UQAC
Université du Québec
à Chicoutimi

MÉMOIRE

PRÉSENTÉ À

L'UNIVERSITÉ DU QUÉBEC À CHICOUTIMI

COMME EXIGENCE PARTIELLE

DE LA MAÎTRISE EN INFORMATIQUE

PAR

CHAFIK MENIAR

VÉRIFICATION DE NORMES D'INTERFACE UTILISATEUR DANS LES APPLICATIONS MOBILES

AOÛT 2018

TABLE DES MATIÈRES

TABLE DES FIGURES

LISTE DES TABLEAUX

LISTE DES SOURCES

RÉSUMÉ

Avec le nombre croissant et l'importance des applications mobiles dans le marché, l'automatisation des tests pour ces applications est devenue un défi de recherche pertinent. Parmi les différents types de tests, l'évaluation de l'interface utilisateur graphique (GUI) est l'un des principaux problèmes.

La conception de l'interface utilisateur d'une application moderne doit respecter un ensemble de spécifications, codifiées dans un document publié par un éditeur de système d'exploitation particulier. Ces spécifications sont destinées à garantir un niveau minimal de qualité et de cohérence afin d'avoir un produit bien conçu, accessible aux utilisateurs de toutes aptitudes (y compris ceux qui ont une déficience visuelle, une cécité, une déficience auditive, une déficience cognitive ou une déficience motrice).

Au cours des dernières années, un certain nombre d'outils automatisés ont été présentés pour gérer les tests de l'interface graphique. Néanmoins, les techniques les plus courantes ont été développées uniquement pour tester la fonctionnalité et la sécurité, sans tenir compte de la vérification des spécifications concernant la conception visuelle. Aucun outil automatisé pour la vérification des spécifications en temps réel pour les applications mobiles n'est actuellement connu. Actuellement, la vérification de la conformité à ces spécifications se fait par des tests

manuels. Dans ce mémoire, nous présentons une méthodologie, basée sur la vérification en temps réel, afin d'automatiser la vérification des spécifications de l'interface utilisateur pour les applications mobiles.

CHAPITRE 1

INTRODUCTION

Au cours de ces dernières années, plusieurs recherches se sont penchées sur le test de logiciels (aussi appelé « software testing ») [1] pour éviter différents bugs qui peuvent s'avérer très dangereux et assez coûteux. La répercussion de ces bugs ne se limite pas à des pertes financières ; dans des cas extrêmes, des pertes de vies humaines ont été déplorées. Plusieurs exemples peuvent être cités, dont parmi les plus célèbres :

— En 1994, un avion de China Airlines s'est écrasé à cause d'un bug logiciel causant plus de 264 morts [2].

— En 2016, des millions de clients de la banque HSBC n'ont pas pu accéder à leurs comptes en ligne via les applications mobiles à cause d'un bug informatique majeur. Le problème n'a pu être réglé qu'après deux jours, causant des millions en dommages à la banque. La même banque avait eu en 2015 un problème avec son système de paiement électronique, ce qui a empêché le traitement de 275 000 paiements individuels et a laissé beaucoup de ses clients sans solde avant la fin de la semaine de la Fête nationale [3].

— En décembre 2015, à cause d'un bug dans une application qui calcule la peine des

1. https://en.wikipedia.org/wiki/Software_testing
2. https://www.airfleets.fr/crash/crash_report_ChinaAirlines_B-1816.htm
3. https://www.theguardian.com/business/2016/jan/05/hsbc-customers-vent-fury-over-online-banking-problems

prisonniers selon leur comportement, plus de 3 200 prisonniers américains ont été libérés 49 jours avant la date prévue de leur libération. Ce bug a perduré pendant 13 ans [4].

— En avril 2015, Nissan a rappelé plus de 3,2 millions de voitures du marché en raison d'une défaillance logicielle dans les détecteurs de coussin gonflable. Cette défaillance avait déjà causé deux accidents graves [5].

— En 2014, plusieurs marchands tiers d'Amazon ont subi de lourdes pertes, parce que le prix de leurs produits a été réduit à 1 point en raison d'un problème logiciel [6].

— En 2012, la société américaine National Grid Gas Company a opté pour un nouveau système SAP. Cependant, ce logiciel a été incorrectement mis en œuvre, entraînant des problèmes tels que des paiements erronés de salaires et des factures de fournisseurs impayées. Un montant dépassant les 945 millions de dollars a été dépensé pour résoudre ce problème [7].

Tous ces exemples démontrent le danger des bugs informatiques et leur impact sur le rendement des logiciels dans différents domaines comme la santé, l'agriculture, le tourisme, l'automobile, la finance, etc.

Avec l'émergence des applications mobiles dans plusieurs domaines, la majorité des nouveaux logiciels possèdent une version mobile (Wikipédia, 2017). Les applications mobiles ne fonctionnent pas seulement sur les téléphones intelligents, mais aussi sur d'autres appareils comme les tableaux de bords des voitures, les avions, les drones, les appareils GPS, etc. Il est à noter que l'utilisation des applications mobiles augmente d'une façon exponentielle : selon

4. http://www.bbc.com/news/technology-35167191
5. https://www.nytimes.com/2016/04/30/business/nissan-recalls-3-5-million-vehicles-for-airbag-problems.html
6. http://www.bbc.com/news/uk-northern-ireland-foyle-west-30475542
7. http://www.businessinsider.com/national-grid-sap-1-billion-upgrade-cost-2014-10

des statistiques récentes [8], environ 205.4 milliards d'applications mobiles ont été téléchargées jusqu'en 2018 et ce nombre devrait atteindre 258.2 milliards de téléchargements en fin de 2022. Les revenus générés par ces applications devraient atteindre les 188.9 milliards de dollars en fin de 2020 [9]. Il existe des applications mobiles qui ont atteint plus d'un milliard d'utilisateurs, comme l'application de réseautage mobile Facebook, l'application de Voip Skype et l'application de PayPal. Le succès commercial de ces applications dépend de leur bon fonctionnement et leur convivialité sur une grande variété d'appareils. Or, ce type de logiciel n'échappe malheureusement pas aux bugs. Au contraire, le nombre d'applications mobiles et la rapidité avec laquelle elles sont produites les rendent encore plus susceptibles de contenir davantage d'erreurs.

L'IMPORTANCE DES TESTS MOBILES

Dans ce mémoire, on s'intéressera particulièrement aux bugs des applications mobiles qui concernent leur **interface graphique**. Les spécifications ou les bonnes pratiques de l'interface graphique définissent les règles à respecter dans l'interface graphique comme le positionnement des menus, le choix de police, conventions sur les icônes, les boutons, les boîtes de dialogue et la disposition fenêtres. Elles peuvent être utilisées pour vérifier également l'utilisation de la couleur, les effets de transparence, l'ombrage ou les animations. Par exemple pour l'utilisation des boutons plats, une règle peut spécifier que le texte de bouton doit être coloré et qu'il faut l'utiliser soit dans les barres d'outils ou les boîtes de dialogue. Plusieurs éditeurs de systèmes ont publié des normes régissant les interfaces de leurs applications, comme Apple IO (Apple, 2017), Android (Android, 2017), GNOME (The GNOME Project,

8. https://www.statista.com/statistics/271644/worldwide-free-and-paid-mobile-app-store-downloads/

9. https://www.statista.com/statistics/269025/worldwide-mobile-app-revenue-forecast/

2017) ou Windows (Microsoft Corporation, 1992).

La seule solution existante jusqu'à présent qui permet d'avoir une interface mobile de qualité consiste à réaliser des tests logiciels ; ceux-ci sont devenus une étape indispensable de la conception de n'importe quel logiciel. Ces tests peuvent être exécutés d'une façon manuelle ou automatisée.

La vérification manuelle consiste à essayer l'application par des testeurs qui essaient de simuler tous les cas de figures d'exécutions possibles en vérifiant si l'application répond aux attentes de l'utilisateur. La deuxième méthode est automatique ; elle consiste à remplacer les testeurs humains par des outils d'automatisation. Les avantages de la deuxième solution (SupInfo, 2016) la rendent plus intéressante et beaucoup plus utilisée, d'autant plus qu'elle permet un gain d'effort, de temps et d'argent par rapport à la première. De plus, elle permet de tester plus de fonctionnalités et augmente la productivité. En effet, l'utilisation des outils automatisés permet aussi de couvrir plusieurs cas des tests possibles et les interactions utilisateur/machine, ce qui n'est pas toujours garanti avec les testeurs humains. D'ailleurs, les outils spéciaux permettent non seulement d'effectuer automatiquement les cas des tests mais de simuler l'interaction de l'utilisateur avec le système.

Actuellement, il existe plusieurs études et approches concernant les tests de la fonctionnalité, de sécurité et de convivialité des applications mobiles, mais le domaine des tests pour vérifier des spécifications de l'interface utilisateur en temps réel n'a pas encore été largement exploré. La situation actuelle montre qu'il existe un besoin pour un outil de test automatisé afin de vérifier des spécifications de l'interface utilisateur pour les applications mobiles. La question qui se pose donc est : comment peut-on vérifier ces règles au niveau de l'interface graphique pour les applications mobiles, et ce en temps réel ? C'est le sujet du présent mémoire.

Le résultat de ce travail est une approche qui peut être utilisée pour vérifier les spécifications

de l'interface graphique dans les applications mobiles. Pour ce fait, il faut développer un outil qui analyse l'état de l'interface, et formaliser les règles de manière à ce qu'un algorithme puisse identifier les règles violées.

Ce travail comprend quatre chapitres. Le premier chapitre donne une idée générale sur les spécifications de l'interface graphique ainsi l'objectif de ce mémoire. Le deuxième chapitre est consacré aux systèmes d'exploitation, les tests GUI et les spécifications de l'interface graphique pour les applications mobiles. Le troisième chapitre est dédié aux travaux connexes concernant les outils de test d'interfaces dans les applications mobiles. Le quatrième chapitre présente la vérification des spécifications en temps réel, l'outil Comidroid, la syntaxe pour exprimer les propriétés, et la manière dont cet outil analyse l'état des éléments de l'interface graphique.

CHAPITRE 2

LES TESTS DANS LES APPLICATIONS MOBILES

Les applications mobiles ont connu un grand succès dans les dernières années, qui peut être mesuré par le nombre de téléchargements et leur utilisation dans presque tous les domaines. Ces applications peuvent être distribuées ou publiées dans les magasins d'applications comme Google Play, Apple Store, ou encore le Windows Phone Store. Ces magasins peuvent contenir des applications payantes ou gratuites ; les applications gratuites sont généralement monétisées avec des publicités. Il est à noter que l'application passe par un système d'abrogation établi par les éditeurs de systèmes mobiles. Selon ce système, l'application doit respecter la politique et les règles propres à chaque magasin.

Dans ce chapitre, nous allons présenter les principaux systèmes d'exploitation afin de connaître les particularités et les contraintes concernant la distribution des applications mobiles, et les différents types de tests qu'on peut effectuer sur les applications mobiles. Nous allons également donner un aperçu sur les tests d'interface utilisateur et les spécifications de l'interface graphique pour les applications mobiles.

2.1 LES SYSTÈMES D'EXPLOITATION MOBILES

Nous commençons par un bref tour de piste des principaux écosystèmes d'applications mobiles. On en dénombre cinq : Android, iOS, Windows Phone et BlackBerry OS.

2.1.1 ANDROID

Android est un système d'exploitation conçu par Google. Il est *open source* et basé sur Linux. Android est le système d'exploitation le plus utilisé dans le monde, avec plus de 85% de parts de marché dans les téléphones intelligents (Zednet, 2017). La plupart des fabricants et des opérateurs cellulaires utilisent et modifient le système d'exploitation Android en fonction de leur matériel.

Le nombre d'appareils Android est en croissance : il y avait 18 796 appareils Android distincts en 2014, comparativement à 11 868 en 2013 (GÖTH, 2015). Cette diversité augmente la complexité du système, les coûts et la complexité des tests. Cependant de nombreux outils et frameworks de test visent principalement Android en raison de sa popularité et du nombre d'applications disponibles dans les boutiques de téléchargement. La boutique principale de téléchargement d'Android est Google Play, mais il existe également d'autres boutiques comme Samsung Apps ou Amazon Store.

Google fournit une liste des informations [1] sur les spécifications de conception, les recommandations relatives aux tests et d'autres informations aux développeurs. Cette liste d'informations doit être vérifiée avant de soumettre une nouvelle application à Google Play. La figure 2.1 montre un exemple d'un cas à respecter. Pendant la soumission, Google utilise un processus

1. `https://developer.android.com/distribute/best-practices/launch/launch-checklist.html`

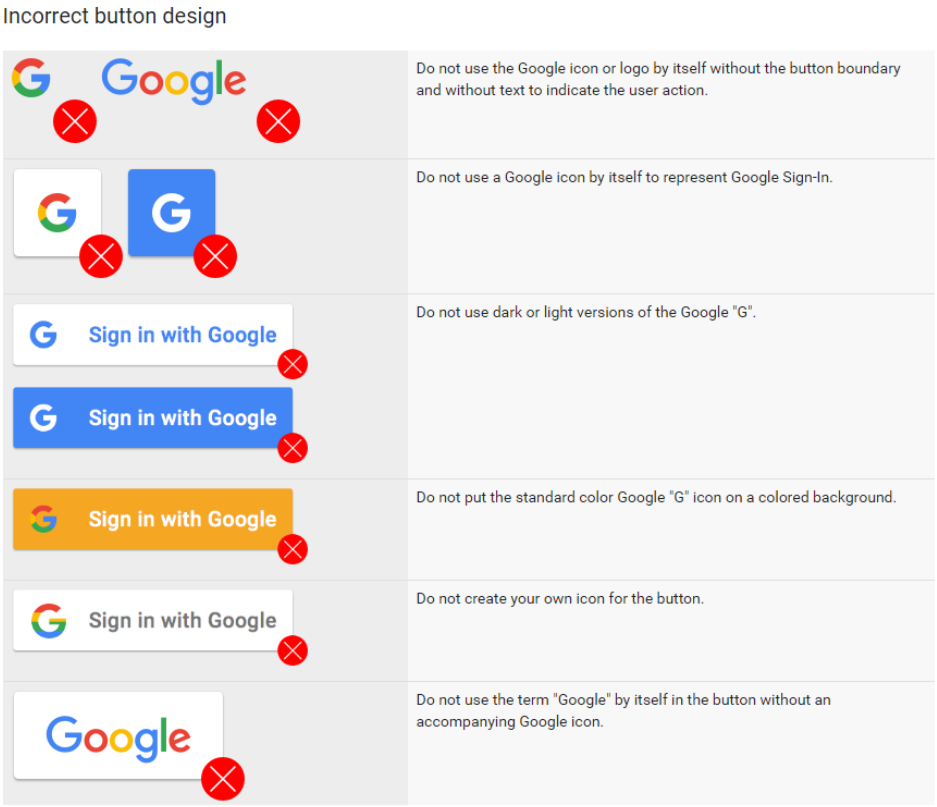

Figure 2.1 – Exemple d'une règle à appliquer sur le bouton Google

d'approbation qui analyse chaque application ou mise à jour. Si l'application ne respecte pas les règles, elle peut être supprimée ou suspendue de la part de Google ; de plus, le temps d'approbation peut prendre jusqu'à sept jours pour publier l'application. C'est pour cette raison qu'il est nécessaire de bien tester et vérifier l'application avant de la soumettre à Google Play.

Android fournit un sous-ensemble des bibliothèques Java adaptées aux composantes d'interface utilisateur graphique et aux événements qui se produisent via l'interface tactile. Les éléments de l'interface peuvent être définis dans un fichier XML et dans le code source. La connexion entre les fenêtres est définie par la programmation dans le code source, et non dans le fichier XML. Les objets du GUI sont structurés hiérarchiquement comme le montre la figure 2.2.

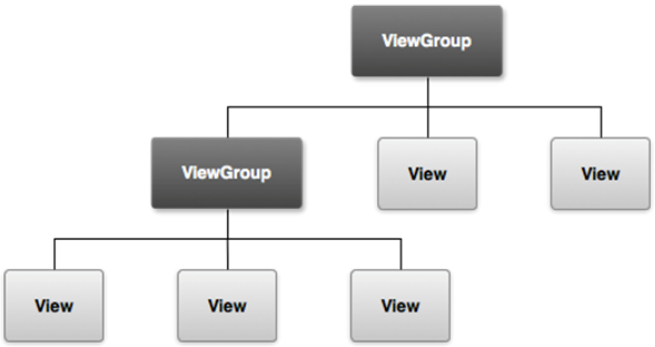

Figure 2.2 – Arborescence de la vue dans le système Android

Les `ViewGroups` (Google, 2016) sont des conteneurs pouvant inclure d'autres éléments `ViewGroup`, lesquels peuvent contenir des éléments `View` tel que les widgets.

