UQAC
Université du Québec
à Chicoutimi

Correction automatique d'erreurs visuelles dans les applications web

par Xavier Chamberland-Thibeault

Mémoire présenté à l'université du québec à chicoutimi comme exigence partielle en vue de l'obtention du grade de Maître ès sciences en Informatique

Québec, Canada

RÉSUMÉ

Cela fait maintenant plusieurs années que les chercheurs se penchent sur le débogage des applications web, chose très complexes pour le développeur dû à l'intrication parfois étrange qu'ont les multiples langages utilisés pour leur conception. Les dernières années n'ont en rien facilité la tâche du débogage puisque les pages web ont de plus en plus fréquemment du contenu généré dynamiquement. Bien que plusieurs outils de la littérature offrent un grand appui pour la détection des bogues, très peu offrent de l'aide à la correction de ceux-ci. Dans ce mémoire sera présenté une étude sur l'évolution de la constitution des pages web permettant de mieux comprendre comment elles sont construites. S'en suivra la présentation d'un outil permettant de corriger plusieurs erreurs visuelles, préalablement détectées par l'outil Cornipickle, accompagné d'exemples réels de son application.

TABLE DES MATIÈRES

LISTE DES TABLEAUX

LISTE DES FIGURES

LISTE DES ABRÉVIATIONS

HTML HyperText Markup Language
SGC Système de Gestion de Contenu
CSS Cascading Style Sheet
DOM Document Object Model
MILP optimisation linéaire en nombres entiers
UQAC Université du Québec à Chicoutimi
OPL Optimization Programming Language

REMERCIEMENTS

Je tiens à remercier toutes les personnes qui m'ont épaulé lors de la rédaction de ce mémoire.

Je voudrais d'abord remercier mon directeur de mémoire M. Sylvain Hallé, professeur en informatique à l'UQAC, pour ses précieux conseils ainsi que tout le support et la confiance qu'il m'a accordés malgré le parcours atypique.

Je tiens aussi à témoigner toute ma reconnaissance aux personnes suivantes, sans qui la réalisation de ce mémoire n'aurait pu se faire :

Mes parents, pour leur soutien et encouragement tout au long de mes études.

Ma conjointe, qui a su me motiver, même les weekends, tout au long de la recherche et de la rédaction.

Thibault Mangold, pour les heures qu'il m'a accordées quand les algorithmes ne voulaient pas coopérer.

INTRODUCTION

Depuis l'avènement du tout premier site web(http://info.cern.ch/) à la fin du 20e siècle, la toile n'a cessé de croître et d'évoluer à un rythme effréné. Après environ 30 ans d'existence, le web est maintenant devenu une plaque tournante de notre quotidien : on y dénombre environ 800 000 nouveaux sites web créés chaque jour (pour un total de 1,78 milliard à l'heure actuelle) où les utilisateurs passent en moyenne 6 heures et 43 minutes quotidiennement [2]. De plus, la toile a connu de vastes changements dans sa diversité d'utilisation et ses méthodes d'accès. En effet, les blogues ont pris plus d'importance que les magazines, les sites de géolocalisation et de référencement ont remplacé les annuaires et les cartes routières, les utilisateurs font de plus en plus leurs achats via cette plateforme et les réseaux sociaux sont la nouvelle façon d'entretenir un lien avec ses contacts. Le tout est dorénavant sur une multitude d'appareils et des dizaines de navigateurs différents, allant du téléphone intelligent à l'ordinateur personnel [2].

Cette nouvelle multitude d'utilités, cette variété de navigateurs et cette pléthore d'appareils accédant aux sites web amènent leur lot de problèmes. Afin de répondre aux nouvelles fonctionnalités attendues par les utilisateurs et de suivre l'évolution des technologies, le code générant l'affichage des sites web devient de plus en plus complexe, notamment au niveau de la structure des pages et des interactions entre les différentes technologies composant les pages, soit le HTML, le CSS et le JavaScript. Les pratiques récentes amènent souvent les développeurs à concevoir des fragments de code qui génèrent dynamiquement des éléments HTML et CSS dans la page, n'aidant pas à simplifier les interactions entre les nœuds HTML du site et les propriétés CSS qui viennent en modifier l'apparence. Loin de simplifier le tout, il existe une grande diversité de navigateurs disponibles aux utilisateurs, navigateurs qui n'interprètent pas tous de la même façon les propriétés CSS. En plus de devoir gérer lesdites interactions parfois

assez complexes et les spécificités des navigateurs, les développeurs doivent faire face à un autre défi : la grande variété d'appareils accédant aux sites web, qui nécessite de développer des affichages qui peuvent s'adapter aux différentes tailles d'écran. La complexité du code mise de pair avec la panoplie d'affichages et de navigateurs à considérer, est propice à l'apparition des bogues d'affichages d'une complexité équivalente.

En plus d'être relativement complexes à corriger, ces erreurs d'affichage sont aussi très difficiles à détecter : il faut tester chacune des pages d'un site web sur plusieurs navigateurs et pour chacune des résolutions des appareils disponibles sur le marché. En conséquence, afin d'alléger le travail de détection et de correction, la communauté scientifique a développé plusieurs outils. Pour ce qui est de la détection de bogues, il existe une grande quantité d'outils différents et fonctionnels [3, 4, 5, 6] : certains outils permettent de déclarer et automatiser des tests comme, par exemple, Capybara, WebDriver et Watij, alors que d'autres, tels que WebSee ou PhantomCSS, utilisent une technique de comparaison de captures d'écran afin d'identifier les erreurs. D'autres encore, tels que Mastermind et Gallen Framework, vont préférer une approche déclarative [7, 8] qui consiste à déclarer, dans un langage spécifique, le comportement attendu de certaines parties de l'affichage pour ensuite valider que ce dernier est conforme aux attentes. Quelques-uns, comme Cornipickle, ont même adapté leur outil afin qu'il puisse tester l'entièreté du site automatiquement, avec un crawler (petits outils servant normalement à naviguer et archiver des documents web [8]).

Bien que l'identification d'erreurs visuelles dans des applications web soit bien avancée, il n'en va pas forcément de même pour leur correction. Plusieurs chercheurs se sont penchés sur le sujet, par exemple Mahajan *et al.* avec l'outil X-Fix [9], ou Hallé et Beroual [10] avec leur modèle générique pour corriger des erreurs abstraites. Cependant ces travaux partagent une même faiblesse : les corrections peuvent prendre plusieurs minutes à être générées, ce qui rend difficile l'application de correctifs à la volée.

Dans ce mémoire, nous nous pencherons donc sur la problématique qu'est la correction automatique, rapide et adéquate d'erreurs visuelles présentes dans les applications web. Ce texte présentera donc comment, en partant de l'outil Cornipickle [8] et de l'implémentation du modèle proposé par Hallé et Beroual [10], appelé Fault-Finder, nous avons trouvé une manière de générer et d'appliquer des corrections adéquates à des erreurs visuelles présentes dans des applications, moyennant l'utilisation d'un solveur numérique. Nous discuterons donc des premiers essais avec Cornipickle et Fault-Finder, de l'utilité et de l'utilisation d'un solveur numérique ainsi que des limitations de l'approche présentée.

Afin de démontrer que cette technique fonctionne, elle a été testée sur quelques exemples de sites fictifs ainsi que sur des erreurs dans deux sites réels. Suite aux résultats moins que convaincants de Fault-Finder, qui arrivait à trouver des solutions à des erreurs très simples en plusieurs dizaines de secondes, l'utilisation d'un solveur s'est imposée. Cet ajustement plus que nécessaire a permis de générer des correctifs adéquats dans des temps raisonnables et ce pour des erreurs visuelles simples ou un peu plus complexes, notamment lorsque la modification d'un élément a des répercussions sur le positionnement d'autres éléments, pouvant ainsi engendrer des erreurs en cascade. Ne restait plus qu'à travailler sur l'application adéquate de ces correctifs. Encore là, il fallait trouver une façon de modifier le style d'un élément afin de corriger l'erreur sans pour autant affecter les autres éléments dans la page et ainsi générer encore plus de bogues visuels. Le script JavaScript développé à cet effet, qui sera expliqué plus loin dans ce texte, permet d'appliquer les correctifs trouvés sans nuire à l'affichage du reste de la page web.

Ce mémoire est structuré comme suit : le chapitre 1 offrira de plus amples explications sur les applications web. La première partie de ce chapitre présentera diverses technologies utilisées dans le développement web dans le but de bien comprendre le fonctionnement de chacune d'entre elles ainsi que comment elles interagissent ensemble. La seconde partie du

chapitre offrira de plus amples détails sur différents types d'erreurs visuelles que l'on peut rencontrer dans des applications web, notamment comment ces erreurs surviennent ainsi que des exemples visuels pour mieux comprendre la problématique.

Le chapitre 2 présentera, en premier lieu, les différents outils et méthodes permettant de faire de la détection d'erreurs visuelles dans une page web. Ensuite, l'avancée des recherches quant à la correction de bogues d'affichage sera présentée. Nous prendrons la peine de regarder un éventail d'outils et de techniques qui sont présentement utilisés et discuterons des problématiques actuelles liées à la correction automatique d'erreurs visuelles.

Ensuite, tout au long du chapitre 3, les résultats d'une étude empirique sur un vaste échantillon de pages web seront présentés. En analysant la structure des pages web, cette étude permettra d'avoir une meilleure conception de la valeur des tests faits à la fois par les autres outils, autres techniques, mais aussi la technique présentée dans ce mémoire. Cela permettra aussi d'avoir un portrait de l'évolution de la structure et de la taille et composition des pages au fil des années. Le tout aidera à quantifier les performances des outils présents pour la correction de page web et à mieux cerner le profil des sites web contemporains. Autrement dit, pouvoir profiler les sites web actuels ainsi que les tendances d'évolution de la conception des sites et de la structure et répartition des éléments d'une page web nous permettra de mieux comprendre les enjeux réels liés à la correction et détection de bogues, notamment en lien avec la taille et complexité des arbres d'éléments composant une page.

Le chapitre 4 nous permettra de discuter des fondements de ce mémoire, soit la détection d'erreurs visuelles grâce à l'outil Cornipickle et des premières tentatives de corrections qui l'accompagnent. De plus amples informations sur ce qu'est Cornipickle et son fonctionnement seront offertes avant de parler des premiers tests de corrections d'erreurs avec Fault-Finder et montrer ainsi pourquoi il a fallu trouver une autre technique pour générer les correctifs.

C'est dans le chapitre 5 que seront présentées les contributions théoriques de cette recherche. Nous y discuterons, notamment, d'une méthode plus fonctionnelle de génération de corrections, notamment grâce au temps d'exécution moindre et à la prise en compte des effets d'une correction sur le reste des éléments de la page, de l'utilité et de l'importance du solveur numérique dans ce processus ainsi que de la technique utilisée pour réussir à appliquer lesdites corrections dans la page web sans générer d'autres erreurs visuelles ni avoir trop d'impacts sur le reste des éléments de la page.

S'en suivra une conclusion où nous présenterons un bref résumé du travail mené, discuterons de la qualité des résultats obtenus et évoquerons les travaux à venir ainsi les recherches complémentaires qu'il pourrait être intéressant de mener suite à cette recherche afin de d'optimiser le processus de détection et de correction d'erreurs visuelles dans les pages web.

La majeure partie du travail présenté dans ce mémoire a été publiée dans les articles qui suivent. Les deux premiers articles présentent l'étude empirique sur la structure et l'évolution des sites web. Le troisième et dernier article présente le travail de Stéphane Jacquet sur la modélisation du solveur numérique ainsi que des problèmes d'optimisation linéaire en nombres entiers (MILP) ainsi que mon travail sur l'interaction avec ledit solveur ainsi que l'application des correctifs dans une page web.

1. X. Chamberland-Thibeault et S. Hallé, "Structural profiling of web sites in the wild," *dans Web Engineering - 20th International Conference, ICWE 2020, Helsinki, Finland, June 9-12, 2020, Proceedings*, M. Bieliková, T. Mikkonen, et C. Pautasso, éds., vol. 12128. Springer, 2020, pp. 27–34
2. ——, "An empirical study of web page structural properties," *Journal of Web Engineering*, vol. 20, no 4, pp. 971–1002, Jul. 2021

3. S. Jacquet, X. Chamberland-Thibeault, et S. Hallé, “Automated repair of layout bugs in web pages with linear programming,” dans *Web Engineering - 21st International Conference, ICWE 2021, Biarritz, France, May 18-21, 2021, Proceedings*, M. Brambilla, R. Chbeir, F. Frasincar, et I. Manolescu, éds., vol. 12706. Springer, 2021, pp. 423–439

CHAPITRE I

BOGUES D'AFFICHAGE DANS LES SITES WEB

Les bogues d'affichage représentent le nerf de la guerre lors du développement web. Nécessitant une large batterie de tests afin d'être détectés et une compréhension parfois assez pointue des intrications entre les multiples technologies présentes dans le web, les erreurs visuelles, soit les rendus qui ne correspondent pas à ce que le développeur attendait, peuvent être un vrai fléau pour ceux qui les développent. Avant de pouvoir se lancer dans la présentation de la façon dont nous avons tenté de circonvenir au problème, il est important de bien comprendre quelles sont les technologies du web impliquées et ce qu'est un bogue d'affichage. Ce chapitre présentera donc les trois technologies principales du web ainsi qu'une description de chacun des grands types d'erreurs visuelles.

1.1 TECHNOLOGIES DU WEB

Depuis le début de ce texte, il est question de pages web et de bogues d'affichage causés par les interactions entre le HTML, le CSS et le JavaScript. Bien sûr, ce ne sont pas les seules technologies du web utilisées pour la création d'application web (il en existe bien d'autres, par exemple le PHP). Toutefois, pour les besoins de ce travail, nous discuterons uniquement des trois premières, puisque ce sont celles qui impactent le rendu de la page une fois dans le navigateur de l'utilisateur.

1.1.1 HTML

À la recherche d'une méthode pour faciliter la diffusion de l'information, Tim Berners-Lee, au sein du Conseil Européen pour la Rercherche Nucléaire (CERN), a travaillé sur le

développement du World Wide Web. Toutefois, pour partager de l'information à grande échelle, il lui fallait une façon standardisée de la représenter. Il donna donc naissance au HyperText Markup Language (HTML). Le HTML est le langage de base utilisé sur la toile : il sert à la publication de l'information. C'est grâce à ce dernier que les navigateurs arrivent à afficher un rendu, puisqu'il permet de définir comment on veut organiser l'information dans la page, par exemple avec l'utilisation des tableaux ou de listes à points. Il est à noter que, bien que le HTML permette de structurer l'information, le rendu final d'un même code HTML dépend fortement de l'application qui l'interprète. Tel que son nom l'indique, c'est un langage de balisage (*markup* en anglais), c'est-à-dire un langage proposant une annotation d'un fichier texte, via l'utilisation de balises prédéfinies, afin de lui donner une structure logique[11]. La première version de HTML ne contenait que quelques balises. C'est en 1992, avec la collaboration de Daver Raggett, que le HTML+, une version élargie de HTML, voit le jour. Depuis, ce langage n'a cessé d'évoluer : la version actuelle est HTML5.

Tel qu'on peut le voir dans la figure 1.1, une page HTML contient plusieurs balises, aussi appelées éléments ou nœuds. Chaque élément ou nœud possède une balise ouvrante, par exemple <*html*> et une balise fermante, le cas échéant <*/html*>. Ce code génère une page qui a pour titre « Titre de la page » et qui a pour contenu un paragraphe contenant le texte « Hello World ! ». La page peut se représenter sous forme d'arbre, tel que présenté dans la figure 1.2 : tous les éléments imbriqués entre la balise ouvrante et celle fermante d'un autre élément est donc un enfant ce celui-ci. Des imbrications multiples créent donc plusieurs niveaux à l'arbre HTML, aussi appelé arbre Document Object Model (DOM).

Absolument tous les sites web ont, pour le rendu final, une page HTML que le navigateur va afficher, ce qui fait de ce langage la base de l'affichage sur la toile. Toutefois, HTML ne fait qu'offrir des indications sur la structure de l'affichage de la page ainsi qu'un petit peu

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

Figure 1.1 : Exemple de code HTML. ©Xavier Chamberland-Thibeault, 2023

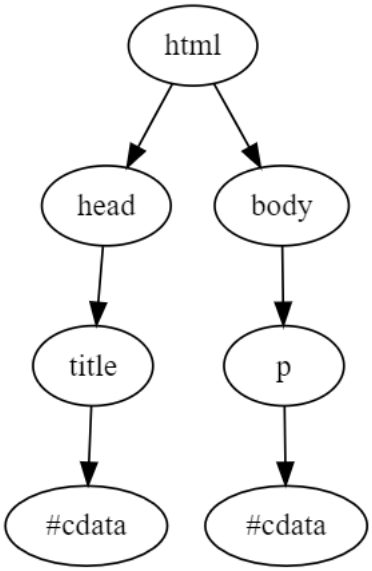


Figure 1.2 : Arbre HTML correspondant au code de la figure précédente. ©Xavier Chamberland-Thibeault, 2023

de design, par exemple indiquer si un élément doit être avant ou après un autre ou bien si le navigateur doit afficher une boîte de texte ou une chaîne de caractères. Afin de générer un visuel plus attrayant et d'avoir un meilleur contrôle sur le positionnement et le rendu des éléments présents dans une page HTML, on peut utiliser du CSS.

1.1.2 CSS

C'est en 1994 que Håkon Wium Lie eu l'idée de tenter d'ajouter à HTML des styles comparables à ceux des éditeurs de textes. C'est seulement deux ans plus tard que le *Cascading Style Sheet (CSS) 1*, soit CSS1, fit son apparition. L'apparition du CSS (Feuilles de Style en Cascade en français) fut une révolution pour le monde du web. Marc Andreessen avait déjà tenté d'ajouter de la stylistique aux pages web en ajoutant une couche supplémentaire au HTML [12]. Or, un nouveau langage à part, comme le CSS, permet de faire une meilleure séparation du code et le rend nettement plus clair. CSS, qui était considéré comme simple comparativement aux autres solutions de l'époque, prit ainsi son envol. Ce langage permettant de gérer tout ce qui a trait à l'affichage, de la taille du texte aux bordures d'un élément en passant par la gestion du positionnement, a, lui aussi, grandement évolué. Alors qu'avant ce langage ne servait qu'à faire de la stylistique comme les éditeurs de texte, le CSS permet maintenant de gérer presque automatiquement plusieurs éléments des sites web réactifs [1] et peut même générer des animations [2].

Alors qu'une page HTML propose une structure d'affichage précise, le CSS peut intervenir pour repositionner à sa guise les nœuds présents en plus d'en modifier l'affichage, et donc le comportement, de base. Qui plus est, le CSS offre aussi plusieurs éléments stylistiques qui automatisent des positionnements ou des affichages, par exemple *inline-block* ou bien *flex*.

1. En anglais Reactive web design
2. Exemple d'animations possibles avec uniquement du CSS https://youtu.be/CG__N4SS1Fc

```css
body{
    background-color: red;
}

#unId{
    border: 2px solid black;
    font-size: 22px;
}

.uneClasse{
    font-size: 24px;
}
```

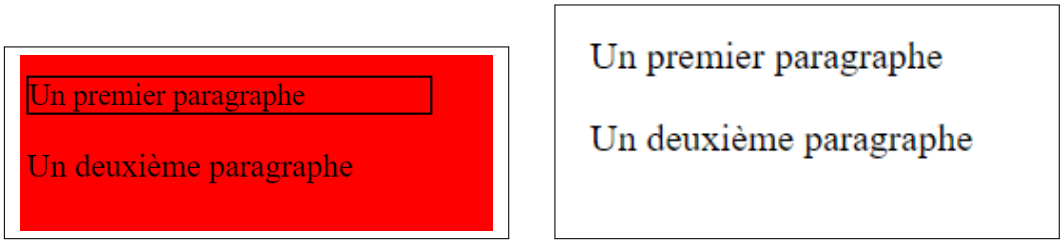
Figure 1.3 : Exemple de code CSS. ©Xavier Chamberland-Thibeault, 2023

Pour l'essentiel, CSS est un ensemble de règles d'affichage qu'on applique à un ou plusieurs éléments de la page HTML. Afin de déterminer à quel élément les règles seront appliquées, il faut utiliser des sélecteurs, notamment le type de balise HTML, l'identifiant d'une balise, la classe d'une balise ou une combinaison de ces sélecteurs. Une liste exhaustive des sélecteurs peut être trouvée sur W3Schools[13].

La figure 1.3 présente un exemple de code CSS. Dans cet exemple, la mise en couleur rouge du fond, soit le *background*, est appliquée à toutes les balises de type *body*. Le sélecteur d'identifiant est utilisé pour appliquer une couleur de bordure, soit *border-color*, et une taille de caractères, soit *font-size*, à l'élément qui a comme identifiant « gunId ». Le sélecteur de classe, quant à lui, est utilisé pour appliquer une taille de caractères à toutes les balises qui appartiennent à la classe « uneClasse ». La figure 1.4 présente le code HTML sur lequel pourrait être appliqué le code CSS vu dans la figure 1.3. La figure 1.5 montre la différence d'affichage du HTML quand le CSS est appliqué ou non.

```html
<!Doctype html>
<html>
    <head>
        <title>Titre de la page</title>
        <link rel="stylesheet" href="styles.css">
    </head>
    <body>
        <p id="unId" class="uneClasse">Un premier paragraphe</p>
        <p class="uneClasse">Un deuxième paragraphe</p>
    </body>
</html>
```

Figure 1.4 : Exemple de code HTML avec des sélecteurs pour appliquer du code CSS. ©Xavier Chamberland-Thibeault, 2023



(a) Avec CSS (b) Sans CSS

Figure 1.5 : Exemple de rendu du code HTML et CSS des figures 1.4 et 1.3. On peut voir en (a) le rendu si le code CSS est appliqué et en (b) s'il n'y a pas de CSS qui s'applique au code HTML. ©Xavier Chamberland-Thibeault, 2023

Bien que le CSS offre une grande lattitude au niveau de l'affichage et facilite la tâche des programmeurs grâce aux éléments stylistiques faisant le gros du travail à leur place, en rendant les pages web beaucoup plus attrayantes, il a aussi apporté son lot de problématiques. Effectivement, puisqu'un développeur peut déplacer comme il veut les éléments HTML dans la page grâce à du CSS, la structure suggérée par ledit code HTML peut donc être complètement différente de celle qui est réellement affichée. Quant au rendu réactif géré de façon presque autonome par le CSS, si le rendu n'est pas celui désiré, le programmeur peut avoir beaucoup de difficulté à identifier la partie du code problématique ainsi qu'à trouver le bon code CSS qui donnera le rendu désiré. De plus, il est compliqué de déterminer quelle règle CSS a préséance sur une autre. Tel que le montre l'exemple de HTML et CSS des figures 1.3 et 1.4, le premier paragraphe est soumis à deux règles contradictoires, celle qui le choisit au moyen de son identifiant, qui lui assigne une taille de caractère de 22 pixels, et celle qui le choisit par la classe à laquelle il appartient pour lui assigner une taille de caractères de 24 pixels.

D'autres notions, telles que la disposition des éléments dans la page sont également complexes à comprendre et à gérer. Il en existe plusieurs types : absolue, relative, fixe, flottante, collante et flexible. Chacun de ces types a son propre fonctionnement, par exemple fixe qui se positionne dans la page versus absolue qui se positionne par rapport au parent le plus proche ayant une disposition spécifiée, et des interactions entre elles qui diffèrent. C'est donc compliqué pour le développeur d'identifier quelle règle sera appliquée et, une fois déterminée, comment les éléments vont interagir entre eux. Une autre problématique qui survient avec le CSS est l'interprétation de celui-ci. En effet, malgré que le CSS soit normé, les navigateurs ne vont pas forcément donner le même rendu pour les mêmes règles CSS, complexifiant d'autant plus la tâche du programmeur. Un exemple de cette disparité de rendu est montré à la figure 1.6.

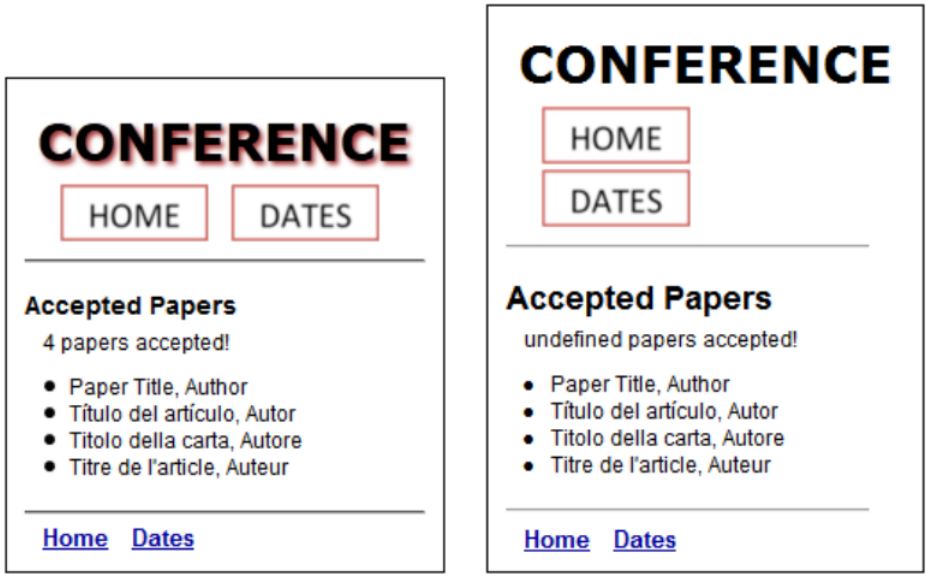


FIGURE 1.6 : Exemple d'erreur d'affichage qui peut arriver lorsque le même code HTML et CSS est interprété par deux navigateurs différents.[14] ©2013, IEEE

Tel qu'expliqué ci-avant, l'interaction entre HTML et CSS est complexe et produit parfois des résultats difficiles à prévoir et à corriger. Pourtant, à ce niveau, on ne parle que de pages dites statiques, que se passe-t-il lorsqu'en rend, en plus, les pages dynamiques ?

1.1.3 JAVASCRIPT

À la recherche d'une façon d'intégrer un langage de script, ce qui était très en vogue à l'époque, et afin d'accroître le dynamisme du site et l'expérience utilisateur, Netscape Communications Corporation a mis une équipe sur la question. En septembre 1995, une version bêta, LiveScript, de leur projet de recherche fut intégrée dans le navigateur. Mais, tel que l'expliquent Wirfs-Brock et Eich [15], ce n'est qu'en décembre 1995 qu'on vit réellement l'apparition de JavaScript avec la version 1.0. Netscape avait réussi son pari, soit de créer un langage de script, ressemblant à Java, beaucoup plus simple que ce dernier et qui serait basé sur les objets au lieu d'être basé sur les classes. Ce nouveau langage devait ainsi pouvoir permettre

```javascript
var hello = "Hello World!";

function sayHello()
{
  alert(hello);
}
```

Figure 1.7 : Exemple de code JavaScript. ©Xavier Chamberland-Thibeault, 2023

d'accéder à toute la puissance d'un langage de programmation, être facile d'apprentissage tout en tenant dans un navigateur en permettant la gestion des interactions entre un utilisateur et un site web. JavaScript permettait donc d'interagir avec l'utilisateur en répondant aux actions qu'il posait (voir la Figure 1.7 pour un exemple de code) et permettait d'accéder programmatiquement au code HTML afin de pouvoir l'adapter et le personnaliser à l'utilisateur en cours.

De nos jours, JavaScript est devenu un des plus gros outils du web. En 2019, on lui dénombrait 24 *frameworks* et 83 librairies [16]. Bien plus qu'un simple outil permettant de réagir aux actions posées par un utilisateur, JavaScript peut remplacer ou générer certaines parties d'une page web et, grâce à certaines librairies telle que NodeJs, peut même gérer tout ce qui a trait aux actions posées du côté du serveur et de la base de données. Bien qu'il ne travaille pas directement sur l'affichage, à proprement parler, de la page web, JavaScript reste une source de problèmes pour les erreurs visuelles. Effectivement, en générant des parties entières d'une page web, application du CSS compris, on vient modifier la structure de la page. Il est aussi possible d'aller modifier les règles de CSS déjà appliquées à la page, venant ainsi altérer le rendu défini par le HTML et CSS originaux. La figure 1.8 montre un tel code. Dans cet exemple, le code JavaScript va aller chercher dans la page l'élément qui a pour identifiant « unId ». Une fois trouvé, cet élément se fera imposer une couleur de bordure rouge et une taille de police de 18px, et ce quel qu'est été le CSS qui régissait ces deux aspects de son affichage

```
function changerParagraphe()
{
    var paragraphe = document.getElementById("unId");

    paragraphe.style.borderColor = "red";
    paragraphe.style.fontSize = "18px";

    paragraphe.innerHTML = "Du nouveau texte ";
    paragraphe.innerHTML += "<a href='#'>un lien</a>";
}
```

Figure 1.8 : Exemple de code JavaScript qui modifie le CSS d'un élément dans la page en plus de lui ajouter un élément enfant. ©Xavier Chamberland-Thibeault, 2023

avant. Une fois cette modification apportée, le JavaScript va ensuite aller modifier le contenu de l'élément : l'ancien contenu sera remplacé par la phrase « Du nouveau texte » auquel on ajoutera un hyperlien. Ainsi, ce simple bout de code JavaScript a modifié non seulement comment l'élément HTML est affiché, mais aussi son contenu et, par le fait même, l'arbre DOM de la page.

Alors qu'il fallait déjà tester les pages sur plusieurs navigateurs et sur plusieurs résolutions afin de couvrir toutes les interprétations possibles du CSS et les problèmes liés à l'espace disponible d'affichage, il faut en plus tester toutes les possibilités d'exécution du langage de script. Ceci revient à dire que, pour éviter de briser le rendu désiré d'une page, il faut tester chacune des parties du code ayant des impacts sur l'affichage. La figure 1.9 montre un bout de code qui modifie l'affichage d'un élément dans la page tout en dépendant de l'entrée reçue par la fonction. Si le texte « Bonne réponse » est reçu, l'élément aura une bordure verte avec une taille de police de 20px. Par contre, si ce n'est pas la bonne réponse, la bordure deviendra rouge, les caractères auront une taille de 18px et le texte sera maintenant ce qu'a reçu la fonction concaténé à « n'est pas la bonne réponse ». Pour cette seule fonction, il faudra au moins tester si la chaîne reçue est « Bonne réponse » ou pas et déterminer si ces deux possibilités

```
function afficherErreur(entreeUtilisateur)
{
    var paragraphe = document.getElementById("unId");
    var message = "";

    if(entreeUtilisateur != "Bonne réponse")
    {
        paragraphe.style.borderColor = "red";
        paragraphe.style.fontSize = "18px";
        message = entreeUtilisateur;
        message += " n'est pas la bonne réponse"
    }
    else
    {
        paragraphe.style.borderColor = "green";
        paragraphe.style.fontSize = "20px";
        message = "C'est la bonne réponse !";
    }

    paragraphe.innerHTML = message;
}
```

Figure 1.9 : Exemple de code JavaScript qui modifie différemment le CSS d'un élément dans la page selon l'entrée fournie par l'utilisateur. ©Xavier Chamberland-Thibeault, 2023

engendrent une erreur d'affichage. Qui plus est, il faut aussi tester avec différentes chaînes de caractères incorrectes : le rendu sera différent selon que la chaîne contienne deux ou cent caractères. Ainsi, alors que CSS et HTML présentaient déjà une intrication complexe à gérer, il faut en plus compter JavaScript puisqu'il a la possibilité de changer le rendu et la structure de la page au chargement tout comme en temps réel à la suite d'une action de l'utilisateur. C'est cette triple interaction qui rend la détection et la correction des bogues d'affichage très ardue pour les développeurs.

1.2 ERREURS D'AFFICHAGE

Tel que mentionné ci-devant, la conception d'une application web sans erreur d'affichage nécessite une compréhension et une maîtrise approfondie du HTML, du CSS, du JavaScript et de l'intrication entre les trois, sans compter le temps nécessaire afin de détecter les bogues d'affichages et trouver les correctifs possibles qui les accompagnent. Une erreur d'affichage, aussi appelée bogue d'affichage, est tout défaut présent dans le visuel de la page qui est présentée à l'utilisateur, soit le rendu produit par le navigateur. Ces bogues peuvent se manifester sous plusieurs formes. Plusieurs équipes de recherches ont travaillé à établir une classification des formes de bogues : Lelli *et al.*[17], Li *et al.* [18], Maji *et al.* [19], Yusop *et al.* [20], Mauser [21] ainsi que Francis Guérin [8]. Par souci de cohérence avec l'outil utilisé pour la détection de bogues, soit Cornipickle, ce mémoire ne tiendra compte que de la catégarisation présentée dans les travaux de Guérin. Selon cette catégorisation, il existe six grandes catégories de bogues : les problèmes d'alignement, les problèmes de chevauchement, les problèmes d'empilage, les éléments s'étendant en dehors de leur conteneur, les erreurs d'encodage et de mojibake et, finalement, les erreurs d'échappement. Dans cette section, chacune des catégories de bogues d'affichage sera présentée plus en détail. À noter que les images utilisées comme exemple pour chacun des types d'erreurs visuelles proviennent de la banque de données recueillies dans le cadre de l'article de Hallé *et al.* [22].

