14, à 1277 pixels 6.7.



(a) Le résultat attendu



(b) Problème de débordement.

Figure 6.7 – Élément qui déborde de son conteneur : capture et suggestion de correction.

CHAPITRE 7

CONCLUSION GÉNÉRALE

Les applications web se multiplient et se diversifient et les exigences de leurs utilisateurs s'accentuent et s'amplifient avec. La relation application web-utilisateur est assurée uniquement via la page Web. Pour que cette relation soit tenue, la page Web doit être entretenue et répondre à un ensemble de critères : se charger rapidement, fournir le service désiré, et être agréable à voir sur tous les appareils : des ordinateurs de bureau ou portables, des tablettes et téléphones mobiles. Cependant, la complexité de la relation entre HTML, CSS et JavaScript, engendre des difficultés considérables pour la mise en page des applications web : le même document peut être affiché dans une variété de tailles, de résolutions, de navigateurs et même de périphériques, entravant, de ce fait, la mise en page. Les « bugs » de mise en page connaissent, par conséquent une présence remarquable, allant de problèmes de particularités relativement simple tels que des éléments superposés ou incorrectement alignés à des problèmes plus sérieux compromettant la fonctionnalité de l'interface utilisateur. Les tentatives, bien que rares, visant à résoudre ces problèmes restent incapables de cerner tous les aspects de ceux-ci (problèmes).

Nous distinguons, dans ce contexte deux familles d'approches servant à tester les interfaces des applications web : l'approche visuelle, se basant sur la comparaison des captures d'écran, pixel par pixel ; et l'approche déclarative : fonctionnant directement sur des informations sur

la mise en page. Si dans la première qui fonctionne mal avec les données dynamiques, le développeur de test se heurte à l'impossibilité de comparer des images de différentes tailles d'écran ; il doit se soumettre dans la seconde aux exigences des descriptions/scripts de test assez verbeux en décrivant les règles de son interface graphique.

L'approche que nous proposons, à savoir l'outil Cornipickle, offre les avantages suivants : elle fonctionne sur la majorité de combinaisons navigateurs/systèmes d'exploitation sans recourir aux plugins spécifiques au navigateur (ou limités par le navigateur). De plus, elle permet : 1. l'évaluation des spécifications en fonction des informations recueillies sur le client en se dispensant de l'évaluation statique de HTML et CSS du côté serveur ; 2. l'interprétation des spécifications de telle manière à éviter une charge de calcul excessive dans le navigateur ; 3. l'utilisation d'un autre langage que JavaScript pour l'implémentation ; 4. la gestion des propriétés comportementales impliquant plus d'un instantané de page par l'outil ; 5. la possibilité à l'utilisateur d'ajouter, de supprimer ou de modifier les spécifications évaluées par l'outil ; 6. l'exclusivité d'exprimer à travers un langage déclaratif des propriétés à propos du document (Document Object Model) et des propriétés CSS d'une page Web. 7. la potentialité de rechercher et détecter automatiquement les bugs comportementaux et RWD (Responsive Web Design) dans les applications web ; 8. la réparation en fournissant un algorithme de base pour calculer les transformations.

Le prototype de preuve de concept de Cornipickle a montré des résultats prometteurs dans sa capacité à exprimer facilement les conditions de bugs de mise en page dans les applications web, et à les détecter efficacement dans des exemples de pages de plus de 35 applications réelles.

L'efficacité de notre outil Cornipickle nous a permis d'attraper automatiquement certains problèmes communs rencontrés dans les applications web modernes (RIA et RWD). Les

propriétés de Cornipickle garantissent que les pages d'une application suivent différents types de contraintes, en particulier le séquençage possible des pages qui est l'objectif de ce volet de notre travail. En combinant les performances de Crawljax pour explorer les états de l'application (crawler state-aware) et un stateful test oracle (spécifications de logiques temporelles du premier ordre) dans Cornipickle, nous avons obtenu des résultats prometteurs. Une petite application a été développée et intégrée afin de tester le rendu visuel dans les différentes fenêtres possibles afin d'attraper les défauts RWD.