Il existe un certain nombre de widgets fournis par le toolkit d'Android [2] qui permettent de créer l'interface utilisateur graphique. Parmi ces widgets, on compte :

— `Button` : Un bouton sur lequel l'utilisateur peut appuyer ou cliquer pour effectuer une action.

— `CalendarView` : Un widget de calendrier pour afficher et sélectionner des dates.

— `CheckBox` : Une case à cocher est un type spécifique de bouton qui peut être coché ou décoché.

— `TextView` : Un élément d'interface utilisateur qui affiche du texte à l'utilisateur.

— `ListView` : Un élément qui affiche une collection de vues défilant verticalement.

— `LinearLayout` : Un Layout qui organise les vues horizontalement dans une seule colonne ou verticalement dans une seule ligne.

— `RelativeLayout` : Un Layout où les les éléments enfants peuvent être positionnés ou placés les unes par rapport aux autres ou au parent.

2. `https://developer.android.com/reference/android/widget/package-summary.html`

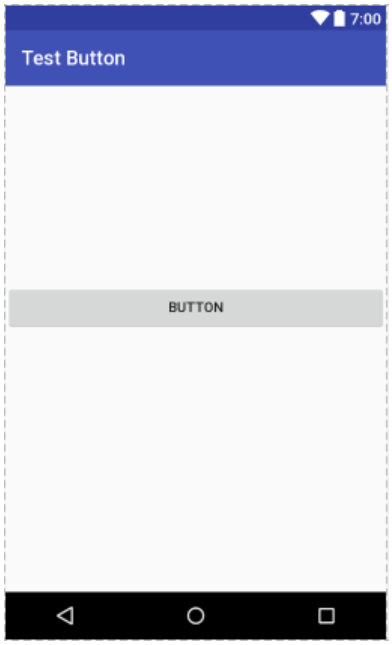

Figure 2.3 – Widget bouton en Android

— `ProgressBar` : Un élément d'interface utilisateur qui indique la progression d'une opération.

— `RadioButton` : Un bouton radio est un bouton à deux états qui peut être coché ou décoché.

— `Spinner` : Une vue qui affiche plusieurs éléments enfant sous forme une liste et permet à l'utilisateur de choisir parmi ces éléments.

— `SeekBar` : Une vue qui étend de l'élément ProgressBar en ajoutant un Icon déplaçable.

— `SnackBar` : Fournissent des commentaires légers sur une opération. Ils affichent un bref message en bas de l'écran sur le mobile.

— `EditText` : Un élément d'interface utilisateur pour entrer et modifier du texte.

— `RatingBar` : Affiche la barre de notation.

— `Datepicker` : Affiche la boîte de dialogue qui peut être utilisée pour choisir la date.

On peut définir les éléments de l'interface graphique dans un fichier XML, ou dans le code source Java. Par exemple, les codes source 2.1 et 2.5 montrent comment créer le widget du bouton présenté dans la figure 2.3.

Code source 2.1 – Définition d'un bouton dans le système Android, au moyen d'une déclaration XML

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
android:layout_width="match_parent"
android:layout_height="wrap_content">
<Button
android:id="@+id/btn"
android:layout_width="wrap_content"
android:layout_height="wrap_content"
android:layout_marginTop="200dp"
android:text="Button" />
</LinearLayout>
```

Code source 2.2 – Définition d'un bouton dans le système Android, au moyen d'instructions en Java

```
LinearLayout linearLayout = findViewById(R.id.rootContainer);
// Instancier le Bouton
Button btn = new Button(this);
btn.setText(Button);
btn.setLayoutParams(new LinearLayout.LayoutParams(ViewGroup.LayoutParams.WRAP_CONTENT, ViewGroup.
    LayoutParams.WRAP_CONTENT));
LinearLayout.LayoutParams lp = (LinearLayout.LayoutParams)
btn.getLayoutParams();
// Définir la marge du bouton de 200 pixels en haut
// Gauche, en haut, droite, en bas
lp.setMargins(0,200,0,0);

// Appliquer les parametres de Layout au bouton
btn.setLayoutParams(lp);
// Ajouter le bouton au Linearlayout
if (linearLayout != null) {
linearLayout.addView(btnShow);
}
```

On peut accéder à ces éléments de l'interface utilisateur en Android avec leur ID, leurs tags et leurs noms qui sont défini dans la structure XML, ou générés automatiquement. Le traitement des événements comme le clic ou le glisser-déposer, quant à eux, sont enregistrés dans des objets appelés `Listener`, comme le montre le code source 2.3.

Code source 2.3 – Instructions en Java pour accéder aux éléments et intercepter leurs événements

```

Button btn = (Button) findViewById(R.id.btn);

// Définir un Listner pour le widget Bouton
btn.setOnClickListener(new View.OnClickListener() {
@Override
public void onClick(View v) {
```

```
8  | // click boutton
9  | }
10 | });
11 |
12 | btn.setOnTouchListener(new View.OnTouchListener() {
13 | @Override
14 | public boolean onTouch(View v, MotionEvent event) {
15 | switch(event.getAction()) {
16 | case MotionEvent.ACTION_DOWN:
17 | System.out.println("␣pressé␣");
18 | return true;
19 | case MotionEvent.ACTION_UP:
20 | System.out.println("␣libéré␣");
21 | return true;
22 | }
23 | return false;
24 | }
25 | });
```

2.1.2 IOS

iOS est un système d'exploitation mobile développé par Apple pour ses propres appareils. Il a été développé à l'origine pour les iPhone, mais maintenant iOS fonctionne sur iPad, iPod Touch et Apple TV. Contrairement à Android, les autres fabricants ne sont pas autorisés à utiliser iOS. Par conséquent, seuls les périphériques fabriqués par Apple peuvent l'utiliser (Zednet, 2017). En septembre 2014, la part de marché d'iOS était de 11%, et environ 90% des utilisateurs possédaient la dernière version (iOS 10) ou bien l'avant-dernière version (iOS 9). Les fonctionnalités de ces versions de iOS dépendent des périphériques (GÖTH, 2015). En conséquence, les tests sur tous les périphériques disponibles ne sont pas aussi difficiles en comparaison avec Android.

La boutique de téléchargement de iOS est le Apple Store. Les applications soumises à l'Apple Store doivent passer par un processus d'approbation difficile. L'application doit répondre aux spécifications de l'Apple Store et aux exigences d'interface utilisateur. Elle doit également être quelque chose d'utile ou d'unique, et répondre à des exigences de qualité élevées. Le processus complet d'approbation des demandes prend entre quatre et onze jours. Au cours de ce processus, l'application passe par des tests automatiques et manuels. Elles peuvent être publiées ou rejetées ; si une application est rejetée, les développeurs reçoivent par la suite des

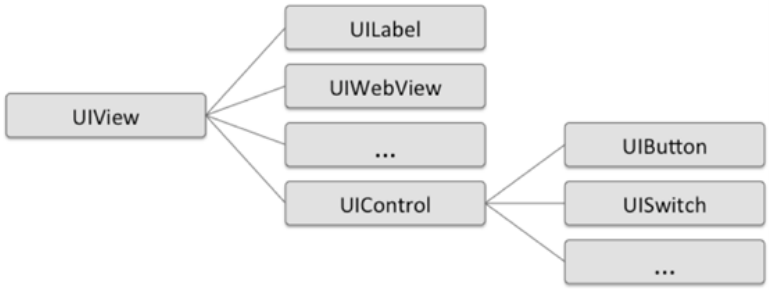

Figure 2.4 – L'arborescence de la vue dans le système iOS

commentaires sur les points non respectés.

Dans iOS, l'environnement de développement pour les applications natives est Xcode qui peut être installé uniquement sur le système d'exploitation OS X d'Apple. Xcode dispose d'un générateur d'interface graphique intégrée pour générer un GUI. La hiérarchie du GUI est sous forme d'un arbre, et est un peu similaire à Android, comme montre la figure 2.4. La vue (View) (Apple, 2016) est générée et stockée dans le fichier XIB, qui est un fichier XML et hiérarchiquement organisé. Pendant la construction, les fichiers XIB sont compilés ; ils ne peuvent pas être édités à la main. Les autres parties de l'application représentent le modèle et le contrôleur. La connexion entre les fenêtres est similaire à l'architecture Android et est définie par programmation dans le code source.

Les vues et les contrôles constituent les blocs de construction visuels de l'interface utilisateur d'une application. Pour dessiner et organiser le contenu d'une application à l'écran, il faut utiliser les éléments du framework Uikit [3] de l'interface utilisateur, par exemple :

— `TextFields` : C'est un élément de l'interface utilisateur qui permet à l'application

3. `https://developer.apple.com/documentation/uikit/views_and_controls`

d'obtenir des informations de la part de l'utilisateur.

— `Buttons` : Il est utilisé pour gérer les actions de l'utilisateur.

— `Label` : Il est utilisé pour afficher le contenu statique.

— `Toolbar` : Il est utilisé si nous voulons manipuler quelque chose en fonction de notre vision.

— `Navigation Bar` : Il contient les boutons de navigation d'un contrôleur de navigation ; les contrôles peuvent être poussés et déplacés.

— `Image View` : Il est utilisé pour afficher une image simple ou une séquence d'images.

— `TextView` : Il est utilisé pour afficher une liste déroulante d'informations textuelles.

— `Pickers` : Il est utilisé pour afficher et sélectionner une donnée spécifique d'une liste.

— `Sliders` : Il est utilisé pour permettre aux utilisateurs de faire des ajustements à une valeur ou un processus dans une plage de valeurs autorisées.

— `ScrollView` : Il est utilisé pour afficher un contenu qui est plus grand que la zone d'écran.

Lorsque l'on appuie sur un bouton ou que l'on sélectionne un bouton qui a le focus, le bouton effectue toutes les actions qui y sont attachées. Un exemple de code source ci-dessous montre comment définir un bouton et gérer l'événement de clic.

Code source 2.4 – La définition d'un bouton dans iOS

```
1
2  -(void)addDifferentTypesOfButton {
3
4  UIButton *roundRectButton = [UIButton buttonWithType:
5  UIButtonTypeRoundedRect];
6  [roundRectButton setFrame:CGRectMake(60, 50, 200, 40)];
7  // Ajouter un titre
8  [roundRectButton setTitle:@"Button" forState:
9  UIControlStateNormal];
10 [self.view addSubview:roundRectButton];
11 }
```

Code source 2.5 – La détection d'un événement de clic dans iOS