1.2.1 ÉLÉMENTS MAL ALIGNÉS

Le premier type de bogue, selon la classification de Guérin [8], survient lorsque deux éléments qui devraient être alignés horizontalement ou verticalement ne le sont pas. Dans certaines circonstances, comme dans la figure 1.10a, repérer l'erreur est rapide et facile pour le développeur. Dans cet exemple, les boutons permettant de lancer la musique devraient apparaître à côté de chacun des titres des pièces musicales, ce qui n'est pas le cas ici. Dans

d'autres cas, comme celui présenté dans la figure 1.10b, l'erreur se joue sur quelques pixels, rendant le tout plus difficile à identifier. Ici, bien que difficile à voir, le menu *Interests* est un tout petit peu plus bas que les autres menus.

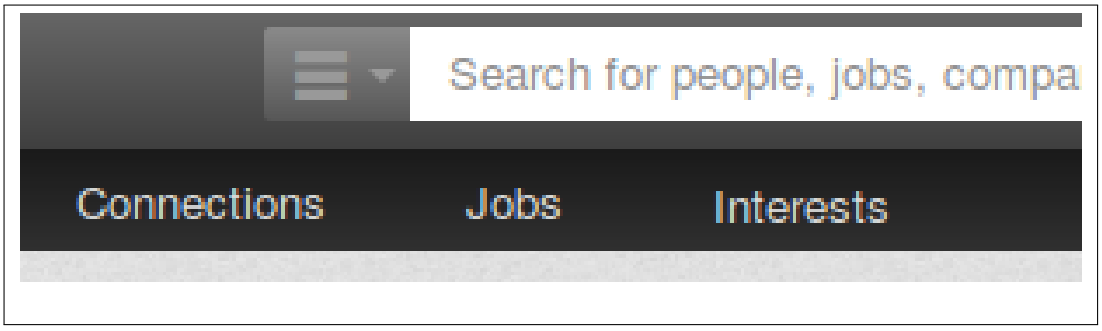
1.2.2 ÉLÉMENTS SE CHEVAUCHANT

Ce deuxième cas de figure se caractérise par la superposition de deux éléments qui ne devraient pas l'être. Ce type d'erreur visuelle peut se diviser en deux grandes catégories. Les bogues de la première catégorie, comme représentés par la figure 1.11a, surviennent lorsque le positionnement de l'élément est incorrect. Dans ce cas-ci, les boîtes de sélection des lieux de départ et d'arrivée sont trop basses et chevauchent donc les options plus basses. Il a été remarqué que ce positionnement fautif venait souvent de l'attribut CSS *position :absolute* [8]. Les bogues de la deuxième catégorie, tels que présentés dans la figure 1.11b, sont dus à un changement de la taille de la chaîne de caractères de l'élément dû au changement de langue de la page. Tel que l'exemple le montre, la chaîne de caractères en français « *connexion* » est beaucoup plus longue que celle en anglais, soit « *login* ». Par la simple différence de taille engendrée par 4 lettres, le bouton empiète sur son voisin de droite créant ainsi un bogue visuel aisément identifiable par l'utilisateur.

1.2.3 ÉLÉMENTS MAL EMPILÉS

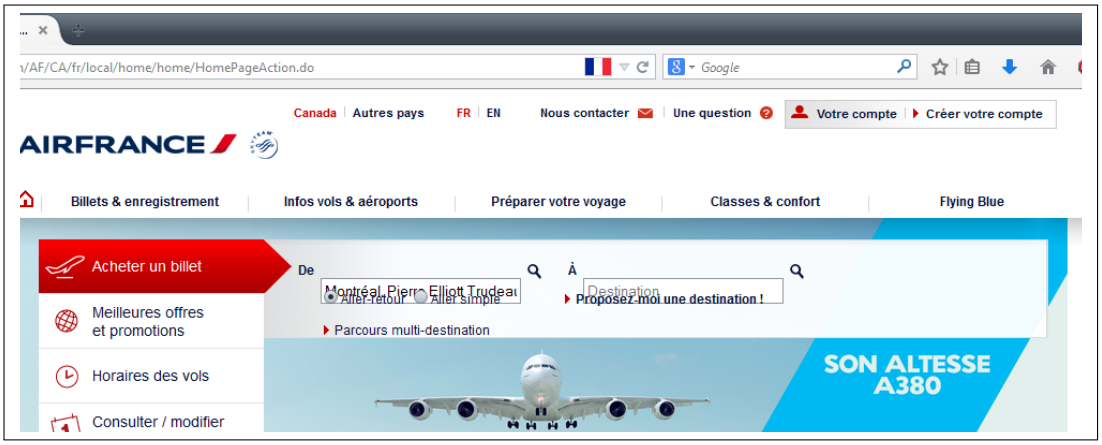
Ce type d'erreurs visuelles survient lorsqu'un élément qui devrait apparaître par-dessus un autre, notamment des *Pop-up*, apparaît en dessous. Dans la figure 1.12, l'utilisateur clique sur le champ permettant d'ajouter des personnes avec qui partager le projet. Le site Bitbucket

(a) Site : 2Frères.

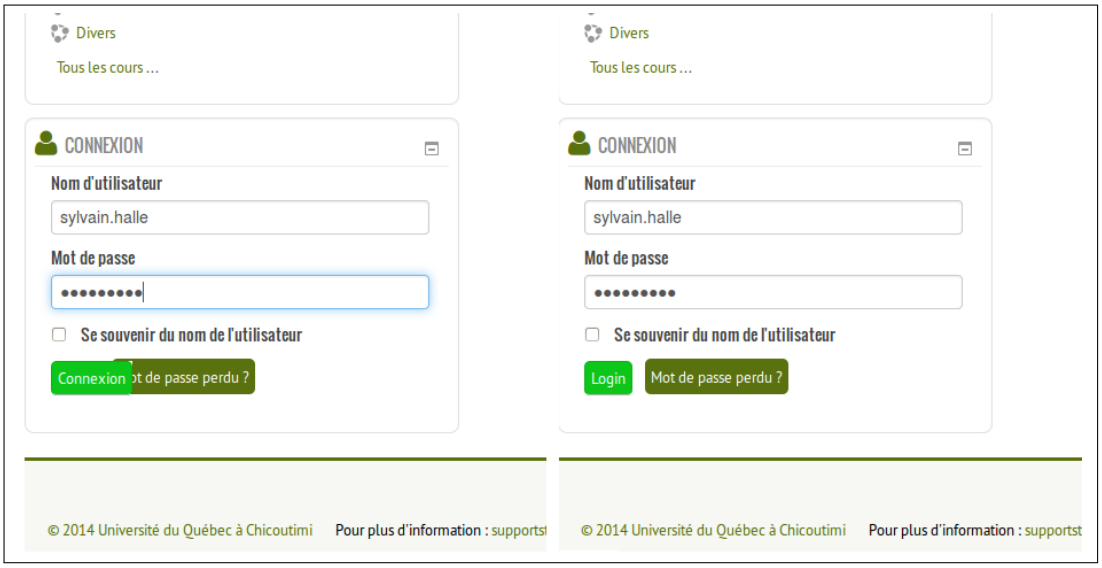


(b) Site : LinkedIn.

FIGURE 1.10 : Exemples de mauvais alignement d'éléments : (a) Les boutons pour jouer la musique ne sont pas alignés avec les chansons correspondantes, (b) *Interests* n'est pas aligné avec le reste des éléments du menu.[22] Autorisation obtenue par Elsevier.



(a) Site : AirFrance.



(b) Site : Moodle.

Figure 1.11 : Exemples de chevauchement d'éléments : (a) Les boîtes de sélections déroulantes et les textes qui devraient être en dessous se chevauchent, (b) le bouton connexion est superposé au bouton à sa droite en français, mais pas en anglais.[22] Autorisation obtenue par Elsevier.

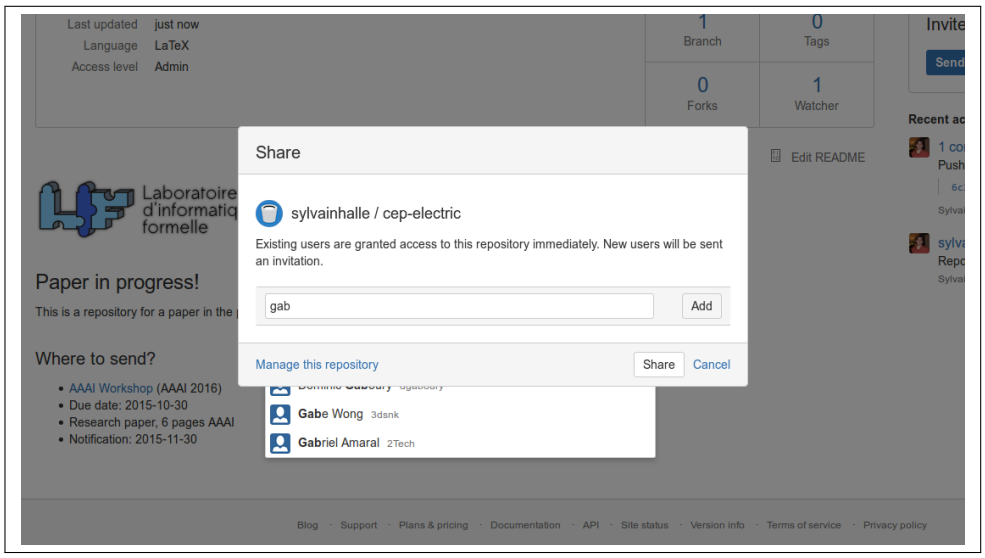
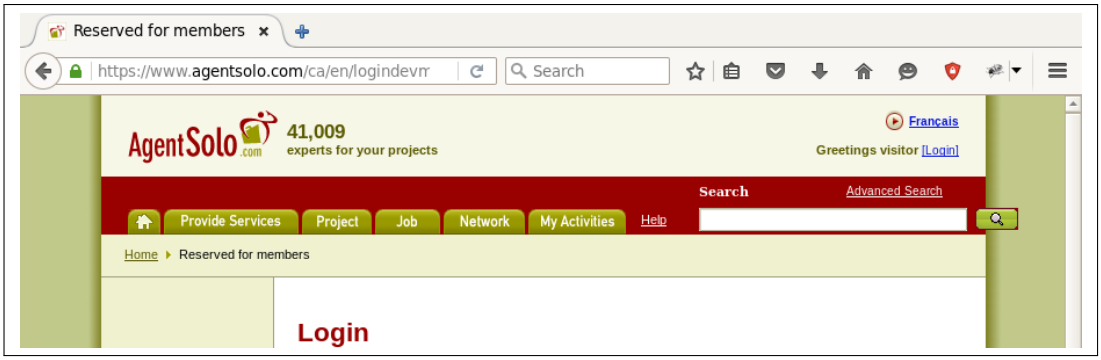


FIGURE 1.12 : Exemple d'empilement d'éléments : la liste de sélection des usagers à qui partager le projet apparaît en dessous de la boîte de partage.[22] Autorisation obtenue par Elsevier.

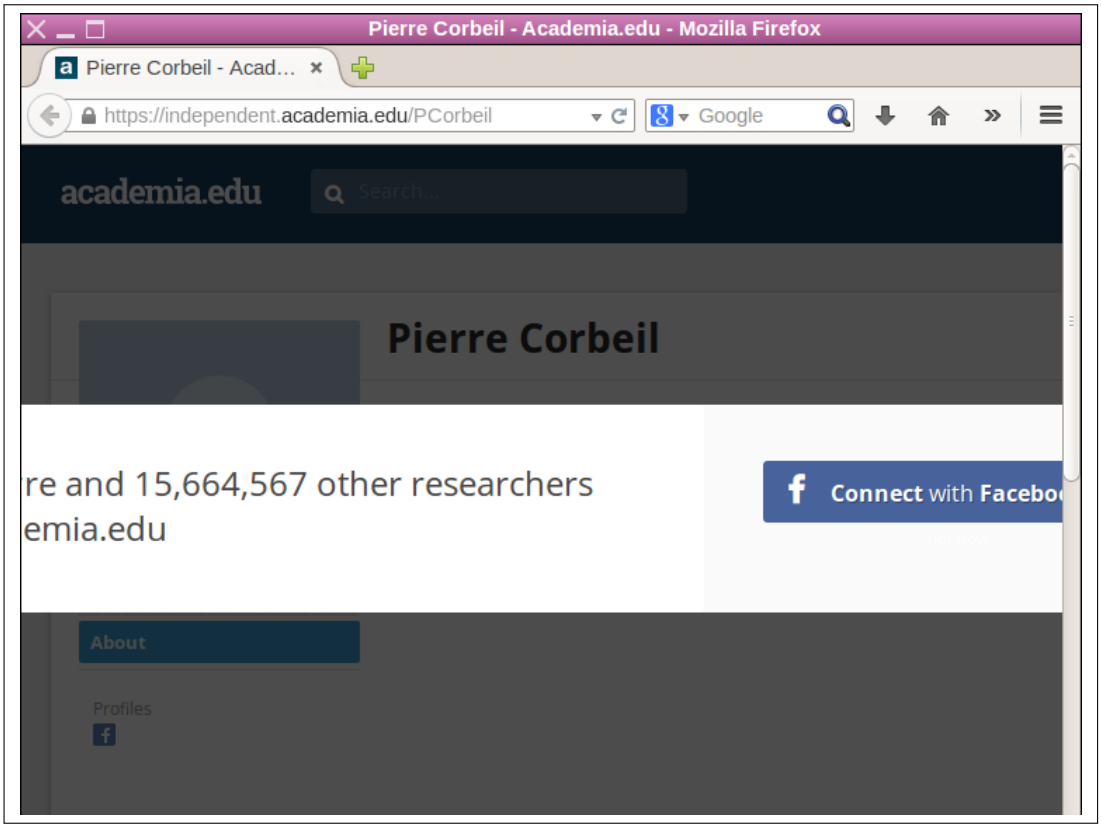
offre une aide à la saisie en faisant apparaître une liste d'usagers connus, or cette liste apparaît en dessous de la boîte de partage alors qu'elle aurait dû apparaître au-dessus.

1.2.4 ÉLÉMENTS S'ÉTENDANT À L'EXTÉRIEUR DE LEUR CONTENEUR

Cette quatrième catégorie de bogues visuels survient lorsqu'un élément de la page qui devrait être contenu par un autre élément, ou même par la page elle-même, ne l'est pas. La figure 1.13a représente le premier cas de figure : le bouton de recherche situé à côté de la barre de recherche sort du cadre rouge dans lequel il est supposé se trouver être. Quant à elle, la figure 1.13b est un exemple clair du deuxième cas de figure puisque le côté gauche de la boîte de dialogue affichée dépasse du cadre de la page et est même totalement inaccessible étant donné que la barre de défilement ne peut pas aller plus à gauche.



(a) Site : AgentSolo.



(b) Site : Academia.edu.

FIGURE 1.13 : Exemples d'éléments s'étendant à l'extérieur du conteneur : (a) Le bouton rechercher, en forme de loupe, apparaît en dehors du cadre rouge, (b) l'encadré affiché est coupé.[22] Autorisation obtenue par Elsevier.

1.2.5 MOJIBAKE ET ERREURS D'ENCODAGE

Ce cinquième type d'erreur ne se rapporte pas au positionnement d'un élément dans la page, mais plutôt à son contenu. Lors de la création d'un site web, on peut en gérer l'encodage des caractères, par exemple UTF-8 ou ASCII. L'encodage choisi déterminera comment la gestion des caractères « étrangers » ou « accentués » se fait. Or, lorsque le type d'encodage ne gère pas certains caractères, l'affichage de ceux-ci sera compromis, comme dans la figure 1.14 où ces caractères sont remplacés par des icônes de point d'interrogation, c'est ce qu'on appelle un mojibake. Cette même erreur peut aussi survenir lorsque le caractère n'est pas lu selon le bon encodage. Puisqu'il y a plusieurs façons d'encoder le même caractère, donc plusieurs séquences d'octets différentes pour représenter le même caractère, il peut y avoir des erreurs lors de l'affichage des caractères si l'interpréteur s'attend à un encodage différent de celui reçu.

1.2.6 ERREURS D'ÉCHAPPEMENT

Le sixième et dernier type d'erreur visuelle traite lui aussi du contenu des éléments d'une page. Toutefois, au lieu d'interférer avec l'affichage du contenu, ces bogues vont altérer le contenu en tant que tel. Lors de la création d'une page web, ce ne sont pas toutes les chaînes de caractères qui sont codées en dur. Plusieurs de ces chaînes vont provenir de sources externes, telles que des bases de données, ou bien être générées par le code. Or, si les bouts de code contiennent des erreurs, des bogues tels que celui de la figure 1.15 surviendront. Ici, au lieu d'afficher les données du prix par mois, le bogue fait en sorte que le navigateur affiche littéralement le code qui sert à générer ou récupérer la valeur voulue, notamment la ligne suivante dans la figure 1.15 : *$data.get($lang).dollarSignEn28$data.get($lang).decimal95 $data.get($lang).dollarsignFr$data.conditions*.

FIGURE 1.14 : Exemple d'erreur d'encodage : Les caractères spéciaux tels que le « é » et le « à » sont remplacés par des icônes de point d'interrogation.[22] Autorisation obtenue par Elsevier.

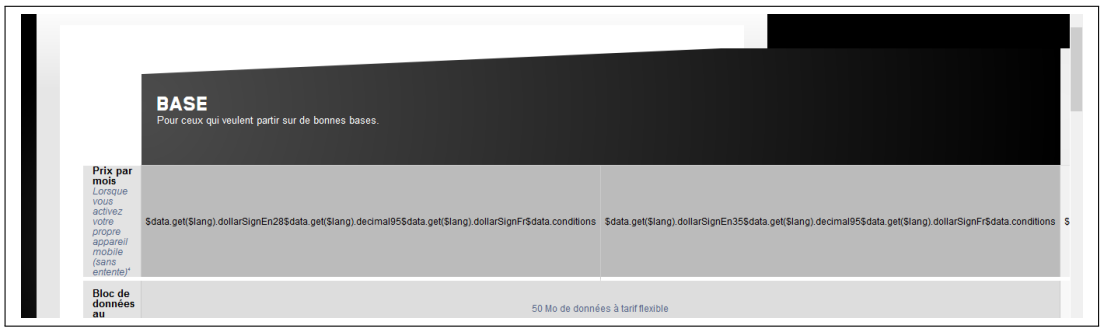


FIGURE 1.15 : Exemple d'erreur d'échappement : Au lieu d'afficher les prix, la ligne « Prix par mois » affiche le code *PHP* qui permet d'aller récupérer l'information voulue sur le serveur. [22] Autorisation obtenue par Elsevier.

L'outil présenté dans le cadre de ce texte travaille uniquement à régler les quatre premiers types d'erreurs visuelles. Les erreurs d'encodage et celles d'échappement ne seront donc pas prises en compte. Les erreurs d'échappement demanderaient une analyse profonde du code qui génère la page, or l'outil présenté analyse uniquement le rendu réel de la page, il est donc impossible, pour le moment, de traiter ce type d'erreur. En ce qui concerne les erreurs d'encodage, elles présentent un problème qui est trop complexe à résoudre automatiquement et ont donc été écartées du projet. En effet, pour résoudre ce type d'erreur, il faudrait analyser le texte pour en déduire la langue puis, identifier tous les mots auxquels pourraient correspondre ceux qui sont mal affichés puis, selon le sens de la phrase, choisir laquelle de ces options est la bonne à afficher.

CHAPITRE II

ÉTAT DE L'ART EN GESTION DES BOGUES D'AFFICHAGE

Maintenant que nous avons donné une définition claire de ce qu'est le web et des types de bogues d'affichage qui seront abordés dans ce texte, il est temps de présenter les recherches existantes sur la détection et la correction de bogues d'affichage. Plusieurs outils ou algorithmes ont été proposés pour soit tester soit corriger les erreurs visuelles. Dans ce chapitre, ces outils seront brièvement présentés avec un accent mis sur ceux qui s'adressent aux applications web.

2.1 OUTILS DE DÉTECTION

Avec la complexification des environnements informatiques et des logiciels, les erreurs sont de plus en plus fréquentes et compliquées à trouver. Ce faisant, la recherche d'une façon d'identifier aisément ces erreurs est devenue très attrayante. Ceci est d'autant plus vrai pour le domaine du développement web, dans lequel il y a des interactions complexes entre plusieurs langages en plus de devoir gérer une multitude de navigateurs différents, et donc d'interpréteurs distincts, et des affichages tout aussi variés.

2.1.1 CRAWLJAX

Un premier outil qui offre un soutien aux développeurs pour tenter de détecter des bogues d'affichage est le crawler qui permet d'exécuter des tests rudimentaires. Cet outil permet d'automatiser le déplacement à travers un site web ainsi que d'automatiser le changement de la taille d'affichage de la page web. En plus de permettre une navigation automatique sur des résolutions variables, les crawlers permettent l'exécution de code JavaScript lors

du déclenchement de certains événements tel que le chargement de la page. Un crawler communément utilisé dans la littérature est l'outil de Mesbah *et al.* : Crawljax [23].

Contrairement aux crawlers traditionnels qui associaient une URL à une page unique, les rendant donc presque obsolètes vis-à-vis des sites dynamiques dont le contenu des pages est constamment modifié par du JavaScript, Crawljax récupère une structure DOM et l'associe à une page unique. Ainsi, au lieu de simplement tester plusieurs URL sur plusieurs tailles d'affichage différentes, Crawljax analyse la page afin d'identifier tous les événements qui pourraient modifier le DOM ou la page en entier. Une fois ces éléments trouvés, Crawljax génère un graphe montrant tous les états possibles du site tout en conservant, pour chaque état, la structure DOM de la page. Crawljax permet ainsi de tester les sites web dynamiques, que ce soit sur les tailles d'affichage ou sur les changements de page ou de contenu, dans leur entièreté, permettant de pallier aux lacunes qu'avaient les crawlers qui l'ont précédé et d'aider les développeurs à automatiser au maximum la détection de bogues sur leurs sites web. Pour utiliser Crawljax, les développeurs n'auront qu'à lui fournir l'URL de base à tester ainsi qu'une série de paramètres représentant les configurations de tests désirés. Par exemple, il est possible de lui spécifier une durée maximale d'exécution ou bien une profondeur maximale à ne pas dépasser.

2.1.2 APOLLO

Afin de pouvoir mieux travailler avec les pages web générées dynamiquement, chose omniprésente dans les applications web d'aujourd'hui, Artzi *et al.* ont développé l'outil Apollo [24, 25]. Cet outil permet de générer des entrées de tests pour les applications web, valider le résultat HTML et surveiller si l'application plante.

Afin d'accomplir cela, Apollo débute son exécution en recevant une page web ou requête HTTP de départ représentant son état initial. C'est depuis cet élément de base qu'Apollo détermine les tests à exécuter. Ensuite, l'outil génère des combinaisons de configurations à tester. Une configuration, dans ce cas-ci, est représentée par des entrées utilisateur générées par Apollo associées à des contraintes relatives à l'accès de la page ou de la requête HTTP, c'est-à-dire quelle URL il faut rejoindre et avec quel passage de paramètres. Une fois ces tests déterminés, un oracle est utilisé afin de vérifier la mise en pratique de chacun des tests et récupérer des informations sur les erreurs rencontrées, le cas échéant. L'outil analyse ensuite les données reçues de l'oracle afin d'identifier les types de problématiques rencontrées, telle qu'une balise fermante manquante ou bien des erreurs de programmation comme un attribut manquant ou des éléments non-définis, et en identifier quelle contrainte fait en sorte que l'exécution observée est une erreur. Une fois toutes ces données recueillies, Apollo produit des rapports à raison d'un rapport par erreur rencontrée. Chacun de ces rapports contient l'erreur rencontrée, le message d'erreur fourni par l'interpréteur du code, la ou les contraintes qui sont violées par l'erreur ainsi que toutes les séquences de test qui ont mené à cette erreur. La figure 2.1 montre la structure de l'outil ainsi que l'interaction entre ses différentes parties.

Afin de générer des tests appropriés, Apollo détermine et résoud des contraintes relatives au flux de contrôle que le code suggère. Entre autres, il se fie à certains éléments spécifiques au langage PHP, comme le *header*, qui permet la redirection de page, le *isset* qui permet de valider si une donnée à été reçue, le *require*, qui demande l'import d'un fichier autre sous peine d'arrêter l'exécution du script, etc. Ainsi, en analysant le code, l'outil est capable d'identifier quels seraient les chemins à suivre et que devrait être une exécution normale dudit code.

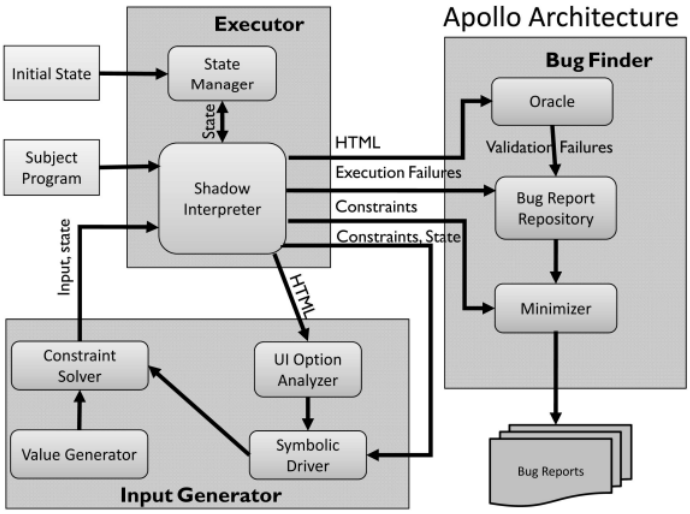


FIGURE 2.1 : Représentation de la structure de l'outil Apollo où il est possible d'observer chacune des parties ainsi que les interactions qu'elles ont entre elles.[25] ©2010, IEEE Autorisation obtenue par IEEE.

2.1.3 WEBDIFF

Roy Choudhary et al, quant à eux, se sont penchés sur le problème des incompatibilités entre navigateurs. Pour ce faire, ils ont commencé par développer l'outil WebDiff[6]. Cet outil utilise un navigateur de référence et y récolte l'arbre DOM de la page web, incluant plusieurs caractéristiques sur chaque élément tel que montré dans la figure 2.2, ainsi qu'une capture d'écran. La même chose est ensuite faite pour tous les autres navigateurs qui sont à valider. Par la suite, les arbres de DOM sont épurés : grâce aux captures d'écrans l'outil va identifier dans l'arbre les éléments qui sont dynamiques et variables, par exemple les publicités. Les nœuds ainsi détectés sont marqués comme étant variables et ne sont donc pas pris en compte lors de la suite du processus. Ensuite, les arbres DOM sont comparés à celui du navigateur de référence pour tenter d'identifier toute différence entre deux éléments qui devraient être identiques, soit tous les éléments qui n'ont pas été marqués comme variables. Chacun des éléments non-concordants est donc considéré comme une erreur visuelle.

```
// Node n1, from Mozilla Firefox's DOM
{ "tagname":"LI", "id":"", "xpath":"/HTML/BODY/.../UL/LI",
  "coord":"140,60", "hash":"0xA56D3A1C", "clickable":"0",
  "visible":"1", "zindex":"0" }

// Node n2, from Internet Explorer's DOM
{ "tagname":"LI", "id":"", "xpath":"/HTML/BODY/.../OL/LI",
  "coord":"140,60", "hash":"0x1C394C1B", "clickable":"0",
  "visible":"1", "zindex":"0" }
```

FIGURE 2.2 : Exemple d'informations récoltées par l'outil WebDiff lorsqu'il parcourt et récupère l'arbre DOM d'une page web.[6] ©2010, IEEE Autorisation obtenue par IEEE.

Une amélioration à WebDiff a aussi été présentée par Roy Choudhary *et al.* : CrossCheck[26]. Premièrement, CrossCheck améliore la comparaison des captures d'écran en implémentant un système d'apprentissage automatique afin de remplacer les heuristiques qu'utilisait WebDiff. Ce changement rend l'outil plus polyvalent. De plus, disposant d'un système de traitement des images plus puissant, la détection des erreurs ne se fait plus seulement avec la comparaison des arbres DOM, mais en comparant aussi les images entre elles. Finalement, Crawljax a été intégré à CrossCheck afin de pouvoir capturer les différences d'affichage lors du lancement et de l'exécution d'événements utilisateur ainsi que de pouvoir parcourir, et donc évaluer, l'entièreté du site. Lors de son parcours et de ces tests d'événements utilisateur, Crawljax récupère les captures d'écran et l'arbre DOM à chacune des étapes. Ainsi, CrossCheck peut faire des validations sur tous les états possibles d'une même page web et également parcourir l'entièreté du site web. Finalement, un algorithme de regroupement d'erreurs a été ajouté afin de produire des rapports finaux présentant des erreurs plus faciles à comprendre pour le développeur et surtout en listant des erreurs plus significatives. La figure 2.3 montre un aperçu du processus que suit CrossCheck sur un site web.

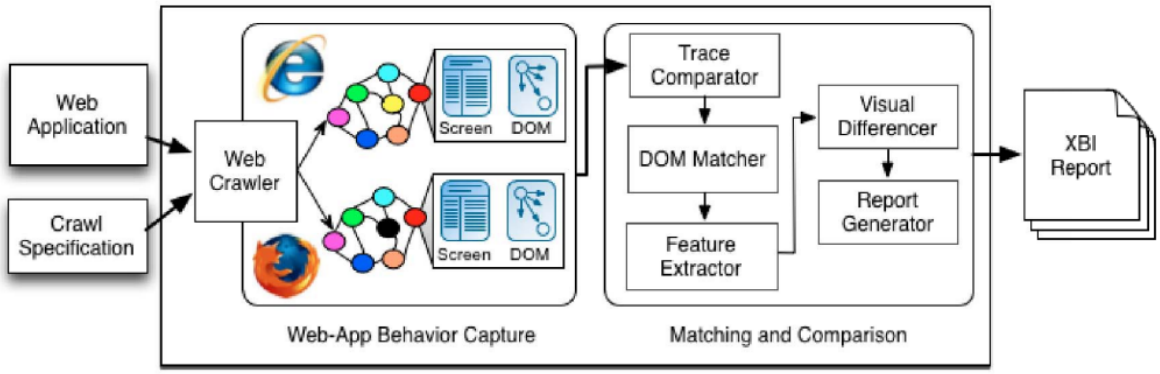


FIGURE 2.3 : Bref aperçu du processus que suit CrossCheck lorsqu'il évalue un site web sur plusieurs navigateurs afin d'en détecter les erreurs inter-navigateur.[26] ©2012, IEEE Autorisation obtenue par IEEE.

Finalement, l'équipe a proposé un dernier outil, X-Pert[14], qui est la version améliorée de CrossCheck. Tout d'abord, l'algorithme de détection d'erreurs a été amélioré afin d'en augmenter la précision, la fiabilité ainsi que de couvrir une plus large gamme d'erreurs visuelles. L'équipe a aussi travaillé à faciliter la lecture et la compréhension des rapports d'erreurs en travaillant sur leur algorithme de détection des éléments fautifs. Malgré le travail réalisé précédemment avec l'outil CrossCheck, plusieurs erreurs sortaient encore en doublons. Par exemple, si un élément est déplacé, pour une quelconque raison, dans la page, cela peut générer un effet domino et déplacer d'autres éléments. Alors que CrossCheck produirait un rapport d'erreur différent pour chacun de ces éléments, tous représentant une erreur différente puisque chaque élément, individuellement, n'est plus à la bonne place, X-Pert produit un seul rapport d'erreur en désignant seulement l'élément fautif qui a provoqué l'effet domino.

2.1.4 SEESS

Liang *et al.* ont développé un prototype pour aider les développeurs à réaliser le développement itératif de fichiers CSS plus rapidement et facilement. L'outil SeeSS [4] qu'ils

présentent permet aux programmeurs de voir l'impact des modifications CSS qu'ils apportent sur l'entièreté du site web. Pour ce faire, l'outil va effectuer un premier parcours du site. grâce à un *crawler*, et, pour chaque page, conserve une capture d'écran et garde en mémoire tous les fichiers CSS qui sont utilisés. Une fois que le site entier à été parcouru, que toutes les captures d'écrans sont enregistrées dans la base de données et que le graphe de dépendances CSS est construit, l'outil est prêt à être utilisé.

À chaque fois que le développeur enregistre une modification dans un des fichiers HTML ou CSS, SeeSS appelle le moteur de rendu pour générer la page HTML modifiée ou toutes les pages HTML qui utilisent le fichier CSS modifié. Pour chacune de ces pages, une capture d'écran est prise et enregistrée dans la base de données. Ensuite, chacune des nouvelles captures d'écran est comparée avec la dernière valide pour cette même page. Cette comparaison va permettre d'identifier tous les changements générés par la modification apportée par le développeur. Quand l'outil détecte un changement dans la page, il génère une image recadrée afin de présenter le changement au développeur. Une fois que la génération des images recadrées est terminée, SeeSS les présente au développeur afin de l'informer de tous les impacts de sa dernière modification. La figure 2.4 présente la structure de SeeSS ainsi que la manière dont les données transitent dans l'outil.