Notre solution a quelques limites et surmonter ces limitations pourrait être la base de futurs travaux. L'utilisation de Cornipickle nous limite à des contraintes se référant uniquement aux éléments qui sont affichés. Cela rend les bugs causés au niveau backend (côté serveur) parfois difficiles à attraper ; il est nécessaire de trouver les éléments affichés qui peuvent indirectement représenter les états du serveur. Dans la même ligne, si Crawljax ne notifie pas un changement d'état lorsque le DOM change, il n'est pas possible d'évaluer cette page où un bug aurait pu se produire. En outre, lorsqu'une propriété est évaluée à false, elle est fausse pour le reste de l'analyse et aucun autre bug ne peut être intercepté avec cette propriété. Cela a causé un problème avec la découverte de bugs RWD observables car la plupart des échecs ne sont pas observables et les propriétés devaient trouver un bug observable comme premier bug.

De plus, la capacité de Cornipickle à renvoyer une explication utile de la violation d'une propriété sur un document Web donné est limitée. C'est pourquoi nous avons introduit une définition du concept de réparation, dont le calcul fournit des informations plus précises sur les changements requis pour une structure afin de satisfaire une spécification donnée. L'étude des réparations et leur calcul fait partie du travail en cours, et de nombreux problèmes sont encore ouverts. Par exemple, un calcul efficace des réparations repose sur la suppression du plus grand nombre possible de transformations candidates ; par conséquent, des techniques pour identifier facilement des endomorphismes qui ne peuvent jamais faire partie d'une solution

pourraient être recherchées. De même, nous prévoyons d'étudier des techniques qui pourraient générer l'ensemble des réparations directement à partir de la spécification et de la structure défectueuse, plutôt que d'utiliser l'algorithme brut de génération et de test présenté.

Le concept de calcul des réparations est en cours de construction et il reste à établir ses liens avec les travaux connexes. Comme nous l'avons vu dans la section précédente, trouver des réparations concerne le concept de résolution de satisfiabilité (SAT), et plus précisément le problème du SAT incrémentiel [64]. Les solveurs SAT traditionnels sont nécessaires pour trouver un seul modèle d'expression. En SAT incrémentiel, un solveur trouve un premier modèle d'expression, mais peut également être demandé à plusieurs reprises de fournir des modèles supplémentaires. Lorsqu'un ensemble de transformations est terminé, l'itération sur les modèles revient à effectuer des itérations sur les réparations.

BIBLIOGRAPHIE

[1] Alm specifications examples. http ://auckland-layout.sourceforge.net/examples/index.html.

[2] Applitools visual test automation. http://www.applitools.comAccessed25April2016.

[3] Blackout repport. https://sites.hks.harvard.edu/hepg/Papers/NYISO.blackout.report.8.Jan.04.pdf.

[4] Bugs catastrophiques. http://www.slideshare.net/wearesocialsg/social-media-for-travel-brands/.

[5] critical-rendering-path. https://developers.google.com/web/fundamentals/performance/critical-rendering-path/.

[6] Définition du viewport. https://www.definitions-marketing.com/definition/viewport/.

[7] Exemple webspecwatij. https ://gist.github.com/tux2323/954186.

[8] Froont. http://froont.com/.

[9] Galen. http://www.swtestacademy.com/galen-framework/.

[10] galen framework. 2017. http://galenframework.com/.

[11] howbrowserswork. 2017. HTTP://taligarsiel.com/Projects/howbrowserswork1.html/.

[12] Html and css w3c standards. https://www.w3.org/standards/webdesign/htmlcss.

[13] Http response. https://www.w3.org/Protocols/rfc2616/rfc2616-sec6.html#sec6.

[14] http ://courses.cs.vt.edu/professionalism/therac_25/therac_1.html. http://courses.cs.vt.edu/professionalism/Therac_25/Therac_1.html.

[15] http ://wearesocial.sg/. http://wearesocial.sg/.