```
1
2  class ViewController: UIViewController {
3  // compteur de clic
4  var clickCount = 0
5
6  @IBAction func buttonClick(sender: UIButton) {
7  // incrémenter le nombre de clic
8  clickCount+=1
9  }
10 }
```

2.1.3 WINDOWS PHONE ET BLACKBERRY OS

Windows Phone est un système d'exploitation mobile développé par Microsoft. En septembre 2014, la part de marché de Windows Phone était de 2,5%. La part de marché de BlackBerry, quant à elle, était descendue à 1% en 2014. Concernant la distribution de l'application dans les boutiques de ces deux systèmes, il y a également des spécifications à respecter. Cependant elles ne se sont pas nombreuses comparé à Android et iOS. Concernant l'interface graphique, Windows Phone OS utilise les langages de la famille .NET pour le développement de l'application. Il utilise aussi le langage XML pour créer l'interface utilisateur. La création d'une interface est très similaire à Android. Quant à BlackBerry OS, il offre un framework appelé *Cascades* pour créer l'interface utilisateur, en utilisant un langage de balisage appelé QML (Qt Modeling Language). C'est un langage déclaratif (similaire à CSS et JSON) pour la conception d'applications.

Étant donné que ces deux systèmes ne spécifient que très peu de normes de conception d'interface, nous n'en parlerons pas en détail dans ce mémoire.

2.2 TYPES DE TESTS POUR LES APPLICATIONS MOBILES

Pour que les applications soient de bonne qualité et conviviale, elles doivent passer par plusieurs types de tests. Le test d'applications mobiles est une activité visant à vérifier si les

Tableau 2.1 – Exemple d'un cas de test

id	Description	Actions	Affichage
T0	vérifier l'addition de 4+5	ouvrir l'application	0
		cliquer sur le premier nombre 4	4
		cliquer sur l'oprateur +	4+
		cliquer sur le nombre 5	4+5
		cliquer sur l'opérateur =	4+5=9

résultats réels correspondent aux résultats attendus et à s'assurer que le système logiciel ne contient pas de défauts en exécutant une série d'actions appelées « cas de test ». Autrement dit, un cas de test est un ensemble d'actions exécutées afin de vérifier la fonctionnalité ou une caractéristique particulière de l'application (GURU, 2010). Le tableau 2.1 montre un exemple d'un cas de test pour vérifier l'addition de deux nombres (4+5) dans une application simple de type calculatrice.

Il existe de nombreux type de tests, tels que présentés dans l'article *Software Testing of Mobile Applications : Challenges and Future Research Directions* (Muccini et al., 2012). Parmi ces tests on trouve :

— Test de connectivité : La connexion réseau des appareils mobiles affecte la fonctionnalité de l'application ; elle doit être testée dans différents réseaux et dans des conditions de connectivité différentes.

— Test de ressources limitées : La particularité des appareils mobiles est la limitation de leurs ressources comme le CPU, le stockage, et la RAM. Le test de ressources limitées sert à vérifier si l'application peut être affectée au niveau de la performance de l'application.

— Test d'autonomie : Tester l'autonomie de l'application consiste à surveiller la consommation d'énergie lors de l'exécution de l'application sur différents périphériques et dans

des situations différentes.

— Test de nouveaux systèmes d'exploitation et diversité de dispositifs : Il existe différents systèmes d'exploitation disponibles pour les appareils mobiles, et aussi de nouvelles versions d'entre eux apparaissent régulièrement. Certains problèmes fonctionnels peuvent être causés par des problèmes d'OS. C'est pourquoi les applications mobiles doivent être testées sur différents OS et différents appareils.

— Test d'entrée/sortie et test de l'écran tactile : Contrairement aux ordinateurs de bureau avec la souris et le clavier comme modalités d'entrée, la plupart des appareils mobiles utilisent les écrans tactiles comme principale mode d'entrée. Le temps de réponse de l'écran tactile dépend du périphérique et de l'état actuel de l'application. La réaction du système à l'entrée tactile doit être testée sur différents appareils et dans des conditions différentes, telles que l'utilisation des ressources, la charge du processeur et la charge mémoire.

— Test de l'interface utilisateur : En fonction des différents appareils mobiles, la taille et la résolution des écrans, l'interface graphique peut réagir différemment au code source. Il est nécessaire de tester comment les éléments de l'interface utilisateur graphiques sont affichés et comment l'utilisateur peut interagir avec eux. Le test GUI est le processus consistant à assurer la bonne fonctionnalité et la convivialité, y compris le respect des normes de l'interface utilisateur graphique (GUI) pour une application donnée (GURU, 2010). Ce type de test est la base de cette recherche.

2.3 LES SPÉCIFICATIONS POUR L'INTERFACE GRAPHIQUE

Une interface graphique peut avoir une présentation visuelle qui n'est pas conviviale, ou une disposition des éléments qui n'est pas correcte. Ceci peut être considéré comme un bug dans

l'application ; ce bug a des effets visibles sur le contenu des fenêtres servies à l'utilisateur. Pour remédier à ces problèmes, les éditeurs des systèmes Apple IO (Apple, 2017), Android (Android, 2017), GNOME (The GNOME Project, 2017) et Windows (Microsoft Corporation, 1992) ont établi dans diverses mesures des règles ou des spécifications à suivre basées sur une étude scientifique de la relation entre l'homme et la machine. Ces règles exploitent toutes les ressources qui puissent être utilisés pour laisser l'utilisateur vivre une meilleure expérience.

Les développeurs échouent souvent à prendre en compte la perspective de l'utilisateur final ; ils sont principalement concentrés à faire fonctionner l'application à proprement parler. Ceci est particulièrement vrai s'ils développent une application de vente au détail, ou un produit qui sera utilisé par des utilisateurs non techniques. Une quantité massive de logiciels à usage interne sont créés pour les entreprises, et un minimum de temps, d'efforts ou d'argent est souvent investi dans la création d'une meilleure interface utilisateur.

Avant d'aller plus loin, il est important de différencier l'*interface* utilisateur et l'*expérience* utilisateur. L'interface utilisateur fait référence aux visuels et aux contrôles de l'application, tandis que l'expérience utilisateur englobe à la fois l'interface utilisateur et le comportement de l'application associée à l'interface utilisateur. Dans ce dernier cas, il ne s'agit pas seulement de concevoir une interface utilisateur attrayante, mais aussi de s'assurer que tout fonctionne bien. Autrement dit, pendant le développement d'une application mobile, la principale source de recommandations pour les concepteurs et les développeurs sont les spécifications de l'interface utilisateur graphique.

La vérification des spécifications pour l'interface graphique est un processus qui permet de contrôler la disposition des éléments de l'interface graphique tel que le positionnement, la taille des widgets dans des écrans différents, la police, les zones de texte, les boutons ainsi que les listes. Cette opération vérifie également l'aspect visuel des composants de l'interface

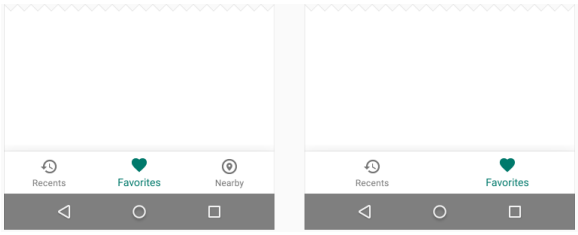

Figure 2.5 – Exemple d'un menu en bas

utilisateur, s'ils sont affichés correctement en prenant en considération l'usage des couleurs, la clarté et la visibilité des logos, des icônes ainsi que des images.

Dans ce qui suit, nous allons présenter quelques exemples des spécifications concernant en particulier l'interface graphique Android, telles que décrites dans la documentation officielle des normes d'interface de ce système [4].

— **Menu en Bas**

Le menu de navigation en bas doit contenir entre trois et cinq éléments. Sinon, il vaut mieux utiliser un autre composant de l'interface graphique (`Tabs`), parce que les boutons deviennent trop nombreux et donc trop serrés et trop petits pour être lus. La figure 2.5 montre un exemple illustrant le nombre d'éléments d'un menu en bas.

— **Les boutons surélevés**

Les boutons "plats" sont des boutons avec des textes, tandis que les boutons surélevés ("raised") ont une forme rectangulaire avec un arrière-plan. Cette règle suggère que, quand il y a beaucoup d'éléments sur l'interface, il faut utiliser les boutons en forme rectangulaire au lieu des boutons plats pour les rendre plus visibles. La figure 2.6 montre la différence entre les deux.

4. `https://developer.android.com/guide/practices/ui_guidelines/index.html`

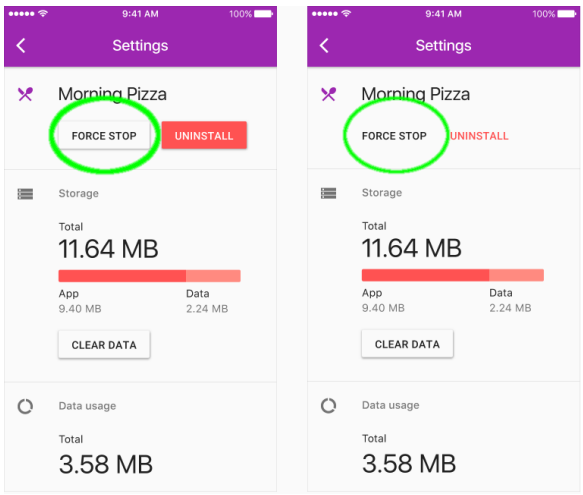

Figure 2.6 – Exemple de bouton surélevé et de bouton plat

— **La taille de la zone de clic**

Pour assurer une bonne utilisation des éléments de l'interface graphique, la zone de clic devrait avoir une taille d'au moins 48×48 de *densité* (dp), où la densité est une unité qui est basée sur la densité physique de l'écran. Dans la plupart des cas, l'espace devrait être de 8 dp ou plus entre les éléments de l'interface pour éviter que les doigts cliquent sur beaucoup d'entre eux. En général, les éléments devraient avoir une taille physique d'environ 9 mm, quelle que soit la taille de l'écran. Un exemple est donné dans la figure 2.7.

— **ListView (Liste)**

Les éléments d'une liste devraient être trié alphabétiquement. Cela rend l'information plus facile à trouver. Par conséquent, les widgets de liste ne doivent pas afficher les éléments dans un ordre aléatoire. La figure 2.8 donne un exemple de cette contrainte.

— **Bouton flottant**

Comme son nom l'indique, un bouton flottant ne doit pas être caché ou masqué par la barre qui s'affiche en bas. Autrement dit, quand on clique sur le bouton flottant, si une

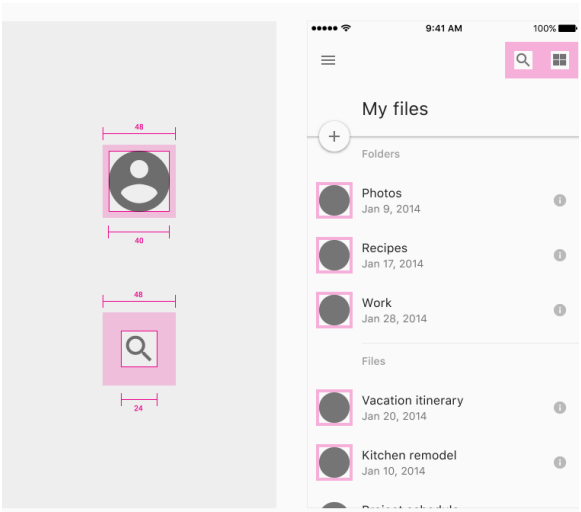

Figure 2.7 – La taille de la zone de clic

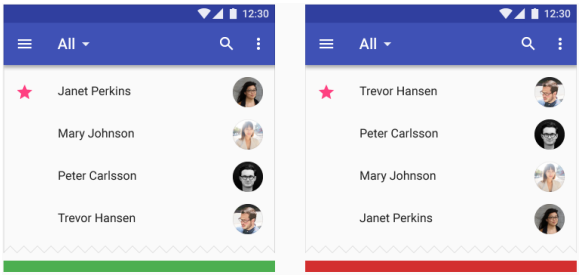

Figure 2.8 – Exemple d'une liste triée (gauche) vs non triée (droite)

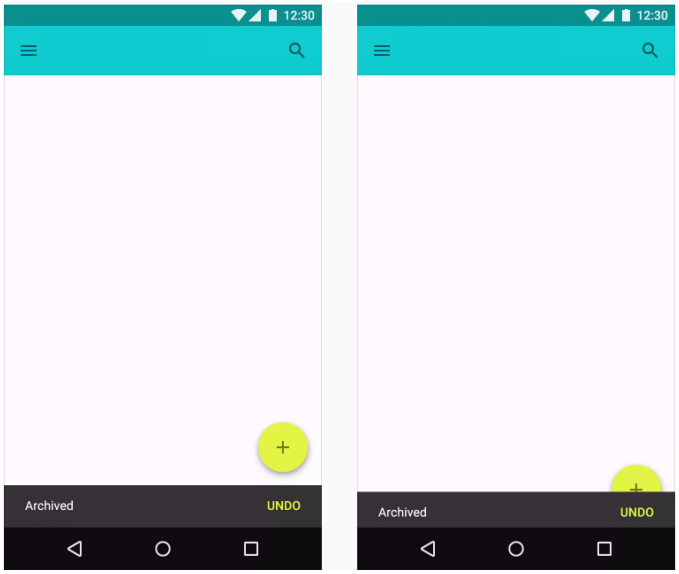

Figure 2.9 – Exemple d'un bouton flottant

barre inférieure apparaît, le bouton doit monter verticalement en haut, comme ce qui est présenté dans la figure 2.9.

— **Mise en miroir**

Pour les langues lues de droite à gauche (RTL) telles que l'arabe et l'hébreu, les fenêtres de l'application doivent être inversées pour garantir la compréhension du contenu. Le contenu RTL affecte également la direction dans laquelle certaines icônes, images et le numéro de téléphone sont affichées. La figure 2.10 montre un exemple pour cette spécification.

— **Regroupement des items**

Cette règle propose de garder les objets liés à proximité les uns des autres. Il peut être utile pour ceux qui ont une vision faible ou ont du mal à se concentrer sur l'écran et la valeur du curseur est placée trop loin du contrôle. Un utilisateur sera en mesure d'afficher à la fois le curseur et la valeur sans effectuer de panoramique dans les deux sens. Voir un exemple dans la figure 2.11.

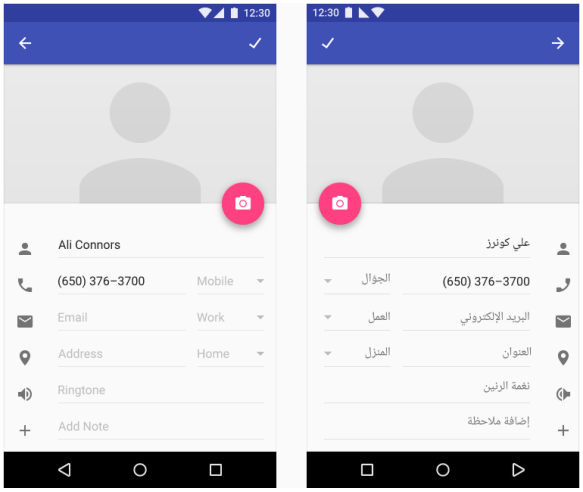

Figure 2.10 – Mise en miroir

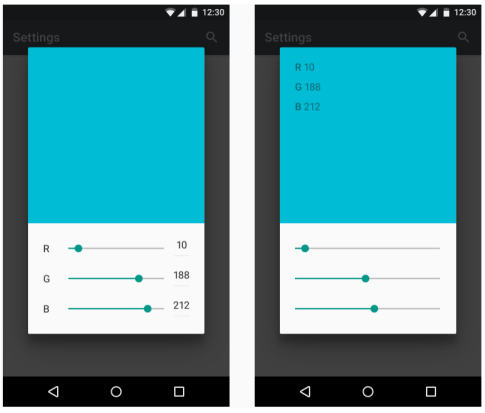

Figure 2.11 – Regroupement des items

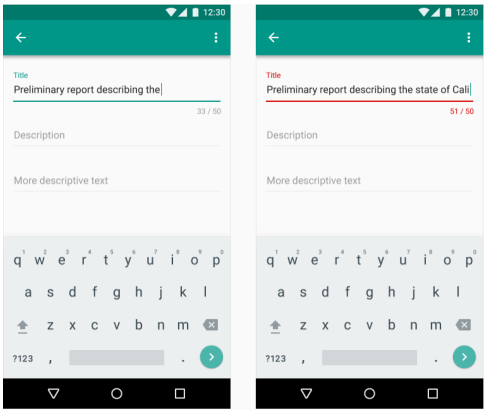

Figure 2.12 – Taille du champ de saisie

— **Taille du champ de saisie**

Dans les zones de texte qui ont un nombre limité de caractères, une simple ligne rouge doit être affichée avec un compteur de caractères indiquant que l'utilisateur s'apprête à dépasser la restriction de caractères. Sinon, la ligne doit être bleue pour indiquer à l'utilisateur qu'il peut encore continuer à entrer d'autres caractères. La figure 2.12 démontre un exemple pour cette règle.

— **Comportement d'un menu**

Les menus sont positionnés sur leurs éléments émetteurs de telle sorte que l'élément de menu réellement sélectionné apparaît au-dessus de l'élément émetteur. Il ne faut pas afficher de doublon de l'élément du menu sélectionné. La figure 2.13 montre un exemple de cette règle.

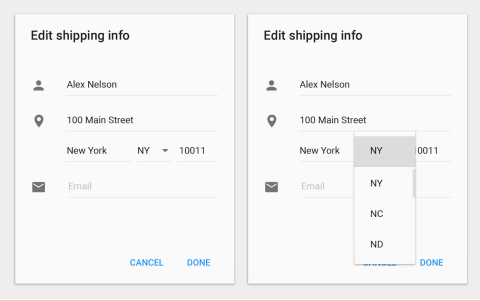

Figure 2.13 – Comportement d'un menu

CHAPITRE 3

ÉTAT DE L'ART

Malgré la relative nouveauté du développement mobile, le besoin d'automatisation de son développement et de ses tests est évident. Ce chapitre présente plusieurs outils existants pour les tests automatisés de l'interface graphique des applications mobiles. Ces outils seront classés selon les approches utilisées. Tous ces outils se basent globalement sur deux approches, l'approche d'analyse de code source et l'approche de capture, que nous allons présenter en détails dans les deux sections suivantes.

3.1 TESTS PAR CAPTURE D'ÉCRANS

L'approche de capture (GURU, 2010) consiste à recueillir et enregistrer à l'aide d'outils d'automatisation les diverses données, telles que l'entrée du clavier et de la souris, les touches tactiles et des captures des écrans pour un élément ou l'ensemble des éléments de l'interface graphique. La figure 3.1 montre que cette approche se fait en deux étapes :

1. le testeur enregistre toutes les actions de tous les cas de test possibles avec un outil d'automatisation ;
2. l'outil reprend toutes les actions enregistrées en simulant toutes les interactions de

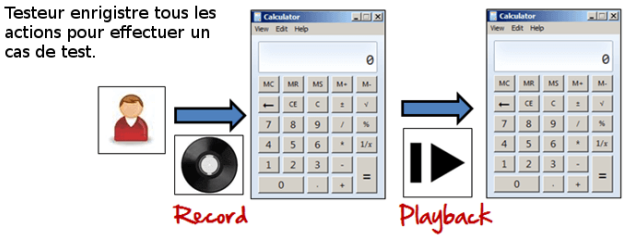

Figure 3.1 – Méthode de capture

l'utilisateur afin de tester la fonctionnalité de l'application.

De plus, cette approche se base soit sur la méthode de la *reconnaissance* d'image, ou sur la méthode de la *comparaison* de l'image. La méthode de la reconnaissance analyse les éléments contenus dans la fenêtre afin de les reconnaître, et cherche un élément semblable dans une banque de référence. La méthode de comparaison d'image, quant à elle, considère tout l'écran comme une image, puis compare les images et affiche la différence entre eux.

Il existe plusieurs outils qui se basent sur cette approche. Nous mentionnons ici les principaux.

3.1.1 BACKSTOPJS

BackstopJS (Elin, 2017) est un outil de test pour les interfaces graphiques des pages web. Il se base sur la technique de capture d'écran des pages web. BackStopJs utilise une configuration JSON pour spécifier les informations des pages à capturer comme l'URL de la page, les éléments de DOM et la taille de la fenêtre. Quant à son fonctionnement, il prend la première capture d'écran d'une page web comme une image de référence, puis à chaque changement, BackstopJS prendra une nouvelle capture d'écran appelée image de test et la comparera au image de référence, ensuite BackstopJS crée une image avec la différence et la visualise dans une page HTML comme le montre la figure 3.2.

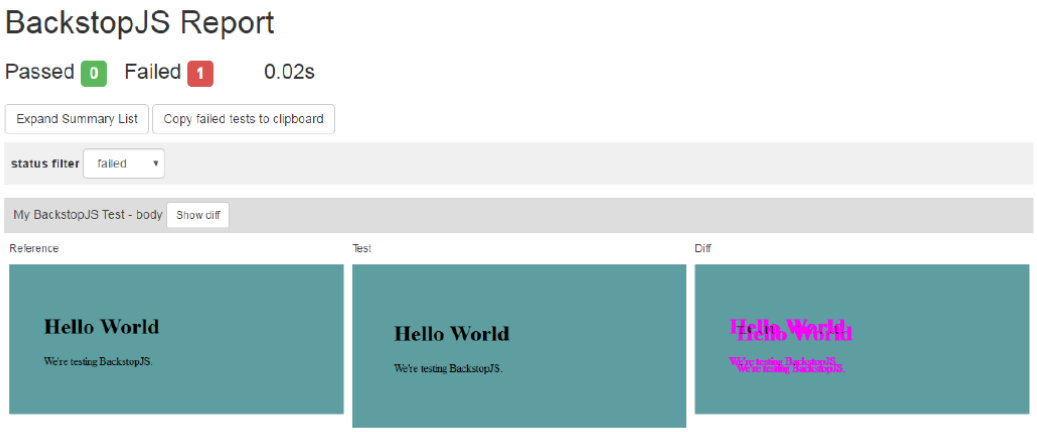

Figure 3.2 – L'interface de BackstopJS

3.1.2 WRAITH

Wraith (Elin, 2017) est un outil de test d'interface utilisateur créé par les développeurs de BBC News. Wraith utilise PhantomJS comme un navigateur web sans interface graphique pour créer des captures d'écran de pages Web sur différents environnements, puis il compare les images avec ImageMagick. ImageMagick est un open source qui permet de créer, éditer, convertir, lire et enregistre les images.

Wraith suit le processus de test UI comme suit :

1. Prendre des captures d'écran des pages Web.
2. Exécuter une tâche de comparaison entre eux.
3. Afficher un fichier PNG diff comparant les deux images et un fichier data.txt contenant le pourcentage de pixels modifiés et afficher les zones affectées surlignées en bleu comme dans la figure 3.3.
4. Rassembler tout cela dans une page html pour être visualisée. Si le diff d'une capture d'écran dépasse le seuil spécifié dans le fichier de configuration, la tâche se termine par

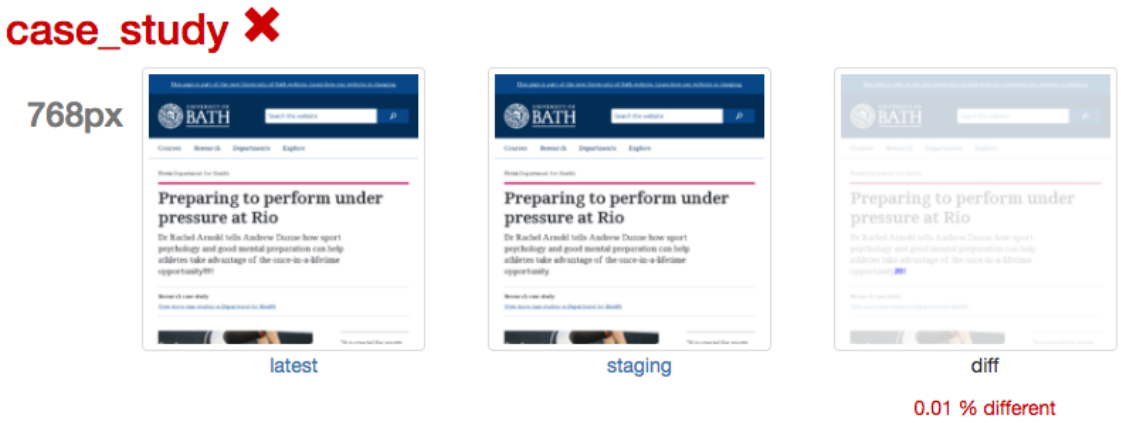

Figure 3.3 – L'interface de Wraith (University Of BATH, 2017)

un code d'erreur système.

3.1.3 SIKULI

Sikuli (Coppola et al., 2016) est un outil d'automatisation qui permet au testeur de prendre une capture d'écran pour chaque élément GUI et de définir l'action à effectuer sur la composante en question. Sikuli n'a pas besoin de code source pour tester l'application car il identifie les éléments de l'interface par leur forme ou image mais pas avec leur identificateur (id). Pendant le processus de test, l'outil recherche une composante donnée sur l'écran avec la technique de la comparaison des images. Ensuite, il compare la capture de la composante ciblée à chaque région sur l'écran en essayant de trouver l'élément le plus semblable. Cependant, il est vraiment vulnérable aux changements de texte et de graphiques, et il n'autorise pas les tests de performances des applications en cours d'exécution.

La figure 3.4 montre comment Sikuli automatise les actions de l'utilisateur. Pour trouver le bouton Enregistrer image(3), Sikuli va procéder comme suit :

1. chercher l'image 1

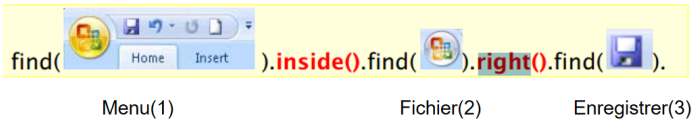

Figure 3.4 – Exemple d'automatisation avec Sikuli

2. chercher l'image 2 à l'intérieur de l'image 1
3. effectuer l'action de clic droit
4. chercher l'image 3 à l'intérieur de l'image 1

3.1.4 RANOREX

Ranorex (Anjum et al., 2017) est principalement utilisé pour les tests d'interface graphique dans Windows Phone. Ranorex fournit une interface graphique rapide et intuitive pour définir des cas de test. Cet outil prend au début le fichier de distribution ou l'exécutable, puis il commence à enregistrer tous les cas de tests en faisant des captures d'écran tout en considérant l'écran comme image. Après avoir fini le processus d'enregistrement, il simule toutes les actions enregistrées, comme le montre la figure 3.5. Il ajoute chaque action générée dans la table d'actions, et à la fin, il génère les résultats du test sous forme de rapport.

3.1.5 MONKEYTALK

MonkeyTalk (Anjum et al., 2017) est un autre un outil de test automatisé gratuit et simple à utiliser. Il permet de faire des tests fonctionnels pour les applications mobiles Android et iOS. Il fonctionne sur des appareils réels ainsi que sur des émulateurs, en utilisant deux composantes. La première composante est un outil de l'environnement Eclipse pour enregistrer,

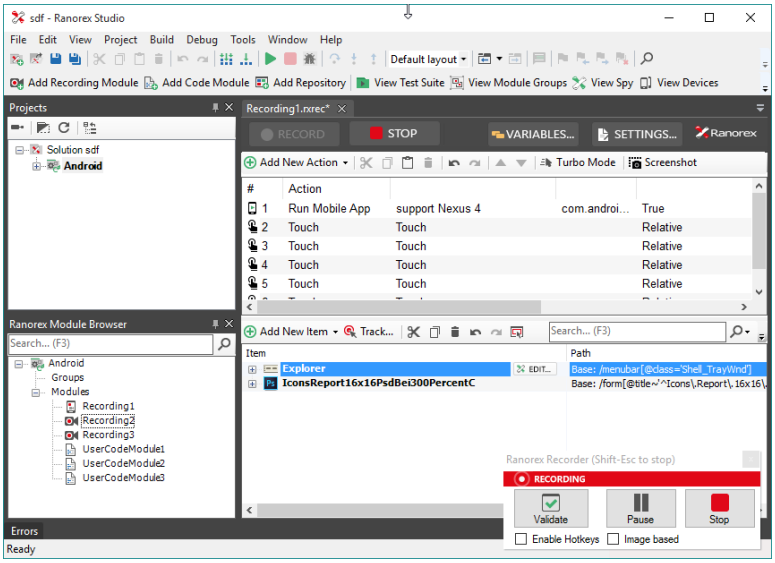

Figure 3.5 – La table d'actions de Ranorex pendant le processus d'enregistrement

démarrer, modifier et gérer les cas de tests, alors que la deuxième composante est un ensemble de librairies qui pointent sur l'application qu'on veut tester pour exécuter les cas de test. La figure 3.6 montre un exemple de la mise à jour de la table d'actions pour chaque cas de test pendant les interactions de l'utilisateur avec le GUI de l'application.

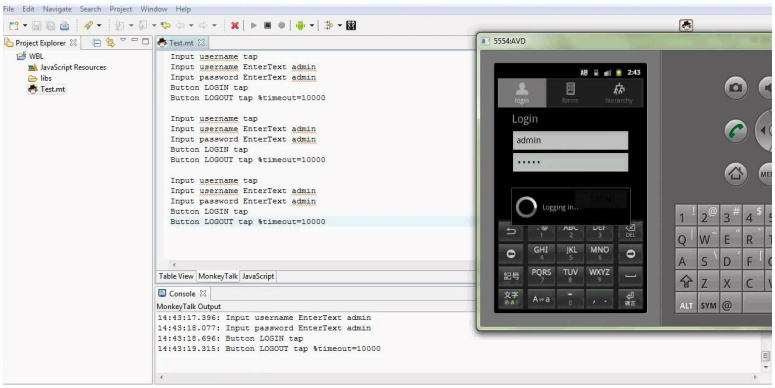

Figure 3.6 – Exemple d'un mise à jour de la table d'actions pendant l'enregistrement

3.1.6 AUTOVALX

Un autre outil d'automatisation Autovalx similaire à MonkeyTalk et Ranorex a été présenté dans l'article intitulé *Automation Framework for testing Android Mobiles* (Anbunathan et Basu, 2014) pour automatiser les tests GUI dans les applications Android. La figure 3.7 montre que cet outil commence par enregistrer toutes les entrées telles que les frappes au clavier et les événements de souris dans le script avec un autre outil de test automatisé(MonkeyTalk, qu'on a présenté dans la section 3.1.5). Il prend des captures d'écran du système avec un outil de capture appelé DDMS (Dalvik Debug Monitor Service). Le DDMS est un outil de débogage utilisé sur la plate-forme Android. Le DDMS est téléchargé depuis le framework du Android SDK. Parmi les services fournis par le DDMS, on compte la redirection de port, la capture d'écran sur le périphérique et les émulateurs, et la surveillance des threads. Lorsque l'automatisation des scripts du cas de test est exécutée, MonkeyTalk lit les entrées enregistrées, reprend les images d'écran en comparant la dernière capture stockée avec l'image actuelle, et à la fin, affiche la différence entre les images en montrant les zones modifiées dans la conception de l'interface utilisateur.

Les outils présentés dans cette sous-section utilisent des méthodes différentes pour faire les tests fonctionnels de l'interface utilisateur graphique à l'aide de deux méthodes, soit la comparaison et la reconnaissance d'image. L'outil Sikuli se base sur la méthode de la reconnaissance d'image, alors que les outils Ranorex, MonkeyTalk, BackstopJS, Wraith et Autovalx se basent sur la méthode de la comparaison d'image. Tous ces outils examinent l'interface graphique en analysant la fonctionnalité, et la façon dont les utilisateurs peuvent interagir avec l'application. Par contre, ils ne prennent pas en compte la disposition visuelle correcte des éléments. Ces outils ne peuvent évaluer que les éléments statiques de l'interface utilisateur graphique, car chaque changement dynamique est considéré comme un échec,

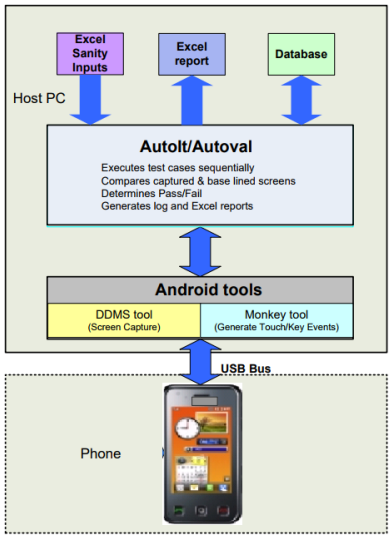

Figure 3.7 – Processus de fonctionnement d'Autovalx

les outils ne peuvent pas relier les composants GUI au code source car ils n'ont aucune information sur les id. En outre, la saisie dans un champ de texte se change dynamiquement en raison de l'entrée de l'utilisateur, ce qui rend difficile d'identifier l'élément recherché. De plus, elles se basent sur une approche manuelle pour prendre des captures d'écrans au moment de l'enregistrement. Ces captures peuvent avoir une présentation visuelle incorrecte parce que le testeur humain ne peut pas identifier ces bugs avec l'œil.

3.2 ANALYSE DE CODE SOURCE

L'analyse de code source est une approche qui permet de faire un test automatisé dans le but de déboguer un programme ou une application informatique avant sa distribution ou sa publication (Itkonen et al., 2007). Le code source est généralement écrit dans un langage de programmation, permettant une meilleure compréhension par des humains. Une fois le code source écrit, il permet de générer une représentation binaire d'une séquence d'instructions sous forme d'un code binaire ou un fichier exécutable par un micro-processeur.

L'analyse de code source peut être divisée en deux types (Margaret Rouse, 2010) : une analyse *statique* et une analyse *dynamique*. L'analyse statique teste l'application dans un environnement non exécutable. Généralement, elle examine le code source de l'application, elle compare la vérification de la syntaxe de code et tous les autres éléments qui ne comprennent pas l'exécution du code de programme. En revanche, l'analyse dynamique vérifie l'application dans un environnement exécutable. Elle examine l'application en cours d'exécution, selon une technique qu'on l'appelle « la boîte blanche » ou « la boîte noire ».

Le test de la boîte blanche est une technique d'analyse structurale, logique du code source de l'application ; et elle nécessite l'accès au code source, contrairement au test de la boîte noire qui n'a pas besoin d'avoir accès au code source. Par exemple si on veut automatiser le test de l'addition présenté au tableau 2.1 avec cette approche, on doit employer l'analyse dynamique, parce que toutes les étapes pour faire l'addition nécessitent que l'application soit en cours d'exécution. De plus pour analyser les éléments du GUI afin de manipuler les éléments de l'interface graphique, il faut savoir comment ces éléments sont eux-mêmes structurées dans les systèmes d'exploitation mobiles.

L'approche de l'analyse de code source est utilisée par plusieurs outils pour tester l'interface graphique pour les applications mobiles en analysant la structure hiérarchique des éléments d'interface graphique. On mentionne dans ce qui suit les plus importants.

3.2.1 WATIJ

Watij (Watij Prahnava, 2018) ou Web Application Testing est un projet de test, implémenté en Java, afin d'automatiser les tests fonctionnels des applications Web. Il est basé sur un outil open source Watir de bibliothèques Ruby (Watir, 2018). Il utilise les expressions XPath pour trouver les éléments HTML dans une page Web, les accéder, ainsi les contrôler facilement à

travers le DOM. Watij permet dans certains cas de connaitre le statut de chargement d'une page web comme si la page est chargée où en cours. Watij fournit un API riche qui permet de créer facilement des cas de tests. Le code 3.1 donne un exemple sur l'utilisation de son API.

Code source 3.1 – Exemple de code Watij (Christian Baranowski, 2018)