2.1.5 CORNIPICKLE

Beroual *et al.* ont employé une technique totalement différente : une approche déclarative. Cette approche, implémentée dans l'outil Cornipickle [7, 22, 27], consiste à déclarer des comportements ou affichages attendus. L'outil vérifiera que ces déclarations sont respectées et, si ce n'est pas le cas, l'affichera au développeur. Guérin *et al.* ont bonifié l'outil en y intégrant Crawljax pour pouvoir se déplacer au travers d'un site web [8, 28]. Cet ajout permet de tester

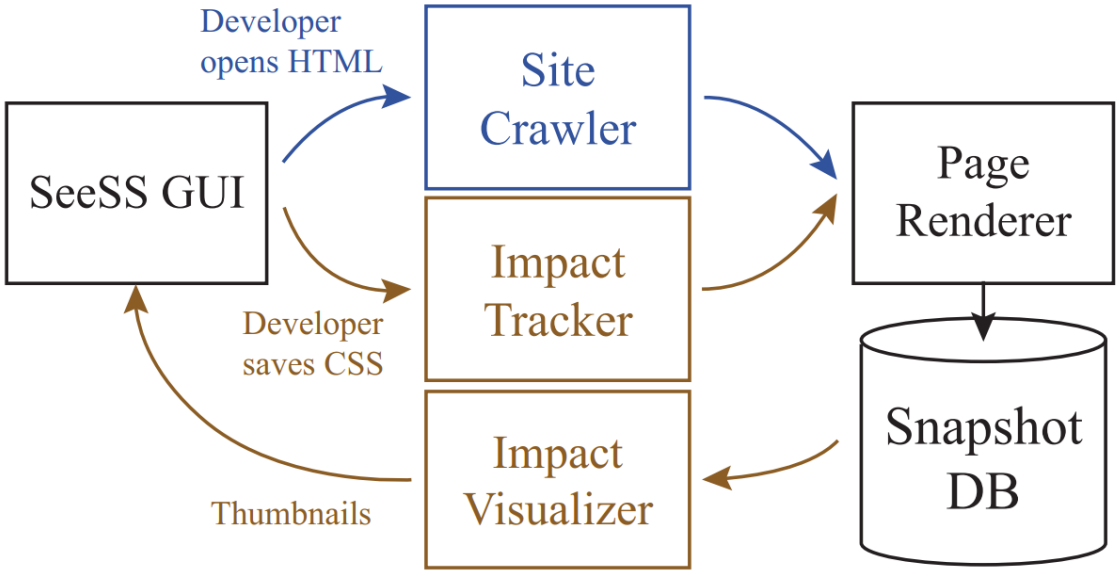


Figure 2.4 : Bref aperçu de la structure de l'outil SeeSS ainsi que du flux de données lors de son utilisation.[4] Autorisation obtenue par Association for Computing Machinery.

plusieurs pages automatiquement en plus de permettre de valider certains comportements au travers du site, permettant ainsi d'effectuer plusieurs cas de tests automatiquement. Une présentation plus détaillée de Cornipickle sera faite au chapitre 4.1.

2.1.6 CASSIUS

Pavel Panchekha et Emina Torlak proposent, quant à eux, l'utilisation de leur environnement Cassius [3] qui, lui aussi, utilise une structure déclarative pour identifier des erreurs de CSS. Cassius nécessite trois entrées : une paire composée d'une page HTML et d'un affichage désiré, un fichier CSS et une liste de déclarations sur les comportements ou rendus attendus. Parmi ces entrées, une et une seule, parmi le fichier CSS ou l'affichage désiré, peut contenir des trous, soit des points d'interrogation au lieu de certaines valeurs. Cassius pourra s'occuper de remplir ces trous avec les valeurs adéquates pour répondre aux attentes ou identifier les

```
(document A
  ([html]
   ([body]                        (stylesheet B
    ([p] "Our products:")           ((tag main)
    ([main]                          [width (% 100)]
      ([div] "A")                    [float left])
      ([div] "B")                   ((tag div)
      ([div] "C")                    [width (px 200)]
      ([div] "D"))                   [height (px 200)]
    ([p] "Buy now!"))))              [float left]))

(layout C
  ([ROOT :w 400 :h 600]
   ([BLOCK :w ? :h ? :x ? :y ?]
    ([BLOCK :w ? :h ? :x ? :y ?]
     ([BLOCK :w ? :h ? :x ? :y ?]
      ([LINE :w ? :h ? :x ? :y ?]
       ([TEXT :w 112 :h 19 :x 0 :y 16])))
     ([BLOCK :w 400 :h 400 :x ? :y ?]
      ([BLOCK :w 200 :h 200 :x 0 :y ?] ...)
      ([BLOCK :w 200 :h 200 :x 200 :y ?] ...)
      ([BLOCK :w 200 :h 200 :x 0 :y ?] ...)
      ([BLOCK :w 200 :h 200 :x 200 :y ?] ...))
     ([BLOCK :w ? :h ? :x ? :y ?]
      ([LINE :w ? :h ? :x ? :y ?]
       ([TEXT :w 76 :h 10 :x 0 :y 451])))))))))
```

FIGURE 2.5 : Exemple d'entrées à fournir à l'outil Cassius. On y voit la page HTML (document A), le fichier CSS (stylesheet B) et le rendu désiré avec trous (layout C).[3] Autorisation obtenue par Association for Computing Machinery.

valeurs pour lesquelles au moins une des déclarations sera violée. La figure 2.5 montre un exemple de fichier HTML, d'affichage désiré, avec trous, et de fichier CSS qui pourraient être fournis à l'outil. Dans tous les cas, Cassius va traiter les entrées reçues pour constituer un problème de satisfaisabilité modulo théories (SMT). C'est grâce à cette conversion qu'il va pouvoir déterminer si les contraintes demandées peuvent être respectée par le HTML et CSS fourni.

Si le fichier troué est celui du rendu désiré, Cassius va tenter de trouver des dimensions d'affichage ou des valeurs qui provoqueraient la violation des contraintes. Pour ce faire, il va générer un nouveau fichier de contraintes où toutes les déclarations seront inversées. Par exemple, si une contrainte indiquait que la taille d'un élément ne peut pas dépasser 300

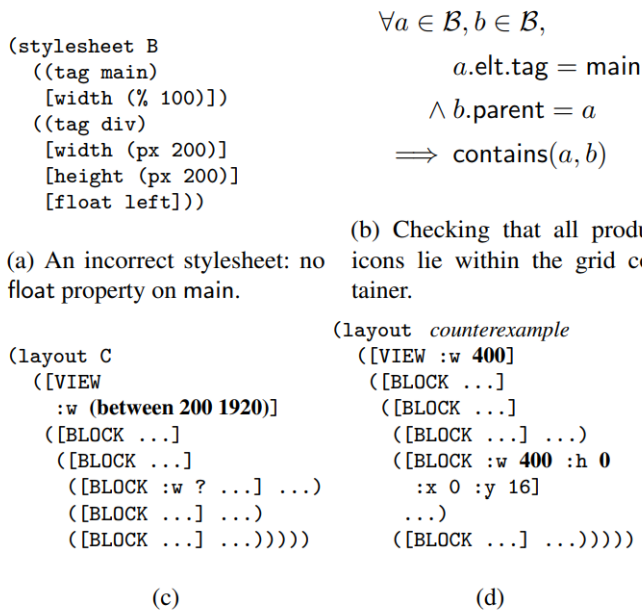


FIGURE 2.6 : Contre-exemple fourni par l'environnement Cassius. En (a) on peut voir le fichier CSS, en (b) la contrainte, en (c) le rendu désiré avec trous et en (d) le contre-exemple.[3] Autorisation obtenue par Association for Computing Machinery.

pixels, alors le nouveau fichier de contraintes indiquera que la taille d'un élément doit être supérieure à 300 pixels. Ensuite, Cassius va concevoir son problème SMT en utilisant ce nouveau fichier et ainsi identifier toutes les valeurs, que ce soit des valeurs entrées ou causées par les dimensions d'affichage, qui respectent ces conditions. Si des valeurs sont trouvées, cela signifie que celles-ci peuvent violer les contraintes fournies par l'utilisateur. Cassius va donc indiquer au développeur chacune de ces valeurs et l'élément auquel les appliquer, afin de lui montrer les données problématiques : l'outil fournit donc à l'utilisateur un contre-exemple. La figure 2.6 montre un tel contre-exemple.

Dans le cas où c'est le fichier CSS qui comporte des trous, Cassius va tenter d'identifier quelles seraient les valeurs, si nécessaire, à donner aux attributs CSS troués afin de répondre aux contraintes fournies. Pour ce faire, il va analyser la structure et le code de la page HTML

ainsi qu'évaluer les déclarations afin de détecter s'il y a une erreur, c'est-à-dire si la page HTML ne respecte pas les contraintes. Une fois cette évaluation faite, il va tenter d'identifier quel attribut aura un impact sur l'erreur trouvée. Ensuite, il pourra résoudre le problème SMT en ayant pour inconnue la valeur de l'attribut CSS identifié. Il pourra ensuite indiquer au développeur quel attribut CSS est requis et suggérer une valeur à cet attribut. Parmi les outils de détection, cette fonctionnalité de Cassius est celle qui se rapproche le plus de la correction d'erreurs visuelles.

2.1.7 FIERYEYE

Mahajan *et al.* ont conçu WebSee [29], un outil permettant d'identifier des regroupements d'éléments HTML qui seraient la cause d'une erreur visuelle. Pour ce faire, l'outil reçoit une URL qui est la page à tester, une représentation de ce à quoi devrait ressembler la page sous forme d'une image et un groupe d'éléments spéciaux. Les éléments dits spéciaux sont soit des sections de la page à ignorer lors de la recherche d'erreur, soit des parties dont le style et l'affichage sont connus, mais pas le contenu (par exemple des plages publicitaires sur les sites web). Avoir connaissance de ces différences de contenu va éviter les faux positifs dans l'identification d'erreurs visuelles en permettant à l'outil, pour ces parties uniquement, d'injecter le style voulu à ces éléments avant de faire la comparaison avec le visuel attendu.

Le processus de détection d'erreurs de WebSee se divise en quatre grandes phases, soit la détection, la localisation, le traitement des résultats obtenus et la gestion des régions spéciales. Au début de la première phase, WebSee commence par normaliser la page web et l'image du rendu désiré. Cette normalisation inclut : s'assurer que le défilement de la page ne cache pas certains éléments qui devraient être visibles et que les dimensions d'affichage du site correspondent à celle de l'image de comparaison reçue. Une fois le tout normalisé et une capture d'écran de la page web effectuée, l'outil utilise une technique basée sur la vision par

ordinateur pour comparer les deux images. Cette technique se base sur la perception humaine : en comparant les images, elle rapporte seulement les erreurs qui seraient perçues par un être humain, et non des erreurs impliquant, par exemple, un seul et unique pixel de différence. Une fois les erreurs détectées entre les deux images, l'outil doit ensuite identifier où sont ces erreurs dans le HTML.

Ainsi débute la deuxième phase, celle de localisation. Pour débuter, WebSee va commencer par adapter l'arbre DOM pour en faire un R-Arbre, une structure de données très utilisée pour l'exploration spatiale, soit arriver à positionner des points ou polygones (dans ce cas-ci des éléments HTML) dans un plan (dans ce cas-ci la page web). Toutes les feuilles de cet arbre seront des éléments HTML visibles alors que tous les éléments non-feuilles seront des rectangles contenant les éléments feuilles. Ensuite, les résultats obtenus à la phase précédente seront utilisés pour parcourir l'arbre et ainsi identifier lesquels des éléments HTML causeraient les erreurs.

Ensuite, lors de la phase de traitement des résultats, la liste des éléments HTML trouvés à la phase précédente seront triés selon la probabilité d'être la cause de l'erreur. Ce tri est appliqué pour éliminer les effets domino qui sont causés par le déplacement d'un élément dans la page.

Finalement, l'outil s'occupe des régions spéciales, qui sont des régions au visuel variable en raison de la génération dynamique de la page web, mais pour lesquels les développeurs sont quand même capable d'établir certaines contraintes de visuel, par exemple la taille ou la position. Ces régions sont séparées en deux grandes catégories : les régions exclues et les régions de texte dynamique. La première catégorie, regroupant des affichages tels que les publicités, ne sera tout simplement pas prise en compte lors du rapport final. Le deuxième groupe, quant à lui, prendra en compte tous les endroits dans la page où le style et la présentation

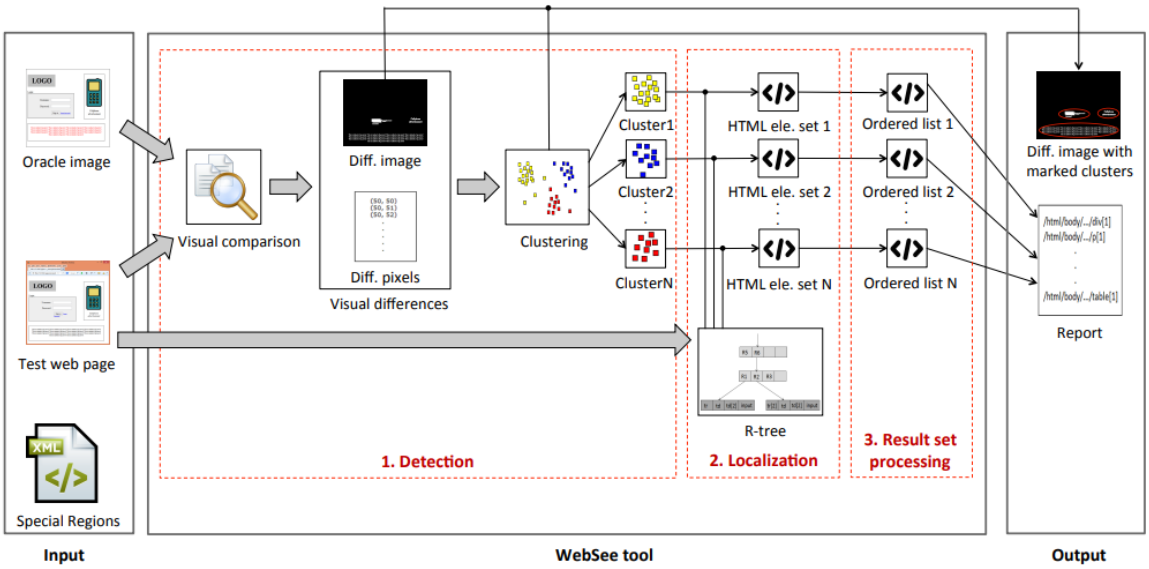


FIGURE 2.7 : Bref aperçu de la structure de l'outil WebSee ainsi que du flux de données lors de son utilisation.[29] ©2015, IEEE Autorisation obtenue par IEEE.

du texte sont connues, mais pas forcément le texte lui même, par exemple les informations de l'utilisateur. Pour tester cette catégorie, le style voulu sera injecté dans la page aux endroits appropriés, puis les phases un à trois seront exécutées à nouveau, mais cette fois en prenant la page modifiée comme page à comparer et la capture d'écran de la page initiale comme page de comparaison. Ainsi, si aucune différence est trouvée, alors on peut conclure que la page originale était correcte. La figure 2.7 présente un schéma du fonctionnement de WebSee.

Tout comme l'équipe de Roy Choudhary avec CrossCheck, Mahajan *et al.* ont développé une deuxième version de leur précédent outil, WebSee, afin d'améliorer l'identification des éléments causant l'erreur visuelle : il s'agit de FieryEye [30]. Dans cette nouvelle version, seule l'identification des éléments fautifs (soit les phases deux et trois de WebSee), a été modifiée. Cette fois-ci, ils introduisent la notion de symptômes visuels, c'est-à-dire, telle une maladie, des erreurs visuelles qui sont des signes pointant vers le problème réel dans le code.

Afin de lier, pour chaque page, les symptômes à l'erreur correspondante, ils implémentent un modèle probabiliste suivant le théorème de Bayes. Pour générer les données qui vont alimenter le modèle pour une page donnée, le logiciel injecte des erreurs dans la page à tester et identifie les symptômes qui en résultent. Le travail identifie trois grands types de propriétés CSS qui peuvent causer des erreurs : les prédéfinis (des attributs CSS avec des valeurs préexistantes tel que *normal* ou *italic* pour le *font-style*), les couleurs (l'attribution d'une couleur quelconque à un élément) et les numériques (tout attribut prenant un nombre comme valeur). Il modifie donc le CSS de la page en jouant sur une de ces trois grandes catégories à chaque fois afin d'identifier tous les symptômes qui en résultent. Ce sont les données recueillies par cette procédure qui permettent au modèle d'associer les symptômes présents dans la page à la cause réelle et ensuite établir quel élément dans la page est le plus probable d'être le responsable des symptômes.

2.1.8 REDECHECK

Dans l'optique d'alléger la tâche que représente le test d'une application web dite réactive, Walsh *et al.* ont publié l'outil ReDeCheck [5, 31, 32]. L'outil se base sur la construction d'un graphe de disposition réactive (reactive layout graph ou RLG), c'est-à-dire une structure de données présentant tous les éléments HTML d'une page ainsi que leur comportement selon différentes dimensions d'affichage. Pour créer un tel graphe, ReDeCheck parcourt l'arbre DOM de la page web et récupére des informations sur chacun des nœuds, notamment une façon d'identifier chaque élément ainsi que ses coordonnées dans la page. Une particularité de ce graphe est que, pour chaque élément de la page HTML, des coordonnées seront conservées pour plusieurs tailles d'affichage différentes. Ce graphe permet donc d'avoir, dans une seule structure, une représentation de l'arbre DOM de la page ainsi que la manière dont les éléments réagissent lorsque la taille d'affichage varie. La figure 2.8 montre un exemple d'un tel graphe.

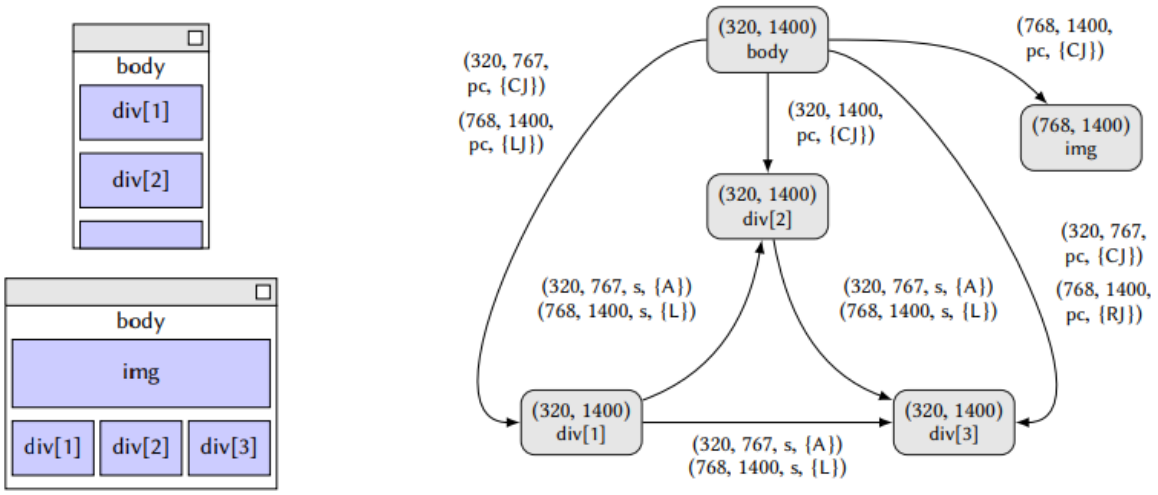


FIGURE 2.8 : À gauche, un exemple de page web affichée selon deux dimensions d'affichage différentes. À droite, un exemple de ce que donne le RLG de la page HTML à sa gauche selon les deux tailles d'affichages présentées.[5] Autorisation obtenue par Association for Computing Machinery.

Afin de déterminer pour quelles tailles d'affichage il est nécessaire de construire le graphe, ReDeCheck analyse tous les fichiers CSS liés à la page. Après cette analyse, l'outil peut identifier des tailles critiques, c'est-à-dire des tailles qui causeraient des problèmes de rendu. Par exemple, si ReDeCheck détecte qu'un élément possède une propriété CSS *width* avec comme valeur 300 pixels, alors il faudra faire un test avec une taille d'affichage plus petite que 300 afin d'observer l'impact sur son affichage.

Une fois le graphe complété, il faudra en faire l'analyse. Celle-ci dépendra du mode d'aide au développement sélectionné : les tests de défaillances courantes ou les tests de régression. Si la première option est choisie, l'outil va parcourir le graphe et analyser chaque nœud, sa position par rapport aux autres nœuds et ses réactions face aux changements de résolution. Si une erreur, telle que le chevauchement ou le débordement du conteneur, est détectée, ReDeCheck produira un rapport indiquant les éléments impliqués dans l'erreur

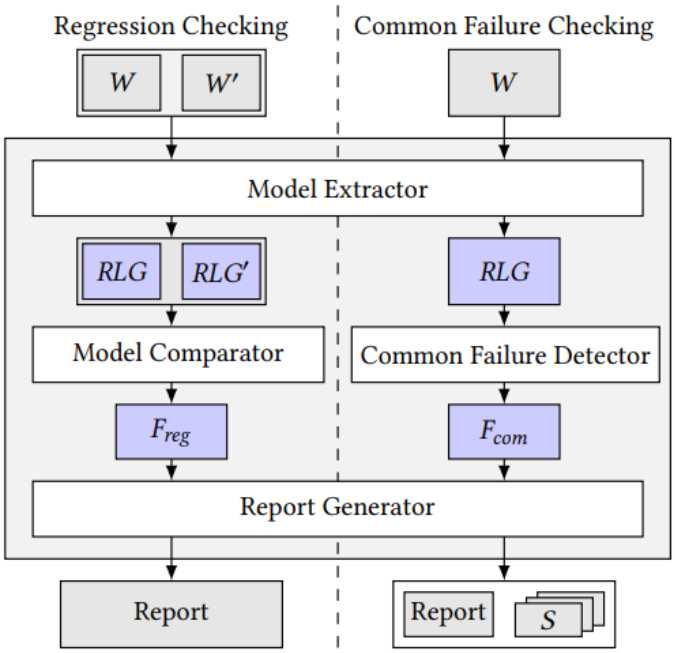


FIGURE 2.9 : Aperçu du fonctionnement de ReDeCheck selon l'option d'aide choisie. À gauche, on peut voir le fonctionnement des tests de régression alors qu'à droite on voit la détection de défaillances communes.[32] Autorisation obtenue par Association for Computing Machinery.

et à quelle dimension d'affichage le bogue survient. Si, toutefois, la deuxième option a été sélectionnée, le fonctionnement varie un peu. Premièrement, au lieu de simplement recevoir une page web à analyser, ReDeCheck en recevra deux : la page web avant modification et la même page web suite aux modifications du développeur. Ensuite, un RLG sera produit pour chacune des pages en utilisant les mêmes dimensions d'affichage. Ensuite, ces deux RLG seront comparés afin de déterminer si la nouvelle version introduit une erreur visuelle ou non. Un même rapport que pour l'option 1 sera produit suite à la découverte d'erreur, si des erreurs sont trouvées. La figure 2.9 présente un schéma montrant le fonctionnement des deux options d'exécution de l'outil.

2.1.9 VISER

Althomali *et al.* ont été inspirés par l'outil de Walsh *et al.* mentionné précédemment. Bien que ReDeCheck soit fonctionnel, ils ont remarqué que plusieurs erreurs trouvées étaient invisibles pour l'humain et donc non dommageables pour l'expérience utilisateur. Ainsi est venue l'idée de Viser [33], un outil qui prend le relais suite à l'analyse d'une page web par ReDeCheck. À la suite de la production des rapports de ReDeCheck, Viser ouvre le site web dans un navigateur, ajuste la taille du navigateur à celle mentionnée dans le rapport d'erreur puis se positionne au bon endroit dans la page, c'est-à-dire défiler jusqu'à l'élément en erreur. Ensuite, Viser fait une brève analyse de l'arbre DOM afin de contre-valider que l'erreur identifiée en est bien une et non un faux positif.

Par la suite, une image est prise et transmise à l'analyseur d'image. Celui-ci identifie la zone d'influence, c'est-à-dire un rectangle se rapportant à l'élément fautif. Une fois cette zone identifiée, Viser modifie le CSS de l'élément HTML contenu dans cette zone afin d'identifier si celui-ci s'affiche au mauvais endroit ou s'il s'affiche par-dessus un autre élément. Une comparaison est ensuite effectuée entre chaque modification et la page de base. Si une différence, hormis celle effectuée par l'outil, est détectée, alors l'erreur est considérée comme valable car elle est détectable par l'utilisateur. Sinon, l'erreur est abandonnée. La figure 2.10 présente une comparaison entre les étapes à suivre, dépendant de ce que l'opération soit effectuée manuellement ou via l'utilisation de Viser, afin d'obtenir la classification des erreurs trouvées par ReDeCheck.

2.1.10 VFDETECTOR

Ryou *et al.* ont, quant à eux, travaillé sur un outil ayant pour but de faciliter la détection d'erreurs visuelles générées par des changements de dimensions de la fenêtre qui affiche la

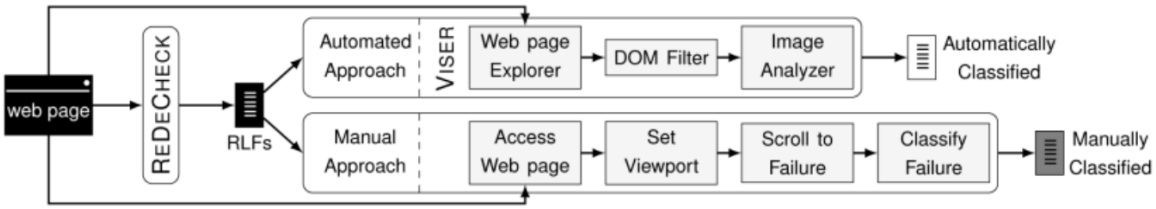


Figure 2.10 : Bref aperçu des étapes de traitement faites par Viser des rapports d'erreurs produits par ReDeCheck.[33] ©2019, IEEE Autorisation obtenue par IEEE.

page web [34]. Pour ce faire, ils utilisent un crawler qui leur permet de naviguer rapidement et automatiquement au sein d'un site web et de répliquer ce même parcours selon différentes dimensions d'affichage. Pour chacune des dimensions d'affichage, le crawler va faire le même parcours au travers du site. À chaque étape de son parcours, soit à chaque fois qu'il déclenche un événement, le crawler va récupérer l'identifiant de chaque élément avec son statut de visibilité pour l'utilisateur. Ce regroupement d'information pour une page donnée à un moment précis formera un nœud. Un exemple d'un tel nœud pourrait être *{(btn1, normal), (btn2, full-cover), (btn3, normal)}*. Dans cet exemple, l'utilisateur voit les boutons btn1 et btn3, mais pas le bouton btn2 qui serait entièrement couvert par un autre élément. Tous les nœuds seront ensuite reliés, selon l'événment menant à la transition de visuel, pour former un graphe de navigation de site indiquant à quoi est supposé ressembler une page suite à un événement précis. Le crawler produira un tel graphe pour chaque dimension d'affichage à tester.

Une fois tous les graphes récupérés, l'outil va comparer les nœuds correspondants de chaque graphe afin de détecter les erreurs visuelles présentes. Puisqu'un noeud représente un affichage précis suite au déclenchement d'un événement quelconque, cette comparaison va donc permettre d'identifier si des visuels, qui devraient être similaires, affichent bien les mêmes éléments. Cette comparaison nécessite simplement de comparer les données présentes dans chacun des nœuds. Si le nœud diffère, que ce soit de par le nombre d'éléments, impliquant

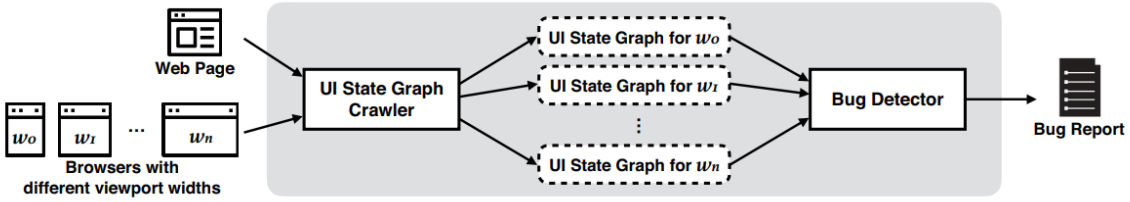


FIGURE 2.11 : Bref aperçu des étapes de traitement de VFDetector afin de détecter des bogues visuels.[34] ©2018, IEEE Autorisation obtenue par IEEE.

qu'il y a plus ou moins d'éléments que prévus présents dans la page (par exemple, si le code JavaScript n'a pas généré le bon nombre d'éléments), ou par le statut de visibilité d'un des éléments notés dans le nœuds, celui-ci est marqué comme étant en erreur. Finalement, l'outil va présenter chacune des erreurs trouvées ainsi que comment recréer chacune d'entre elles, soit la dimension d'affichage requise ainsi que la suite d'événements à faire pour obtenir l'erreur. La figure 2.11 présente le fonctionnement de VFDetector.

2.1.11 DÉTECTION D'ERREURS SUR D'AUTRES PLATEFORMES

Les précédents outils portaient tous sur la détection de bogues d'affichage sur des applications web, mais ça ne représente qu'une partie de la littérature. En effet, il existe plusieurs outils qui aident à la détection d'erreurs sur d'autres plateformes. Notamment, plusieurs équipes se sont penchées sur la détection d'erreurs visuelles pour les applications Android.

Meniar *et al.* présentent l'outil Cornidroid[35] qui, à la manière de Cornipickle, utilise un langage déclaratif, dans un environnement client-serveur, afin d'établir des contraintes pour l'affichage. Grâce à ces déclarations, qui sont évaluées au niveau du serveur, une évaluation de l'affichage, qui est prise du côté client par une sonde pour envoyer au serveur, peut être faite

afin de détecter les éléments qui violent une des déclarations. L'application niveau serveur renvoie les erreurs trouvées à la sonde afin qu'elle puisse identifier dans l'affichage client les erreurs.

Une approche similaire a été employée par l'outil Venus[36] de Zhang *et al.* En employant lui aussi un langage déclaratif pour définir les règles d'affichage à respecter, Venus offre la possibilité de détecter des erreurs visuelles, mais aussi d'identifier des publicités dérangeantes ou frauduleuses ainsi que des violations du règlement général sur la protection des données. En créant une forêt, soit un regroupement d'arbres, pour créer une abstraction de l'application contenant uniquement les informations utiles telles que la taille ou la position des éléments, Venus va pouvoir établir une expression logique correspondant à l'application. Cette formule pourra ensuite être comparée aux règles fournies et ainsi déterminer quelles parties de l'application violent une des spécifications.

Moran *et al.* présentent une technique qui vise à détecter les erreurs d'implémentation de l'affichage. Leur outil implémentant cette technique, GVT[37], vise à fournir de l'aide au développement d'affichage en récupérant l'affichage affichée par l'application ainsi qu'en se faisant fournir une maquette, c'est-à-dire le prototype de l'interface utilisateur désirée. Une fois que GVT a identifié quel élément de l'interface graphique correspond à quelle partie de la maquette, il va employer des heuristiques et des techniques de vision par ordinateur afin de détecter à quel endroit l'affichage diffère du prototype. Un rapport HTML présente la liste des erreurs trouvées ainsi qu'une description et une capture d'écran de chaque erreur.

De leur côté, Wang *et al.* ont travaillé à détecter des erreurs reliées au texte présent dans une application. Afin d'arriver à leur fin, ils ont opté pour l'utilisation d'un réseau de neurones convolutifs (convolutionnal neural network ou CNN)[38]. Leur processus implique de prendre une image de l'affichage de l'application, puis d'y appliquer un processus de détection de

texte afin d'obtenir plusieurs fragments, où chaque fragment est une partie de l'image de base contenant du texte. Ces fragments d'images sont ensuite envoyés à un CNN, préalablement entraîné, qui attribue un score à chacun des fragments. Si on déduit de ce score qu'un fragment est anormal, l'erreur sera encadrée en rouge pour l'indiquer au développeur.

Il est à noter que ces outils visent une catégorie d'applications différente de celle de ce travail, d'où leur présentation moins détaillée. En effet, bien que ces outils fassent de la détection de bogues d'affichage, ils n'interagissent pas avec du HTML, CSS et, par extension, du JavaScript. Or, c'est précisément l'intrication complexe, menant parfois à des comportements difficiles à comprendre, entre ces trois langages qui amène autant de problèmes dans les applications web.

2.2 CORRECTIONS

À contrario de la détection de bogues d'affichage, pour laquelle de multiples outils ont été suggérés et dont la recherche est abondante, la correction d'affichage est un domaine où il reste beaucoup de travail à faire. Effectivement, les quelques solutions actuellement proposées, bien que fonctionnelles, présentent plusieurs lacunes importantes, la plus notable étant le temps d'exécution.

