Detección Automática de Incompatibilidades Web utilizando Procesamiento Digital de Imágenes

Maximiliano Agustin Mascheroni[1], Mariela Katherina Cogliolo[1], Emanuel Irrazábal[1]

[1] Universidad Nacional del Nordeste. Facultad de Ciencias Exactas y Naturales y Agrimensura. Departamento de Informática, Corrientes, Argentina
{mascheroni, mcogliolo, eirrazabal}@exa.unne.edu.ar

Resumen. La compatibilidad de un sitio web es la propiedad que lo permite ser apreciado idénticamente en todos los navegadores. La creciente necesidad de que los sitios web puedan ser compatibles, ha llevado a que las empresas realicen análisis de múltiples factores en materia de visibilidad durante las etapas de desarrollo. Pero la continua evolución de los navegadores hace que este análisis no sea suficiente para evitar incompatibilidades entre ellos, cobrando relevancia las pruebas de compatibilidad web. Por otro lado, hoy en día las empresas buscan alcanzar el enfoque de desarrollo de software continuo, donde el objetivo es mantener la producción de software en ciclos cortos de tiempo, acelerando los procesos de pruebas. Uno de los desafíos que existe actualmente, es la aceleración de las pruebas sobre la interfaz de usuario, entre ellas, las de compatibilidad. En este trabajo se presenta una técnica para automatizar la detección de defectos de incompatibilidad, mediante el uso del procesamiento digital de imágenes. Los resultados de su implementación indican que se adapta a los requerimientos de los procesos de desarrollo continuo, aumentando el rendimiento y la velocidad de las pruebas y disminuyendo la cantidad de falsos positivos que normalmente aparecen en este tipo de pruebas.

1 Introducción

El Desarrollo de Software Continuo (DSC), es un conjunto de técnicas (integración continua, despliegue continuo, entrega continua y pruebas continuas) de la Ingeniería de Software, que permiten a los equipos de desarrollo producir software en ciclos muy cortos de tiempo, asegurando el lanzamiento de diferentes versiones del producto en cualquier momento [1].

Por las ventajas mencionadas, el DSC se convierte en un factor clave en las organizaciones como elemento diferenciador de la competencia y que además permite alcanzar mejoras rápida y eficientemente en las entregas a los clientes, brindándoles a los mismos un producto fiable [2]. Sin embargo, uno de los principales desafíos es mantener elevada la calidad del producto software. Al realizarse los despliegues del sistema con mayor frecuencia, aparecen más defectos en el producto [3], [4]. Esto se

debe, principalmente, a que el tiempo para realizar los ciclos de prueba es muy reducido.

Como una solución a este problema se ha intentado mantener la calidad del producto software utilizando las pruebas automatizadas. Mediante las mismas, se busca acortar los tiempos de la ejecución de las pruebas sobre las diferentes versiones del producto [5]. Así, por ejemplo, Google genera lotes de pruebas automatizadas utilizando herramientas especializadas [6], adquiriendo una mayor cobertura de pruebas antes de liberar una nueva versión del producto. Por el contrario, Mike Cohn y Anand Bagmar, ubican a las pruebas unitarias y a las de aceptación como base en las pirámides de pruebas [7], [8]; y las pruebas de interfaz gráfica (UI) se encuentran en el último lugar (ver Figura 1).

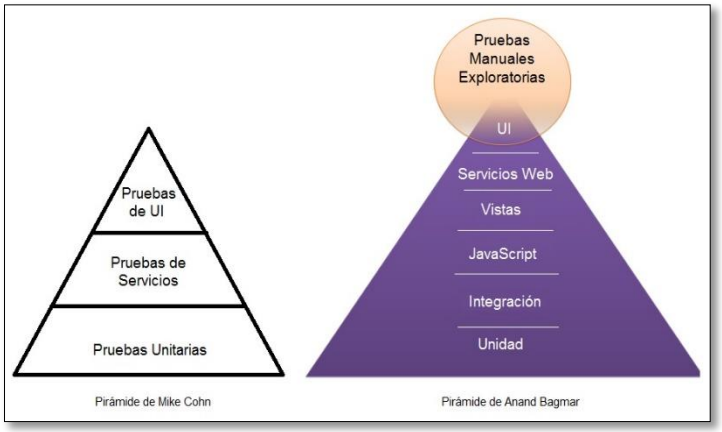

Fig. 1. Pirámide de Pruebas.

Esto lleva a que muchas organizaciones pongan su mayor esfuerzo en automatizar pruebas unitarias y especialmente, pruebas de aceptación. Así, por ejemplo, puede verse que los pequeños equipos de desarrollo de software en el nordeste argentino, sólo realizan pruebas de aceptación en forma manual [9].