```
1
2       @Test
3         public void testSearchWikipedia() throws Exception {
4
5                 spec.open("http://de.wikipedia.org/wiki/Wikipedia:Hauptseite");
6                 spec.find.input().with.id("searchInput").set.value("Softwaretest");
7                 spec.find.button().with.id("searchButton").click();
8                 assertEquals("Softwaretest", spec.find.h1().with.id("firstHeading").get.innerText());
9                 spec.find.a().with.innerText("Qualite").click();
10                assertEquals("SoftwareQualite", spec.find.h1().with.id("firstHeading").get.innerText());
11
12        }
13
14
15        @Test
16        public void testOpenGoogle() throws Exception {
17
18                spec.open("http://www.google.de/");
19                spec.jquery("input[name='q']").set.value("Testing");
20                spec.find.input().with.name("btnG").click();
21                assertTrue(spec.find.div().with.id("res").find.a().get.innerText().startsWith("Software␣
                      testing"));
22
23      }
```

3.2.2 CAPYBARA

Capybara (Robbins, 2013) est un framework de test d'acceptation pour les applications web. Il permet aux développeurs les tests d'acceptation, ou d'intégration en simulant les actions des utilisateurs sur une page Web et de faire des assertions en fonction du contenu de la page. Il fournit également un API pour interagir avec les elements de la page Web. Par défaut, il s'exécutera en mode headless en utilisant Rack : :Test, il peut également utiliser plusieurs autres pilotes, tels que PhantomJS, pour tester les pages web avec JavaScript. Il utilise aussi d'autres drivers tels que selenium-webdriver, capybara-webkit afin d'exécuter des actions comme genre : clics de boutons, de liens, etc. Voici un exemple 3.2 qui montre la façon dont le test d'enregistrement des utilisateurs est effectué à l'aide de Capybara. Le test est pour vérifier si l'utilisateur peut continuer le processus d'inscription, ou s'il a des problèmes avec ca. S'il possède les informations d'identification requises, il sera enregistré puis redirigé vers la page

"Welcome".

Code source 3.2 – Exemple de code Capybara (Wikipedia, 2018)

```
describe 'UserRegistration' do
    it 'allows a user to register' do
      visit new_user_registration_path
      fill_in 'First name', :with => 'New'
      fill_in 'Last name', :with => 'User'
      fill_in 'Email', :with => 'newuser@example.com'
      fill_in 'Password', :with => 'userpassword'
      fill_in 'Password Confirmation', :with => 'userpassword'
      click_button 'Register'
      page.should have_content 'Welcome'
    end
  end
```

3.2.3 UIAUTOMATOR

UIAutomator (Anjum et al., 2017) est un framework de test d'Android intégré dans l'environnement de développement fourni par Google. Cependant, ce dernier a arrêté de donner le support à cet outil. C'est un outil simple à utiliser et prend moins de temps à mettre en œuvre. Il fonctionne uniquement à partir de la version Android 4.1. UIAutomator repose sur l'analyse de code source tout en identifiant les éléments de l'interface graphique afin de simuler l'interaction de l'utilisateur sur l'application ciblée en exécutant les actions possibles. La figure 3.3 montre comment l'outil démarre une application et cherche les boutons concernés pour exécuter un test de connexion.

Code source 3.3 – Exemple de code pour UIautomator