2.2.1 CONTRE-EXEMPLES

Le contre-exemple consiste à donner une preuve qu'il existe un élément ne respectant pas une règle ou contrainte donnée, causant donc un bogue visuel. Bien qu'il ne s'agisse pas de correction à proprement parler, le contre-exemple permet aux développeurs non seulement de savoir quel élément dans la page crée l'erreur visuelle, mais aussi d'identifier la ou les contraintes qu'il ne respecte pas. De plus, un contre-exemple permet d'identifier une valeur

précise pour laquelle, lorsque appliqué à un attribut spécifique d'un élément HTML donné, un affichage n'est plus fonctionnelle. Cette information supplémentaire permet d'identifier la cause du problème plus aisément et donc de réparer le bogue plus rapidement. Hallé *et al.* fournissent un exemple concret du contre-exemple grâce à l'outil Cornipickle [22]. Cet outil identifie, dans une page web, les éléments en faute et offre, pour chacun des éléments fautifs, une explication de la règle qui est enfreinte.

L'outil Cassius[3], mentionné précédemment, se sert lui aussi de contre-exemples, tel que montré dans la figure 2.6. Grâce aux contre-exemples, il est en mesure d'identifier précisément des valeurs problématiques pour lesquelles un correctif doit être appliqué afin d'éviter de causer une erreur d'affichage. Ainsi, bien que le contre-exemple ne soit une correction à part entière, c'est la première étape d'identification de valeurs précises qui peuvent mener à la mise en place d'un correctif adéquat.

2.2.2 RÉPARATION D'AFFICHAGE WEB

Une fois qu'un élément fautif a été détecté et qu'il est possible d'identifier pourquoi celui-ci n'est pas correct, il est envisageable de regarder comment trouver et appliquer un correctif audit élément, soit réparer les programmes. Plusieurs équipes de recherche se sont penchées sur ce sujet : Winter *et al.*[39] en ont fait la revue. Toutefois, seules deux équipes se sont attelées à la correction d'erreurs visuelles dans les applications web.

Beroual et Hallé [10] ont travaillé sur la libraire Fault-Finder. En ce faisant fournir une contrainte, sous forme d'expression logique, que devrait respecter un élément ainsi que la liste des éléments enfreignant la contrainte reçue, Fault-Finder, qui se charger de trouver une valeur adéquate à donner à un ou des élément(s) de la liste afin de valider ladite expression logique. Pour se faire, Fault-Finder va générer une liste de transformations, soit une liste de

modifications ou combinaisons de modifications qui, lorsque l'on applique l'une d'entre elles, permet aux éléments reçus de valider la contrainte fournie en entrée. Fault-Finder sera expliqué plus en détails dans le chapitre 4.2.

Mahajan *et al.* ont préféré se pencher sur certains types d'erreurs en particulier, soit les erreurs inter-navigateur [9, 40], celles d'internationalisation [41] et celles liées aux rendus sur mobile [42]. XFix, l'outil développé pour les erreurs inter-navigateur, débute en appelant l'outil X-Pert, mentionné ci-avant, afin de détecter les différences présentes entre la page du navigateur témoin et la page du navigateur à tester. Pour chaque erreur détectée, XFix identifie les attributs CSS qui peuvent être en cause pour ce type d'erreur grâce à un schéma erreur-propriété intégré à l'outil. S'en suit une recherche des correctifs probables. En comparant les attributs CSS trouvées et l'erreur, XFix identifie tous les correctifs possibles. Chacun de ces correctifs est pondéré par un score déterminant son niveau d'efficacité. Une fois tous les correctifs possibles trouvés, l'outil calcule, grâce aux scores d'efficacité, la meilleure combinaison de correctifs à appliquer. En effet, ce ne sont pas tous les correctifs trouvés qui sont réellement applicables : certains n'auront aucun effet lorsque combinés avec d'autres alors que certains vont démultiplier leurs effets. Ensuite, XFix applique les correctifs choisis à la page du navigateur à tester, puis compare à nouveau cette page à celle du navigateur témoin. Si la comparaison détermine qu'il reste des erreurs, XFix recommence le processus en ce servant de cette nouvelle page et des nouvelles erreurs détectées comme point de départ. Une fois que l'outil juge les correctifs satisfaisants, il produit un fichier CSS réparé, contenant toutes les modifications suggérées. Un aperçu de ce processus est montré dans la figure 2.12.

Mahajan et al. ont aussi travaillé sur une approche, implémentée via l'outil IFix[41], pour réparer les erreurs d'internationalisation qui fonctionne similairement à celle de Xfix : compater une page dont le visuel est correct à la même page traduite dans une autre langue afin

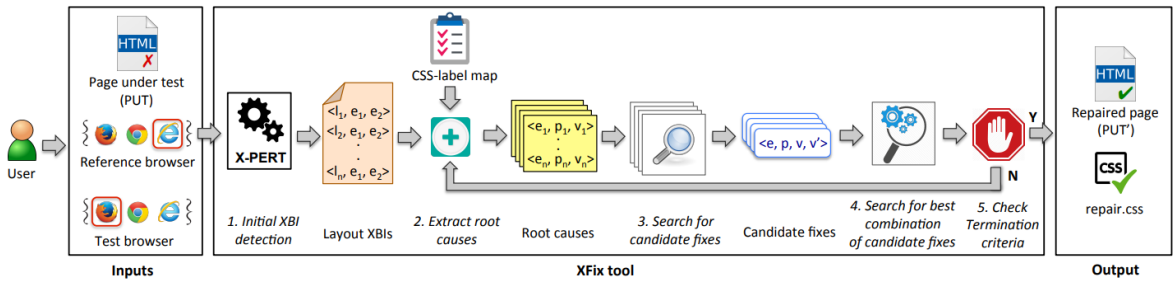


FIGURE 2.12 : Bref aperçu des étapes de traitement faites par XFix afin de détecter et corriger des bogues visuels.[9] Autorisation obtenue par Association for Computing Machinery.

d'identifier les problèmes liés à la traduction. En effet, lors de la traduction d'un texte, celui-ci peut prendre plus ou moins de place qu'avant le changement de langue. Cette différence de longueur de texte peut ainsi faire varier la taille d'un élément de la page web et donc en changer l'affichage. En comparant une page avec un affichage considéré correct et la même page traduite, il est possible de détecter de telles erreurs.

Une fois cela fait, l'outil IFix identifie des groupes d'éléments qui devraient être modifiés ensemble, par exemple, tous les boutons constituant le menu de navigation. Ce regroupement permet de garder une cohésion dans la page lors des changements. Afin de faire un tel regroupement, IFix identifie les éléments qui ont des similitudes. Les premières similitudes considérées sont celles visuelles. L'outil identifie les éléments qui ont des hauteurs et largeurs identiques, qui ont un alignement horizontal ou vertical correspondant et des attributs CSS similaires, comme la couleur ou la police. Ensuite, IFix considère les similitudes au niveau du DOM. Il vérifie si les éléments ont le même type de balise HTML, si les éléments ont les mêmes ancêtres et si les éléments partagent des classes CSS. Des éléments répondant à tous ces critères sont fort probablement liés et manipulés ensemble, d'où l'idée de les placer dans un même ensemble.

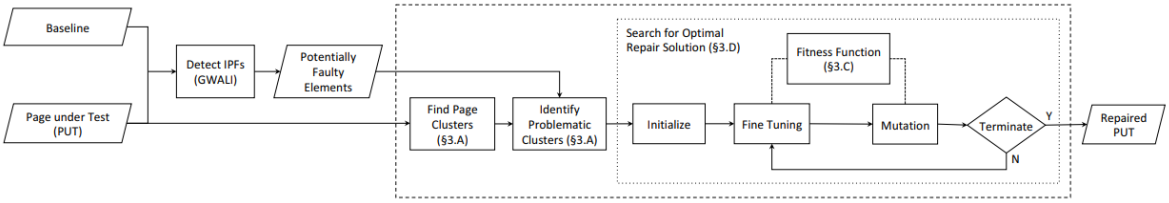


FIGURE 2.13 : Bref aperçu des étapes de traitement faites par IFix afin de détecter et corriger des bogues visuels.[41] ©2018 IEEE Autorisation obtenue par IEEE.

Ensuite, pour chaque groupe contenant une erreur, il faut établir la liste de toutes les réparations possibles. Une réparation, dans ce cas-ci, représente toute combinaison de modifications faites dans la page menant à la résolution du bogue visuel. Chaque réparation est testée. Lors de ce test, l'outil observe l'impact qu'a la modification sur le reste de la page et lui attribue une note indiquant la quantité d'altérations qu'elle cause. Cette étape est cruciale puisque, lors de la réparation d'une erreur causée par la traduction, qui revient fréquemment à modifier la taille d'un ou plusieurs éléments de la page, on peut créer un effet domino, déplaçant moult autres éléments et déformant ainsi la page.

Suite à la génération et la notation des réparations, celle ayant la meilleure note est appliquée dans la page. Ensuite, la page sera de nouveau comparée avec la page témoin : si le correctif choisi a bel et bien réparé l'erreur, alors le programme s'arrête, sinon, il teste le prochain correctif présentant le meilleur score. Ce processus est présenté dans la figure 2.13.

Leur dernière approche, présentée via l'outil MFix[42], cherche à identifier et corriger les erreurs qui surviennent lorsqu'un site web s'affiche sur un appareil mobile alors qu'il n'a pas été conçu pour une telle utilisation. La technique présentée par MFix se sépare en trois phases : la segmentation, la localisation et la réparation. La figure 2.14 illustre cette technique. Lors de la première phase, l'outil identifie des ensembles dans la page. Tout comme pour l'outil

IFix, ces ensembles servent à regrouper des éléments communs afin que, lors de l'application de modifications, il reste une cohésion dans la page.

La prochaine étape sert à identifier les erreurs dans la page. Cette étape se sépare en deux grandes parties, soit la détection de segments problématiques et la détection d'attributs CSS problématiques. Afin d'identifier les ensembles en erreurs, MFix utilise un *Mobile Friendly Oracle* (MFO). Ces oracles spécialisés dans la détection d'erreurs présentes dans une page web affichée sur un appareil mobile permettent de trouver des sections problématiques de la page et identifier le code HTML lié celles-ci, mais pas le code CSS correspondant. Pour arriver à identifier ce code CSS, des graphes de dépendance des propriétés (property dependancy graphs ou PDG) sont conçus. Ces PDG permettent d'identifier tous les éléments HTML ainsi que chacun des attributs CSS pertinents qui est appliqué à chaque nœud, que ces attributs soient donnés explicitement ou hérités de leurs parents. Chaque PDG a une fonction spécifique, soit gérer les erreurs de police, gérer les erreurs de contenus et gérer les erreurs liées aux sections utilisables par l'utilisateur. Chacun de ces PDG récupère seulement les attributs CSS qui sont pertinents à sa fonction. Grâce à ces PDG, il est possible de relier le HTML identifié comme problématique par le MFO aux propriétés CSS l'affectant.

La troisième et dernière phase consiste à trouver et appliquer un correctif. Pour chacun des attributs CSS qui a été indiqué comme potentiellement problématique, MFix va tenter de trouver des nouvelles valeurs ou combinaison de valeurs qui vont corriger l'erreur tout en minimisant les impacts subséquents qu'auront ces corrections. Pour chaque candidat possible de corrections, ceux ayant le meilleur ratio de correction-impact seront testés. Lorsqu'un correctif viable aura été testé, MFix produira un correctif final. Pour ce faire, il devra appliquer les nouvelles valeurs trouvées tout en considérant l'impact des attributs CSS et de leur héritage. Ainsi, au lieu de bêtement appliquer la valeur trouvée dans le fichier CSS, MFix va calculer les impacts qu'ont les attributs des ancêtres des éléments à modifier sur ceux-ci. Ainsi, lors de

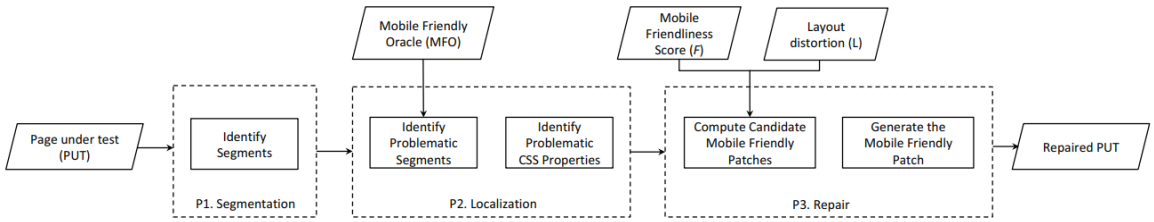


FIGURE 2.14 : Bref aperçu des étapes de traitement faites par MFix afin de détecter et corriger des bogues visuels dans une application web affichée sur un appareil mobile.[42] Autorisation obtenue par Association for Computing Machinery.

l'application finale des valeurs correctrices, les impacts des parents auront été pris en compte afin que la valeur appliquée mène à la solution désirée sans que d'autres règles CSS viennent interférer.

CHAPITRE III
PORTRAIT DES SITES WEB

Tel que constaté dans le chapitre précédent, il y a eu de grosses avancées dans le monde du web : plusieurs outils et techniques ont été mis au point pour aider à détecter, même des fois déboguer, des erreurs et analyser une page web. Dans plusieurs cas, cette aide offerte au développeur s'effectue vie l'analyse de l'arbre DOM de la page ainsi que de son CSS. Puisque plusieurs de ces pratiques sont réalisables uniquement en travaillant avec la structure de la page, la taille de celle-ci, soit la quantité de nœuds présents dans l'arbre DOM ainsi que leur disposition, a un impact direct sur les performances.

Afin de prouver le fonctionnement d'un tel outil, la majorité des équipes de recherche présenteront les performances de leur outil suite à une évaluation empirique de son exécution sur quelques pages web. Par exemple, une telle analyse présentée par Walsh *et al.* [31] a été effectuée sur des pages ayant jusqu'à 196 éléments dans la page, donc un arbre DOM contenant 196 nœuds. Par manque de données à ce sujet, il est impossible de dire si une telle page est considérée comme petite ou volumineuse. Quelle est donc la taille d'une page « réelle » ?

Cette question n'en est qu'une parmi tant d'autres. Il est raisonnable d'assumer que les pages développées durant les quelques dernières années diffèrent des pages produites dans les années 90, mais en quoi le sont-elles ? Quand ces changements de façon de faire sont-ils arrivés ? À quelle vitesses ces changements surviennent-ils ? Ce chapitre va tenter d'amener des réponses à ces questions en présentant les résultats d'une étude contemporaine [43, 44] du monde du web ainsi qu'une longitudinale [44].

La première étude, portant sur 708 sites web, avait pour but de mesurer la structure et la taille d'une page web grâce à variété de paramètres : la taille de l'arbre DOM, le degré

de distribution des nœuds, la profondeur, la distribution des balises HTML, la diversité des classes CSS, etc. La seconde étude permet d'avoir un aperçu des caractéristiques des sites web archivés par la *Wayback Machine*(https://archive.org/) entre 1996 et 2020. Grâce aux quelques 250 captures, récupérées sur différents sites à certains moments dans le temps, il possible d'avoir une meilleure compréhension de l'évolution des sites web au courant des 25 dernières années.

3.1 COLLECTE DE DONNÉES

Afin de pouvoir faire les analyses voulues sur les sites web, il a été nécessaire de trouver une façon de récolter le DOM de chacune des pages de la liste de site web, plus précisément réussir à récolter le type de balise des éléments présents dans la page, leurs classes CSS, leur statut de visibilité ainsi que des informations relatives à leur positionnement dans l'arbre DOM. Pour ce faire, un programme en JavaScript a été conçu dans l'optique de se promener à travers chaque nœud de l'arbre afin de récolter les informations nécessaires et produire deux fichiers : l'un en JSON contenant la structure des données recueillies ainsi que des statistiques sur la page web et l'autre un fichier DOT utilisable par la librairie Graphviz (https://graphviz.org) afin de générer une représentation visuelle de l'arbre correspondant à la page HTML.Un exemple du fichier JSON est présenté dans la figure 3.1 alors que la figure 3.2 présente un exemple de graphique produit par Graphviz. On peut y voir le graphe complet de la page web où chaque nœud représente un et un seul élément HTML de lapge. Il est à noté que chaque couleur de ce graphe représente un type de balise HTML différent et que le carré noir correspond à la racine de la page, soit la balise *BODY*. Afin d'exécuter ce script sur chacune des pages, l'extension TamperMonkey (https://www.tampermonkey.net) fut utilisée ; cette extension permet d'exécuter automatiquement des scripts sur une sélection de sites, et ce à un moment désiré, soit après le chargement de la page dans ce cas-ci. La liste complète des

données recueillies et le script sont disponibles en ligne[45]. Il est à noter que la méthode de collecte de données a fait en sorte que des pages de publicité et des popups ont aussi été récoltés.

Une seconde collecte a eu lieu en utilisant la *Wayback Machine*, un outil qui permet de visualiser des pages web telles qu'elles étaient à un moment précis dans le temps. Toutefois, l'utilisation de cet outil ralentissait grandement le processus de collecte de données. Ainsi, une première collecte sur un échantillon de dix sites populaires, soit Amazon [3], Ebay [4], Microsoft [5], Yahoo [6], Nytimes [7], Cnn [8], Apple [9], Blogger [10], Adobe [11] et Mozilla [12], a été effectuée afin de voir si certaines tendances, telles que la taille d'une page web, les balises utilisées ou même l'utilisation de classes CSS, se dessinaient. Ces analyses étant concluantes, la taille de l'échantillon a été augmentée. Par concluante, il est entendu ici qu'il a été possible de récupérer l'entièreté de l'arbre DOM d'une page ainsi que de récupérer les classes CSS reliées à chaque nœud de cet arbre en plus d'aller chercher plusieurs informations sur le rendu final, comme, par exemple, la position de l'élément dans la page ou son statut de visibilité. De plus, il a été possible, lors de la récupération de ces informations, d'ignorer le contrôleur de la *Wayback Machine*, ce qui est hautement désirable puisqu'il ne fait pas parti de la page web originale. Ce contrôleur, qu'on peut voir dans la figure 3.3, est ajouté dans la page web par la *Wayback*

3. http://amazon.com
4. http://ebay.com
5. http://microsoft.com
6. http://yahoo.com
7. http://nytimes.com
8. http://cnn.com
9. http://apple.com
10. http://blogger.com
11. http://adobe.com
12. http://mozilla.org

Machine. Celui-ci, superposé au contenu de la page, permet de naviguer au travers des captures qu'a fait la *Wayback Machine* d'une page au fil du temps. Dans le cas de cette figure, on peut remarquer que la capture a été effectuée en août 2011.

Afin d'augmenter la taille de l'échantillon, il fallait récupérer aléatoirement des sites web afin de ne pas induire un biais involontaire. Pour ce faire, nous sommes partis de la liste complète de sites mentionnée précédemment, de laquelle nous avons retiré les dix sites ayant servis de cobayes pour la récolte, et l'avons mélangée de façon aléatoire avant d'en sélectionner les cent premiers. Ce procédé a donc permis d'avoir un échantillon d'une centaine de sites choisis aléatoirement. Ensuite, le même processus que pour la première collecte de données, soit la collecte de test sur les dix sites mentionnés ci-avant, a ensuite été employé sur ce nouvel échantillon, à deux détails près. Premièrement, chacun des sites a été récolté plusieurs fois, mais à des années différentes comprises entre 1996 et 2020 grâce à la *Wayback Machine*. Deuxièmement, tel quel mentionné plus tôt, le contrôleur de la *Wayback Machine* était retiré par le script. Le tableau 3.1 montre le nombre de sites récoltés pour chaque tranche de cinq ans.

3.2 ANALYSE DES DONNÉES

Une fois les fichiers générés pour chacun des sites web de la liste, il fallait arriver à traiter la grande quantité de données récoltées. Pour ce faire, l'outil LabPal [46] a été utilisé puisqu'il permet de regrouper et évaluer un grand volume de données selon les paramètres désirés en plus de permettre de générer plusieurs statistiques, tableaux et graphiques à propos des données recueillies. Cet outil a pour but de créer des expériences facilement réutilisables et répétables pour la recherche scientifique. Ainsi, au sein d'un projet LabPal, il y aura une ou plusieurs expériences, soit un scénario qui prend certaines données en entrée, effectue

```
{
  "nbElementTotal": 1617,
  "profondeurMinArbre": 1,
  "profondeurMaxArbre": 20,
  "degreMinArbre": 1,
  "degreMaxArbre": 52,
  "nbNoeudsInvisibles": 1041,
  "p": 1,
  "div": 13,
  "a": 107,
  "span": 17,
  "circle": 1,
  "ellipse": 1,
  "svg": 109,
  ...
}
```


Figure 3.1 : Extrait d'un fichier JSON produit par le script de récolte de données.[44] Autorisation obtenue par River Publishers.

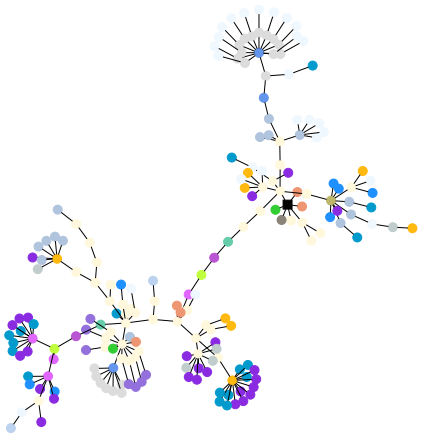


Figure 3.2 : Exemple d'un arbre DOM produit par le script de récolte de données. Chaque couleur représente une balise HTML différente, chaque point représente donc un élément de la page. Le carré noir représente la racine de l'arbre, soit l'élément *BODY*.[44] Autorisation obtenue par River Publishers.

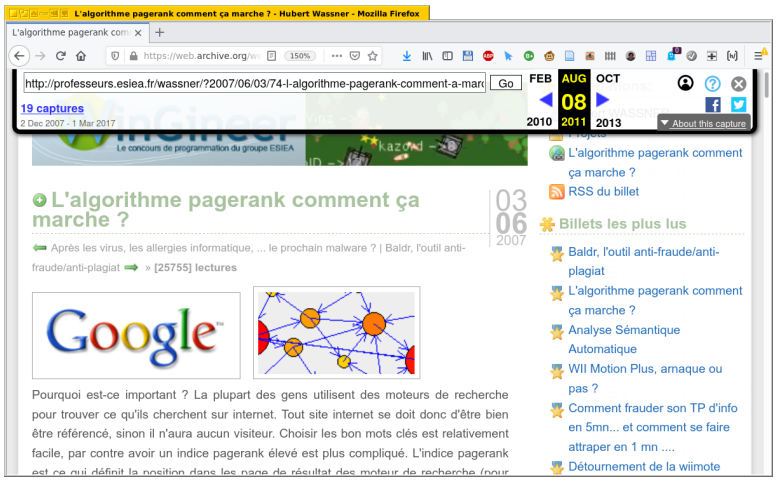


FIGURE 3.3 : Exemple de rendu d'une page en utilisant la Wayback Machine. On peut y voir le controleur dans le haut qui est superposé au contenu de la page ; celui-ci indique que cette page a été pise en août 2011.[44] Autorisation obtenue par River Publishers.

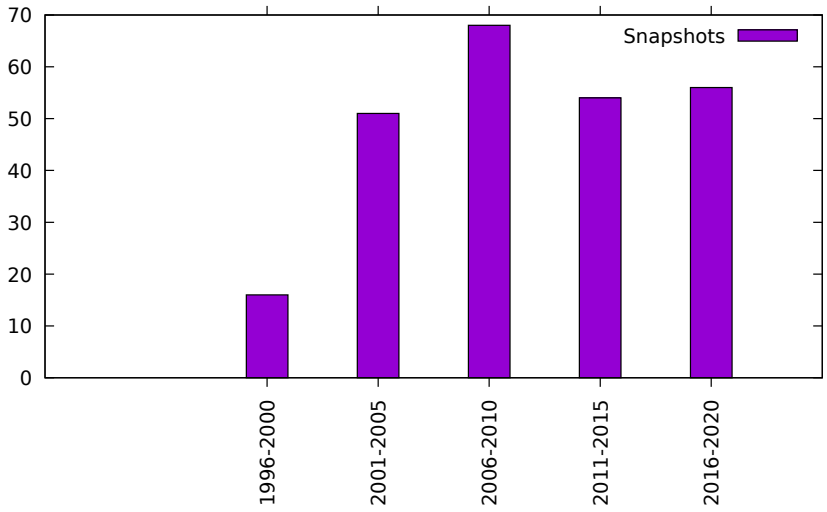


TABLEAU 3.1 : Nombre de pages récoltées pour chaque tranche de 5 ans.[44] Autorisation obtenue par River Publishers.

un traitement sur ces données puis produit un résultat sous forme de graphique. Il est même possible d'effectuer des traitements sur plusieurs expériences en même temps afin de produire des statistiques ou résultats plus génériques. Ainsi, en définissant chaque fichier JSON, donc chaque site, comme une expérience, il a été possible de traiter les données reçues site par site tout comme en lot afin d'obtenir les données suivantes. Il est à noter que ce ne sont pas tous les fichiers JSON récoltés qui ont été utilisés : tout fichier contenant moins de cinq nœuds DOM ou dont l'URL fait partie d'une liste de domaines identifiés comme de la publicité a été retiré. Cette règle générale, qui permet d'éliminer les cites publicitaires et popups, a été préférée à l'exclusion manuelle des fichiers, qui aurait présenté un travail fastidieux.

3.2.1 ANALYSE LONGITUDINALE DE SITE WEB

Le traitement des données recueillies via l'utilisation de la *Wayback Machine* vont permettre d'avoir un aperçu de l'évolution de la programmation des pages web au fil des ans, et ainsi permettre de mieux comprendre comment les sites web ont évolué.

STRUCTURE DES PAGES

Tel que montré dans la figure 3.4, la taille médiane des pages n'a cessé d'augmenter depuis 1996. Alors que la page médiane entre les années 1996 et 2000 contenait un peu moins de 300 éléments, vingt ans plus tard la page médiane en compte un peu plus de 800. Toutefois, la figure 3.5 apporte une nuance intéressante à la figure précédente en montrant que la profondeur médiane des pages web, quant à elle, n'a pas eu de réelle augmentation ou diminution au fil des ans, restant relativement stable entre 12 et 18 nœuds de profondeur. Il est donc possible d'en conclure qu'en 20 ans, les arbres DOM représentant un site web sont devenus plus volumineux mais en gagnant en largeur plutôt qu'en profondeur.

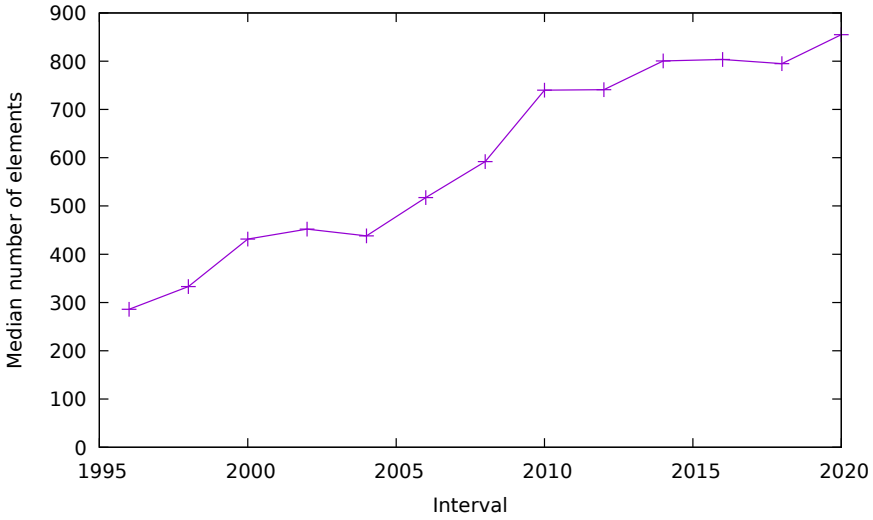


FIGURE 3.4 : Taille médiane de l'arbre DOM au fil des ans.[44] Autorisation obtenue par River Publishers.

UTILISATION DE L'ATTRIBUT `CLASS`

Notre analyse a ensuite porté sur l'utilisation du CSS, plus précisément l'utilisation des classes. Il a été remarqué qu'entre 1996 et 2000 presque aucun site n'utilisait les classes, préférant l'attribution du CSS grâce à l'attribut *id* ou en passant directement par une attribution globale à toutes les balises de même type, par exemple appliquer des règle CSS à toutes les balises *p* de la page. En regardant la figure 3.6, qui compare le nombre de classes et la taille des pages web pour deux intervalles de 5 ans, il est possible de distinguer certaines tendances. Alors que dans le premier graphique les points forment grossièrement une ligne horizontale, ne montrant aucune différence entre le nombre de classes et la taille des pages, le second semble indiquer une tendance plus significative. Le tableau 3.2, montrant les coefficients de corrélation de Pearson, montre la même tendance : au fil des ans le lien entre le nombre de classes et la taille des pages semble de plus en plus fort.

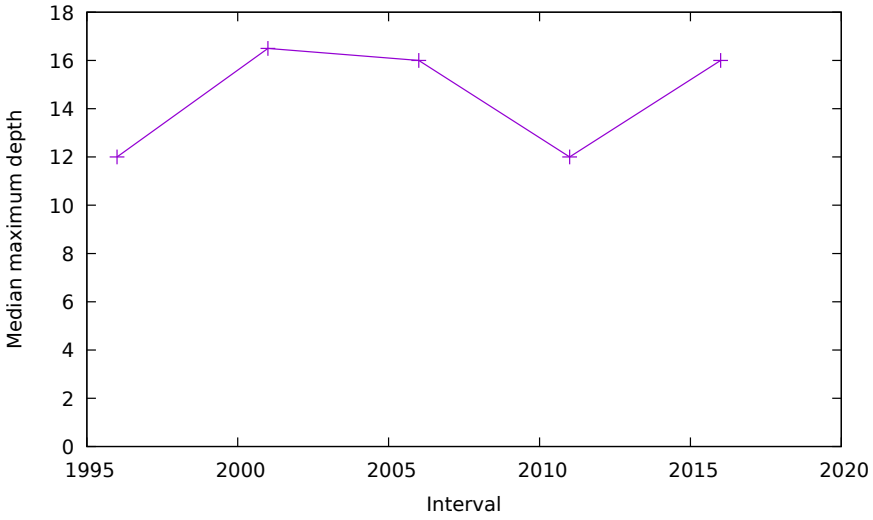
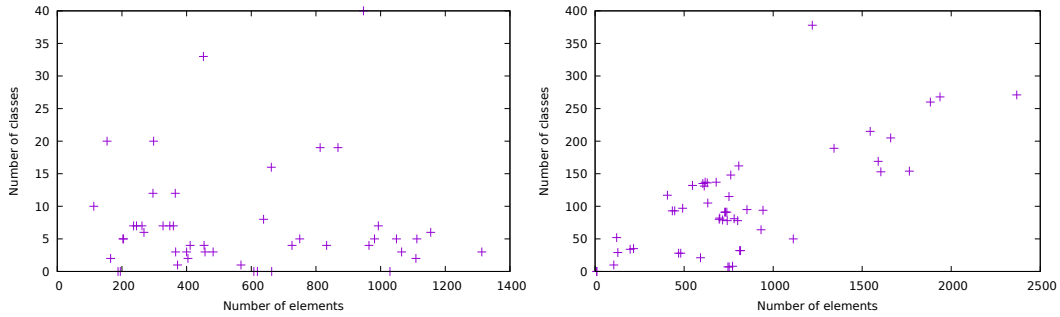


FIGURE 3.5 : Profondeur médiane de l'arbre DOM au fil des ans.[44] Autorisation obtenue par River Publishers.



(a) 2001-2005 (b) 2011-2015

FIGURE 3.6 : Nuage de points représentant le nombre de classes vs nombre d'éléments, pour deux intervalles de 5 ans.[44] Autorisation obtenue par River Publishers.

Years	r
2001–2005	-0.001
2006–2010	0.652
2011–2015	0.734
2016–2020	0.835

TABLEAU 3.2 : Coefficient de corrélation de Pearson (r) entre la taille du DOM et le nombre de classes, pour chaque intervalle de 5 ans.[44] Autorisation obtenue par River Publishers.

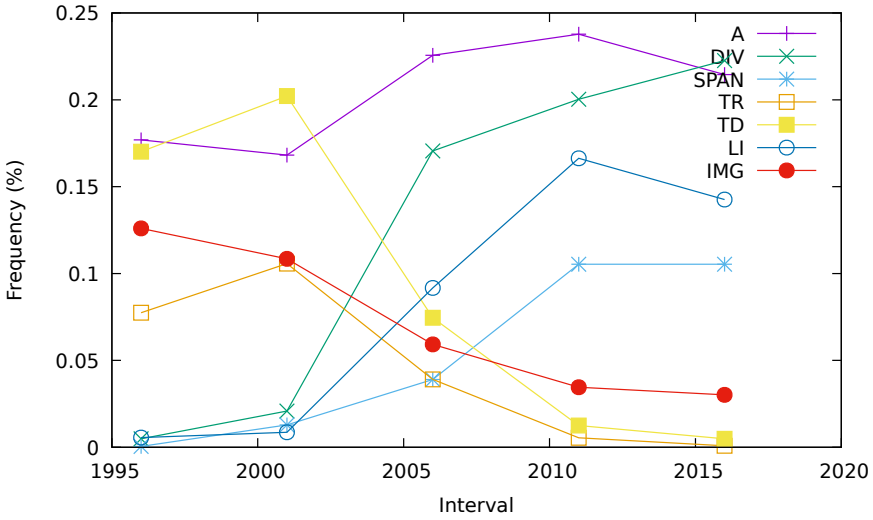


FIGURE 3.7 : Évolution de l'utilisation des balises au fil des ans.[44] Autorisation obtenue par River Publishers.