[16] http ://www.cnn.com/2003/us/08/14/power.outage/. http://www.cnn.com/2003/US/08/14/power.outage/.

[17] http ://www.yfolire.net/humr/humeur13.htm. http://www.yfolire.net/humr/humeur13.htm.

[18] Les bases de sahiscript. https ://sahipro.com/docs/scripting/sahi-scripting-basics.html.

[19] mobile and tablet internet usage exceeds desktop for first time worldwide. http://gs.statcounter.com/press/mobile-and-tablet-internet-usage-exceeds-desktop-for-first-time-worldwide.

[20] Phantomcss. 2017. https://github.com/Huddle/PhantomCSS.

[21] Principe de fonctinement de sahi. https ://www.thoughtworks.com/insights/blog/introduction-sahi-part-1.

[22] Respondr. http://respondr.io/.

[23] Responsinator. https://www.responsinator.com/.

[24] Responsivepx.com/. http://responsivepx.com/.

[25] Reuters ; us-blackout-newyork 2003. https://www.reuters.com/article/us-blackout-newyork/spike-in-deaths-blamed-on-2003-new-york-blackout-idUSTRE80Q07G20120127.

[26] Rwdbookmarklet. https://www.sitepoint.com/responsive-web-design-tool/.

[27] Screenfly. http://quirktools.com/screenfly/.

[28] Software bugs found to be cause of toyota acceleration death. https://www.google.fr/amp/s/www.computerworld.com/article/2573466/disaster-recovery/software-failure-cited-in-august-blackout-investigation.amp.html.

[29] Software failure cited in august blackout investigation. https://www.computerworld.com/article/2573466/disaster-recovery/software-failure-cited-in-august-blackout-investigation.html.

[30] Utilisation de capybara. https ://www.sitepoint.com/basics-capybara-improving-tests/.

[31] Vpresizer. http://lab.maltewassermann.com/viewport-resizer/.

[32] Washingtonpost. toyota reaches 12-billion settlement to end criminal probe.2014. https://www.washingtonpost.com/business/economy/

toyota-reaches-12-billion-settlement-to-end-criminal-probe/2014/03/19/5738a3c4-af69-11e3-9627-c65021d6d572_story.html?utm_term=.4826d81e2aa6.

[33] Watir. http ://watir.com/.

[34] websiteresponsivetest. http://www.websiteresponsivetest.com/.

[35] A. Arora and M. Sinha. Web application testing : A review on techniques, tools and state of art. *International Journal of Scientific I& Engineering Research*, 3(2) :1–6, 2012.

[36] K. Benjamin, G. Von Bochmann, M. E. Dincturk, G.-V. Jourdan, and I. V. Onut. A strategy for efficient crawling of rich internet applications. In *11th international conference on Web engineering ser.ICWE'11*, page 74–89. Heidelberg : Springer-Verlag, 2011.

[37] Tim Berners-Lee, Roy Fielding, and Larry Masinter. Uniform resource identifier (URI) : Generic syntax. Technical report, January 2005. RFC 3986.

[38] Oussama Beroual, Francis Guerin, and Sylvain Hallé. Searching for behavioural bugs with stateful test oracles in web crawlers. In *10th IEEE/ACM International Workshop on Search-Based Software Testing, SBST@ICSE 2017, Buenos Aires, Argentina, May 22-23, 2017*, pages 7–13, 2017.

[39] T.-H. Chang, T. Yeh, and R.C. Miller. Gui testing using computer vision. In *the SIGCHI Conference on H man Factors in Computing Systems, CHI '10*, pages 1535–1544, New York, NY, USA, may 2010. ACM.

[40] S. Choudhary, E. Dincturk, S. Mirtaheri, G.-V. Jourdan, G. Bochmann, and I. Onut. Building rich internet applications models : Example of a better strategy. In *Web Engineering, ser. Lecture Notes in Computer Science, F. Daniel, P. Dolog, and Q. Li*, volume 7977, page 291–305. Springer Berlin Heidelberg, 2013.