```
public void testInscription throws Exception {
// Démarrer l'application:
getUiDevice().wakeUp(); // se situer sur l'écran d'Accueil
getUiDevice().pressHome(); // appuyez sur le bouton Accueil
new UiObject(new UiSelector().description("Facebook")).click(); // Selectioner Facebook et cliquer
new UiObject(new UiSelector().text("Connexion")).click(); // Localiser 'Inscription' et cliquer
UiObject signInButton = new UiObject(signIn);
// vérifier si le bouton existe puis
// insérer le nom d'utilisateur et mot passe
if (signInButton.exists()) {
signInButton.click();
new UiObject(new
UiSelector().className("android.widget.EditText").instance(0)).setText("username");
new UiObject(new
UiSelector().className("android.widget.EditText").instance(1)).setText("password");
new UiObject(new UiSelector().className("android.widget.Button").
text("Sign In").instance(0)).click();
getUiDevice().waitForWindowUpdate(null, 2000);
getUiDevice().waitForWindowUpdate(null, 30000);
}
```

3.2.4 ESPRESSO

Un autre outil gratuit aussi développé par Google s'appelle Espresso (Coppola et al., 2016). Il fournit une interface pour la création des tests automatisés du GUI. Cet outil applique l'approche de la « boîte grise », qui consistent à examiner à la fois le fonctionnement d'une application et sa structure interne, et les fonctionnalités de l'application. Il identifie directement les références des éléments GUI et les ressources textuelles, parce qu'il est intégré dans l'environnement de développement Android. Cet outil a la possibilité de connaître le temps de chargement de l'application. Ainsi, son API a des méthodes pour simuler les boutons physiques. On peut voir dans le code source 3.4 un exemple d'utilisation de l'API d'Espresso.

Code source 3.4 – Exemple de code Espresso

```
public void testEspresso() {
//  récupérer le bouton avec la classe id for 'sign_in'
// cliquer sur le Button
 onView(withId(getInstrumentation().getTargetContext().getResources()
 .getIdentifier("com.twitter.android:id/sign_in", null, null))).perform(click());
 // récupérer le Button avec la class id login_username
 onView(withId(getInstrumentation().getTargetContext().getResources()
 .getIdentifier("com.twitter.android:id/login_username", null, null))).perform((typeText("username")));
// récupérer le Button avec la class id login_password

 onView(withId(getInstrumentation().getTargetContext().getResources()
 .getIdentifier("com.twitter.android:id/login_password", null, null))).perform((typeText("password")));
// récupérer le Button avec la class id login_password
//cliquer sur le Botton pour se connecter
 onView(withId(getInstrumentation().getTargetContext().getResources()
 .getIdentifier("com.twitter.android:id/login_login", null, null))).perform(click());
}
```

3.2.5 SELENDROID

Selendroid (Coppola et al., 2016) tire profit d'un framework déjà existant : Selenium, à l'origine conçu pour les tests d'application web. Les fonctionnalités proposées sont légèrement plus complètes qu'en ce qui concerne Espresso et UIAutomator. Selendroid n'a pas besoin de modifier le code binaire de l'application pour l'automatiser, parce qu'il utilise des plugins intégrés dans les navigateurs. Son API permet d'analyser le code et retourner les éléments de l'interface pour automatiser les tests. Le code source 3.5 montre un exemple d'utilisation de

cette API.

Code source 3.5 – Exemple de code Selendroid

```
public void selendroidTest() throws Exception {
// Trouver le champ de texte de saisie
WebElement inputField = driver.findElement(By.id("edtText"));
// Entrer un texte dans le champ de texte
inputField.sendKeys("Bonjour Liflab");
WebElement button = driver.findElement(By.id("btnShow"));
button.click();
Thread.sleep(1000);
WebElement txtView = driver.findElement(By.id("txtView"));
String expected = txtView.getText();
// Vérifier que le texte saisit dans le champ de texte est identique au texte retourné
Assert.assertEquals(expected, inputField.getText());
 }
```

3.2.6 APPIUM

Appium (Anjum et al., 2017) est un outil open source multiplateforme qui permet d'écrire des cas de test sur plusieurs plates-formes en utilisant la même API. Appium repose sur l'analyse du code en utilisant l'interface WebDriver pour accéder aux éléments de l'interface graphique, comme le montre la figure 3.6. Lorsqu'un script de test est exécuté, Appuim envoie la commande à UIAutomator ou à Selendroid pour tester les commandes afin d'exécuter l'action sur Android, ce qui permet par exemple de désinstaller ou d'installer une application automatiquement.

Code source 3.6 – Exemple de code Appium

```
public void testEspresso() {
 textFields = driver.find_elements_by_tag_name('textField')
 assertEqual(textFields[0].get_attribute("value"), "Hello")
 # cliquer sur le bouton sign-in
 driver.find_elements_by_name('Sign in')[0].click()
 # trouver les champs de texte pour username and password
 textFields = driver.find_elements_by_tag_name('textField')
 textFields[0].send_keys("twitter_username")
 textFields[1].send_keys("passw0rd")
 # cliquer sur le Button connexion pour se connecter
 driver.find_elements_by_tag_name('button')[0].click()
```

Tous ces outils (Espresso, Appuim, Selendroid,Capybara et Watij) se basent sur la méthode d'analyse de code pour analyser les éléments de l'interface graphique. Ils analysent la fonctionnalité de l'application, y compris la façon dont les utilisateurs peuvent interagir avec elle.

Cependant, ces outils ne prennent pas en compte la présentation visuelle correcte, et aucun de ces outils n'a exploré la vérification des spécifications de l'interface graphique en temps réel.

3.3 CONCLUSION

Cet aperçu de la recherche montre que la majorité des outils présentés fonctionnent avec la génération de cas de test automatisés pour imiter ou simuler l'interaction de l'utilisateur avec l'application. Le tableau 3.1 donne un récapitulatif de ces outils en fonction des types d'applications ciblés et des techniques de test. La plupart des techniques existantes peuvent être divisées en approches qui examinent la structure de l'application en trouvant tous les éléments interactifs et leurs connexions, et d'autres qui capturent l'écran pour trouver les composants GUI requis.

Cependant, ces approches pour l'évaluation de l'interface graphique analysent la fonctionnalité, c'est-à-dire la façon avec laquelle les utilisateurs peuvent interagir avec l'application. Aucune d'entre elles ne prend en compte la conception visuelle correcte, la facilité d'utilisation de l'interface ou la disposition des éléments. En outre, à ce jour aucun outil pour le test automatisé de la conception de l'interface n'est connu. Or, on a vu que les applications de qualité exigent non seulement une fonctionnalité correcte, mais elles doivent également être visuellement attrayantes. La convivialité de l'interface utilisateur graphique dépend de plusieurs propriétés de conception et règles directives proposées par les éditeurs de systèmes d'exploitation ; c'est pourquoi le développement d'un outil de test automatisé pour vérifier en temps réel les lignes directrices de l'interface graphique d'une application s'avère d'une grande importance.

Tableau 3.1 – Tableau recaputilatif des outils presentés dans ce chapitre

Outils de test	Web	Mobile	Desktop	Tests par capture d'ecrans	Analyse de code source
BackstopJS	-			-	
Wraith	-				
Sikuli	-	-	-	-	
Ranorex	-	-	-	-	
MonkeyTalk		-		-	
Autovalx		-		-	
WATIJ	-				-
Capybara	-				-
UIAutomator		-			-
Espresso		-			-
Selendroid		-			-
Appuim	-	-	-		-

CHAPITRE 4

VÉRIFICATION DES SPÉCIFICATIONS GUI EN TEMPS RÉEL AVEC CORNIDROID

De nombreux écosystèmes d'application ont défini des ensembles de contraintes sur les possibilités d'utilisation des widgets d'interface utilisateur, tels que les boutons, les curseurs et les listes. Malheureusement, un grand nombre d'applications ne suivent pas ces spécifications, ce qui entraîne une interaction de moins bonne qualité avec l'utilisateur et des incohérences dans le comportement des composants entre les applications du même système d'exploitation.

Dans l'état actuel des choses, le respect des spécifications de l'interface utilisateur décrites dans la section précédente est laissé au développeur de l'application. Ainsi, le guide de qualité Android Core UX-B1 exige qu'une application soit testée pour la conformité de l'interface utilisateur (Meniar et al., 2017) ; cependant, l'écosystème Android ne fournit aucun outil automatisé permettant de vérifier que les contraintes suggérées sont effectivement suivies par des applications développées par des tiers et publiées en ligne sur Google Play Store.

Pour résoudre ce problème, nous avons développé Cornidroid, un système de vérification d'exécution conçu spécifiquement pour la vérification des contraintes basées sur l'interface dans les applications Android. Cornidroid est une version modifiée et étendue d'un outil antérieur appelé Cornipickle, qui a été utilisé pour le test des applications web (Hallé et al.,

2016).

Dans ce chapitre, nous décrivons une méthodologie pour instrumenter une application Android d'une manière générique, et nous montrons comment un moniteur peut détecter automatiquement les violations des normes de l'interface utilisateur à chaque fois que l'état de l'interface change. À cette fin, les normes sont écrites dans un langage de spécification de haut niveau basé sur la Logique Temporelle Linéaire du premier ordre dont la syntaxe sera présentée plus loin.

Ces concepts ont été implémentés dans un nouvel outil de vérification en temps réel appelé Cornidroid.

Ce travail comprend quatre chapitres. Le premier chapitre donne une idée générale sur les spécifications de l'interface graphique ainsi l'objectif de ce mémoire. Le deuxième chapitre est consacré aux systèmes d'exploitation, les tests GUI et les spécifications de l'interface graphique pour les applications mobiles. Le troisième chapitre est dédié aux travaux connexes concernant les outils de test d'interfaces dans les applications mobiles. Le quatrième chapitre présente la vérification des spécifications en temps réel, l'outil Cornidroid, la syntaxe pour exprimer les propriétés, et la manière dont cet outil analyse l'état des éléments de l'interface graphique.

Ce Chapitre comprend quatre sections, on decrit :

— En 4.1, le langage de Cornidroid basé sur le langage Cornipickle (un outil existant).

— En 4.2, le développement d'une sonde pour Android.

— En 4.3, la codification des guidelines d'Android en expressions Cornidroid.

— En 4.4, l'expérimentation sur des applications.

4.1 SPÉCIFICATION DU LANGAGE

Cornidroid est organisé selon une architecture client-serveur. Le code côté serveur prend en charge la collecte et l'évaluation des spécifications. À cette fin, les utilisateurs peuvent écrire des contraintes sur le contenu d'une interface utilisateur Android en utilisant un langage déclaratif basé sur une extension de premier ordre de la Logique Temporelle Linéaire appelée LTL-FO$^+$ (Hallé et al., 2016). La grammaire du langage est conçue pour utiliser de nombreux mots et constructions anglaises simples, résultant en un ensemble de contraintes relativement lisibles.

De plus, les propriétés peuvent être accompagnées de métadonnées de type Javadoc ; ces éléments peuvent être utilisés pour afficher des informations utiles au développeur lorsque la propriété correspondante est violée. Dans un cas d'utilisation typique, un développeur lance d'abord l'interpréteur Cornidroid, et l'alimente avec un ensemble d'instructions déclaratives représentant les contraintes de l'interface utilisateur à tester. Cet ensemble d'instructions est stocké en mémoire et reçoit un identifiant unique.

Grammaire

Comme déjà mentionné, Cornidroid est une version modifiée et étendue d'un outil antérieur appelé Cornipickle. Pareillement à Cornipickle, les propriétés sont exprimées sous forme d'assertions sur le contenu et les attributs d'une *capture* d'une fenêtre prise à un instant donné. La manière précise par laquelle ces captures sont prises à partir d'une application web donnée sera détaillée dans la suite.

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

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

Énoncés

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

Logique du premier ordre

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

Expressions temporelles

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

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

Sélection d'éléments

Les *éléments* de la fenêtre sont l'unité principale dans Cornidroid, ils sont sélectionnés pour exprimer les propriétés. Ces propriétés peuvent s'appliquer à tous les éléments sélectionnés, ou au moins un. Par conséquent, la sélection de l'élément se fait via la quantification de premier ordre classique, en utilisant l'anglais pour la syntaxe comme : `For each $x in S` ou `There exists $x in S`(pour dire chaque `$x` dans `S` ou Il existe `$x` dans `S`). Dans ces expressions, `S` désigne soit un sélecteur des éléments ou un autre ensemble précédemment défini par l'utilisateur.

Un sélecteur d'élément est une expression qui permet de récupérer les éléments de l'interface graphique. Ces expressions sont définies par une grammaire régulière. Les éléments de l'interface graphique peuvent être sélectionnés soit par la valeur de leurs attributs *tag* et *id*, ou alors de leurs *instances objets* en utilisant la syntaxe de jQuery `$(...)`. Si `$x` est une variable quantifiée en utilisant le mécanisme décrit ci-dessus, on peut accéder aux hiérarchies des fenêtres pour chaque activité, et aux attributs de cet élément en utilisant `$x's property` où `property` est l'attribut ou le champ de l'élément ciblé. Par exemple, pour mentionner la largeur d'un élément, on écrit : `$x's height`.

Les attributs des éléments sont soit des chaînes de caractères ou de chiffres. Ils peuvent alors être comparés en utilisant des mots clés comme `is greater than` ou `equals`. La comparaison d'expressions régulières est faite à travers le prédicat `match`, et l'inclusion de chaîne est affirmée par l'assertion `contains`. Des assertions de base sur les éléments peuvent également être combinées en utilisant des connecteurs booléens classiques : `and`, `or`, `if...then`, `not`.

Opérateurs

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

Constructions auxiliaires

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

Sélecteur des élements

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

Attributs

⟨*elements-attribute*⟩ : := | value | height | width | top | left | right | bottom | color | id | text | background
| border | event | cornipickleid | display | size | accesskey | checked | disabled | min | menuItemText
| length | bgcolor | parent | widthdp | heightdb | position | name ;

Propriétés des éléments

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

Expressions régulières

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

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

Événements et opérateurs temporels

Dans Cornidroid, les événements déclenchés par l'utilisateur tels que les touches et les clics sont illustrés comme des attributs de l'élément en question. Par exemple, un élément qui a été touché ou appuyé par l'utilisateur possède un attribut `event` avec une valeur `touch`. Ainsi, pour affirmer qu'un élément `$x` a été touché, on peut écrire `$x's event is 'touch'`.

De plus, Cornidroid fournit des opérateurs prise de la logique temporelle Linéaire (LTL) pour exprimer des assertions sur le contenu d'un document, grâce à l'inclusion d'événements dans le langage Cornidroid, ce qui a conduit naturellement à la notion de captures instantanées. Par exemple, l'utilisation du mot-clé `Always` φ nous permet de faire l'assertion suivante : l'expression φ elle doit être vraie (`true`) dans tous les états de l'interface. On peut remplacer φ par n'importe quelle expression valide du langage. De même, on utilise `Eventually` φ pour dire que φ sera vraie dans au moins un futur état de l'interface. Finalement, `Next` φ est utilisé pour dire que φ est vrai dans le prochain état de l'interface.

Une autre expression spéciale, appelée, `The next time` φ `then` ψ affirme que ψ doit être vraie, mais seulement une fois que φ devient vraie. Par exemple, on peut utiliser cette fonction pour exprimer que quelque chose doit être observé après qu'un élément ait été cliqué. Cependant l'assertion reste en attente jusqu'à l'instant où la condition φ est vraie, ce qui en fait une légère réécriture de l'opérateur *until* de la logique LTL.

Le but particulier des opérateurs temporels est d'évaluer l'état du même élément de l'interface en plusieurs moments. Cela peut être fait dans Cornidroid avec la construction `When $x is now $y` φ. Si `$x` se réfère à l'état d'un élément capturé dans un état antérieur, par conséquent `$y` va contenir l'état du même élément dans la capture actuelle.

Ensemble défini en extension

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

Prédicats définis par l'utilisateur

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

Énoncés définis par l'utilisateur

⟨*userdef-stmt*⟩ : := empty

Tableau 4.3 – Extensions de la grammaire BNF pour Cornipickle

Tous ces mots clés peuvent être librement mélangés. Par exemple, la propriété ci-dessous exprime que chaque élément d'une liste (`li`) se déplacera vers le bas de la fenêtre, à un certain moment :

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

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

Les prédicats peuvent être définis avec la construction `We say that...when`. Le texte entre `that` et `when` est interprété comme une chaîne de caractères qui peut contenir des variables. Puis, le texte après `when` décrit comment cette expression doit être évaluée en termes du

vocabulaire existant. Par exemple, on peut définir l'expression « left-aligned » comme suit :

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

Par la suite, l'expression `$x and $y are left-aligned` peut être réutilisée avec différents noms de variables dans des assertions ultérieures. Les utilisateurs peuvent également définir des ensembles : soit des ensembles de chaînes de caractères, ou des ensembles d'éléments à partir d'une fenêtre. Pour ce faire, on les énumère en utilisant la phrase `A...is any of` :