UTILISATION DES BALISES

La figure 3.7, qui montre les proportions relatives de plusieurs balises HTML dans les pages web, dénote un changement de tendance intéressant entre les années 2001 et 2006. Alors qu'en 2001 la présence des balises `tr` et `td` était proéminente dans les pages et que `div` et `span` étaient presque inexistant, la situation en 2006 est totalement inversée. Ce changement de tendance s'explique par le fait que les vieilles pratiques voulaient que la structure des pages soit générée grâce à des tableaux alors que les pratiques plus récentes poussent à l'utilisation de balises non sémantiques associées à du CSS pour concevoir la structure de la page.

Suite à cette découverte de changement de tendance, il semblait pertinent de regarder l'évolution de la présence de balises non sémantiques dans une page. On peut voir les résultats de cette analyse dans la figure 3.8, où il est possible de constater une augmentation de la proportion d'éléments non sémantiques dans une page à raison d'environ 1% par année.

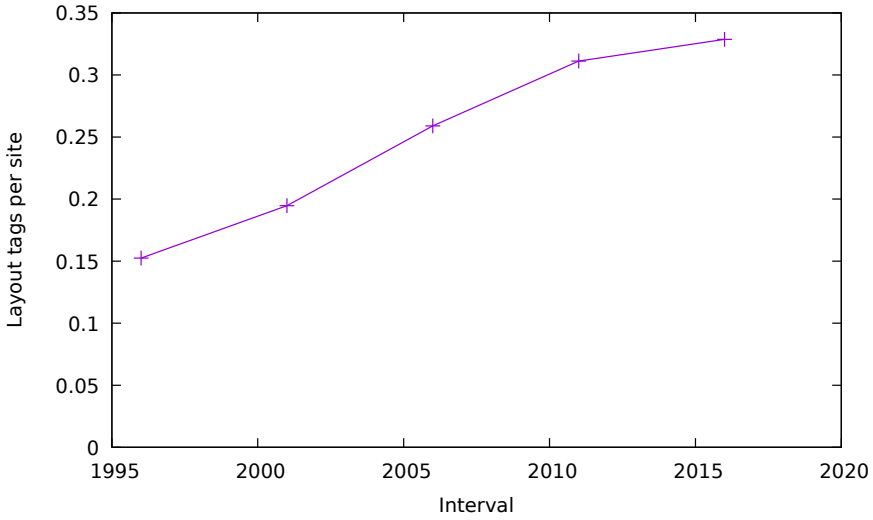


FIGURE 3.8 : Fraction de balises de mise en page utilisées dans une page.[44] Autorisation obtenue par River Publishers.

UTILISATION DU HTML5

Une analyse a aussi porté sur l'utilisation du HTML5, dont une première version a été publiée en 2008 [47] et une version finale en 2014 [48]. La figure 3.9 montre la proportion de sites utilisant au moins une des 25 nouvelles balises introduites par HTML5 ; exceptionnellement, cette figure utilise des intervalles de 2 ans afin de mieux observer l'évolution de l'utilisation de ce nouveau standard. La figure montre clairement que l'utilisation du HTML5 a débuté dès la sortie de sa première version en 2008. L'évolution de son utilisation semble suivre une courbe logistique où le ralentissement survient autour de 2015 pour plafonner à un peu plus de 80% d'utilisation entre 2018 et 2020. Il est à noter que la présence de HTML5 avant 2008 est attribuable uniquement à l'utilisation de la balise `wbr` dans quelques sites, chose très surprenante considérant que, selon W3Schools[49], les navigateurs ne supportaient pas cette balise avant 2008.

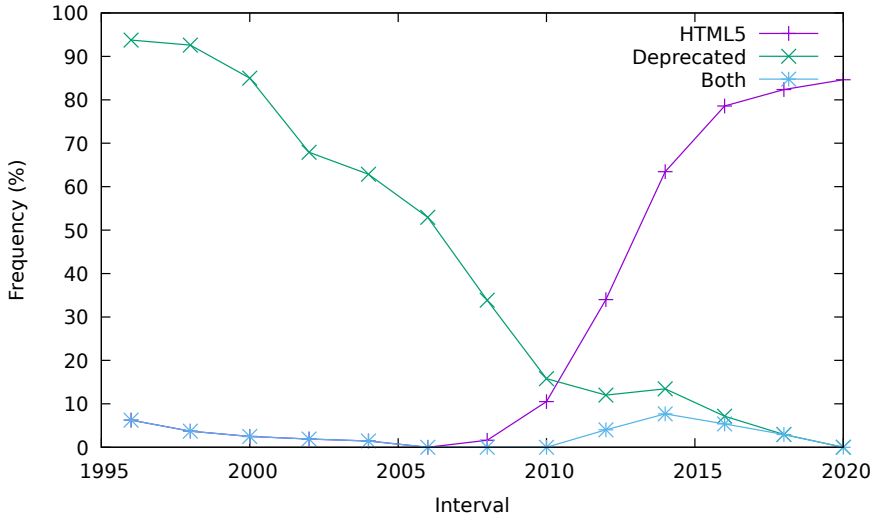


FIGURE 3.9 : Proportion de sites utilisant HTML5.[44] Autorisation obtenue par River Publishers.

La deuxième ligne du graphique montre la proportion de sites qui utilisent des balises devenues obsolètes avec HTML5. Alors qu'au début presque tous les sites en utilisaient, à la publication de HTML5 moins du quart des sites utilisaient encore des balises obsolètes. La troisième et dernière ligne de la figure 3.9 montre la proportion de sites utilisant à la fois des balises apparues avec HTML5 et des balises rendues obsolètes à cause d'HTML5, créant des documents qui sont invalides selon tout standard. Ce pic de documents hybrides est survenu lors de la sortie finale de HTML5, montrant une période de transition lors de l'arrivée de la version finale de ce nouveau standard. Ces hybrides ne sont maintenant plus présents, tel que le montre la troisième courbe qui retourne à 0% en 2020.

PRÉVALENCE DU JAVASCRIPT

De la même façon que pour HTML5, il est possible d'analyser l'évolution de l'adoption du JavaScript en comptant les sites qui contiennent au moins une balise `script`, élément inclus dans une page peu importe si le script a été écrit directement dans la page ou qu'il

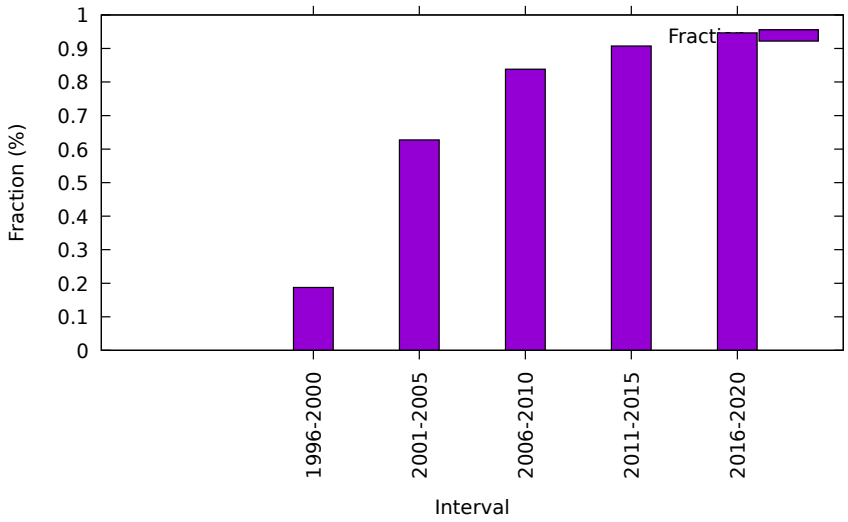


FIGURE 3.10 : Proportion de sites utilisant les balises `script` et `embed`.[44] Autorisation obtenue par River Publishers.

provient d'une source externe. La figure 3.10 montre que le nombre de sites utilisant la balise `script` est passé d'environ 10% à presque 100%. Bien que cette observation indique que presque tous les sites utilisent du JavaScript, elle ne nos permet pas de déterminer à quel point ils en utilisent. Le script de récupération de données présenté plus tôt ne permet pas d'obtenir cette information. Toutefois, en comptant le nombre de balises `script` présentes dans la page, on peut en avoir une mesure indirecte. La figure 3.11 montre ces résultats, où il est possible de constater une augmentation constante du nombre de balises `script` présentes dans une page, à raison d'environ une de plus par année. Il est donc possible d'en déduire que le JavaScript est utilisé en quantité croissante.

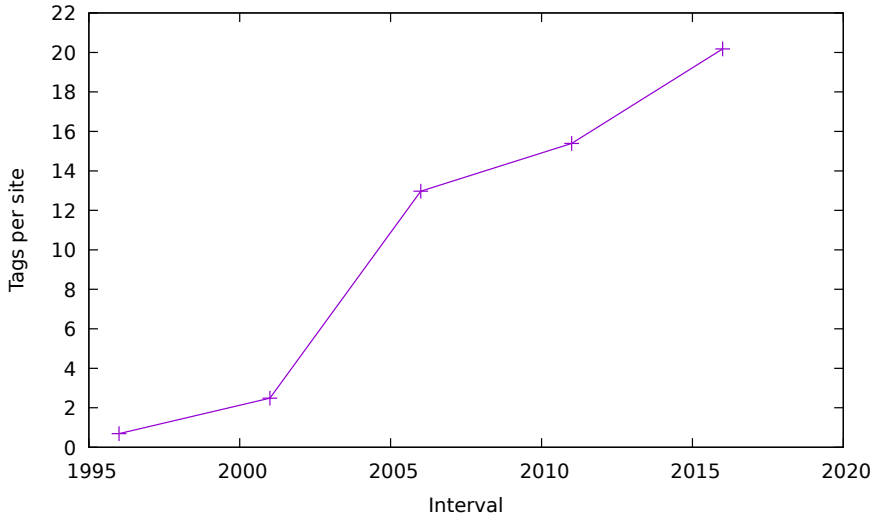


FIGURE 3.11 : Nombre moyen de balises `script` par page.[44] Autorisation obtenue par River Publishers.

ÉLÉMENTS INVISIBLES

La figure 3.12 montre les résultats obtenus en étudiant les méthodes utilisées, au fil des ans, pour rendre un élément dans la page invisible à l'utilisateur. Alors que dans les 20 premières années, les éléments sont rendus invisibles majoritairement grâce à des positions x ou y très grandes afin de les faire sortir de la page, les années suivantes montre un changement de pratique en attribuant majoritairement une valeur z, soit le niveau de superposition de l'élément, négative afin de le cacher en arrière des autres. Alors que la pratique de donner une dimension de 0 à un élément n'a jamais été utilisée, on observe une certaine constance au fil des ans quant à l'utilisation des attributs `display` et `visibility` pour gérer l'invisibilité.

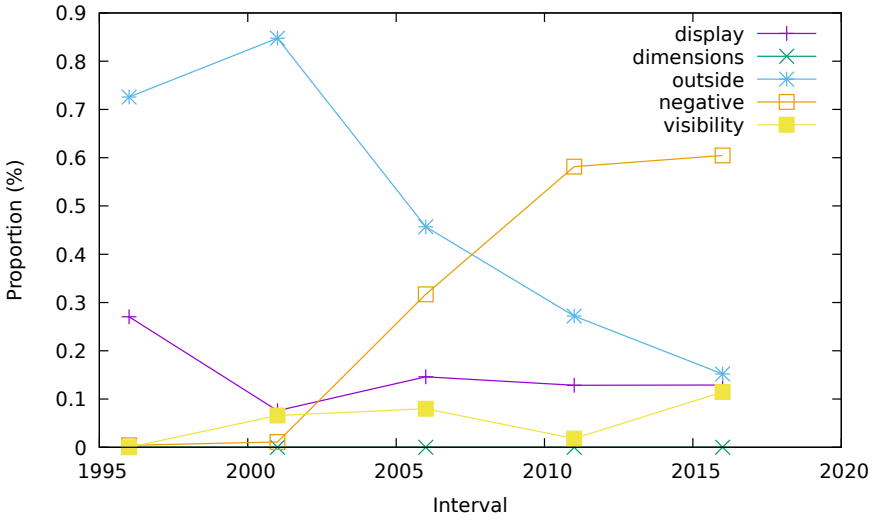


FIGURE 3.12 : Évolution de l'invisibilité au fil des ans.[44] Autorisation obtenue par River Publishers.

3.2.2 ANALYSE EMPIRIQUE DE SITE WEB

À partir des résultats de cette étude longitudinale, il est possible de remarquer plusieurs transitions marquées et importantes, notamment l'utilisation du HTML 5, l'augmentation dramatique de l'utilisation de scripts JavaScript, l'augmentation de la taille des pages ainsi que les différences d'utilisation des balises. Ces changements importants justifient de s'attarder plus en profondeur à la structure des pages web contemporaines.

STRUCTURE DE L'ARBRE DOM

Les premières statistiques recueillies cherchent à aider à mieux comprendre à quoi ressemble le profil type de l'arbre DOM d'un page HTML. La figure 3.13, montrant la distribution cumulative des sites quant à la taille de leur arbre DOM, permet de constater que la répartition suit une fonction exponentielle inversée. Cette courbe, suivant la règle $f(x) = 1 - \frac{0.83}{e^{0.0011x}}$ avec un coefficient de détermination $R^2 = 0.999$, montre que 90% des sites

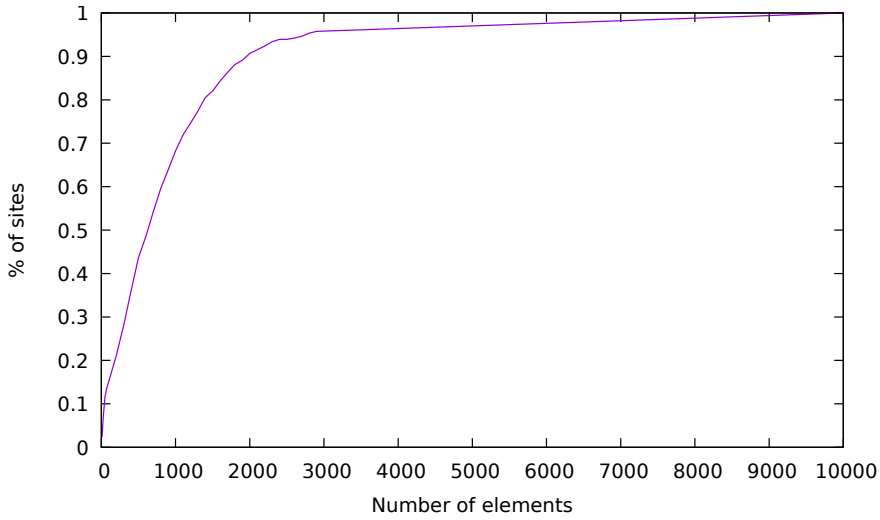


Figure 3.13 : Répartition des sites web selon la taille de leur arbre DOM.[44] Autorisation obtenue par River Publishers.

web sont constitués de 2000 nœuds HTML ou moins. En utilisant ladite règle, il est possible de conclure qu'un site de taille moyenne, soit $f(x) = 0.5$, comporte environ 460 nœuds HTML.

La figure 3.14, quant à elle, représente la distribution des sites web selon la profondeur de l'arbre de DOM, autrement dit le nombre maximal d'imbrications d'éléments HTML au sein de la page web. D'après ce graphique, 39% des sites web ont une profondeur d'arbre DOM maximale variant de 10 à 16.

Une analyse portant sur le degré de l'arbre DOM a aussi été réalisée. On peut voir dans les figures 3.15 et 3.16, que 50% des sites ont un degré maximal de 22. Il est à noter que cette nouvelle courbe suit la règle suivante $f(x) = 1 - \frac{4.91}{x^{0.92}}$. Comparativement à la première, cette fonction présente un coefficient de détermination $R^2 = 0.894$.

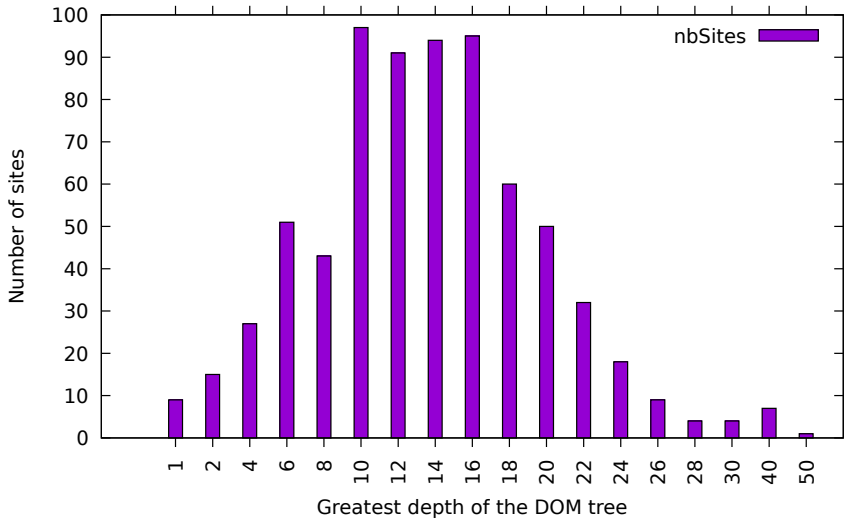


FIGURE 3.14 : Distributions des sites web selon la profondeur de l'arbre DOM.[44] Autorisation obtenue par River Publishers.

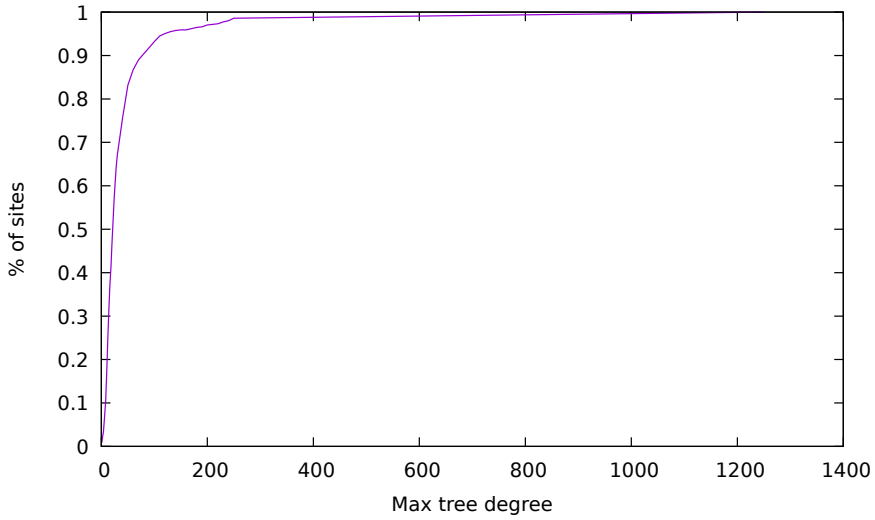


FIGURE 3.15 : Répartition des sites web selon le degré maximal des nœuds.[44] Autorisation obtenue par River Publishers.

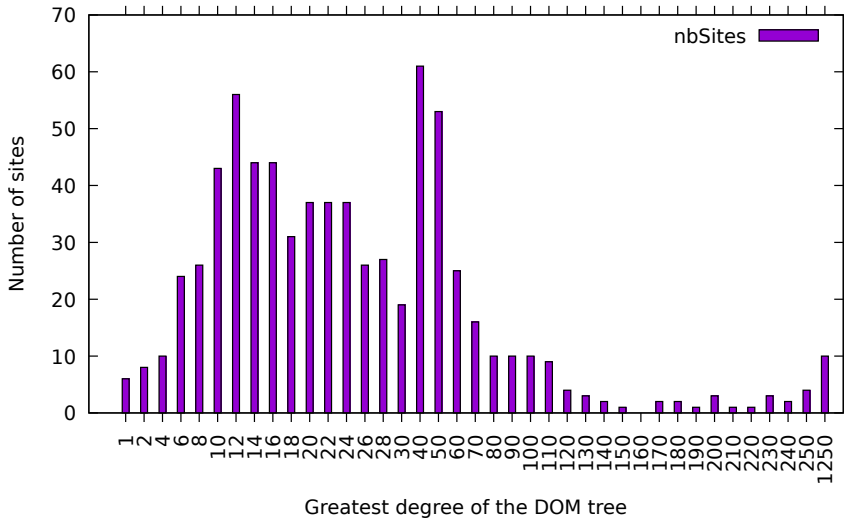


FIGURE 3.16 : Distribution des sites web selon le degré maximal des nœuds.[44] Autorisation obtenue par River Publishers.

Une dernière analyse a été menée sur la structure de l'arbre DOM, soit les nœuds qui le composent. Les nœuds liés au balisage en ligne de SVG n'ont pas été compilés puisque c'est la structure de la page qui est pertinente dans ce cas-ci. La figure 3.17 montre la répartition des nœuds dans les pages. Avec ses 26%, le `div` est l'élément le plus fréquent dans les pages web, suivi par le `a` avec 19%, le `li` avec 11% et le `span` avec 10%. Ces quatre éléments représentent les deux tiers des éléments présents dans des pages web, alors que tout autre élément n'a pas de prévalence dépassant 4%. Autre fait saillant : les `div` et `span`, qui ne servent qu'à contenir d'autres éléments, représentent 36% des éléments trouvés dans des pages web. On en déduit donc que les éléments non sémantiques représentent environ le tiers des éléments de balisage HTML utilisés.

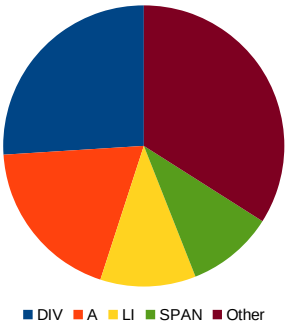


FIGURE 3.17 : Proportion relative d'utilisation des balises HTML dans les pages web.[44] Autorisation obtenue par River Publishers.

STATUT DE VISIBILITÉ

De nos jours, la majorité des sites web contiennent des éléments qui ne sont pas visibles par l'utilisateur ; des exemples fréquents de tels éléments sont les objets d'un carrousel. La figure 3.18 montre un exemple d'un tel carrousel. Le code produisant ce carrousel peut être retrouvé dans la figure 3.19. Toutefois, l'analyse des sites a révélé qu'il y avait beaucoup plus d'éléments cachés que simplement ceux des carrousels : 54% de tous les éléments HTML analysés étaient invisibles à l'utilisateur. La figure 3.20 montre la distribution du nombre de sites selon le pourcentage d'éléments qui sont invisibles à l'utilisateur. La distribution entre chaque intervalle de 10% est assez uniforme au travers du graphique.

On remarque aussi que les tendances observées dans la Section 3.2.1 se maintiennent. Tel qu'il peut être vu dans le tableau 3.3, aucun des éléments invisibles rencontrés n'utilisait la propriété CSS `display : none` ou avait une largeur ou hauteur de 0. Au lieu de cela, environ 15% utilisent la propriété CSS `visibility : hidden`, environ 24% des éléments se font attribuer une position à l'extérieur de la page et 60% se font attribuer une superposition négative, c'est-à-dire que l'élément est invisible puisqu'il est en dessous d'un autre.

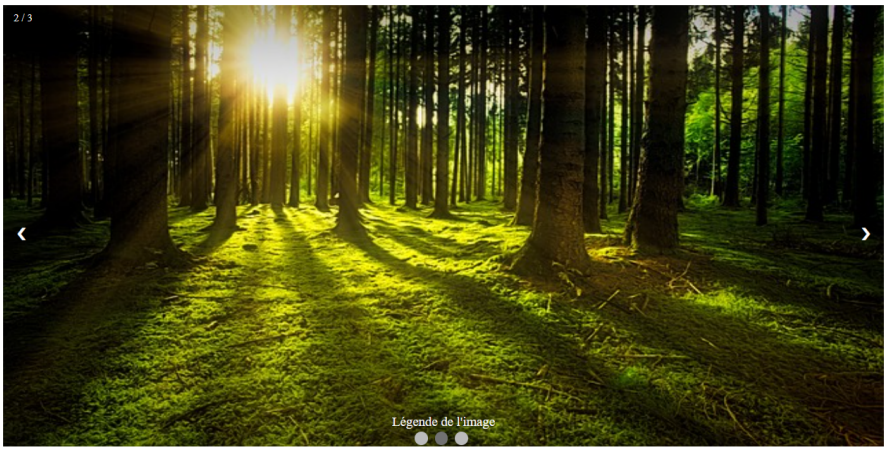


FIGURE 3.18 : Exemple de rendu d'un carrousel dans une page web. ©Xavier Chamberland-Thibeault, 2023

Elements	Invisible Type
0	Display
0	Width or height
131334	Visibility
213840	Outside position
536700	Negative position

TABLEAU 3.3 : Nombre total d'éléments utilisant chacun des types d'invisibilité.[44] Autorisation obtenue par River Publishers.

CLASSES CSS

Le dernier aspect de cette analyse porte sur les classes CSS et leur utilisation. Toutes les classes étant appliquées, directement dans le HTML ou par JavaScript, à au moins un élément du DOM ont été traitées. La figure 3.21a montre la relation entre le nombre de classes CSS présentes dans le document et la taille de l'arbre DOM. Grâce à ce graphique, il est possible de constater une dépendance assez faible entre les deux variables.

```html
<!DOCTYPE html>
<html>
    <head>
        <title>Exemple de carrousel</title>
        <link rel="stylesheet" href="carrousel.css">
    </head>
    <body>
        <div class="conteneur">
            <div class="mesImages visible">
                <div class="nombre">1 / 3</div>
                <img src="img/img1.jpg">
                <div class="texte">Légende de l'image</div>
            </div>
            <div class="mesImages">
                <div class="nombre">2 / 3</div>
                <img src="img/img2.jpg">
                <div class="texte">Légende de l'image</div>
            </div>
            <div class="mesImages">
                <div class="nombre">3 / 3</div>
                <img src="img/img3.jpg">
                <div class="texte">Légende de l'image</div>
            </div>
            <a class="precedent" onclick="changerImage(-1)">&#10094;</a>
            <a class="suivant" onclick="changerImage(1)">&#10095;</a>
            <div class="points">
                <span class="point actif" onclick="imageActuelle(1)"></span>
                <span class="point" onclick="imageActuelle(2)"></span>
                <span class="point" onclick="imageActuelle(3)"></span>
            </div>
        </div>
    </body>
    <script src="carrousel.js"></script>
</html>
```

(a) Code HTML

```css
* {box-sizing:border-box}
img {
  width: 1000px;
  height: 500px;
}
.conteneur {
  max-width: 1000px;
  position: relative;
  margin: auto;
}
.mesImages {display: none;}
.precedent, .suivant {
  position: absolute;
  top: 50%;
  width: auto;
  margin-top: -22px;
  padding: 16px;
  color: white;
  font-weight: bold;
  font-size: 18px;
  transition: 0.6s ease;
  border-radius: 0 3px 3px 0;
  user-select: none;
}
.suivant {
  right: 0;
  border-radius: 3px 0 0 3px;
}
.texte {
  color: #f2f2f2;
  font-size: 15px;
  padding: 8px 12px;
  position: absolute;
  bottom: 15px;
  width: 100%;
  text-align: center;
}
.nombre {
  color: #f2f2f2;
  font-size: 12px;
  padding: 8px 12px;
  position: absolute;
  top: 0;
}
.point {
  height: 15px;
  width: 15px;
  margin: 0 2px;
  background-color: #bbb;
  border-radius: 50%;
  display: inline-block;
}
.actif{
  background-color: #717171;
}
.points{
   position: absolute;
   bottom: 2px;
   left: 0;
   right: 0;
   width: 70px;
   margin: auto;
}
```

(b) Code CSS

```javascript
let imageAffichee = 1;
afficherImage();
function changerImage(n) {
    imageAffichee += n;
    afficherImage();
}
function imageActuelle(n) {
    imageAffichee = n;
    afficherImage();
}
function afficherImage() {
    let images = document.getElementsByClassName("mesImages");
    let points = document.getElementsByClassName("point");
    if (imageAffichee > images.length) {imageAffichee = 1}
    if (imageAffichee < 1) {imageAffichee = images.length}
    for(let i = 0; i < images.length; i++)
    {
        images[i].style.display = "none";
    }
    for(let i = 0; i < points.length; i++)
    {
        points[i].className = points[i].className.replace(" actif", "");
    }
    images[imageAffichee-1].style.display = "block";
    points[imageAffichee-1].className += " actif";
}
```

(c) Code JavaScript

FIGURE 3.19 : Exemple de code permettant d'afficher un carrousel dans une page web. ©Xavier Chamberland-Thibeault, 2023

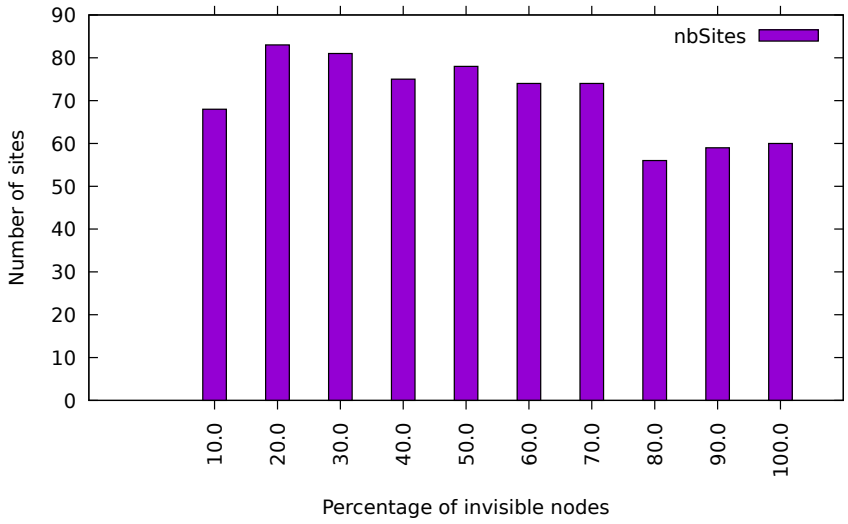
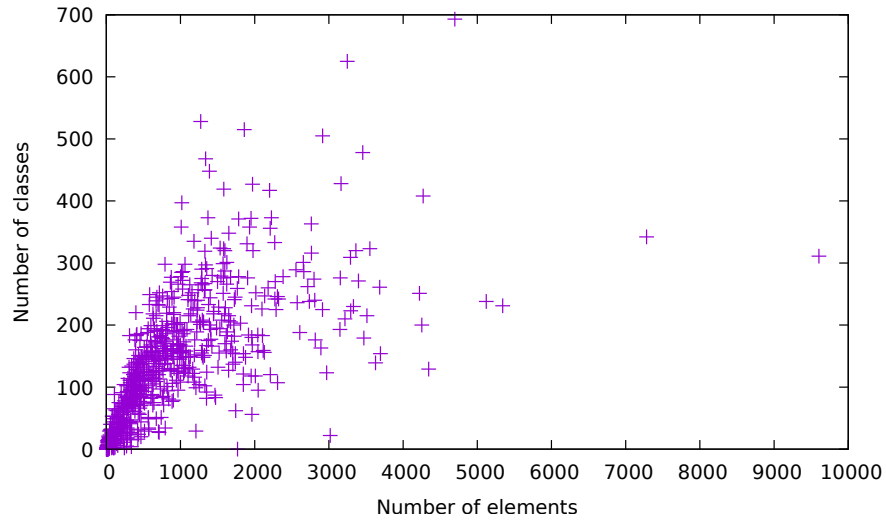


FIGURE 3.20 : Distribution des sites web selon le pourcentage de nœuds du DOM qui sont invisibles.[44] Autorisation obtenue par River Publishers.

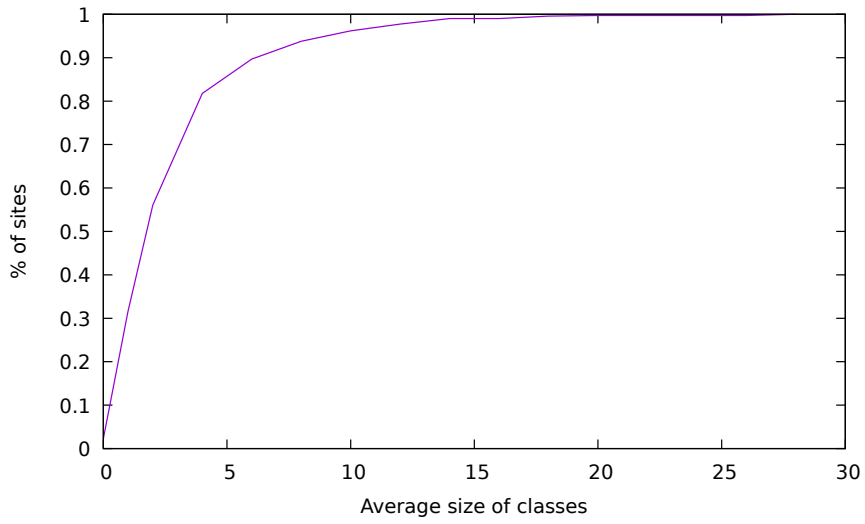
Une analyse portant sur la taille des classes CSS, soit le nombre d'éléments distincts qui sont associés à cette classe, a été réalisée. La figure 3.21b en montre les résultats. Encore une fois, il est possible d'observer une courbe exponentielle inversée, répondant à l'équation $f(x) = 1 - \frac{1.72}{x^{1.93}}$,où $f(x)$ donne la proportion des sites ayant une taille de classe CSS moyenne de x. Cette fonction, dont le coefficient de détermination est $R^2 = 0.931$, nous permet de conclure que plus de 90% des sites web ont des classes CSS qui s'appliquent à moins de 10 éléments dans la page.