[41] S. Choudhary, M. E. Dincturk, S. M. Mirtaheri, A. Moosavi, G. von Bochmann, G.-V. Jourdan, and I.-V. Onut. Crawling rich internet applications : the state of the art. In *CASCON*, page 146–160. IBM Corp., 2012.

[42] Shauvik Roy Choudhary, Mukul R. Prasad, and Alessandro Orso. X-PERT : accurate identification of cross-browser issues in web applications. In David Notkin, Betty H. C. Cheng, and Klaus Pohl, editors, *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*, pages 702–711. IEEE / ACM, 2013.

[43] V. Dallmeier, M. Burger, T. Orth, and A. Zeller. Webmate : Generating test cases for web 2.0. In *D. Winkler, S. Biffl, J. Bergsmann (Eds.), SWQD*, volume 133 of Lecture Notes in Business Information Processing, page 55–69. Springer, 2013.

[44] M. E. Dincturk. "model-based crawling – an approach to design efficient crawling strategies for rich internet applications. Master's thesis, EECS - University of Ottawa, 2013.

[45] M. E. Dincturk, S. Choudhary, G. von Bochmann, G.-V. Jourdan, and I.-V. Onut. A statistical approach for efficient crawling of rich internet applications. *ICWE*, page 362–369, 2012.

[46] Mustafa Emre. *Model-based Crawling - An Approach to Design Efficient Crawling Strategies for Rich Internet Applications*. PhD thesis, University of Ottawa, 2013.

[47] Jesse James Garrett. Ajax : A new approach to web applications - adaptive path, 2005.

[48] Alan Grosskurth and Michael Godfrey. A case study in architectural analysis : The evolution of the modern web browser. *Software Maintenence and Evolution : Research and Practice.EMSE*, 2007.

[49] Allan Grosskurth and Michael Godfrey. A reference architecture for web browsers. *Software Maintenence and Evolution : Research and Practice*, page 1–7, 2006.

[50] A. Guttman. R-trees : a dynamic index structure for spatial searching. June 1984.

[51] Sylvain Hallé, Nicolas Bergeron, Francis Guerin, Gabriel Le Breton, and Oussama Beroual. Declarative layout constraints for testing web applications. *J. Log. Algebr. Meth. Program.*, 85(5) :737–758, 2016.

[52] Sylvain Hallé and Oussama Beroual. Fault localization in web applications via model finding. In *Proceedings First Workshop on Causal Reasoning for Embedded and safety-critical Systems Technologies, CREST@ETAPS 2016, Eindhoven, The Netherlands, 8th April 2016.*, pages 55–67, 2016.

[53] Sylvain Hallé and Roger Villemaire. Constraint-based invocation of stateful web services : The Beep Store (case study). In *4th International ICSE Workshop on Principles of Engineering Service-Oriented Systems, PESOS 2012, June 4, 2012, Zurich, Switzerland*, pages 61–62, 2012.

[54] S. Hallé, G. Le Breton, F. Maronnaud, A. Blondin Massé, and S. Gaboury. Exhaustive exploration of ajax web applications with selective jumping. In *ICST*, page 243–252. IEEE Computer Society, 2014.

[55] Arnaud Le Hors, Philippe Le Hégaret, Lauren Wood, Gavin Nicol, Jonathan Robie, Mike Champion, and Steve Byrne. Document object model level 2 core, 2000. http ://www.w3.org/TR/DOM-Level-2-Core.

[56] Jaakko Järvi, Mat Marcus, Sean Parent, John Freeman, and Jacob Smith. Algorithms for user interfaces. In *Proceedings of the Eighth International Conference on Generative*

Programming and Component Engineering, GPCE '09, pages 147–156, New York, NY, USA, 2009. ACM.

[57] Jaakko Järvi, Mat Marcus, Sean Parent, John Freeman, and Jacob N. Smith. Property models : from incidental algorithms to reusable components. In Yannis Smaragdakis and Jeremy G. Siek, editors, *GPCE*, pages 89–98. ACM, 2008.