```
A Ensemble is any of "1,2,3"
```

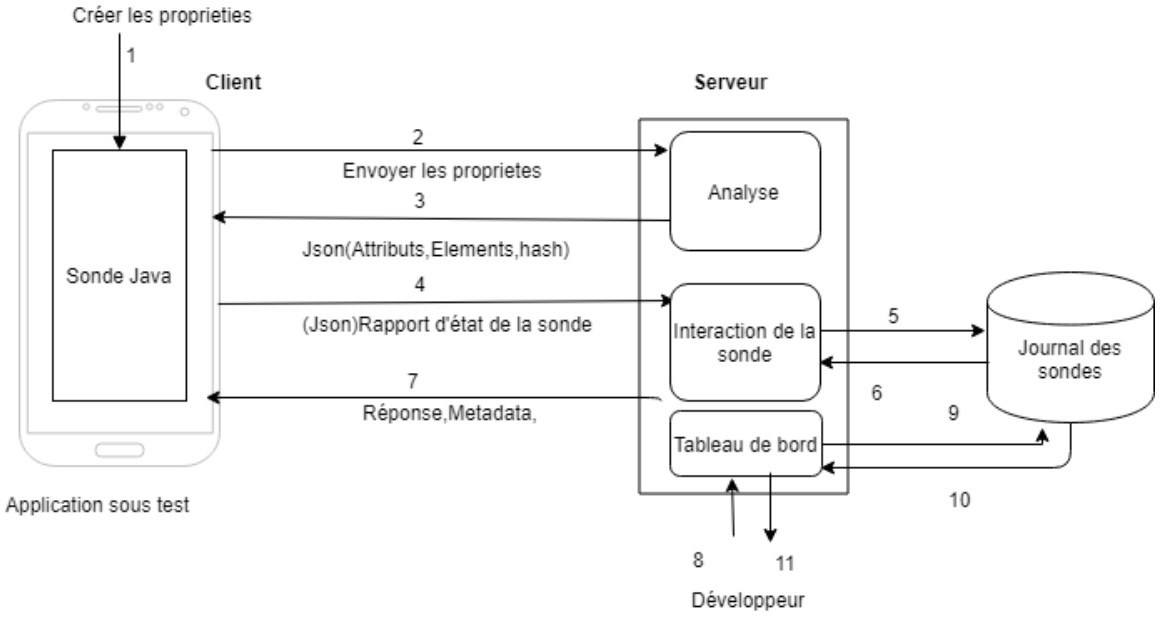

Figure 4.1 – Architecture de Cornidroid

4.2 SONDE CÔTÉ CLIENT

Cornidroid dispose d'une architecture client-serveur comme le montre la figure 4.1. La figure montre l'interaction entre la sonde et l'interpréteur. Les étapes (1-2-3) seront présentées dans les sections 4.2.1 et 4.2.2, les étapes (3-4-5-6) seront expliquées dans la section 4.2.3, et l'étape (7) sera détaillées dans la section 4.2.4 et concernant les étapes (8-9-10-11), elles montrent que le côté serveur prend en charge la collecte, l'évaluation des spécifications, et l'afficher sous forme des résultats aux développeurs via un tableau de bord.

La tâche du code côté client (la « sonde ») est d'interroger les informations sur l'état actuel de la fenêtre et le relayer au serveur pour un traitement ultérieur. La sonde est un objet unique prenant en charge l'analyse des éléments de l'interface graphique de l'application à surveiller. Elle est adaptée à la manière dont les éléments de l'interface utilisateur sont créés dans un système d'exploitation Android.

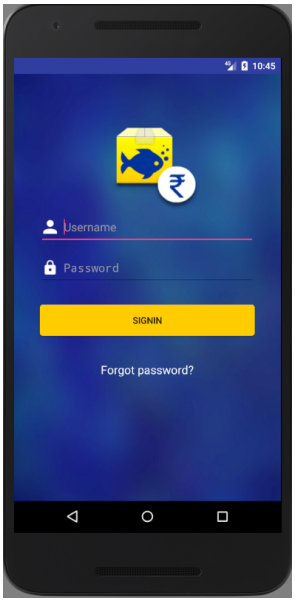

Figure 4.2 – L'interface de l'application

Dans cette section, on va présenter une simple application de connexion (« Login ») [1] en vérifiant si elle respecte les spécifications mentionnées dans la section en haut 2.3, tous les éléments affichés sur lesquels on peut cliquer, appuyer, ou avec lesquels il est possible d'interagir, doivent être assez grands pour permettre des interactions fiables.

4.2.1 LA MISE EN PLACE

Cette application dispose d'un écran de connexion où l'utilisateur peut se connecter. L'interface contient des objets layout, des labels et des boutons, comme le montre la figure 4.2.

Pour insérer la sonde dans l'application, on suit les étapes suivantes :

1. Ouvrir Android Studio, naviguer vers l'option du projet, sélectionner l'option Android et ajouter le package `ca.liflab.sonde` dans le répertoire `java` de l'application, comme

1. `https://github.com/androidcss/android-material-design-user-interface-login-screen/`

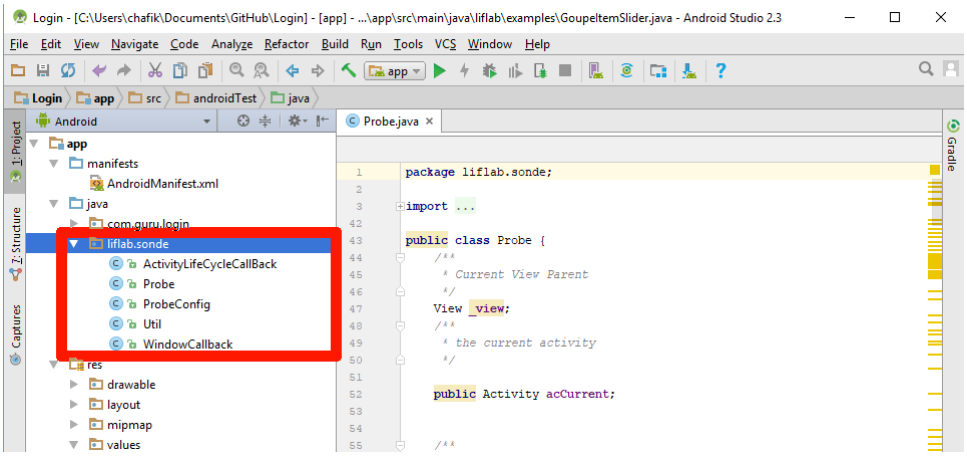

Figure 4.3 – L'ajout de la sonde

le montre la figure 4.3

2. Sélectionner le dossier de l'application. Faire un clic droit sur le dossier de l'application ; aller à Nouveau / Dossier / Dossier Assets. Android Studio ouvre alors une boîte de dialogue. Choisir l'option `main` et cliquer sur Terminer. Une fois cette opération complétée, le répertoire `assets` apparaît dans le projet. Il faut alors ajouter les propriétés créées avec le langage Cornidroid, en donnant à ces dernières le même nom que les activités d'Android voir (la figure 4.4). De plus, il faut ajouter le fichier `config.properties` en renseignant le nom (*hostname*) du serveur et le nom de l'application.

4.2.2 GESTION DES PROPRIÉTÉS

Dans l'univers Android, les activités sont les objets les plus utilisés. Chaque fenêtre ou écran de l'application est implémenté par une classe qui hérite de la classe Activity. Une seule activité est lancée au démarrage d'une application. Chaque activité, pour qu'elle soit lancée, doit être déclarée dans un fichier nommé `Manifest`, dans une balise enfant `<activity>` de la balise `<application>`, comme le montre le code source 4.1.

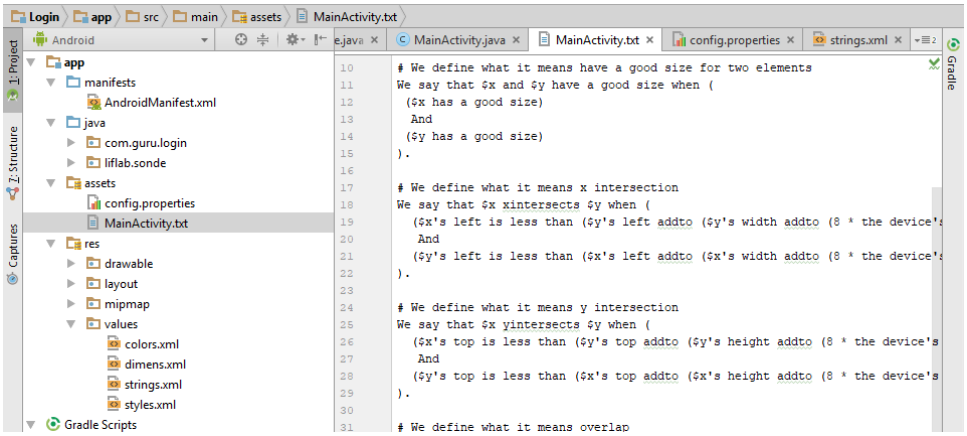

Figure 4.4 – Le dossier `assets`

Code source 4.1 – Fichier Manifest

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
package="com.guru.com" >
<application
android:allowBackup="true"
android:icon="@mipmap/ic_launcher"
android:label="@string/app_name"
android:roundIcon="@mipmap/ic_launcher_round"
android:supportsRtl="true"
android:theme="@style/AppTheme" >
<activity android:name="com.guru.login.MainActivity" >
<intent-filter>
<action android:name="android.intent.action.MAIN" />

<category android:name="android.intent.category.LAUNCHER" />
</intent-filter>
</activity>
</application>
</manifest>
```

Pour la spécification de notre exemple, selon le nom de l'activité, on nomme le fichier avec le nom `activity_main`. Cette propriété vérifie si les éléments du layout s'intersectent, sont proches l'un de l'autre de 8 dp sur les deux axes x et y. Elle vérifie également, pour chaque élément, s'il a une taille d'au moins 48×48 de densité (dp), soit d'environ 9 mm. On obtient les définitions suivantes :

```
# We define what it means have a good size for an element
We say that $x has a good size when (
    (($x's width / the device's device-density) is greater than 47)
     And
    (($x's height / the device's device-density) is greater than 47)
```

```
).

# We define what it means have a good size for two elements
We say that $x and $y have a good size when (
 ($x has a good size)
  And
 ($y has a good size)
).

# We define what it means x intersection
We say that $x xintersects $y when (
  ($x's left is less than
      ($y's left + ($y's width +
        (8 * the device's device-density))))
   And
   ($y's left is less than ($x's left +
        ($x's width + (8 * the device's device-density))))
).

# We define what it means y intersection
We say that $x yintersects $y when (
  ($x's top is less than
        ($y's top + ($y's height +
          (8 * the device's device-density))))
   And
  ($y's top is less than
         ($x's top + ($x's height +
          (8 * the device's device-density))))
).

# We define what it means overlap
We say that $x and $y overlap when (
  ($x xintersects $y)
   And
  ($x yintersects $y)
).

# We define the negation to simplify the grammar of overlap
We say that $x and $y don't overlap when (
  Not(
    $x and $y overlap
  )
```

```
).

"""
  @name Touch Target Size
  @description To ensure balanced information density and usability...
  @link https://material.io/guidelines/layout/metrics-keylines.html
  @severity Error
"""
There exists  $x in $(.tag) such that (
  For each $y in $(.tag) (
   ($x's id equals $y's id)
    Or
   (($x and $y don't overlap)
    And
   ($y and $x have a good size))
  )
).
```

Selon le nom de l'activité chargée, la sonde s'occupe d'envoyer au serveur la propriété dont le nom correspond au nom de l'activité ; le serveur stocke et lit (*parsing*) l'ensemble des déclarations des propriétés dans la mémoire du serveur. La sonde reçoit du serveur des données sous format JSON, comme le montre la figure 4.5. Ces données contiennent les éléments de l'interface graphique et les attributs à récupérer.

4.2.3 ANALYSER LES ÉLÉMENTS GUI

Selon le JSON illustré dans la figure 4.5, la sonde analyse le contenu des éléments de l'interface et envoie l'état instantané des éléments de l'interface utilisateur pertinents lors de chaque événement déclenché par l'utilisateur. Comme on l'a vu plus tôt, le contenu des éléments de l'interface Android peut être décrit avec du code Java ou dans un document XML distinct. L'extrait de code 4.2 explique mieux la figure 4.2.

Code source 4.2 – XML du GUI

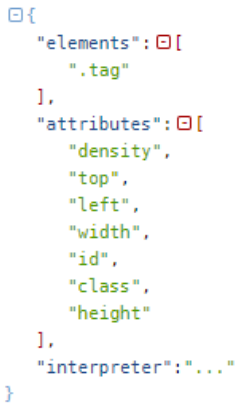

Figure 4.5 – Exemple du contenu JSON pour les éléments et les attributs de GUI d'une propriété

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
android:layout_width="match_parent"
android:layout_height="match_parent"
android:orientation="vertical"
android:background="@mipmap/back_login"
android:padding="40dp">

<EditText
android:id="@+id/username"
android:layout_width="match_parent"
android:layout_height="wrap_content"
android:layout_marginTop="170dp"
android:drawableLeft="@mipmap/account"
android:drawablePadding="5dp"
android:ems="10"
android:hint="Username"
android:inputType="textEmailAddress"
android:paddingLeft="0dp"
android:tag="tag"
android:textColor="#FFF"
android:textColorHint="#999" />

<EditText
android:id="@+id/password"
android:layout_width="match_parent"
android:layout_height="wrap_content"
android:layout_marginTop="10dp"
android:drawableLeft="@mipmap/lock"
android:drawablePadding="5dp"
android:ems="10"
android:hint="Password"
android:inputType="textPassword"
android:paddingLeft="0dp"
android:tag="tag"
android:textColor="#FFF"
android:textColorHint="#999" />

<Button
android:id="@+id/button"
android:layout_width="match_parent"
android:layout_height="wrap_content"
android:layout_gravity="center_horizontal"
android:layout_marginTop="30dp"
android:background="@drawable/selector_xml_btn_yellow"
android:onClick="checkLogin"
android:tag="tag"
android:text="SignIn" />

<TextView
android:id="@+id/textView3"
android:layout_width="wrap_content"
android:layout_height="wrap_content"
android:layout_gravity="center_horizontal"
android:layout_marginTop="40dp"
android:tag="tag"
android:text="Forgot␣password?"
```

```
57 android:textAppearance="?android:attr/textAppearanceMedium"
58 android:textColor="#FFF" />
59
60 </LinearLayout>
```

Un fichier XML est cependant très lourd à parcourir, donc construire un arbre de vues prend du temps et des ressources. De plus, on ne peut pas récupérer les éléments créés dynamiquement pendant l'exécution. Ceci survient, par exemple, dans le cas d'un objet `Layout` qui affiche une liste où le nombre des éléments change en fonction de nombre des utilisateurs stockés dans la base de données.

Sur Android, il existe plusieurs façons d'intercepter les événements à partir de l'interaction d'un utilisateur avec l'application. Lorsqu'on examine des événements dans l'interface utilisateur, l'approche consiste à capturer les événements de l'objet View spécifique avec lequel l'utilisateur interagit. La classe View fournit les fonctions pour le faire. Dans les différentes classes View, il y a plusieurs méthodes de rappel (callbacks) publiques qui semblent utiles pour les événements de l'interface utilisateur. Ces méthodes sont appelées par le framework Android lorsque l'action correspondante se produit sur cet objet. Par exemple, lorsqu'une vue telle qu'un bouton est cliqué, la méthode onTouchEvent est appelée sur cet objet. Toutefois, afin d'intercepter cela, on doit étendre la classe et remplacer la méthode. Cependant, étendre chaque objet View afin de gérer un tel événement dans chaque activité ne serait pas pratique et dynamique pour chaque activité chargée c'est pourquoi on a défini une classe avec des callbacks pour capturer l'interaction de l'utilisateur avec l'interface utilisateur, comme suit :

Code source 4.3 – Callbacks

```
1  public class WindowCallback extends ProbeConfig implements Window.Callback {
2  Window.Callback localCallback;
3
4  public WindowCallback(Window.Callback localCallback, Activity a) {
5    nameFile = a.getClass().getSimpleName()+".txt";
6    this.a = a;
7    this.localCallback = localCallback;
8  }
9
10 @Override
11 public boolean dispatchKeyEvent(KeyEvent event) {
12   sendActivityUiToServer(null);
13   return localCallback.dispatchKeyEvent(event);
14 }
15
```