3.3 MENACES POUR LA VALIDITÉ

Tel que précisé dans l'article [44], nous avons identifié plusieurs éléments pouvant porter préjudices à la validité des résultats présentés ci-dessus : les sites faisant partie de l'échantillon testé, le navigateur grâce auquel l'étude à été mené, l'analyse de pages d'accueil ainsi que l'utilisation de la *Wayback Machine*.



(a) Taille de l'arbre DOM vs. nombre de classes CSS.



(b) Répartition des sites web selon la taille moyenne des classes CSS.

FIGURE 3.21 : Statistiques à propos de l'utilisation des classes CSS.[44] Autorisation obtenue par River Publishers.

3.3.1 COMPOSITION DE L'ÉCHANTILLON

Il est important de comprendre qu'il existe une dépendance entre les distributions et statistiques présentées ci-avant et les sites constituant l'échantillon à analyser. Afin de tenter de palier à cette problématique, nous nous sommes servis de deux méthodes d'échantillonnage différentes. Nous avons exploité la liste des 500 sites les plus visités de Moz [50] afin d'avoir un échantillonnage contenant une bonne fraction de sites très fréquentés. Ensuite, nous avons aussi recueilli des sites web suggérés par des gens, selon l'usage quotidien qu'ils font du web. Ce duo d'échantillonnage combiné à la quantité de sites recueillis réduisent les chances d'avoir un échantillon uniquement composés de cas exceptionnels. Malgré tout, les sites ayant été conçus avec le même Système de Gestion de Contenu (SGC) ou utilisant les mêmes patrons auront une structure semblable. Ainsi, une structure de page web pourrait être sur-représentée.

Malheureusement, dû au temps de récupération des données largement accru lors de l'utilisation de la *Wayback Machine* et la difficulté de savoir pour quels sites il existe une capture à un moment précis dans le temps, l'étude longitudinale [44] s'est faite avec un échantillon moindre comparativement à l'étude contemporaine [43]. Dans le but de mitiger cette réduction d'effectifs, nous avons regroupé les sites en grappe par année de la capture, variant de deux à cinq ans selon l'évaluation qui était appliquée. Cependant, dû à l'inconsistance de la présence des sites au travers du temps dans la *Wayback Machine*, chaque grappe ne contient pas forcément la même quantité de sites. Malgré tout, la majeure partie des résultats ont été obtenus grâce à l'analyse d'au moins 60 pages et démontrent des tendances si fortes qu'on peut en dédire que la taille de l'échantillon serait assez grande pour être représentative.

3.3.2 L'ANALYSE DE PAGES D'ACCUEIL

Pour chacun des sites visités, seule la page d'accueil a été analysée afin de pouvoir avoir une méthodologie uniforme pour chacun des sites. La majeure partie des sites web de la liste utilisée ont une page d'accueil entièrement fonctionnelle et utilisable, même si certains sites offrent des sections différentes à un utilisateur connecté plutôt qu'à un utilisateur dit « invité ». Il y a toutefois des exceptions à cette règle, tel que Facebook, qui ne présenteront rien de plus qu'un formulaire d'authentification pour tout utilisateurs non-connectés. Ces sites ont évidemment des répercussions sur la taille d'une page et son arbre DOM.

Il est impossible d'éviter ce phénomène sans amener une grande complexité lors de la collecte des données. En effet, il serait possible de choisir une page « représentative » pour chacun des sites web et récupéré seulement le contenu de ces pages. Hors, pour arriver à cela, il faudrait, pour chaque site web, déterminer les étapes à suivre pour permettre au *crawler* d'atteindre et cette page sans compter qu'il faudrait aussi introduire le concept de « page représentative », qui en soit, pourrait être une menace à la validité.

Aussi, les pages autres que les pages d'accueil sont propices à la génération de contenu dynamiquement, notamment via la nouvelle tendance au défilement infini. Bien que ça semble être une menace à la validité de l'étude, le contenu généré dynamiquement respecte souvent la même structure, utilisation de balises et méthode pour rendre les éléments invisibles que le contenu précédemment affiché. Ainsi, les proportions des pages, hormis leur taille, ne devraient pas être impactées par ce phénomène.

3.3.3 VARIANCES DUES AUX NAVIGATEURS

Il est à noter qu'un seul et unique navigateur fut utilisé pour cette étude : Mozilla Firefox. Ce choix de navigateur n'est en rien dû à des technicités, étant donné que les scripts,

via TamperMonkey, peuvent être exécutés sur n'importe quel navigateur. Puisqu'il existe certaines différences dans le rendu d'une page selon le navigateur utilisé, l'arbre DOM pourrait varier selon le navigateur utilisé pour récupérer les sites constituant l'échantillon. Par contre, considérant que la majeure partie des navigateurs ont grandement amélioré leur conformité [51] et que la majorité des erreurs de conformité récentes affectent le visuel de la page sans pour autant impacter son contenu, les statistiques présentées ne sont pas affectées par cette problématique. Il est à noter que, pour cette étude, la taille de la fenêtre du navigateur était fixée et qu'une variation dans la taille d'affichage peut avoir un impact sur les résultats obtenus.

Un argument pourrait être fait sur l'étude longitudinale quant à l'utilisation d'un navigateur récent pour afficher des sites datant de quelques années à quelques décennies. En effet, certaines règles de présentations d'éléments ont pu varier au fil du temps, telle que la taille de police par défaut ou les marges applicables à un élément. Toutefois, toutes ces différences ne s'appliquent qu'au visuel final présenté par le navigateur et ne modifie en rien le contenu de la page, ainsi les statistiques n'en sont pas affectées, tel que mentionné dans le paragraphe précédent.

3.3.4 UTILISATION DE LA WAYBACK MACHINE

Cette menace s'applique uniquement aux statistiques portant sur plusieurs années, puisque pour les données contemporaines l'utilisation de la *Wayback Machine* n'était pas nécessaire. Bien que la majorité des modifications faites par la *Wayback Machine* soient facilement réversibles, tel que l'ajout du contrôleur, ou sans impact sur le rendu final, telle que la modification d'un chemin d'accès à une ressource externe comme une image, la *Wayback Machine* n'offre pas un rendu exact des pages.

Deux autres questions se posent sur l'outil, notamment la composition des pages [52] et l'intégrité des captures d'un site [53]. En effet, une page de la *Wayback Machine* peut être différente du rendu réel qu'elle avait à un moment précis dans le temps puisqu'elles peuvent être des assemblages de plusieurs captures faites sur une période de temps rapprochée. Hors, il est peu probable que cet assemblage ait un effet significatif sur la structure de l'arbre DOM puisque l'utilisation d'une balise donnée a peu de chance de varier, pour un même site, lors de deux captures faites à des moments rapprochés dans le temps. En ce qui concerne l'intégrité des pages, dû à des failles de sécurité, les captures d'une page peuvent être modifiés par une personne tierce. Il est impossible d'identifier quel sites et encore moins quelles captures ont été modifiées. Ceci, bien que peu probable, peut avoir un impact sur les résultats présentés dans ce chapitre.

CHAPITRE IV
CORRECTION DES BOGUES : APPROCHE ITÉRATIVE

Le chapitre 1 nous a permis de faire l'éventail des types de bogues visuels qui peuvent se manifester lors du rendu d'une page web. Le chapitre 3 nous a amené à mieux connaître le profil structurel d'une page web contemporaine en plus de nous permettre d'avoir une idée de ce qu'est une page réaliste lors de tests d'un outil. Le chapitre 2 démontre que la majeure partie des travaux antérieurs, réalisés pour aider les développeurs lors de la création d'une page web, se concentrent majoritairement sur la détection des bogues, mais très peu sur leur correction. C'est le problème auquel va s'attaquer le reste du mémoire.

L'approche proposées repose sur la notion de contraintes et s'inscrit dans la suite de l'outil développé à l'Université du Québec à Chicoutimi (UQAC), soit la librairie Cornipickle[22]. Dans ce chapitre, nous débuterons avec la présentation de l'outil permettant de détecter les violations de contraintes dans une page tout en identifiant les éléments qui en sont la cause. Ensuite, nous allons présenter une extension développée dans le cadre de ce mémoire. Celle-ci se sert des résultats de Cornipickle pour aller un pas plus loin : au lieu de simplement indiquer à l'utilisateur quels éléments violent des contraintes, ainsi que la contrainte violée, l'extension va proposer des corrections potentielles permettant aux contraintes d'être à nouveau respectées.

4.1 CORNIPICKLE

Afin d'aborder la problématique qu'est la détection de bogues, Hallé *et al.*, au travers de l'outil Cornipickle, ont développé un langage déclaratif de haut niveau permettant d'établir des contraintes que devrait respecter la page. Le but premier de ce nouveau langage, hormis d'aider à la détection de bogues, est d'établir des contraintes grâce à des phrases en langage naturel. Ce

langage offre plusieurs fonctionnalités : sélectionner des éléments dans une page web, accéder aux propriétés des éléments sélectionnés, énoncer des conditions, grâce à des connecteurs logiques et quantificateurs, que ces éléments devront respecter et, finalement, corréler l'état de plusieurs pages successives grâce à des opérateurs de logique temporelle. Dans l'optique de rendre la grammaire la plus versatile possible, ils ont rendu possible pour l'utilisateur de définir ses propres opérateurs, constantes et regroupement d'éléments qui viendront s'ajouter à ceux de Cornipickle. La figure 4.1 montre un exemple de ce langage déclaratif. Les trois premières lignes de cet exemple montrent la déclaration d'une contrainte nommée *aligned*. Celle-ci spécifie que deux éléments, dans ce cas-ci *x* et *y*, doivent avoir le même alignement gauche. Dans cette déclaration de contrainte on peut voir comment le langage permet d'accéder aux propriétés, ici la propriété *left*, d'un élément. Les lignes permettent d'indiquer quand cette contrainte devrait s'appliquer. Les deux boucles présentes vont permettre d'aller chercher tous les *li* contenus dans un *ul*, autrement dit tous les éléments d'une liste à puces à un seul niveau, pour ensuite les comparer entre eux. En bref, cet exemple dit que tous les éléments d'une liste à puces devraient avoir le même alignement gauche. Une description plus approfondie de la grammaire ainsi que plusieurs exemples sont disponibles dans l'article *Declarative layout constraints for testing web applications*[22].

Arriver à concevoir un tel langage est bien, mais il faut arriver à l'interpréter correctement sans quoi il ne permettra pas d'aider à la détection de bogues et n'aura donc pas lieu d'être. Cette tâche, complexifiée par la possibilité pour l'utilisateur de définir ses propres règles, fut menée à bien grâce au développement de Bullwinkle (https://github.com/sylvainhalle/Bullwinkle) : un analyseur de forme de Backus-Naur, notation sur laquelle est basée la grammaire ci-dessus. Ainsi, suite à l'ajout de cet analyseur, Cornipickle était en mesure d'interpréter toutes contraintes données, à condition qu'elles respectent la grammaire.

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

For each $i in $(ul li) (
   For each $j in $(ul li) (
      $i and $j are aligned
   )
).
```

Figure 4.1 : Exemple de contrainte faite grâce au langage déclaratif de Cornipickle. ©Xavier Chamberland-Thibeault 2023

Il fallait maintenant arriver à récupérer la page web afin de l'analyser. Or cette tâche ne pouvait se faire qu'une fois la page entièrement rendue par le navigateur puisque chaque navigateur interprète le CSS différemment. Une sonde JavaScript a donc été conçue afin d'interagir avec l'application serveur. La figure 4.2 montre l'interaction entre les deux. La sonde va se faire fournir un ensemble de contraintes, préalablement indiquées par le développeur, et un identifiant. Cet identifiant sera injecté dans la page HTML afin que, suite au chargement complet de la page, Cornipickle puisse lui envoyer la bonne sonde. Ensuite, lorsque l'appel au serveur, pour avoir la sonde sera fait, Cornipickle va générer le code JavaScript de la sonde demandée afin qu'elle prenne en compte uniquement les contraintes qui s'appliquent à cette page en particulier. Une fois cette sonde injectée dans la page, elle va récupérer tous les nœuds qui sont pertinents face aux contraintes fournies et leur assignera un identifiant numérique unique nommé *CorniId*. Elle récupérera aussi les propriétés CSS applicables aux contraintes, comme la position de l'élément pour l'exemple de contrainte de la figure 4.1. Toute l'information collectée par la sonde sera ensuite renvoyée à Cornipickle, sous forme de JSON, afin de procéder à l'analyse de la page. Une fois le JSON reçu, Cornipickle va l'analyser en validant que chacune des contraintes est respectée. Grâce à son algorithme de validation des contraintes sur une page web, Cornipickle va être en mesure d'identifier les éléments qui sont en erreur dans la page ainsi que la contrainte que viole chacun des éléments. Il est à noter

que le programme est aussi en mesure d'identifier des paires d'éléments qui, ensemble, violent des contraintes.

Une fois l'analyse complétée, Cornipickle va en transmettre le résultat à la sonde : soit la liste de toutes les contraintes ainsi que, pour chacune de celles-ci, la liste des identifiants numériques correspondant aux éléments qui sont en lien avec sa violation, le cas échéant. Finalement, la sonde va apporter des modifications à la page web afin de faire une rétroaction à l'utilisateur. Tout élément étant mentionné comme potentiellement responsable de la violation de l'une ou l'autre des contraintes sera encadré de rouge. La sonde ajoutera aussi des commentaires indiquant quelle contrainte chacun des éléments fautifs enfreint afin que l'utilisateur sache ce qui ne fonctionne pas dans l'affichage de son site. Un exemple d'un tel rendu est donné dans la figure 4.3 . Dans cet exemple, on peut voir deux éléments d'une même liste à puces qui sont encadrés en rouge. On peut remarque qu'un de ces deux éléments est décalé par rapport au reste des éléments de la liste. On voit aussi qu'en passant la souris sur un des encadrés rouges, un pop-up en bas à droite de l'écran est apparu afin de fournir plus d'informations sur la contrainte violée. Ici, il est indiqué que les éléments d'une liste devraient tous être alignés soit verticalement, donc selon leur alignement gauche comme dans l'exemple de la figure 4.1, soit alignés horizontalement.

4.2 FAULT-FINDER

Puisque Cornipickle arrive à identifier à la fois les contraintes enfreintes et les éléments qui sont potentiellement responsables, il serait possible de tirer parti de cette information pour cibler les corrections à apporter à une page afin de restaurer la validité de ces contraintes. C'est en se basant sur ce principe que Sylvain Hallé et Oussama Beroual [10] ont créé Fault-Finder. Cette nouvelle librairie a pour but de déduire, des propriétés d'un élément fautif et de la

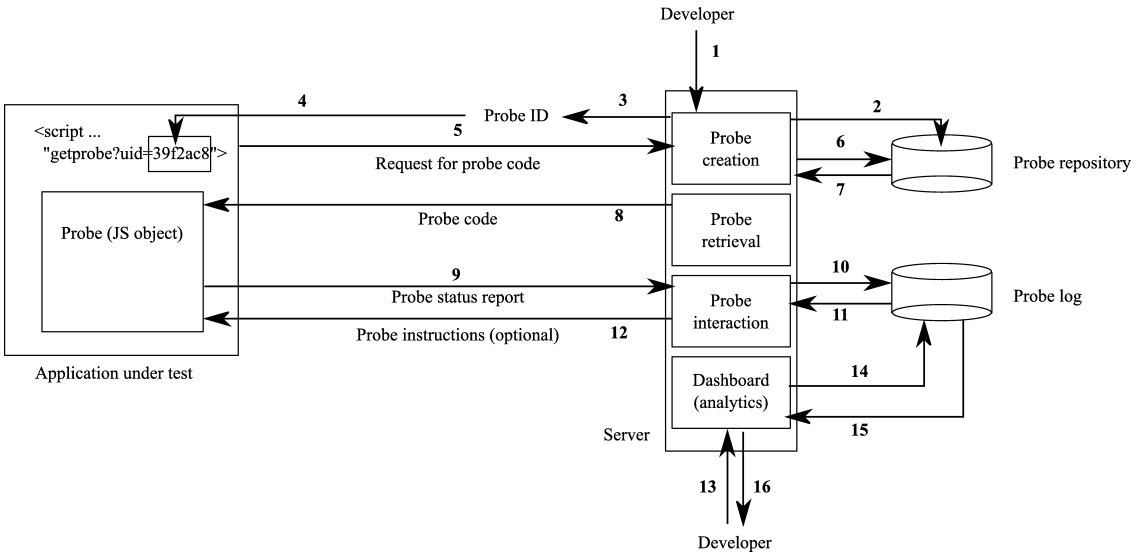


FIGURE 4.2 : Schéma présentant la suite des interactions entre la sonde JavaScript et l'application serveur Cornipickle [22]. Autorisation obtenue par Elsevier.

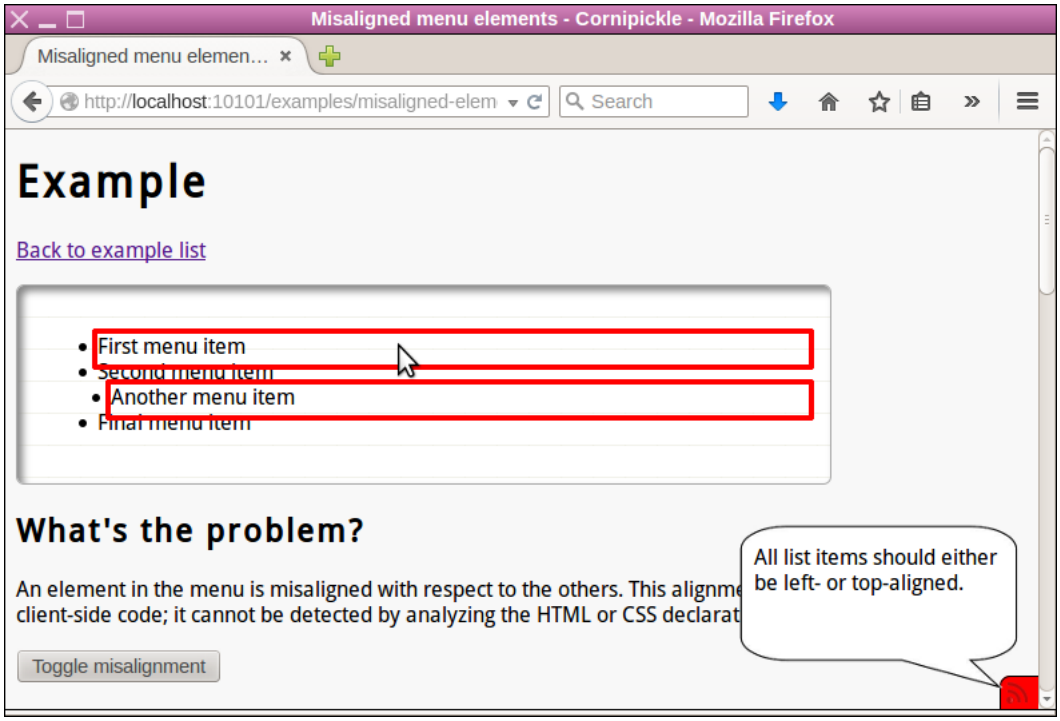


FIGURE 4.3 : Exemple d'un rendu de Cornipickle [22]. Autorisation obtenue par Elsevier.

contrainte qu'il enfreint, un ensemble de modifications à son état qui seraient de potentielles réparations.

Pour ce faire, la librairie utilise les principes suivants. Elle utilise un ensemble arbitraire d'objets, nommé O. Sur cet ensemble est défini un ensemble fini de fonctions de la forme $T : O \to O$, nommé T, qu'on appelle des transformations. Chaque élément de T est une fonction qui transforme des objets en d'autres du même type. Ces transformations peuvent être composées de manière habituelle, soit $t \circ t' = t(t'(o))$ pour tout O. Le dernier principe correspond aux propriétés de o, qui est une fonction quelconque de la forme $P : O \to \{\top, \bot\}$. Un objet o satisfait une propriété p si et seulement si $p(o) = \top$.

Lorsqu'un objet o ne satisfait pas une propriété, Fault-Finder tente d'y trouver une « correction », soit une transformation ou composition de transformations qui, appliquées à o, produisent un objet o', c'est-à-dire l'objet « corrigé », tel que $p(o') = \top$. Pour arriver à ce résultat, la librairie procède à une énumération : elle commence par appliquer toutes les transformations de T une par une ; si elle en trouve une qui corrige l'objet, celle-ci est retournée. Sinon, le processus s'applique de nouveau en énumérant toutes les compositions de deux transformations, puis de trois, etc. Par construction, la transformation qui est retournée sera forcément la plus simple, c'est-à-dire celle impliquant la plus petite combinaison, puisque celles-ci sont énumérées en ordre croissant de longueur.

Par exemple, supposons que O est l'ensemble des nombres entiers, que T est l'ensemble composé des deux fonctions $t_1(x) = x + 1$ et $t_2(x) = x - 1$, et $p(x)$ est la propriété $x = 42$. On voit que $O = 43$ ne satisfait pas la propriété et que l'application de $t1$ à cet objet produit un objet qui la satisfait. Donc, $t1$, qui correspond à l'incrémentation de 1 d'un objet, est la correction à appliquer à o afin de satisfaire la propriété p. La correction trouvée, voire même

l'existence d'une correction, dépend donc de l'ensemble T : si T ne contient que la fonction $t2$, alors aucune composition de transformations ne réussira à satisfaire la propriété.

Il est à noter que Fault-Finder est une librairie générique. Ainsi, bien que l'article la présentant suggère une application possible pour des erreurs dans les pages web, aucune implémentation n'a été réalisée. L'objectif est donc d'utiliser Fault-Finder dans Cornipickle afin de trouver des corrections à appliquer à une page lorsqu'une condition n'est pas respectée.

4.3 INTÉGRATION DE FAULT-FINDER

Pour arriver à intégrer Fault-Finder à un outil, il suffit d'arriver à définir ce que sont O, T et P. Dans le cas de Cornipickle, la liaison semble se faire naturellement : P est la liste de contraintes que retourne l'outil et O est la liste d'objets violant une contrainte donnée que retourne Cornipickle. Il ne restait qu'à générer T. Or, puisque Cornipickle nous fournit, sous forme de fonctions mathématiques, les propriétés qui ne sont pas respectées et la valeur des objets potentiellement responsables de l'erreur à envoyer à ces fonctions afin des les violer, il suffit de trouver la solution à l'inéquation pour trouver une correction.

Toutefois, en s'attardant plus longtemps sur les types d'erreurs visuelles, telles que présentées dans le chapitre 1.2, nous avons pu constater que ce ne sont pas tous les types de bogues pour lesquels la résolution d'une inéquation est triviale. Le premier type qui est sorti du lot est l'erreur d'échappement, soit les erreurs dues à la mauvaise rédaction du HTML. Afin de trouver une correction à ce type d'erreur, il faudrait développer un interpréteur de HTML. Grâce à cet interpréteur, il serait possible de trouver la cause exacte du problème, par exemple une balise ouvrante qui n'a pas de balise fermante correspondante. Une fois la source du problème détectée, il serait possible d'appliquer le correctif correspondant, soit ajouter la bonne balise fermante pour l'exemple précédent. Or, où mettre cette balise fermante ?