[58] Sonal Mahajan and William G. J. Halfond. WebSee : A tool for debugging HTML presentation failures. In *8th IEEE International Conference on Software Testing, Verification and Validation, ICST 2015, Graz, Austria, April 13-17, 2015*, pages 1–8, 2015.

[59] Ethan Marcotte. *Responsive web design*. Eyrolles, 4 edition, 2013.

[60] A. Mesbah, A. van Deursen, and S. Lenselink. Crawling Ajax-based web applications through dynamic analysis of user interface state changes. *ACM Transactions on the Web (1)*, 6, 2012.

[61] S. M. Mirtaheri, D. Zou, G. V. Bochmann, G.-V. Jourdan, and I. V. Onut. Dist-ria crawler : A distributed crawler for rich internet applications. In *8th International Conference on P2P, Parallel, Grid, Cloud and Internet Computing*, pages 105–112. IEEE Computer Society Washington, 2013.

[62] Seyed M. Mirtaheri, Mustafa Emre Dincturk, Salman Hooshmand, Gregor V. Bochmann, and Guy-Vincent Jourdan. A brief history of web crawlers. In *CASCON '13 Proceedings of the 2013 Conference of the Center for Advanced Studies on Collaborative Research*, pages 40–54. IBM Corp. Riverton, NJ, USA ©2013, 2013.

[63] M.Tamm. Http response. `https://de.slideshare.net/MichaelTamm/fighting-layout-bugs`.

[64] Alexander Nadel and Vadim Ryvchin. Efficient SAT solving under assumptions. In *SAT*, pages 242–255, 2012.

[65] C. Olston and M. Najork. Web crawling. *Foundations and Trends in Information Retrieval*, 4 :175–246, 2010.

[66] Sean Parent, Mat Marcus, and Foster Brereton. ASL overview. Technical report, Adobe Systems, 2007. `http://stlab.adobe.com/group__asl__overview.html`.

[67] Pedro A. Szekely, Piyawadee Noi Sukaviriya, Pablo Castells, Jeyakumar Muthukumarasamy, and Ewald Salcher. Declarative interface models for user interface construction tools : the MASTERMIND approach. In Leonard J. Bass and Claus Unger, editors, *Proceedings of the IFIP TC2/WG2.7 Working Conference on Engineering for Human-Computer Interaction*, volume 45 of *IFIP Conference Proceedings*, pages 120–150. Chapman & Hall, 1995.

[68] Seyed M. Mir Taheri. *Distributed Crawling of Rich Internet Applications*. PhD thesis, University of Ottawa, 2015.

[69] Michael Tamm. Fighting layout bugs, 2009. `https://www.youtube.com/watch?v=WY3C6FHqSqQ`.

[70] Hideo Tanida, Mukul R. Prasad, Sreeranga P. Rajan, and Masahiro Fujita. Automated system testing of dynamic web applications. volume 303, page 181–196. Springer Berlin Heidelberg, 2013.

[71] te (testing experience) :The Magazine for Professional Testers. Test automation - does it make sense ? a simplified automation solution using watij. www.testingexperience.com.

[72] Thomas A. Walsh, Gregory M. Kapfhammer, and Phil McMinn. Automated layout failure detection for responsive web pages without an explicit oracle. In *Proceedings*

of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis, Santa Barbara, CA, USA, July 10 - 14, 2017, pages 192–202, 2017.

[73] Thomas A. Walsh, Gregory M. Kapfhammer, and Phil McMinn. Redecheck : an automatic layout failure checking tool for responsively designed web pages. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis, Santa Barbara, CA, USA, July 10 - 14, 2017*, pages 360–363, 2017.

[74] Thomas A. Walsh, Phil McMinn, and Gregory M. Kapfhammer. Automatic detection of potential layout faults following changes to responsive web pages (N). In *30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015, Lincoln, NE, USA, November 9-13, 2015*, pages 709–714, 2015.