```
@Override
public boolean dispatchKeyShortcutEvent(KeyEvent event) {
  return localCallback.dispatchKeyEvent(event);
}
// ...
}
```

De plus, les connexions entre les différentes activités sont définies dans le code Java, donc pour pouvoir récupérer toutes les informations sur les éléments de l'interface utilisateur à l'exécution, il est important d'analyser le code source en entier.

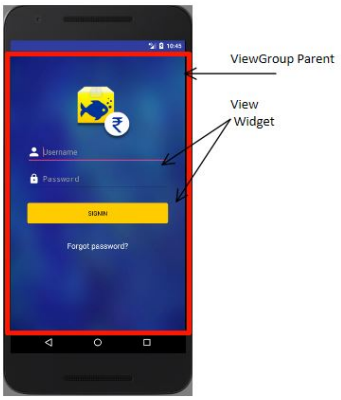

Figure 4.6 – Les types d'objets pour l'interface graphique

Visuellement, tous les objets `Layout` sont composés d'une hiérarchie de vues, comme le montre la figure 2.2. Chacune de ces vues hérite des classes `View` ou `ViewGroup`. Un `ViewGroup` est simplement un conteneur qui peut contenir d'autres `ViewGroups` ou `Views`. `View`, quant à elle, est la classe ancêtre de chaque widget Android. Par exemple, dans la figure 4.6, `bouton` "signin" est un descendant de `View`.

Lorsqu'un événement d'interface utilisateur est détecté dans l'application, la sonde parcourt récursivement tous les nœuds parents et enfants de l'objet `View` supérieur et elle recherche les nœuds d'un certain type, comme le montre l'extrait du code source 4.4. En général, la sonde ne récupère que les informations nécessaires à l'évaluation des spécifications fournies par

l'utilisateur ; en effet, elle reçoit des instructions sur les widgets de l'application mentionnés dans une propriété, et les renvoie au serveur sous format JSON, comme le montre la figure 4.7.

Code source 4.4 – Analyser les éléments de GUI

```
Public void analyseViews(ViewGroup v,JSONArray jsArrayChildren, MotionEventevent) {
// ...
Final int childCount=v.getChildCount();
// ...
try {

// ...

if (canIncludeThisView(jNode,v)) {
  addAttributeIfDefined(jNode,v,event);
  // ...
  } else {
  for(inti=0; i<childCount; i++) {
    View child=v.getChildAt(i);
    JSONObject jNodeChild=new JSONObject();
    if ((child instanceof ViewGroup) && child.getTag()!=hashTagResult) {
      jNodeChild.put("children",jArrayChild);
      analyseViews((ViewGroup)child,level+1,jArrayChild,event);
    } else {
    // ...
    if (canIncludeThisView(jNodeChild,child)) {
      addAttributeIfDefined(jNodeChild,child,event);
    }
  // ...
  }
  // ...
}
```

4.2.4 GESTION DE LA RÉPONSE

Lors de son utilisation, la sonde affiche une petite icône dans le bas-côté droit de la fenêtre, cette icône peut être soit verte dans le cas où la propriété a été respectée, ou bien rouge si la propriété a été violée comme le montre la figure 4.8.

Il est également possible d'analyser systématiquement une assertion quand cette dernière est évaluée à faux, et identifier les éléments qui sont responsables du non-respect de la règle en les entourant par des rectangles rouges. Cela fournit davantage de feedback au testeur qu'un simple verdict oui/non. À cet effet, la réponse reçue de l'interpréteur Cornicpikle comprend plusieurs informations telles que le verdict, les identifiants des éléments violés, etc. La figure 4.9 donne un extrait des informations fournies par l'interpréteur et renvoyées à la sonde dans l'application Android. Le code source , quant à lui, montre comment cette réponse de

```
{
  "element":"window",
  "aspect-ratio":1.6611111164093018,
  "orientation":"portrait",
  "width":1080,
  "height":1794,
  "device-width":1080,
  "device-height":1794,
  "url":"",
  "device-aspect-ratio":1.6611111164093018,
  "device-density":2.625,
  "device-langue":"en_US",
  "children":[
    {
      "children":[
        {
          "children":[
            {…},
            {…},
            {
              "element":"Button",
              "tag":"tag",
              "cornipickleid":2,
              "id":"button",
              "width":870,
              "height":126,
              "left":105,
              "top":1000
            },
            {
              "element":"TextView",
              "tag":"tag",
              "cornipickleid":3,
              "id":"textView3",
              "width":376,
              "height":63,
              "left":352,
              "top":1231
            }
          ]
        }
      ]
    }
  ]
}
```


Figure 4.7 – Les propriétés des éléments de l'interface utilisateur sont sérialisées par l'objet sonde dans un document JSON envoyé à l'interpréteur.

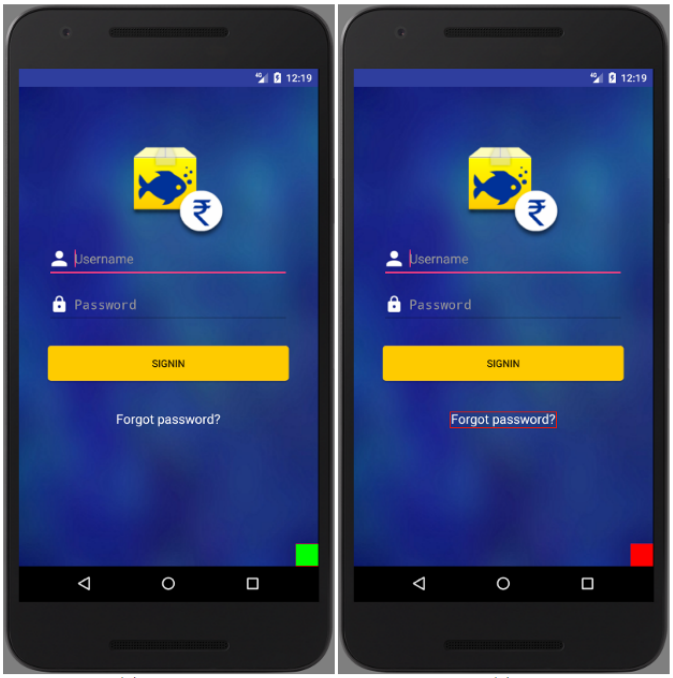

Figure 4.8 – Une capture d'écran de Cornidroid en action

l'interpréteur est traitée dans l'application Android.

Code source 4.5 – Traitement de la reponse

```
JSONObject jsonObj = new JSONObject(result.toString());
String global-verdict= jsonObj.getString("global-verdict");
JSONArray hightlight = jsonObj.getJSONArray("highlight-ids");
if (global-verdict.toLowerCase().equals("true") || global-verdict.toLowerCase().equals("inconclusive"))
  btn.setBackgroundColor(Color.GREEN);
else
  btn.setBackgroundColor(Color.RED);
for(int j=0; j < _hightlight.length(); j++) {
  JSONObject set_of_tuples=(JSONObject)_hightlight.get(j);
  JSONArray jsIds=set_of_tuples.getJSONArray("ids");
  final String jlink=set_of_tuples.getString("link");
});

for (int z=0; z < js.length(); z++) {
  JSONArray ids=jsIds.getJSONArray(0);
  for(int h=0; h < ids.length(); h++) {
    int key = ids.getInt(h);
    if(idMap.containsKey(key)){
      View v = acCurrent.findViewById(idMap.get(key).id);
    if(v != null)
      Util.customViewBorder(v,Color.RED);
    }
  }
}
```

Dans l'exemple illustré dans la figure 4.8, l'application affiche deux images A et B. L'image A montre que la règle est violée puisque l'icône est rouge, et le dernier élément en bas a

```
{"num-false":1,
"highlight-ids":
[{"link":"https://material.io/guidelines\...",
"ids":[[3,0] ],
"caption":"To ensure balanced information density and usability, ... "}],
"global-verdict":"FALSE",
"num-inconclusive":0,
"interpreter":...}
```


Figure 4.9 – Un extrait des informations fournies par l'interpréteur et renvoyées à la sonde dans l'application Android.

été identifié comme un élément fautif, parce que l'attribut de hauteur a été renseigné par la valeur "wrap_content", ce qui a entraîné que sa hauteur est inférieure à 48 dp. Cependant dans l'image B, la propriété a été respectée parce qu'on a modifié la valeur de `height`, et de cette manière on s'assure que l'utilisateur peut cliquer sans aucun problème sur l'élément en question.

4.2.5 COMMUNICATION AVEC LE SERVEUR

Le serveur reçoit le document JSON et met à jour en temps réel le verdict de chaque propriété surveillée, en incluant aussi les métadonnées de la propriété comme lien vers la règle et le niveau de l'erreur. Les informations sur l'état actuel des propriétés sont retransmises à la sonde, qui, comme on l'a vu, interagit avec l'interface utilisateur de l'application pour fournir un retour visuel à l'utilisateur.

Même si la sonde et le serveur sont des entités séparées, dans un appareil Android, les deux tournent à l'intérieur de la même machine. Le code d'application s'exécute dans le thread principal, et chaque instruction est donc exécutée dans une séquence. Si nous devons effectuer de longues tâches / opérations, le thread principal est bloqué jusqu'à la fin de l'opération correspondante. Pour fournir une bonne expérience utilisateur, nous utilisons donc la classe

`AsyncTasks`, qui s'exécute dans un thread séparé. Cette classe va tout exécuter dans la méthode `doInBackground`, à l'intérieur d'un autre thread qui n'a pas accès à l'interface graphique où toutes les vues sont présentes. La méthode `onPostExecute` de cette classe se synchronise à nouveau avec le thread principal de l'interface utilisateur et lui permet d'effectuer des mises à jour. Cette méthode est appelée automatiquement après que la méthode `doInBackground` ait terminé son travail.

4.3 LA CODIFICATION DES GUIDELINES D'ANDROID

Dans cette section, nous allons présenter plusieurs exemples en expressions Cornidroid, pour vérifier les spécifications présentées dans la section 2.3.

Menu en bas 2.3

```
"""
  @name Bottom Navigation
  @description  The Bottom Navigation can accommodate between 3 and 5 destinations.
  @link https://material.io/guidelines/...
  @severity Warning
"""
For each $x in $(BottomNavigationMenuView) (
    ($x's size is greater than  2)
     And
    ($x's size is less than  5)
).
```

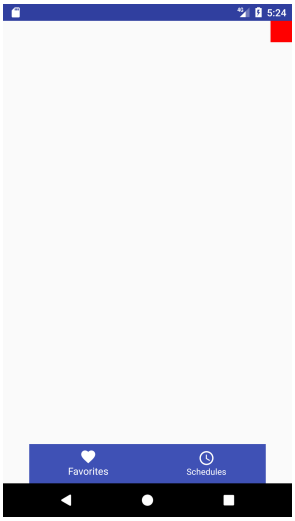

Figure 4.10 – Menu en Bas

Les boutons surélevés 2.3

```
# Define what it means the raised Button and not a flat button
We say that $x is RaisedButton when (
  ( (($x's width / the device's device-density) is greater than 47 ) And
     (($x's height / the device's device-density) is greater than 47)
   ) And
  ( Not ($x's background matches "RGB\(0,0,0\)")
   )

).

"""
  @name Using raised buttons
  @description It is better to use raised buttons instead of
               flat buttons when there are many details
  @link https://material.io/guidelines/components/buttons.html...
  @severity Warning
"""
  For each $z in $(Button) (
  $z is RaisedButton
  ).
```

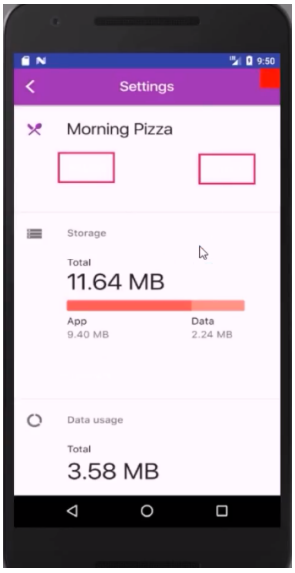

Figure 4.11 – Les boutons surélevés

ListView (Liste)

```
"""
 @name ListView sort
 @description The list can be sorted according to its needs
 @link https://material.io/guidelines/components/lists.html...
 @severity Warning
"""

For each $x in $(ListView TextView) (
   For each $y in $(ListView TextView) (
       If ($y's top is greater than $x's top)
       Then ($y's text is greater than $x's text)
   )
).
```

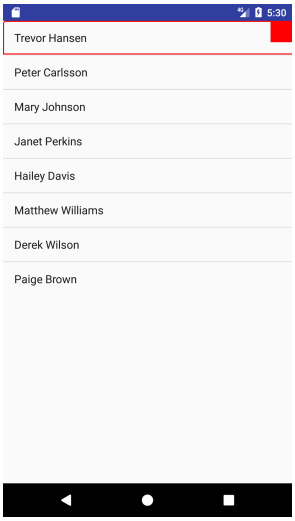

Figure 4.12 – ListView

Bouton flottant 2.3

```
"""
  @name Floating action button
  @description  the floating action button should not be hidden
  @link https://material.io/guidelines/components/...
  @severity Error
"""
Always (
     For each $x in $(#fab)  (
         For each $y in $(TextView)(
             If ($y's parent equals "SnackbarLayout")
             Then (($y's top  - $x's top) is greater than $x's height)
         )
     )
).
```

Mise en miroir 2.3

```
We say that $x has a numbers in the correct direction when(
 If($x's id equals "txtPhone")
```

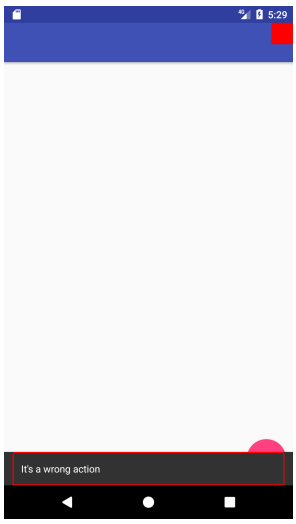

Figure 4.13 – Bouton flottant

```
 Then ($x's text equals $y's text)
).

# We define what it means right-to-left(rtl)
We say that $x is rtl when(
  Always (
    When $x is now $y (
    If (the device's device-langue equals "ar")
    Then (
         ($x's left equals $y's right) And (
          $x has a numbers in the correct direction
        )
     )
    )
  )
).

"""
  @name UI Mirroring
  @description When mirroring the layout, padding...
  @link https://material.io/guidelines/usability/...
  @severity Error
"""
For each $x in $(LinearLayout TextView)(
  $x is rtl
```

).

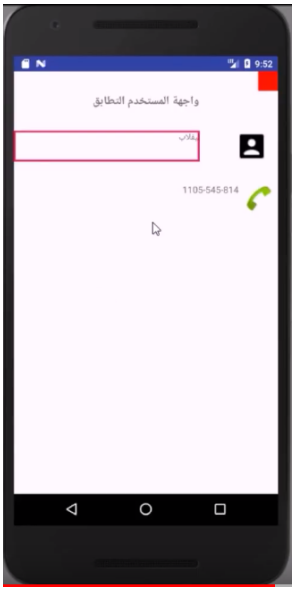

Figure 4.14 – La mise en miroir

Regroupement des items 2.3 .

```
"""
  @name Grouping items
  @description All TextViews are aligned next to the sliders...
  @link https://material.io/guidelines/usability/...
  @severity Warning
"""
For each $z in $(RelativeLayout SeekBar) (
    There exists $x in $(RelativeLayout TextView) such that (
        $z's top equals $x's top
    )
).
```

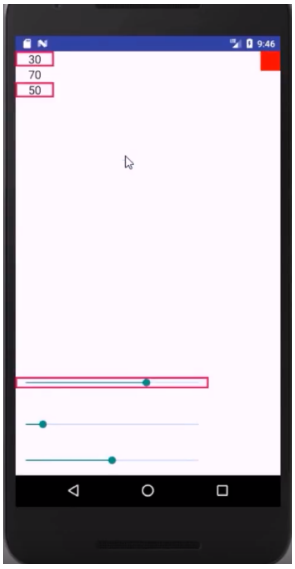

Figure 4.15 – Le regroupement des éléments

Taille du champ de saisie 2.3

```
# We define what it means to be a redline
We say that $x and $y are redline when (
  ($x's length is greater than 10) And
  ($y's color matches "RGB\(255,0,0\)")
).