```
1.    <body>
2.        <div class="conteneur">
3.            <div class="mesImages visible">
4.                <div class="nombre">1 / 3</div>
5.                <img src="img/img1.jpg">
6.                <div class="texte">Légende de l'image</div>
7.            </div>
8.            <div class="mesImages">
9.                <div class="nombre">2 / 3</div>
10.               <img src="img/img2.jpg">
11.               <div class="texte">Légende de l'image</div>
12.           </div>
13.           <div class="mesImages">
14.               <div class="nombre">3 / 3</div>
15.               <img src="img/img3.jpg">
16.               <div class="texte">Légende de l'image</div>
17.           </div>
18.           <a class="precedent" onclick="changerImage(-1)">&#10094;</a>
19.           <a class="suivant" onclick="changerImage(1)">&#10095;</a>
20.           <div class="points">
21.               <span class="point actif" onclick="imageActuelle(1)"></span>
22.               <span class="point" onclick="imageActuelle(2)"></span>
23.               <span class="point" onclick="imageActuelle(3)"></span>
24.           </div>
25.
26.   </body>
```

FIGURE 4.4 : Exemple de code HTMl dans lequel il y a une erreur d'échappement. ©Xavier Chamberland-Thibeault 2023

Prenons l'exemple de la figure 4.4, qui correspond au code du carrousel de la figure 3.18 dans lequel une erreur a été introduite. On peut y voir une *div* dont l'ouverture est faite à la ligne 2 mais qui n'est jamais fermée. Afin d'avoir un carrousel bien affiché, cette balise devrait se fermer à la ligne 25. Toutefois, pour générer automatiquement une correction à cette erreur, il est beaucoup plus compliqué d'identifier où mettre la balise fermante. Elle pourrait aller directement après la balise ouvrante sur la ligne 2. Ou encore entre les lignes 7 et 8 après le premier bloc « mesImages ». Ou bien elle pourrait aller entre les lignes 17 et 18, à la suite de tous les blocs « mesImages » car le code détecterait que ce sont des groupes d'éléments similaires qui doivent rester ensemble. Dans ce cas, comment justifier qu'il faut aussi que les balises *a* et le groupe d'éléments « points » soit compris dans le *div* « conteneur » ? Vu la complexité d'une telle correction, nous avons décidé de mettre de côté ce type d'erreur.

Les erreurs d'échappement n'étaient pas les seules à être problématiques : nous avons aussi dû mettre de côté la correction des mojibakes et erreurs d'encodage. Afin de détecter ces

erreurs dans une page web, il faut utiliser des expressions régulières afin d'identifier les parties du texte en erreur. Ceci implique que, pour arriver à générer des correctifs adéquats, il faut identifier quelle partie de l'expression régulière n'est pas correcte. Ensuite, si on a réussi à identifier une partie précise de l'expression régulière, il faut arriver à identifier à quel caractère « normal » cette partie de l'expression régulière fait référence. Or, tel que mentionné dans le chapitre 1.2.5, il existe plusieurs encodages différents pour un même caractère, il faudrait donc arriver à déterminer quel est l'encodage attendu pour le rendu de la page, chose qui n'a pas forcément été spécifiée par le développeur. Encore une fois, ce type d'erreur a été mis de côté, car identifier des solutions possibles à de telles erreurs engendre un niveau de complexité bien plus élevé que la simple résolution d'une inéquation.

La correction de tous les autres types d'erreurs visuelles mentionnés au chapitre 1.2 semblait envisageable. En effet, tous ces types d'erreurs sont identifiés via des inéquations exprimées en fonction des propriétés des éléments. Si on se rapporte à l'exemple de la figure 4.1, une inéquation de la forme $x_{left} = y_{left}$ sera énoncée pour chacune des paires d'éléments de la liste à puces. Ainsi, il suffit de résoudre les inéquations, ou systèmes d'inéquations comme dans l'exemple précédent, pour trouver une correction au bogue.

Maintenant que ces bases ont été établies, un premier jet à l'intégration, à Cornipickle, de Fault-Finder a été conçu. La figure 4.5 montre l'interaction entre Cornipickle et Fault-Finder. Suite à son analyse, soit l'étape 1, Cornipickle demandera à Fault-Finder de trouver des correctifs pour une contrainte à la fois. Ainsi, il enverra à la librairie une contrainte non-respectée, soit P, ainsi que la liste des éléments potentiellement en cause dans la violation de cette contrainte, soit O. C'est l'étape 2 de la figure 4.5. Fault-Finder se chargera de générer T, soit l'étape 3. Afin de pouvoir générer efficacement des transformations, il faut travailler avec les inéquations les plus simples possibles. Une inéquation dite simple comprend au plus un seul inconnu et utilise un des opérateurs suivants : plus grand, plus petit, plus grand ou égal,

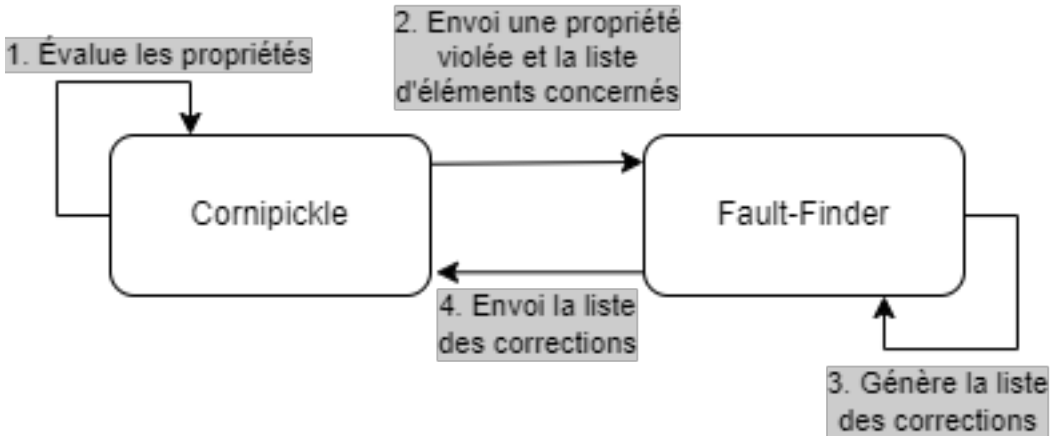


Figure 4.5 : Schéma présentant l'interaction entre Cornipickle et Fault-Finder. ©Xavier Chamberland-Thibeault, 2023

plus petit ou égal et égal. Ainsi, en arrivant à décomposer l'inéquation en plusieurs petites parties, que nous appellerons sous-inéquations, il sera possible de générer des transformations simples qui, par elles-mêmes ou combinées entre elles, permettront de valider l'inéquation de base.

Une fois ce découpage d'inéquation effectué, Fault-Finder peut procéder à la génération des transformations. Puisque toute modification amenée dans une page web peut avoir un effet domino sur le positionnement des autres éléments de la page, chaque transformation vise à faire la modification la plus minime possible. En considérant comme exemple la sous-inéquation $x = y$, Fault-finder va générer l'ensemble des transformations T comprenant $x + 1$, $x - 1$, $y + 1$, $y - 1$, x affecté à y et y affecté à x. Le principe étant qu'une transformation de plus ou moins un pixel, ou une combinaison de telles transformations, sur un élément ou bien l'affectation de la valeur d'un des éléments à l'autre va forcément aboutir à l'alignement des deux éléments. Un tel ensemble de transformations sera généré pour chacune des sous-inéquations. Une fois chaque ensemble généré, ils sont regroupés en un ensemble contenant toutes les transformations simples applicables à l'inéquation globale pour atteindre une possible correction. Grâce à cet

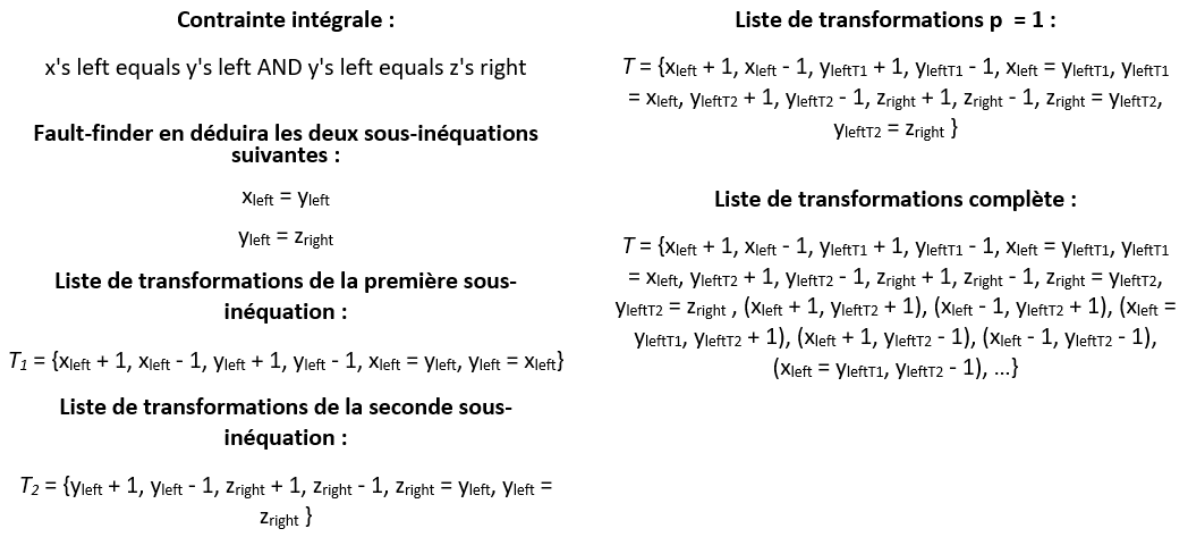


Figure 4.6 : Exemple de génération de transformations. ©Xavier Chamberland-Thibeault, 2023

ensemble de transformations, Fault-Finder va tenter de trouver des corrections possibles. Cette liste de possibles corrections correspond donc à la combinaison de transformations C_n^1 où n est le nombre de transformations dans l'ensemble T. Si aucune de ces transformations permet de valider l'inéquation globale, Fault-Finder va tenter les combinaisons C_n^p, où p varie de 2 à n, en ordre croissant de p. La figure 4.6 montre un exemple de ce processus avec $n = 2$.

Une fois la liste de corrections terminée, Fault-finder la retourne au programme l'ayant appelé, soit Cornipickle dans ce cas-ci tel que montré à l'étape 4. Il est important de noter que Fault-Finder arrête de produire des solutions après 40 secondes d'exécution afin d'éviter de tourner trop longtemps. Si aucune solution n'a été trouvée durant ces 40 secondes, Fault-Finder considère qu'il n'existe pas de solution. Les étapes 2, 3 et 4 vont se répéter pour chacune des propriétés violées. De légères modifications ont été amenées à Cornipickle ainsi qu'à sa sonde afin qu'ils prennent en compte le retour de Fault-Finder. Au lieu de simplement identifier dans la page les éléments fautifs en citant la règle enfreinte, une suggestion de correctif est aussi

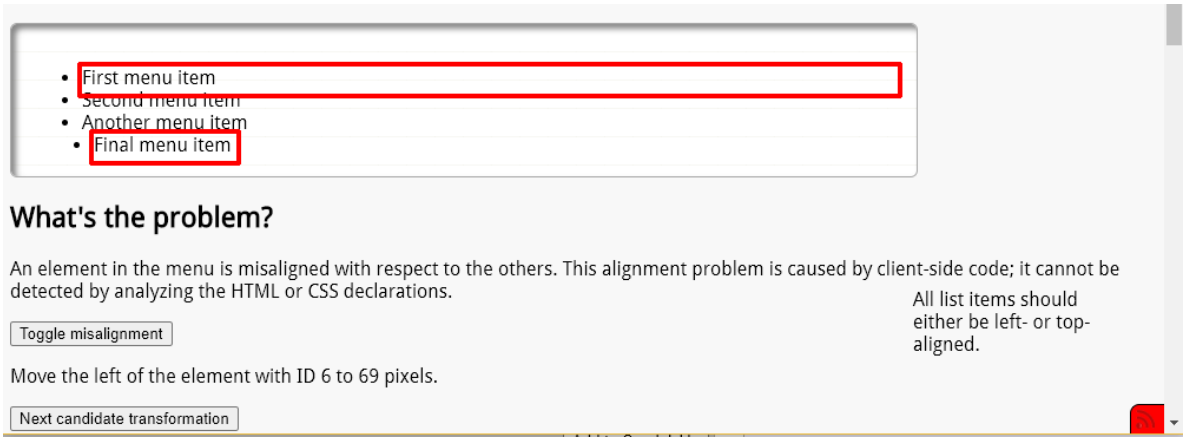


FIGURE 4.7 : Exemple de rendu de Cornipickle et Fault-Finder sur une page présentant une erreur d'alignement. ©Xavier Chamberland-Thibeault, 2023

donnée à l'utilisateur. L'utilisateur peut aussi boucler au travers des transformations suggérées jusqu'à en trouver une qui convient. La figure 4.7 montre un exemple du rendu de Fault-Finder et Cornipickle pour une erreur d'alignement d'un élément dans une liste. On peut y voir une recommandation de correctifs ainsi qu'un bouton pour aller aux corrections suivantes.

4.4 LIMITATION DE FAULT-FINDER

Une fois l'intégration faite, nous avons pu débuter les tests. Notre premier essai fut sur la page d'exemple d'erreur d'alignement de Cornipickle. Tel que le montre la figure 4.8, Fault-Finder avait bien identifié un correctif à la problématique. Lors de l'application de la correction suggérée, nous nous sommes rendu compte que notre implémentation de Fault-Finder ne prenait pas en compte le *padding* ni le *margin*. De plus, afin d'appliquer des valeurs précises à la gauche d'un élément, il doit forcément être en *position :fixed* ou *position :absolute*. Ainsi, pour appliquer le correctif suggéré, il a fallu mettre le dernier élément de la liste en *position :absolute*, lui enlever tout *padding* et *margin* pour ensuite mettre sa

FIGURE 4.8 : Exemple du rendu de la page web de la figure 4.7 suite à l'application du correctif suggéré par Fault-Finder. ©Xavier Chamberland-Thibeault, 2023

propriété *left* à 69. Heureusement, c'était le dernier élément de la liste, sinon mettre sa position en *absolute* ou *fixed* aurait créé une superposition d'éléments de la liste.

Ce premier test, bien que relativement concluant, nous a mené à deux constats. Premièrement, nous allions devoir changer la prise en compte de valeurs de *padding* et *margin* des éléments, sans quoi la position suggérée ne sera jamais la bonne. Deuxièmement, l'application d'un correctif via l'utilisation de *position :absolute* et *position :fixed* est la plus simple étant donné que cela diminue au minimum les interactions fortuites entre plusieurs règles CSS, mais en contrepartie nécessite de gérer les impacts de ce positionnement sur tous les autres éléments. Nous avons fait le même test avec l'exemple d'erreur de chevauchement de Cornipickle et avons obtenu des résultats similaires.

Bien que le premier constat est une correction mineure, le second est nettement plus compliqué. Sans changement à l'implémentation, c'est le développeur qui, après chaque correctif appliqué, devra rouler à nouveau Fault-Finder pour corriger les possibles erreurs engendrées par le correctif. Si on se lance dans un changement du fonctionnement, il faut arriver à identifier et gérer les effets en cascade d'une modification. Ce fut la première limitation importante que nous avons rencontré face à Fault-Finder.

Nombre d'éléments mal alignés	Temps d'exécutions (ms)										Moyenne des temps d'exécutions (ms)
1	48	29	21	22	102	97	91	39	64	8	52,1
2	76712	73524	73188	75207	73072	70939	71738	72908	72805	72839	73293,2

TABLEAU 4.1 : Temps d'exécutions de Fault-Finder sur la page d'exemple d'erreur d'alignement de Cornipickle, selon le nombre de d'éléments mal alignés dans la liste. ©Xavier Chamberland-Thibeault 2023

Avant de se pencher plus sur la problématique de l'application du correctif, nous avons procédé à d'autres tests afin de découvrir s'il y avait d'autres limites à l'approche utilisée. Pour ce faire, nous avons effectué un deuxième test sur la même page web, mais cette fois-ci avec deux éléments mal alignés. Une autre problématique s'est immédiatement révélée à nous : le temps d'exécution. Comme on peut le voir dans le tableau 4.1, le temps d'exécution était plus que raisonnable lors de nos tests ayant un seul élément mal aligné, avec une moyenne de 0,05 secondes. Or, les tests avec deux éléments mal alignés se sont révélés hautement décevants, prenant en moyenne 73 secondes pour s'exécuter : le temps maximal de 40 secondes avait été retiré afin de voir si Fault-Finder arriverait à suggérer des corrections, chose qu'il n'arrivait pas à faire en 40 secondes. Ces temps d'exécution sont majoritairement explicables par la quantité exponentielle de combinaisons à tester. En effet, en ajoutant un nouvel élément mal aligné, le nombre de sous-inéquations a doublé, ce qui a un effet exponentiel sur le nombre de cas à tester. En plus de devoir générer chacune des nouvelles combinaisons, elles doivent toutes être testées. Ceci nous a immédiatement menés à nous questionner sur l'extensibilité de notre méthode. Aucun développeur ne serait prêt à attendre 73 secondes pour se faire suggérer des corrections pour possiblement régler un bogue d'affichage tout en risquant d'en créer un nouveau de par l'effet en cascade du *position :absolute* ou *position :fixed*.

Qui plus est, ce ne sont pas toutes les corrections suggérées qui sont pertinentes. En effet, une suggestion proposée pourrait être de décaler les trois éléments qui ont le même alignement

afin qu'ils aient l'alignement du quatrième, que nous considérons comme l'élément mal placé. Effectivement, cette transformation répond à la propriété que tous les éléments d'une même liste doivent être alignés, mais c'est une correction très contre-intuitive. Ainsi, malgré une possible solution trouvée, il faudrait obtenir l'approbation de l'utilisateur. Si aucune solution n'est approuvée par l'utilisateur, Fault-Finder pourrait continuer sa recherche de combinaisons. Toutefois, cela créerait plusieurs échanges entre le développeur et Fault-Finder, augmentant encore plus le délais nécessaire pour trouver une solution. De plus, ces allers-retours nous éloigneraient encore plus de l'objectif de cette recherche, soit d'arriver à faire une correction automatique des erreurs visuelles.

La solution proposée fait donc face à trois grandes limites, soit la gestion de correctifs en cascade, le temps d'exécution et l'incapacité d'établir un jugement critique sur la correction suggérée. Il serait possible de gérer les effets en cascade et d'intégrer des heuristiques pour proposer des solutions plus adéquates, mais cela viendrait au coût d'inéquations largement plus complexes et donc d'un temps d'exécution démesuré. Nous nous sommes donc tournés vers une nouvelle solution, inspirée de Fault-Finder et de ces transformations, mais qui pourrait gérer plus facilement les inéquations : les solveurs.

CHAPITRE V

CORRECTION DES BOGUES : APPROCHE PAR SOLVEUR

Dans ce dernier chapitre, sera premièrement discutée une nouvelle méthode pour générer des listes de transformations à appliquer à une page web afin de la corriger, méthode qui permet de circonvenir à plusieurs des lacunes rencontrées avec Fault-finder. Une seconde partie présentera pourquoi un solveur numérique est nécessaire à la correction d'erreurs visuelles dans les pages web et comment l'utiliser. Finalement, une méthode pour l'application de correctifs dans une page web sera présentée avec quelques exemples réels.

5.1 ZONE D'INFLUENCE

Après avoir observé les multiples lacunes de la technique présentée au chapitre précédent, il a fallu trouver une méthode plus efficace pour trouver un correctif aux erreurs présentes dans une page web tout en prenant en considération les impacts qu'une telle correction pourrait avoir sur le reste de la page.

Premièrement, afin d'essayer de réduire les délais d'exécution, il a fallu trouver une façon de réduire le nombre d'éléments de la page qui sont analysés. Pour ce faire, lorsqu'un nœud est identifié comme problématique, au lieu de sérialiser toute la page, une « zone d'influence » sera calculée et seuls les nœuds faisant partie de cette zone seront sérialisés. Le principe de la zone d'influence repose sur les effets d'une modification amenée à un élément d'une page web. Lorsqu'un élément est modifié, que ce soit un déplacement ou un changement de taille, ce changement peut décaler les éléments frères, soit tous les éléments qui ont le même parent que celui modifié. Or, ce décalage peut entraîner un changement dans la taille du parent. Ce dernier changement produit un effet cascade : si ce parent est modifié, ses frères et son parent

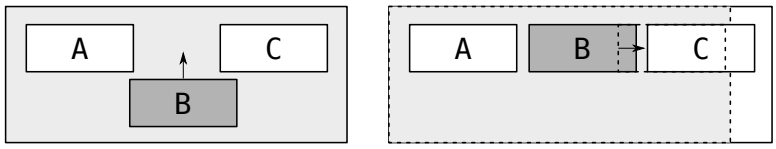


FIGURE 5.1 : Exemple de répercussion pouvant se produire sur les éléments adjacents lors du déplacement d'un élément fautif.[1] Autorisation obtenue par Springer Nature.

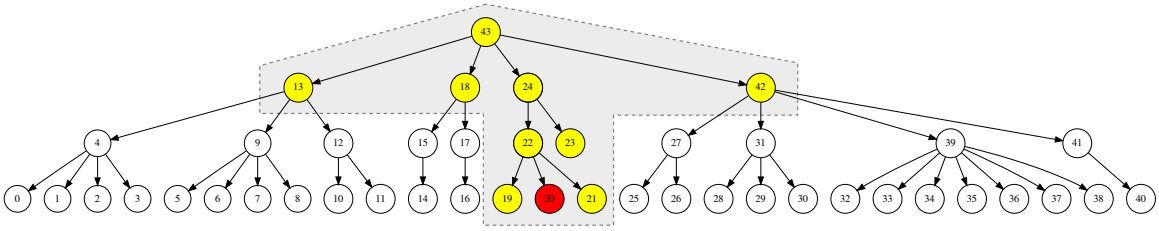


FIGURE 5.2 : Un exemple du concept de zone d'influence.[1] Autorisation obtenue par Springer Nature.

peuvent être affectés, et ainsi de suite. La figure 5.1 montre un exemple des répercussions que peut engendrer le déplacement d'un élément.

Tel que démontré par Jacquet *et al.* [1], la zone d'influence, représentant tous les éléments qui peuvent être affectés par la modification du nœud fautif, contient tous les frères de l'élément fautif. À cette sélection de nœuds, on rajoute récursivement le parent du nœud fautif ainsi que ses frères. Ce processus est répété pour le parent suivant et ses frères, et ce jusqu'à atteindre la racine de la page web. La Figure 5.2 montre un exemple d'une zone d'influence. Dans cet exemple, le nœud 20, qui est en rouge, est considéré comme le nœud fautif. On voit en jaune tous les nœuds compris dans la zone d'influence du nœud 20, soit tous ses frères, ses parents et les frères de ses parents.

Grâce à la zone d'influence, il est maintenant possible de récupérer, dans une page web, uniquement le nœud fautif ainsi que tous les nœuds qui peuvent être impactés par une modification appliquée au nœud problématique. Ceci réduit au minimum le nombre d'éléments

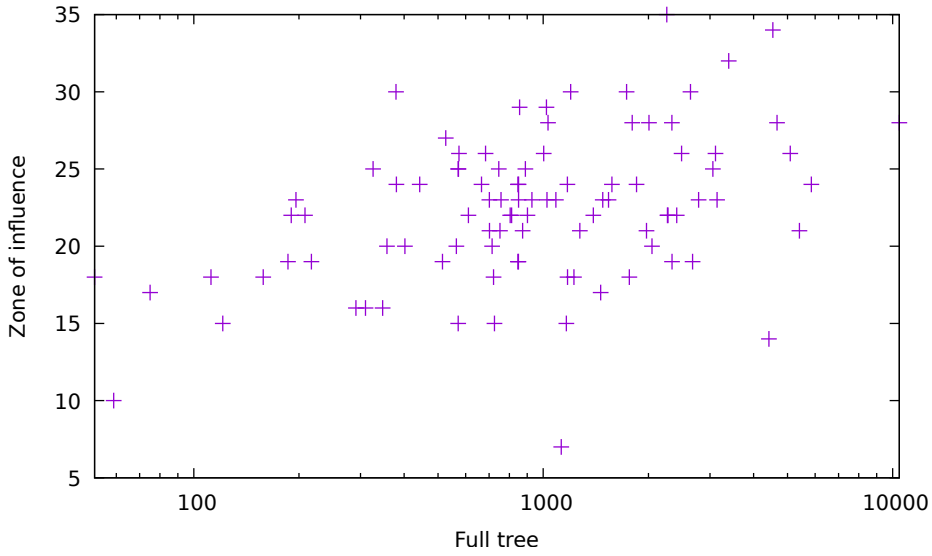


FIGURE 5.3 : Impact de la zone d'influence.[1] Autorisation obtenue par Springer Nature.

à traiter pour trouver une solution. La figure 5.3 montre la différence de taille entre la zone d'influence et la page web : sur l'axe des x on peut voir la taille de l'arbre DOM de la page web et sur l'axe des y la taille de l'arbre DOM de la zone d'influence. Toutefois, ça ne résout pas le problème de trouver une correction dans un temps raisonnable tout en prenant en compte l'effet cascade que la correction peut amener. Bien que Fault-Finder arrive à trouver des solutions à certaines erreurs, la puissance de calcul de notre implémentation était beaucoup trop faible. Pour pallier à cette lacune, nous nous sommes tournés vers les solveurs.

5.2 INTERACTIONS AVEC LE SOLVEUR NUMÉRIQUE

Mais qu'est-ce qu'un solveur ? C'est un logiciel qui, en se basant sur les informations reçues en entrée, va trouver une solution à un problème donné. Ces logiciels, ou librairies, cherchent, dans un délai imparti, une liste de solutions possibles au problème reçu puis identifient la meilleure de ces solutions. Afin d'obtenir les résultats escomptés, soit d'optimiser des inéquations mathématiques représentant des contraintes visuelles et des éléments ne respectant

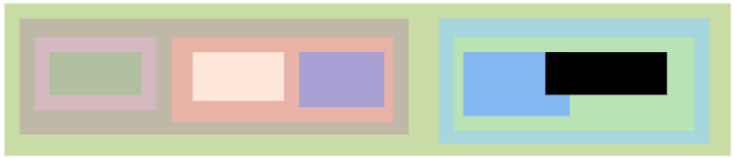


Figure 5.4 : Un exemple d'un site web synthétique où il y a une erreur de chevauchement. ©Xavier Chamberland-Thibeault, 2023

pas ces contraintes, nous utilisons un solveur numérique, plus précisément CPLEX. Comme dans la littérature, nous n'irons pas en détails dans le fonctionnement des solveurs numériques, toutefois il reste pertinent d'expliquer plus en détail les entrées que nous fournissons au solveur et les sorties que nous récupérons.

Un solveur numérique travaille avec des systèmes d'inéquations à résoudre ainsi que les variables présentes dans ces inéquations et une fonction objectif : ce sont les éléments qui doivent lui être fournis en entrée. Dans notre cas, il faut arriver à lui fournir les valeurs des propriétés CSS des éléments enfreignant une contrainte ainsi que la contrainte elle-même. Une fois les inéquations reçues, le solveur peut optimiser le plus possible la fonction objectif tout en se basant sut le système, soit trouver les valeurs qui permettent de valider les inéquations tout en minimisant ou maximisant la fonction objectif, selon les spécifications données par l'usager. Une fois la solution optimale trouvée ou le temps d'exécution épuisé, le solveur retournera les affectations de valeurs à donner aux variables afin de résoudre le système d'inéquations. Il nous faut donc récupérer ces affectations et arriver à les appliquer dans la page web afin de la corriger. La suite du texte explique, en se basant sur l'exemple de la Figure 5.4, comment nous parvenons à transformer le site web en entrée valide pour le solveur ainsi que comment nous traitons la sortie du solveur pour l'appliquer dans le site web. Dans cet exemple, on peut y voir un site web dans lequel un des éléments, le rectangle noir dans ce cas-ci, chevauche un de ses frères.

Afin de permettre au solveur de bien faire son travail, il faut donc faire quelques manipulations avec le site web afin d'obtenir les inéquations et la liste de variables à envoyer au solveur. Tout premièrement, il faut identifier, à l'aide de CorniId, tous les éléments fautifs de la page web. Une fois cette identification faite, la sonde JavaScript va récupérer chacun des éléments identifiés ainsi que la zone d'influence de chacun, créant ainsi un arbre représentant une partie de la page web. Cette partie correspond à tous les éléments qui peuvent potentiellement être modifiés, ou impactés par une modification, lors de la résolution de l'erreur visuelle. La sonde va ensuite sérialiser cet arbre sous forme de JSON et l'envoyer à l'application Java.

La Figure 5.5 montre le JSON représentant les éléments fautifs et la zone d'influence du site web de la figure précédente. L'attribut `level`, tout comme l'indentation, représente le niveau de profondeur de l'élément dans l'arbre. Ainsi, l'élément ayant l'attribut `"level":0` se trouve à être le `body` de la page, celui ayant `"level":1` est un niveau plus profond dans l'arbre, donc un de ses enfants (dans ce cas-ci, le plus grand rectangle vert), et ainsi de suite. Les attributs `x` et `y` représentent respectivement les positions `top` et `left` de l'élément dans la page. Quant aux attributs, `height` et `width`, ils représentent les dimensions des éléments tels qu'affichés dans la page lors de la récupération des données. L'attribut `"children":[...]` est la liste des enfants de l'élément. Finalement, l'attribut `nodename` est le CorniId attribué à l'élément et qui sert ici d'identifiant unique.

L'application va ensuite désérialiser le JSON pour en récupérer l'arbre. Elle traversera ensuite l'arbre pour en produire une copie simplifiée sous forme d'imbrication de « boîtes ». La Figure 5.6 montre un exemple d'arbre de boîtes généré grâce au JSON de la figure précédente. Tel qu'on peut le voir dans cet exemple, chacune des boîtes, ayant une hauteur, une largeur, une position `top`, une position `left` ainsi qu'un identifiant, représente un et un seul élément de la

```
{"level":0,"x":8,"y":8,"width":913,"height":500,"children":[
  {"level":1,"x":4,"y":4,"width":95.59375,"height":20.1875,"children":[
    {"level":2,"x":6,"y":6.140625,"width":51.171875,"height":15.1875,"children":[],"nodeName":3},
    {"level":2,"x":61.171875,"y":6.140625,"width":35.578125,"height":16.421875,"children":[
      {"level":3,"x":63.34375,"y":8.28125,"width":31.578125,"height":12.421875,"children":[
        {"level":4,"x":64.515625,"y":10.421875,"width":13.734375,"height":8.421875,"children":[],"nodeName":"6"},
        {"level":4,"x":75.265625,"y":10.421875,"width":15.828125,"height":5.34375,"children":[],"nodeName":"7"}
      ],"nodeName":"5"}
    ],"nodeName":"4"}
  ],"nodeName":"2"}
],"nodeName":"1"}
```


FIGURE 5.5 : Un exemple du JSON récupéré depuis le site web de la Figure 5.4. ©Xavier Chamberland-Thibeault, 2023

```
id: 1, x: 8.0, y: 8.0, w: 913.0, h: 500.0
 id: 2, x: 4.0, y: 4.0, w: 95.59375, h: 20.1875
  id: 3, x: 6.0, y: 6.140625, w: 51.171875, h: 15.1875
  id: 4, x: 61.171875, y: 6.140625, w: 35.578125, h: 16.421875
   id: 5, x: 63.34375, y: 8.28125, w: 31.578125, h: 12.421875
    id: 6, x: 64.515625, y: 10.421875, w: 13.734375, h: 8.421875
    id: 7, x: 75.265625, y: 10.421875, w: 15.828125, h: 5.34375
```

FIGURE 5.6 : Un exemple d'un arbre de boîtes généré à partir du JSON de la Figure 5.5. ©Xavier Chamberland-Thibeault, 2023

page web. En résulte de cette procédure un arbre de boîtes imbriquées représentant la structure de la page web. Tel qu'on peut le voir en comparant les deux figures, l'arbre de boîtes est une représentation plus simple de la page web et présentant uniquement des paires variable-valeur pour chaque élément. Cette version simplifiée de l'arbre permet de faciliter la conversion de la page et des contraintes en ligne de code Optimization Programming Language (OPL), soit le langage reconnu par le solveur CPLEX. Ce sont ces lignes de codes qui servent d'entrées au solveur puisqu'elles contiennent le système d'inéquations et les variables à utiliser dans ce système.

L'application Java va donc envoyer l'arbre de boîtes à l'outil PageGen [13]. Cet outil est capable, en se basant sur l'arbre de boîtes, de générer les systèmes d'inéquations correspondant à certaines contraintes : même alignement, pas de chevauchement et contenu l'un dans l'autre. En effet, ces trois types de contraintes sont relativement simples à convertir en système

13. https://github.com/sylvainhalle/pagegen

d'inéquations : elles correspondent toutes à la comparaison d'un ou plusieurs des attributs des éléments. Par exemple, pour une contrainte de même alignement, il faut simplement comparer les `x` ou `y` des éléments, donc écrire une inéquation par comparaison nécessaire. De plus, PageGen est aussi en mesure de convertir et agréger les boites de l'arbre en tableaux de valeurs qui sont les variables des inéquations. Un tel exemple de conversion est présenté dans la Figure 5.7. Dans la Figure 5.7a, on peut voir que des tableaux ont été créés. Chacun de ces tableaux contient les informations des boîtes de l'arbre de la figure précédente. Par exemple, le tableau `ini_top` contient toutes les valeurs de l'attribut `x` des boîtes de la Figure 5.6. On constate aussi que l'identifiant de la boîte correspond à sa position dans le tableau afin de pouvoir retrouver aisément quelle valeur correspond à quelle boîte. Dans la Figure 5.7b, on peut voir les contraintes qui ont été générées pour représenter la problématique de chevauchement. La première ligne indique au solveur qu'il doit minimiser la variation des variables lors de sa recherche de correction. Cette ligne est ajoutée afin de forcer le solveur à fournir une solution qui modifiera visuellement le moins possible la page tout en corrigeant le bogue visuel. Chacune des lignes suivantes correspond à une partie de la contrainte de non-chevauchement.

Une fois l'arbre de boîtes converti en code OPL, il est maintenant possible de l'envoyer à CPLEX afin qu'il puisse optimiser le système d'inéquations et ainsi trouver une correction à l'erreur d'affichage : soit le déplacement vertical ou horizontal d'un ou plusieurs éléments de la page ou le redimensionnement vertical ou horizontal d'un ou plusieurs éléments de la page. Une fois une correction trouvée, CPLEX va retourner la position et la dimension à donner à chacun des éléments afin de corriger la page. Un exemple d'une telle sortie est présenté dans la Figure 5.8. On y retrouve quatre tableaux contenant respectivement la position `top` de l'élément, sa position `left` ainsi que sa hauteur et sa largeur. Comme pour les tableaux du code OPL, la position dans le tableau correspond à l'élément auquel il faut appliquer ses valeurs.

```
int nb_rectangles=7;
{int} rectangles_id={1, 2, 3, 4, 5, 6, 7};
float ini_Height[rectangles_id]=[500.0, 20.1875, 15.1875, 16.421875, 12.421875, 8.421875, 5.34375];
float ini_Width[rectangles_id]=[913.0, 95.59375, 51.171875, 35.578125, 31.578125, 13.734375, 15.828125];
float ini_left[rectangles_id]=[8.0, 4.0, 6.0, 61.171875, 63.34375, 64.515625, 75.265625];
float ini_top[rectangles_id]=[8.0, 4.0, 6.140625, 6.140625, 8.28125, 10.421875, 10.421875];
dvar float Height[rectangles_id];
dvar float Width[rectangles_id];
dvar float left[rectangles_id];
dvar float top[rectangles_id];
```

(a) Arbre de boîtes converti en variables OPL

```
minimize sum(i in rectangles_id)(abs(top[i]-ini_top[i])+abs(left[i]-ini_left[i])+Height[i]-ini_Height[i]+Width[i]-ini_Width[i]);
subject to {
left[1]==ini_left[1];
top[1]==ini_top[1];
top[5]<=top[7];top[5]+Height[5]>= top[7]+Height[7];left[5]<=left[7];left[5]+Width[5]>= left[7]+Width[7];
top[1]<=top[2];top[1]+Height[1]>= top[2]+Height[2];left[1]<=left[2];left[1]+Width[1]>= left[2]+Width[2];
top[2]+Height[2]<= top[4] || top[4]+Height[4]<= top[2] || left[2]+Width[2]<= left[4] || left[4]+Width[4]<= left[2];
top[2]<=top[3];top[2]+Height[2]>= top[3]+Height[3];left[2]<=left[3];left[2]+Width[2]>= left[3]+Width[3];
top[3]<=top[2];top[3]+Height[3]>= top[2]+Height[2];left[3]<=left[2];left[3]+Width[3]>= left[2]+Width[2];
top[3]<=top[4];top[3]+Height[3]>= top[4]+Height[4];left[3]<=left[4];left[3]+Width[3]>= left[4]+Width[4];
top[4]<=top[5];top[4]+Height[4]>= top[5]+Height[5];left[4]<=left[5];left[4]+Width[4]>= left[5]+Width[5];
top[5]<=top[6];top[5]+Height[5]>= top[6]+Height[6];left[5]<=left[6];left[5]+Width[5]>= left[6]+Width[6];
top[6]+Height[6]<= top[7] || top[7]+Height[7]<= top[6] || left[6]+Width[6]<= left[7] || left[7]+Width[7]<= left[6];
forall(k in rectangles_id)
Width[k]>=ini_Width[k];
forall(l in rectangles_id)
Height[l]>=ini_Height[l];
```

(b) Contraintes OPL

FIGURE 5.7 : Exemple du code OPL généré afin de corriger des erreurs de chevauchement présentes dans l'arbre de boîtes de la Figure 5.6.©Xavier Chamberland-Thibeault, 2023

```
top = [8.0 4.0 6.140625 6.140625 8.28125 10.421875 10.421875];
left = [8.0 4.0 6.0 61.171875 63.34375 64.515625 78.425484];
Height = [500.0 20.1875 15.1875 16.421875 12.421875 8.421875 5.34375];
Width = [913.0 95.59375 51.171875 35.578125 31.578125 13.734375 15.828125];
```

FIGURE 5.8 : Un exemple de correction proposée par CPLEX afin de régler l'erreur présente dans la page web de la Figure 5.4. ©Xavier Chamberland-Thibeault, 2023

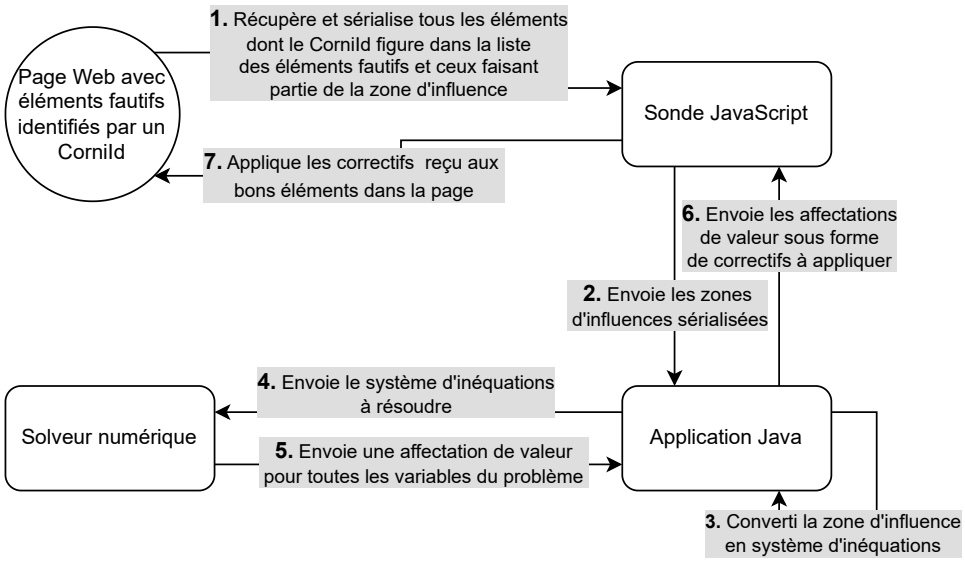


FIGURE 5.9 : Un exemple d'interaction entre la page web, la sonde, l'application Java et le solveur numérique. ©Xavier Chamberland-Thibeault, 2023

L'application Java va récupérer cette liste de valeurs et, pour chacune des boîtes, va générer du code JavaScript permettant d'appliquer ces valeurs aux bons éléments. L'application fait ensuite exécuter le code JavaScript par la sonde présente dans la page afin d'appliquer les correctifs. La Figure 5.9 montre les interactions mentionnées ci-dessus entre la page web, la sonde, l'application Java et le solveur.

5.3 APPLICATION DE CORRECTIFS

Appliquer les correctifs reçus du solveur n'est pas aussi simple qu'on peut le croire. En effet, lorsque le navigateur donne, par exemple, la largeur d'un élément dans la page, ce n'est

FIGURE 5.10 : Un exemple d'application de la correction proposé par CPLEX dans la Figure 5.8 sans traitement préalable des valeurs. ©Xavier Chamberland-Thibeault, 2023

pas sa valeur réelle : la largeur d'un élément comprend à la fois sa réelle largeur, mais aussi le `padding` et les bordures qui lui sont attribuées. Ce qui implique que, si nous appliquons les valeurs fournies par le solveur telles quelles, la correction ne sera pas adéquate. Un exemple d'une telle application est présenté dans la Figure 5.10. Cet exemple montre clairement que l'application des valeurs proposées par CPLEX, sans aucun traitement préalable afin d'obtenir les valeurs réelles, ne corrige pas du tout la page web, voir même la brise encore plus. Ainsi, afin d'arriver à appliquer les correctifs du solveur adéquatement, le script JavaScript doit faire quelques opérations afin de trouver et d'appliquer les bonnes mesures ou positions.

Afin d'appliquer aux éléments pointés la bonne hauteur et largeur, il faut récupérer dans la page la quantité de pixels prévus pour le `padding`, soit de gauche et de droite si on veut modifier la largeur, soit du haut et du bas si on tente de modifier la hauteur de l'élément, ainsi que la taille, toujours en pixels, des bordures de l'élément. On soustrait ensuite ces deux valeurs à la largeur ou à la hauteur que le solveur recommande. Cette nouvelle valeur est ensuite appliquée à l'élément, résultant en un élément ayant exactement la hauteur et la largeur de la correction, bordure et `padding` inclus.

La modification de la position d'un élément est un peu plus compliquée. Afin d'obtenir une position juste, il faut d'abord trouver la position du coin en haut à gauche du premier parent ayant comme positionnement la valeur `absolute`, ou sinon le `BODY` de la page s'il n'y

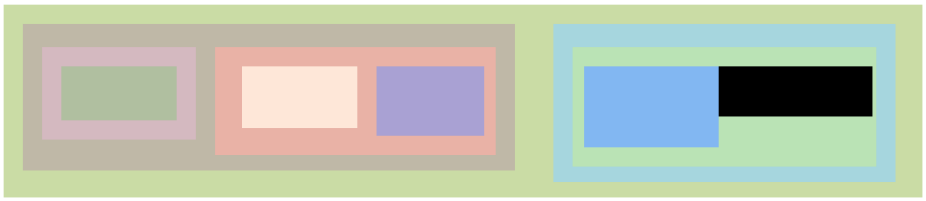


FIGURE 5.11 : Un exemple du site web synthétique de la Figure 5.4 où l'erreur de chevauchement a été corrigée. ©Xavier Chamberland-Thibeault, 2023

a pas de parent répondant à ce critère. Il faut ensuite récupérer la taille des marges de gauche et du haut qui sont attribuées à l'élément à repositionner. Une fois la position du premier parent `absolute` et la taille des marges récupérées, il est temps d'appliquer le correctif de positionnement à l'élément. Pour ce faire, on change son type de positionnement afin qu'il devienne lui aussi `absolute`. Ensuite, on lui attribue la valeur suggérée par le solveur, en y soustrayant la position du parent et la taille des marges. La Figure 5.11 montre à quoi ressemble la page web après l'application des correctifs de CPLEX tout en prenant soin d'appliquer les bonnes valeurs à chacun des éléments afin de ne pas injecter de nouvelle erreur.

5.4 RÉSULTATS EXPÉRIMENTAUX

Le processus complet a été appliqué sur des sites synthétiques, mais aussi sur deux sites réels. Comme on le verra dans la suite, les résultats des expériences sur les sites synthétiques, où une correction était toujours apportée et correcte, montrent de bonnes performances d'exécution : pour des sites variant de quelques dizaines d'éléments à presque dix mille éléments, le temps d'exécution était généralement bien en dessous d'une seconde. Seront ensuite présentés les résultats obtenus lors de la correction d'erreurs sur deux sites réels.

5.4.1 PAGES SYNTHÉTIQUES

En plus d'être en mesure de faire la transition entre les arbres de boîtes et du code OPL, l'outil PageGen est aussi capable de générer aléatoirement des pages web constituées uniquement d'éléments `div` aléatoirement imbriqués. Pour ce faire, PageGen utilise une fonction récursive pour peupler la page. Pour un élément n, un nombre aléatoire d'enfants e sera choisi. Pour chacun de ses enfants, une profondeur p lui sera attribuée aléatoirement : si $p = 0$, alors un élément m sera généré, avec des dimensions aléatoires, et ajouté comme enfant à n, sinon m est peuplé récursivement avant d'être ajouté à n. Une fois tous les enfants ajoutés à n, ils sont disposés horizontalement ou verticalement et séparés par des marges identiques. Une fois la disposition faite, les dimensions de n sont ajustées afin d'être en mesure de contenir tous ses enfants. Le type d'élément, tout comme son contenu, n'a pas d'impact ici puisque notre approche vise à corriger des erreurs de position et dimension uniquement.

PageGen ne se limite pas simplement à la génération de pages web synthétiques, il est aussi capable d'insérer des erreurs aléatoirement au sein des pages qu'il crée. Une fois que les éléments disposés dans leur parent, un lancer de pièce est effectué pour chacun des éléments afin de déterminer s'il doit rester placé comme il est ou bien s'il doit être affecté par une des erreurs suivantes : être mal aligné par rapport à ses frères, chevaucher un de ses frères ou être agrandi pour s'étendre à l'extérieur de son parent. Ceci permet donc d'avoir des pages avec des dispositions aléatoires et de multiples erreurs au travers de chacune des pages.

Cet outil a été utilisé pour générer 100 pages aléatoires dont la taille variait entre 2 et 10450 éléments. Pour chacune de ces pages, notre outil a été utilisé pour trouver et appliquer les correctifs adéquats. Le but de l'outil étant d'être utilisé pour corriger des pages web à la volée, nous avons imposé un temps d'exécution maximal au solveur de deux secondes. La Figure 5.12 montre les temps d'exécution obtenus lors de la correction de chacune des pages

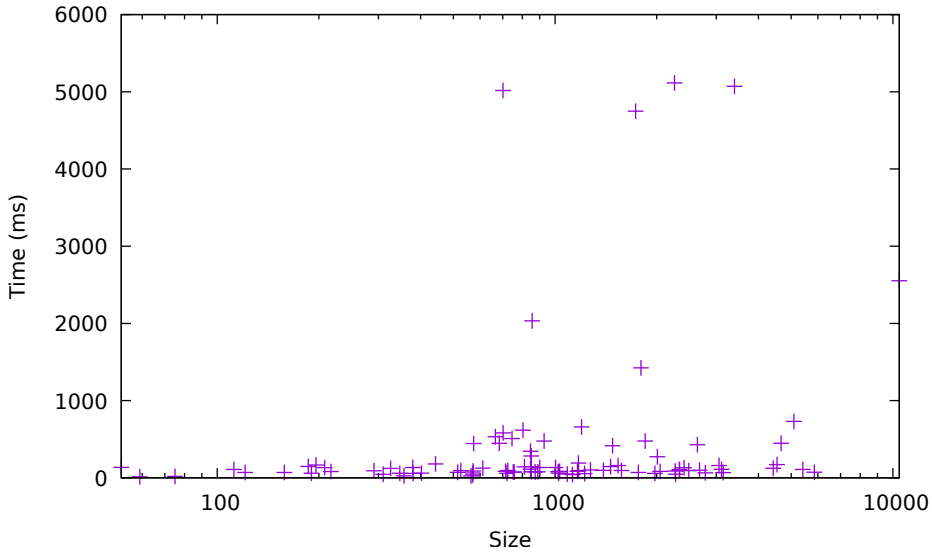
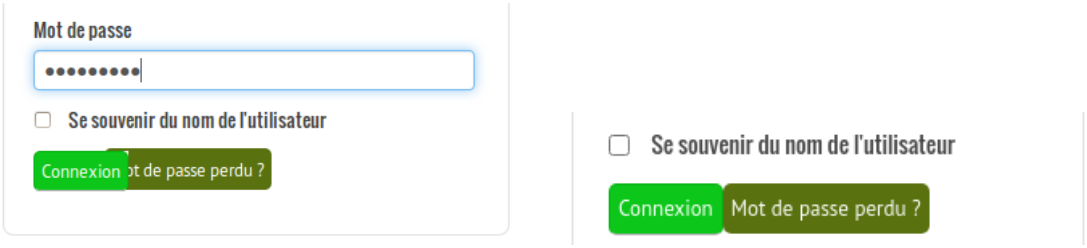


FIGURE 5.12 : Résultats expérimentaux des tests de performance.[1] Autorisation obtenue par Springer Nature.

synthétiques générées. On y constate que toutes les pages sauf cinq ont pris moins de deux secondes à être corrigées et pour la grande majorité moins d'une seconde. Il est possible de comparer ces temps d'exécution à ceux de l'outil X-Fix, qui présente un temps de résolution médian de 841 secondes pour un échantillon de sites ayant un arbre DOM d'une taille moyenne de 425 nœuds.

5.4.2 SITES WEB RÉELS

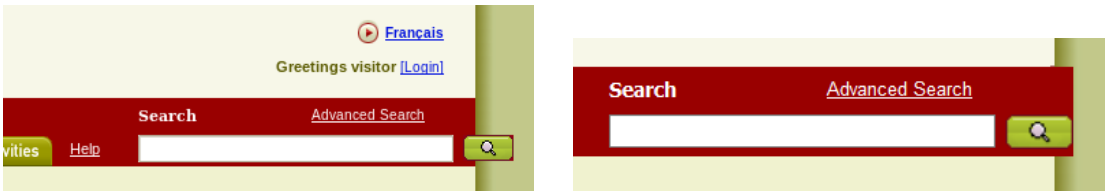
À la suite de ces tests, nous avons tenté de corriger deux sites web réels afin de déterminer s'il est vraiment possible, avec cette technique, de corriger de vraies erreurs visuelles. La Figure 5.13a montre un premier exemple où le bouton pour le mot de passe, lorsque mis en français, chevauche le bouton de connexion. La Figure 5.13b montre la correction appliquée par l'application : ici le bouton a simplement été décalé vers la droite. La Figure 5.14b montre un exemple où le parent a été élargi juste assez pour que le bouton de recherche, qui dépassait



(a) Moodle avec erreur visuelle.

(b) Moodle corrigé.

Figure 5.13 : Exemple de correction d'une erreur de chevauchement sur le site Moodle.[1] Autorisation obtenue par Springer Nature.



(a) AgentSolo avec une erreur de sortie du parent.

(b) AgentSolo Corrigé

Figure 5.14 : Exemple d'une correction d'un élément sortant de son parent sur le site AgentSolo. [1] Autorisation obtenue par Springer Nature.

tel que vu dans la Figure 5.14a, soit contenu à l'intérieur. Dans les deux cas, on peut voir que les sites ont été corrigés avec des modifications minimes, mais appropriées.

CONCLUSION

Dans ce mémoire, qui s'inscrit dans la suite des travaux réalisés sur Cornipickle, nous cherchions une manière de transposer les contraintes visuelles enfreintes détectées par Cornipickle en correctifs à appliquer dans une page. Pour ce faire, nous avons considéré y intégrer Fault-Finder. Bien que l'outil, combiné à la librairie, arrive à détecter les bogues visuels et parvient à produire certains correctifs, il n'arrive à générer des solutions que pour quelques types de bogues visuels en un temps d'exécution trop important.

Nous nous sommes donc tournés vers une autre solution, soit l'intégration d'un solveur numérique pour générer les correctifs ainsi que la mise en place d'une zone d'influence pour minimiser le nombre d'éléments à considérer lors de la génération des corrections. Une fois le solveur opérationnel, la conversion de la page web en équation linéaire utilisable par le solveur et la récupération des éléments uniquement présents dans la zone d'influence de l'élément problématique, nous avons pu procéder à l'application des correctifs.

Pour ce faire, l'outil Java récupère les positions et tailles de tous les éléments tels que trouvées par le solveur. Ensuite, tous les éléments compris dans la solution du solveur se verront attribuer la position et taille indiquées. Pour ce faire, l'application va générer un script JavaScript par élément pour lui appliquer la position trouvée, en prenant en compte la position des parents pour arriver à mettre la position exacte, la taille trouvée, en considérant autant le *margin* que le *padding* de celui-ci, ainsi que le type de positionnement *absolute*. Chacun des scripts sera exécuté afin de corriger la page.

Bien que l'approche présentée soit fonctionnelle, celle-ci fait face à quatre limites. Premièrement, la zone d'influence n'est valide que sous de fortes hypothèses quant à la structure de la page. En effet, lors de l'utilisation de la zone d'influence, il est présumé que, si un élément x est le parent d'un élément y dans l'arbre DOM, alors x contient forcément y

visuellement. Il est aussi présumé que deux éléments frères dans l'arbre DOM sont disjoints visuellement. Or, il est possible qu'une page web ait un visuel qui, via l'utilisation de CSS pour modifier et repositionner des éléments, ne correspond pas à la structure de l'arbre DOM. Dans un tel cas, la zone d'influence ne serait pas fonctionnelle.

Deuxièmement, l'application du correctif partage une hypothèse avec la zone d'influence : un élément parent dans l'arbre DOM contient visuellement l'élément enfant. Si cette hypothèse s'avère incorrecte, l'application du correctif ne fonctionnera pas. En effet, pour appliquer la correction, on prend en compte la position du parent ainsi que son *padding*. Si l'élément pour lequel le correctif appliqué n'est pas contenu dans son parent visuellement, alors le positionnement sera forcément faussé par la position de celui-ci.

Troisièmement, les corrections que peut faire l'outil ne s'appliquent qu'à la géométrie de la page. En effet, l'outil est capable de gérer tout ce qui a trait au positionnement d'un élément dans la page ou sa taille, mais pas les autres types de bogues, comme le contenu. En effet, il faudra être en mesure de convertir la détection d'erreur de contenu, par exemple via l'utilisation d'expressions régulières, en équation linéaire pour que le solveur puisse le gérer. Il serait possible de gérer les erreurs manquantes via l'application Java directement, mais pour ce faire il faudrait y intégrer des algorithmes de correction de bogues comme le font les outils XFix, IFix et MFix. Toutefois, ce type de logique n'est pas présent dans l'outil actuellement et celui-ci, comparativement aux trois solutions proposées par Mahajan *et al,* n'utilise pas de page témoin permettant d'identifier la correction à appliquer.

La quatrième et dernière limite applicable à l'approche est le type de correction appliqué. En effet, l'approche telle qu'implémentée présentement, ne permet pas de faire des corrections durables. En effet, les corrections sont directement appliquées dans le navigateur, corrigeant ainsi le visuel présent au moment de l'exécution du script. Toutefois, si la page est rafraîchie

ou chargée de nouveau, tous les correctifs appliqués ne seront plus présents. Il faudra donc refaire le processus complet d'analyse du visuel et de génération de correction pour avoir, de nouveau, une page fonctionnelle. Il faudrait pouvoir modifier directement le fichier CSS afin que les correctifs soient permanents. Or, les corrections telles que nous les appliquons ne sont pas dans la prolongation du code CSS présent : nous appliquons une position *absolute* à tous les éléments modifiés ainsi qu'une position au pixel près. Pour arriver à faire des correctifs s'intégrant proprement au code CSS présent, il faudrait être en mesure de l'analyser pour détecter quelles parties causent l'erreur visuelle détectée. L'approche présentée ne va présentement pas dans ce sens.

Puisque l'approche introduite dans ce mémoire présente des résultats prometteurs, autant au niveau de l'application des corrections que la génération des corrections, des travaux futurs pourraient être réalisés afin de faire face à ces limites. Notamment, il faudrait être en mesure d'avoir une zone d'influence et une application des correctifs qui ne dépendent pas d'hypothèses. Il faudrait donc être en mesure soit de s'assurer que ces hypothèses sont tout le temps valides ou bien être en mesure de prendre en compte que le visuel ne correspond pas à la structure de la page. Ensuite, des recherches plus approfondies sur les erreurs de contenus, comme les erreurs d'encodage ou d'échappement, seraient nécessaires afin de trouver une façon de les corriger via la méthode utilisée par l'application. En effet, bien que l'utilisation d'un langage déclaratif permette de détecter ces erreurs, la transposition d'une telle déclaration en problème d'équation linéaire s'apparente plus à un problème NP-complet. Finalement, il serait intéressant de trouver une méthode d'application des corrections, soit une façon de positionner un élément à un endroit précis dans la page tout en lui donnant une taille précise, sans avoir recours à des méthodes statiques telles que le positionnement absolu et les tailles en pixel. En effet, bien que fonctionnelle, cette méthode nuit à l'affichage réactif des pages web.

BIBLIOGRAPHIE

[1] S. Jacquet, X. Chamberland-Thibeault, et S. Hallé, "Automated repair of layout bugs in web pages with linear programming," dans *Web Engineering - 21st International Conference, ICWE 2021, Biarritz, France, May 18-21, 2021, Proceedings*, ser. Lecture Notes in Computer Science, M. Brambilla, R. Chbeir, F. Frasincar, et I. Manolescu, éds., vol. 12706. Springer, 2021, pp. 423–439. [En ligne]. Repéré à : https://doi.org/10.1007/978-3-030-74296-6_32

[2] E. Delattre, "100 statistiques et chiffres clés sur les sites internet en 2021," février 2021. [En ligne]. Repéré à : https://blog-fr.orson.io/web-marketing/100-statistiques-sites-internet-2018

[3] P. Panchekha et E. Torlak, "Automated reasoning for web page layout," dans *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2016, part of SPLASH 2016, Amsterdam, The Netherlands, October 30 - November 4, 2016*, E. Visser et Y. Smaragdakis, éds. ACM, 2016, pp. 181–194. [En ligne]. Repéré à : https://doi.org/10.1145/2983990.2984010

[4] H. Liang, K. Kuo, P. Lee, Y. Chan, Y. Lin, et M. Y. Chen, "Seess : seeing what i broke - visualizing change impact of cascading style sheets (css)," dans *The 26th Annual ACM Symposium on User Interface Software and Technology, UIST'13, St. Andrews, United Kingdom, October 8-11, 2013*, S. Izadi, A. J. Quigley, I. Poupyrev, et T. Igarashi, éds. ACM, 2013, pp. 353–356. [En ligne]. Repéré à : https://doi.org/10.1145/2501988.2502006

[5] T. A. Walsh, G. M. Kapfhammer, et P. McMinn, "Automated layout failure detection for responsive web pages without an explicit oracle," dans *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis, Santa Barbara, CA, USA, July 10 - 14, 2017*, T. Bultan et K. Sen, éds. ACM, 2017, pp. 192–202. [En ligne]. Repéré à : https://doi.org/10.1145/3092703.3092712

[6] S. R. Choudhary, H. Versee, et A. Orso, "WEBDIFF : automated identification of cross-browser issues in web applications," dans *26th IEEE International Conference on Software Maintenance (ICSM 2010), September 12-18, 2010, Timisoara, Romania*, R. Marinescu, M. Lanza, et A. Marcus, éds. IEEE Computer Society, 2010, pp. 1–10. [En ligne]. Repéré à : https://doi.org/10.1109/ICSM.2010.5609723

[7] O. Beroual, "Détection et correction automatique des bugs d'interface dans les applications

web," Thèse de doctorat, Université du Québec à Chicoutimi, 2018. [En ligne]. Repéré à : https://constellation.uqac.ca/4706/

[8] F. Guérin, "Testing Web Applications Through Layout Constraints : Tools and Applications," Mémoire de maîtrise, Université du Québec à Chicoutimi, 2017. [En ligne]. Repéré à : https://constellation.uqac.ca/4472/

[9] S. Mahajan, A. Alameer, P. McMinn, et W. G. J. Halfond, "Xfix : an automated tool for the repair of layout cross browser issues," dans *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis, Santa Barbara, CA, USA, July 10 - 14, 2017*, T. Bultan et K. Sen, éds. ACM, 2017, pp. 368–371. [En ligne]. Repéré à : https://doi.org/10.1145/3092703.3098223

[10] S. Hallé et O. Beroual, "Fault localization in web applications via model finding," dans *Proceedings First Workshop on Causal Reasoning for Embedded and safety-critical Systems Technologies, CREST@ETAPS 2016, Eindhoven, The Netherlands, 8th April 2016*, ser. EPTCS, G. Gößler et O. Sokolsky, éds., vol. 224, 2016, pp. 55–67. [En ligne]. Repéré à : https://doi.org/10.4204/EPTCS.224.6

[11] "Definition of MARKUP," 2023. [En ligne]. Repéré à : https://www.merriam-webster.com/dictionary/markup

[12] J. Hoffmann, "A Look Back at the History of CSS," octobre 2017. [En ligne]. Repéré à : https://css-tricks.com/look-back-history-css/

[13] "Css selector reference," 2023. [En ligne]. Repéré à : https://www.w3schools.com/cssref/css_selectors.php

[14] S. R. Choudhary, M. R. Prasad, et A. Orso, "X-PERT : accurate identification of cross-browser issues in web applications," dans *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*, D. Notkin, B. H. C. Cheng, et K. Pohl, éds. IEEE Computer Society, 2013, pp. 702–711. [En ligne]. Repéré à : https://doi.org/10.1109/ICSE.2013.6606616

[15] A. Wirfs-Brock et B. Eich, "Javascript : the first 20 years," *Proc. ACM Program. Lang.*, vol. 4, nº HOPL, pp. 77 :1–77 :189, 2020. [En ligne]. Repéré à : https://doi.org/10.1145/3386327

[16] Huspi, “What Javascript Framework Is the Best For My Project in 2021 ? - HUSPI,” 2019. [En ligne]. Repéré à : https://huspi.com/blog-open/what-javascript-framework-to-choose-in-2020-a-comparison/

[17] V. Lelli, A. Blouin, et B. Baudry, “Classifying and qualifying GUI defects,” *CoRR*, vol. abs/1703.09567, 2017. [En ligne]. Repéré à : http://arxiv.org/abs/1703.09567

[18] N. Li, Z. Li, et X. Sun, “Classification of software defect detected by black-box testing : An empirical study,” *Software Engineering, World Congress on*, vol. 2, pp. 234–240, 12 2010.

[19] A. K. Maji, K. Hao, S. Sultana, et S. Bagchi, “Characterizing failures in mobile oses : A case study with android and symbian,” dans *IEEE 21st International Symposium on Software Reliability Engineering, ISSRE 2010, San Jose, CA, USA, 1-4 November 2010*. IEEE Computer Society, 2010, pp. 249–258. [En ligne]. Repéré à : https://doi.org/10.1109/ISSRE.2010.45

[20] N. S. M. Yusop, J. Grundy, J. Schneider, et R. Vasa, “A revised open source usability defect classification taxonomy,” *Inf. Softw. Technol.*, vol. 128, p. 106396, 2020. [En ligne]. Repéré à : https://doi.org/10.1016/j.infsof.2020.106396

[21] D. Mauser, A. Klaus, R. Zhang, et L. Duan, “Gui failure analysis and classification for the development of in-vehicle infotainment,” *VALID 2012 - 4th International Conference on Advances in System Testing and Validation Lifecycle*, pp. 79–84, 01 2012.

[22] S. Hallé, N. Bergeron, F. Guerin, G. L. Breton, et O. Beroual, “Declarative layout constraints for testing web applications,” *J. Log. Algebraic Methods Program.*, vol. 85, n° 5, pp. 737–758, 2016. [En ligne]. Repéré à : https://doi.org/10.1016/j.jlamp.2016.04.001

[23] A. Mesbah, A. van Deursen, et S. Lenselink, “Crawling ajax-based web applications through dynamic analysis of user interface state changes,” *ACM Trans. Web*, vol. 6, n° 1, pp. 3 :1–3 :30, 2012. [En ligne]. Repéré à : https://doi.org/10.1145/2109205.2109208

[24] S. Artzi, A. Kiezun, J. Dolby, F. Tip, D. Dig, A. M. Paradkar, et M. D. Ernst, “Finding bugs in dynamic web applications,” dans *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2008, Seattle, WA, USA, July 20-24, 2008*, B. G. Ryder et A. Zeller, éds. ACM, 2008, pp. 261–272. [En ligne]. Repéré à : https://doi.org/10.1145/1390630.1390662

[25] ——, “Finding bugs in web applications using dynamic test generation and explicit-state model checking,” *IEEE Trans. Software Eng.*, vol. 36, n° 4, pp. 474–494, 2010. [En ligne]. Repéré à : https://doi.org/10.1109/TSE.2010.31

[26] S. R. Choudhary, M. R. Prasad, et A. Orso, “Crosscheck : Combining crawling and differencing to better detect cross-browser incompatibilities in web applications,” dans *Fifth IEEE International Conference on Software Testing, Verification and Validation, ICST 2012, Montreal, QC, Canada, April 17-21, 2012*, G. Antoniol, A. Bertolino, et Y. Labiche, éds. IEEE Computer Society, 2012, pp. 171–180. [En ligne]. Repéré à : https://doi.org/10.1109/ICST.2012.97

[27] S. Hallé, N. Bergeron, F. Guerin, et G. L. Breton, “Testing web applications through layout constraints,” dans *8th IEEE International Conference on Software Testing, Verification and Validation, ICST 2015, Graz, Austria, April 13-17, 2015*. IEEE Computer Society, 2015, pp. 1–8. [En ligne]. Repéré à : https://doi.org/10.1109/ICST.2015.7102635

[28] O. Beroual, F. Guérin, et S. Hallé, “Detecting responsive web design bugs with declarative specifications,” dans *Web Engineering - 20th International Conference, ICWE 2020, Helsinki, Finland, June 9-12, 2020, Proceedings*, ser. Lecture Notes in Computer Science, M. Bieliková, T. Mikkonen, et C. Pautasso, éds., vol. 12128. Springer, 2020, pp. 3–18. [En ligne]. Repéré à : https://doi.org/10.1007/978-3-030-50578-3_1

[29] S. Mahajan et W. G. J. Halfond, “Websee : A tool for debugging HTML presentation failures,” dans *8th IEEE International Conference on Software Testing, Verification and Validation, ICST 2015, Graz, Austria, April 13-17, 2015*. IEEE Computer Society, 2015, pp. 1–8. [En ligne]. Repéré à : https://doi.org/10.1109/ICST.2015.7102638

[30] S. Mahajan, B. Li, P. Behnamghader, et W. G. J. Halfond, “Using visual symptoms for debugging presentation failures in web applications,” dans *2016 IEEE International Conference on Software Testing, Verification and Validation, ICST 2016, Chicago, IL, USA, April 11-15, 2016*. IEEE Computer Society, 2016, pp. 191–201. [En ligne]. Repéré à : https://doi.org/10.1109/ICST.2016.35

[31] T. A. Walsh, P. McMinn, et G. M. Kapfhammer, “Automatic detection of potential layout faults following changes to responsive web pages (N),” dans *30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015, Lincoln, NE, USA, November 9-13, 2015*, M. B. Cohen, L. Grunske, et M. Whalen, éds. IEEE Computer Society, 2015, pp. 709–714. [En ligne]. Repéré à : https://doi.org/10.1109/ASE.2015.31

[32] T. A. Walsh, G. M. Kapfhammer, et P. McMinn, “Redecheck : an automatic layout failure checking tool for responsively designed web pages,” dans *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis, Santa Barbara, CA, USA, July 10 - 14, 2017*, T. Bultan et K. Sen, éds. ACM, 2017, pp. 360–363. [En ligne]. Repéré à : https://doi.org/10.1145/3092703.3098221

[33] I. Althomali, G. M. Kapfhammer, et P. McMinn, “Automatic visual verification of layout failures in responsively designed web pages,” dans *12th IEEE Conference on Software Testing, Validation and Verification, ICST 2019, Xi’an, China, April 22-27, 2019*. IEEE, 2019, pp. 183–193. [En ligne]. Repéré à : https://doi.org/10.1109/ICST.2019.00027

[34] Y. Ryou et S. Ryu, “Automatic detection of visibility faults by layout changes in HTML5 web pages,” dans *11th IEEE International Conference on Software Testing, Verification and Validation, ICST 2018, Västerås, Sweden, April 9-13, 2018*. IEEE Computer Society, 2018, pp. 182–192. [En ligne]. Repéré à : https://doi.ieeecomputersociety.org/10.1109/ICST.2018.00027

[35] C. Meniar, F. Opalvens, et S. Hallé, “Runtime verification of user interface guidelines in mobile devices,” dans *Runtime Verification - 17th International Conference, RV 2017, Seattle, WA, USA, September 13-16, 2017, Proceedings*, ser. Lecture Notes in Computer Science, S. K. Lahiri et G. Reger, éds., vol. 10548. Springer, 2017, pp. 410–415. [En ligne]. Repéré à : https://doi.org/10.1007/978-3-319-67531-2_27

[36] Z. Zhang, Y. Feng, M. D. Ernst, S. Porst, et I. Dillig, “Checking conformance of applications against GUI policies,” dans *ESEC/FSE ’21 : 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Athens, Greece, August 23-28, 2021*, D. Spinellis, G. Gousios, M. Chechik, et M. D. Penta, éds. ACM, 2021, pp. 95–106. [En ligne]. Repéré à : https://doi.org/10.1145/3468264.3468561

[37] K. Moran, B. Li, C. Bernal-Cárdenas, D. Jelf, et D. Poshyvanyk, “Automated reporting of GUI design violations for mobile apps,” dans *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, M. Chaudron, I. Crnkovic, M. Chechik, et M. Harman, éds. ACM, 2018, pp. 165–175. [En ligne]. Repéré à : https://doi.org/10.1145/3180155.3180246

[38] Y. Wang, H. Xu, Y. Zhou, M. R. Lyu, et X. Wang, “Textout : Detecting text-layout bugs in mobile apps via visualization-oriented learning,” dans *30th IEEE International*

Symposium on Software Reliability Engineering, ISSRE 2019, Berlin, Germany, October 28-31, 2019, K. Wolter, I. Schieferdecker, B. Gallina, M. Cukier, R. Natella, N. R. Ivaki, et N. Laranjeiro, éds. IEEE, 2019, pp. 239–249. [En ligne]. Repéré à : https://doi.org/10.1109/ISSRE.2019.00032

[39] E. Winter, V. Nowack, D. Bowes, S. Counsell, T. Hall, S. Ó. Haraldsson, et J. R. Woodward, "Let's talk with developers, not about developers : A review of automatic program repair research," *IEEE Trans. Software Eng.*, vol. 49, nº 1, pp. 419–436, 2023. [En ligne]. Repéré à : https://doi.org/10.1109/TSE.2022.3152089

[40] S. Mahajan, A. Alameer, P. McMinn, et W. G. J. Halfond, "Automated repair of layout cross browser issues using search-based techniques," dans *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis, Santa Barbara, CA, USA, July 10 - 14, 2017*, T. Bultan et K. Sen, éds. ACM, 2017, pp. 249–260. [En ligne]. Repéré à : https://doi.org/10.1145/3092703.3092726

[41] ——, "Automated repair of internationalization presentation failures in web pages using style similarity clustering and search-based techniques," dans *11th IEEE International Conference on Software Testing, Verification and Validation, ICST 2018, Västerås, Sweden, April 9-13, 2018*. IEEE Computer Society, 2018, pp. 215–226. [En ligne]. Repéré à : https://doi.ieeecomputersociety.org/10.1109/ICST.2018.00030

[42] S. Mahajan, N. Abolhassani, P. McMinn, et W. G. J. Halfond, "Automated repair of mobile friendly problems in web pages," dans *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, M. Chaudron, I. Crnkovic, M. Chechik, et M. Harman, éds. ACM, 2018, pp. 140–150. [En ligne]. Repéré à : https://doi.org/10.1145/3180155.3180262

[43] X. Chamberland-Thibeault et S. Hallé, "Structural profiling of web sites in the wild," dans *Web Engineering - 20th International Conference, ICWE 2020, Helsinki, Finland, June 9-12, 2020, Proceedings*, ser. Lecture Notes in Computer Science, M. Bieliková, T. Mikkonen, et C. Pautasso, éds., vol. 12128. Springer, 2020, pp. 27–34. [En ligne]. Repéré à : https://doi.org/10.1007/978-3-030-50578-3_3

[44] ——, "An an empirical study of web page structural properties," *J. Web Eng.*, vol. 20, nº 4, pp. 971–1002, 2021. [En ligne]. Repéré à : https://doi.org/10.13052/jwe1540-9589.2044

[45] X. Chamberland-Thibeault et S. Hallé, “Structural profiling of web sites in the wild,” mars 2020. [En ligne]. Repéré à : https://doi.org/10.5281/zenodo.3718598

[46] S. Hallé, R. Khoury, et M. Awesso, “Streamlining the inclusion of computer experiments in a research paper,” *Computer*, vol. 51, n° 11, pp. 78–89, 2018. [En ligne]. Repéré à : https://doi.org/10.1109/MC.2018.2876075

[47] I. Hickson et D. Hyatt, “HTML 5 : A vocabulary and associated APIs for HTML and XHTML (working draft),” World Wide Web Consortium, Rapport Technique, 2008, http://www.w3.org/TR/2008/WD-html5-20080122/.

[48] I. Hickson, R. Berjon, S. Faulkner, T. Leithead, E. D. Navara, et T. O. S. Pfeiffer, “HTML 5 : A vocabulary and associated APIs for HTML and XHTML (recommendation),” World Wide Web Consortium, Rapport Technique, 2014, http://www.w3.org/TR/2014/REC-html5-20141028/.

[49] “Html <wbr> tag,” 2023. [En ligne]. Repéré à : https://www.w3schools.com/TAgs/tag_wbr.asp

[50] I. SEOMoz, “The Moz top 500 websites,” 10 2019. [En ligne]. Repéré à : https://moz.com/top500

[51] D. Perry, “Acid3 test simplified ; all modern browsers score 100,” 2011, https://www.tomsguide.com/us/acid3-browser-test-web-standard-compatibility-IE9,news-12583.html, Retrieved January 14th, 2020.

[52] S. G. Ainsworth, M. L. Nelson, et H. V. de Sompel, “Only one out of five archived web pages existed as presented,” dans *Proceedings of the 26th ACM Conference on Hypertext & Social Media, HT 2015, Guzelyurt, TRNC, Cyprus, September 1-4, 2015*, Y. Yesilada, R. Farzan, et G. Houben, éds. ACM, 2015, pp. 257–266. [En ligne]. Repéré à : https://doi.org/10.1145/2700171.2791044

[53] A. Lerner, T. Kohno, et F. Roesner, “Rewriting history : Changing the archived web from the present,” dans *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, B. M. Thuraisingham, D. Evans, T. Malkin, et D. Xu, éds. ACM, 2017, pp. 1741–1755. [En ligne]. Repéré à : https://doi.org/10.1145/3133956.3134042