#We define what it means to be a blueline
We say that $x and $y are blueline when (
 ( $x's length is less than 11)
 And ($y's color matches "RGB\(0,0,255\)")
).


"""
  @name Text field input
  @description Reporting by a color if the number of ...
  @link https://material.io/guidelines/patterns/errors...
  @severity Warning
"""
```

```
For each $z in $(#txtEdit1) (
For each $t in $(#colorLineCounter1) (
    ($z and $t are blueline)  Or
    ($z and $t are redline)

)
).
```

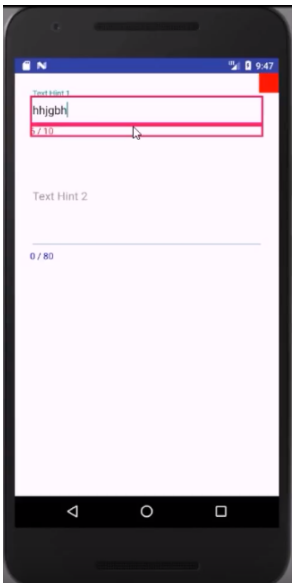

Figure 4.16 – Taille du champ de saisie

Comportement du menu 2.3

```
# attribute "event". Its value is the event's type

We say that I click on Spinner when (
  There exists $b in $(Spinner) such that (
    $b's event is "ACTION_UP"
  )
).

# We define what it means occurrence

We say that $x and $y are not equals when (
```

```
  Not(
        $x’s text equals $y’s text
    )
).


"""
  @name Menu behavior
  @description Do not display a duplicate of the  menu item
  @link https://material.io/guidelines/components/menus...
  @severity Warning
"""

If (I click on Spinner)
Then (
For each $x in $(Spinner item) (
    For each $y in $(Spinner item) (
    ($x’s position equals $y’s position) Or
       ( $x and $y are not equals)
    )
 )
).
```

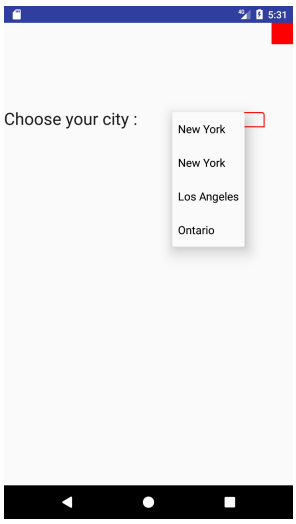

Figure 4.17 – Comportement du Menu

Pour plus d’informations et des exemples, le lecteur est référé à un dépôt de code contenant

plusieurs normes d'interface Android codifiées, ainsi que tout le code de la sonde Android et du serveur [2]. On peut également voir une vidéo de Cornidroid en action sur la chaîne YouTube [3].

4.4 EXPÉRIMENTATION

Le but de l'expérience est de mesurer le temps que la sonde prenait à construire le JSON, d'avoir une idée approximative du temps que la sonde prend en fonction de nombre des éléments de l'interface graphique, ainsi de mesurer le temps pour évaluer les propriétés en fonction de la taille du document JSON. Le plus important est de tester différents cas et différents attributs pour simuler autant de scénarios possibles d'utilisation que possible. On a choisi pour l'expérience, une machine avec un processeur Intel Core i7, simulateur mobile Nexus 5 API 25, et le CPU est un Google Play Intel Atom (x86).

```
For each $x in $(LinearLayout Button) (
  For each $y in $(LinearLayout Button) (
      If ($y's top is greater than $x's top)
      Then ($y's text is greater than $x's text)
  )
).
```

On a créé une application qui contient de 0 à 3000 éléments combinés avec des layouts, boutons et des textes. La propriété mentionnée ci-dessus utilisé indique à la sonde de construire un JSON seulement avec les éléments Button qui se trouve à l'intérieur d'un LinearLayout.

Selon les résultats affichés dans la figure 4.18, on remarque que le temps pour récupérer l'état de l'interface et pour construire le JSON, il s'augmente progressivement en fonction

2. https://bitbucket.org/chafdev/cornipickle-sonde-mobile
3. https://www.youtube.com/watch?v=YNxxV8hIIzY

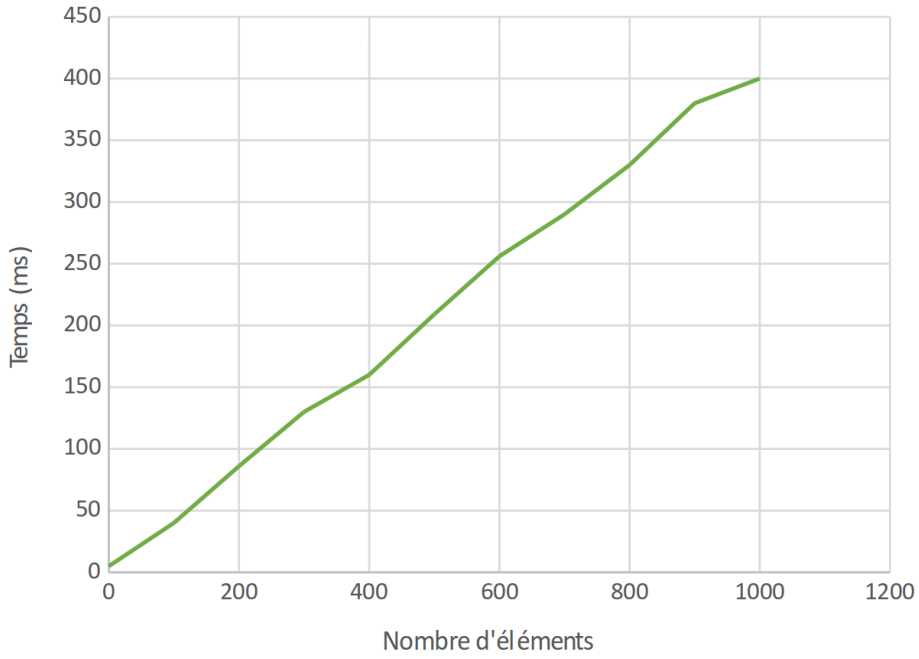

Figure 4.18 – Le temps que prend la sonde pour récupérer l'état de l'interface

de nombre d'éléments de l'interface, sous forme d'une fonction linéaire. Quant au temps de l'évaluation de la propriété présenté dans la figure 4.19, ce temps augmente d'une façon exponentielle dans le cas où la propriété évaluée est vraie. Par contre, si le résultat est faux, il peut dans certains cas de trouver une condition fausse au début de l'évaluation du document JSON, ce qui rend la propriété entière fausse sans nécessiter d'évaluer le reste du document.

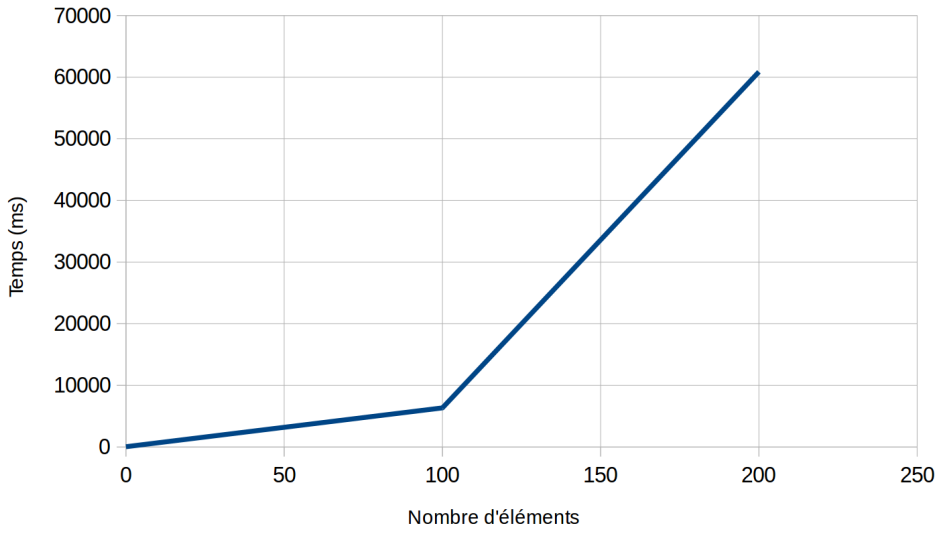

Figure 4.19 – Le temps que prend l'interpreteur pour evaluer JSON

CONCLUSION

L'émergence des applications mobiles dans tous les domaines rend l'utilisation de ces applications très intéressant ce qui augmente la demande pour les applications mobiles d'une façon exponentielle. Pour que ces applications soient choisies parmi les milliards des applications existantes, il faut concevoir un bon produit, et pour cette raison les éditeurs de systèmes d'exploitation mobiles ont établi des spécifications à vérifier concernant l'interface graphique.

Ces spécifications sont destinées à garantir la qualité et la cohérence des applications afin d'avoir un produit bien conçu, accessible aux utilisateurs de toutes aptitudes, y compris ceux qui ont une déficience visuelle, une cécité, une déficience auditive, une déficience cognitive ou une déficience motrice.

Plusieurs outils de tests automatisés pour les applications mobiles ont été présentés dans ce mémoire, qui analysent l'interface graphique afin d'évaluer la fonctionnalité et la sécurité des applications. La plupart des techniques existantes sont divisées en approches qui examinent la structure de l'application en trouvant tous les éléments interactifs et leurs connexions, et d'autres qui capturent l'écran pour trouver les composants GUI requis. À ce jour, aucun outil pour le test automatisé pour la vérification des spécifications en temps réel de l'interface graphique n'est connu.

Dans ce mémoire, nous avons présenté un nouvel outil appelé Cornidroid, qui propose une approche automatisée pour détecter et signaler les violations des spécifications de l'interface utilisateur Android –ceci afin de réduire le temps d'essai et d'augmenter la productivité de la production de l'application. Cornidroid nous permet d'attraper automatiquement de nombreux problèmes communs rencontrés dans de vraies applications Android. Cette application est aussi utile dans le cas de test de régression : lorsqu'une erreur se produit et ne correspond à aucune spécification existante, le développeur peut écrire une règle et l'utiliser pour s'assurer que cette erreur ne réapparaisse jamais dans les futures modifications apportées à l'application. Nous avons obtenu des résultats préliminaires prometteurs, qui devraient être confirmés en envisageant des études de cas supplémentaires. Évidemment, un plus grand nombre de contraintes devrait être codifié avec le langage de Cornidroid, et ces contraintes devraient également être classées selon le niveau de gravité de l'erreur.

La version actuelle de Cornidroid permet de vérifier les spécifications Android concernant les layouts, les polices de caractères, les tailles, les couleurs, les alignements, les images, les boutons. La prochaine version sera focalisée sur la gestion du son. En effet, il existe certaines applications qui donnent aux utilisateurs l'option de naviguer dans en utilisant le son, en ajoutant des étiquettes descriptives aux éléments de l'interface utilisateur. Lorsque on utilise un lecteur d'écran tel que TalkBack et que l'on navigue par moyen tactile, les labels sont prononcés à haute voix lorsque les utilisateurs touchent les éléments de l'interface utilisateur.

De plus, certaines sous-vues peuvent servir de conteneurs pour regrouper les autres éléments à l'intérieur et ne sont pas visibles à l'écran. Cependant, il est nécessaire d'évaluer uniquement les objets visibles. La combinaison de l'analyse du code source et de la reconnaissance d'image est en général une bonne approche pour les travaux futurs. Il peut apporter différents avantages. La reconnaissance d'image de la capture d'écran prend en compte la représentation réelle des éléments réellement visibles dans la vue en cours. Ainsi, la technique de reconnaissance

d'image peut être utilisée pour trouver tous les composants de l'interface utilisateur à l'écran et définir les éléments de l'objet testé. Après cela, l'outil peut rechercher ces objets dans le code source en définissant les coordonnées à l'écran ou en comparant les captures d'écran de chaque élément et effectuer toutes les opérations nécessaires avec eux. Cette technique peut apporter des résultats plus précis, en particulier pour les éléments dynamiques.

Comme étape future, nous prévoyons également trouver une approche appropriée par laquelle des suggestions qui seront données aux développeurs permettraient de corriger l'interface graphique. Un nouvel algorithme pourrait être créé, qui permettrait de récupérer les éléments qui portent intérêt pour une spécification donnée et générer des moyens de modifier ces éléments de manière à satisfaire la propriété Cornidroid.

BIBLIOGRAPHIE

Anbunathan, R. et A. Basu. 2014. « Automation framework for testing android mobiles », *International Journal of Computer Applications*, vol. 106, no. 1.

Android. 2017. User interface guidelines. `https://developer.android.com/guide/practices/ui_guidelines/index.html`, Retrieved May 7th, 2017.

Anjum, H., M. I. Babar, M. Jehanzeb, M. Khan, S. Chaudhry, S. Sultana, Z. Shahid, F. Zeshan, et S. N. Bhatti. 2017. « A comparative analysis of quality assurance of mobile applications using automated testing tools », *INTERNATIONAL JOURNAL OF ADVANCED COMPUTER SCIENCE AND APPLICATIONS*, vol. 8, no. 7, p. 249–255.

Apple. 2016. Ui kit. `https://developer.apple.com/reference/uikit`, Retrieved December 1th, 2017.

———. 2017. Apple iOS human interface guidelines. `https://developer.apple.com/ios/human-interface-guidelines/`, Retrieved May 7th, 2017.

Christian Baranowski. 2018. Simple demo of a watij webspec. `https://gist.github.com/tux2323/954186`, Retrieved July 2018.

Coppola, R., E. Raffero, et M. Torchiano. 2016. « Automated mobile ui test fragility : An exploratory assessment study on android ». In *Proceedings of the 2Nd International Workshop on User Interface Test Automation*. Coll. « INTUITEST 2016 », p. 11–20, New York, NY, USA. ACM.

Elin, H. 2017. Designing a visual regression testing tool : Decrease fear-driven development and enhance the quality assurance.

Google. 2016. Ui overview. `https://developer.android.com/guide/topics/ui/overview.html`, Retrieved December 1th, 2017.

GÖTH, R. 2015. « Testing techniques for mobile device applications ». Thèse de Doctorat, Masarykova univerzita, Fakulta informatiky.

GURU. 2010. Mobile testing. `https://www.guru99.com/mobile-testing.html`, Retrieved December 1th, 2017.

Hallé, S., N. Bergeron, F. Guérin, G. Le Breton, et O. Beroual. 2016. « Declarative layout constraints for testing web applications », *J. Log. Algebr. Meth. Program.*, vol. 85, no. 5, p. 737–758.

Itkonen, J., M. V. Mantyla, et C. Lassenius. 2007. « Defect detection efficiency : Test case based vs. exploratory testing ». In *Empirical Software Engineering and Measurement, 2007. ESEM 2007. First International Symposium on*, p. 61–70. IEEE.

Margaret Rouse. 2010. source code analysis. `http://searchsoftwarequality.techtarget.com/definition/source-code-analysis`, Retrieved December 1th, 2017.

Meniar, C., F. Opalvens, et S. Hallé. 2017. « Runtime verification of user interface guidelines

in mobile devices ». In *International Conference on Runtime Verification*, p. 410–415. Springer.

Microsoft Corporation. 1992. *The Windows Interface : An Application Design Guide*. Microsoft Press.

Muccini, H., A. Di Francesco, et P. Esposito. 2012. « Software testing of mobile applications : Challenges and future research directions ». In *Proceedings of the 7th International Workshop on Automation of Software Test*, p. 29–35. IEEE Press.

Robbins, M. 2013. *Application Testing with Capybara*. Packt Publishing Ltd.

SupInfo. 2016. Les tests automatisés. `https://www.supinfo.com/articles/single/2585-tests-automatises`, Retrieved December 1th, 2017.

The GNOME Project. 2017. GNOME human interface guidelines. `https://developer.gnome.org/hig/stable/`, Retrieved May 7th, 2017.

University Of BATH. 2017. Visual regression testing with wraith. `http://blogs.bath.ac.uk/digital/2017/04/07/visual-regression-testing-with-wraith/`, Retrieved July 2018.

Watij Prahnava. 2018. watij webspec. `http://watij.com/`, Retrieved July 2018.

Watir. 2018. Watir project. `http://watir.com/`, Retrieved July 2018.

Wikipedia. 2018. User-registration process. `https://en.wikipedia.org/wiki/Capybara_(software)`, Retrieved July 2018.

Wikipédia. 2017. Application mobile. `https://fr.wikipedia.org/wiki/Application_mobile`, Retrieved December 1th, 2017.

Zednet. 2017. Chiffres clés : les os pour smartphones. http://www.zdnet.fr/actualites/chiffres-cles-les-os-pour-smartphones-39790245.htm, Retrieved December 1th, 2017.