Sin embargo, para otras empresas, es vital que el producto en desarrollo sea compatible en los distintos navegadores tanto en un computador, como en un dispositivo móvil. Las pruebas que se utilizan para determinar la compatibilidad de una aplicación en diferentes navegadores se denominan “pruebas de compatibilidad web” o “pruebas cruzadas entre navegadores” (más conocido en inglés como Cross-Browser Testing). En este trabajo se propone una técnica de detección de incompatibilidades utilizando el procesamiento digital de imágenes.

Además de esta sección introductoria, el trabajo está dividido en 5 secciones. En la sección 2 se brinda un breve marco teórico sobre la compatibilidad web y las pruebas que la verifican, incluyendo un breve estado del arte de las técnicas existentes. En la sección 3 se presenta el algoritmo propuesto. Los resultados de la implementación del algoritmo se muestran en la sección 4. Finalmente, en la sección 5 se perfilan las conclusiones y trabajos futuros.

2 Compatibilidad Web

Las páginas web a menudo son interpretadas de manera diferente a través de múltiples navegadores y plataformas. Cuando estas diferencias afectan al usuario final, estos problemas de interpretación son llamados incompatibilidades web. Las incompatibilidades, pueden tomar la forma de texto no visible o fuera de fase, fuentes distorsionadas o botones no renderizados. Para poder identificar este tipo de defectos se utilizan las denominadas pruebas de compatibilidad.

Que un sitio web sea compatible con todos los navegadores significa que se vea igual en todos ellos. Algunos autores consideran como suficiente si se consiguen visualizaciones similares en los navegadores más importantes, como Internet Explorer, Chrome, Opera, Safari y Mozilla [10]. Sin embargo, no siempre se logra ver todos los sitios y páginas web como lo concibieron sus autores. Esto ocurre con gran frecuencia y la necesidad de que esto no suceda ha aumentado. El problema radica en que no todos los navegadores interpretan el código HTML y las hojas de estilo (CSS) de la misma manera [11]. Algunas de esas diferencias son tan importantes que provocan el mal funcionamiento del sitio o la pérdida de visualización.

Adicionalmente, las personas navegan en Internet con diferentes sistemas operativos, navegadores, resoluciones de pantalla, complementos, velocidades de conexión, ancho de banda y recursos físicos. Todos estos factores pueden afectar a la compatibilidad de un sitio web.

La Figura 2 muestra un ejemplo de un defecto de incompatibilidad entre navegadores. En el ejemplo se aprecia una página web que tiene el menú distorsionado en el navegador Internet Explorer 9 (IE9). Esta incompatibilidad es causada por un error en la etiqueta final: en vez de utilizar la etiqueta final apropiada , el elemento se cierra con <a/>, el cual es interpretado por algunos navegadores como un nuevo elemento. En IE9, la etiqueta incorrecta final hace que el siguiente elemento a nivel de bloque sea también envuelto en una etiqueta de anclaje (<a>), lo que resulta en una disposición inadecuada. En la mayoría de los navegadores, incluyendo Chrome, la etiqueta final no se extiende en el elemento adyacente, por lo que el diseño no se ve afectado. Este tipo de control de errores demuestra un comportamiento incoherente entre los navegadores.

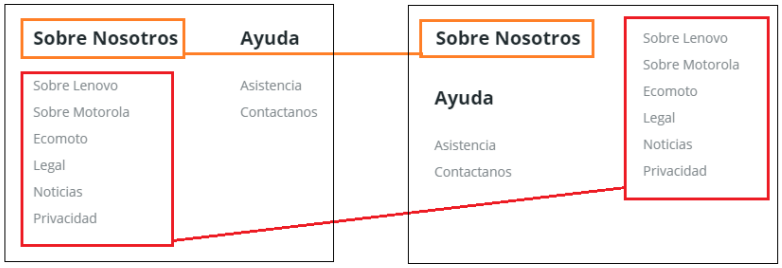

Fig. 2. Menús en Google Chrome (izquierda) e Internet Explorer 9 con un defecto (derecha).

Los factores que tienen relación con los defectos de incompatibilidad son varios y a todos se les debe prestar atención. Los más comunes, de acuerdo con [12], son: la resolución de pantalla, el uso de macros, el sistema operativo, la incorrecta implementación de hojas de estilo en cascada, la utilización de diferentes tecnologías (javascript, ajax, angular) y el uso de diferentes versiones de HTML.

2.1 Pruebas de Compatibilidad

Uno de los objetivos de la construcción de un sitio web es que pueda ser visitado por el mayor número de personas y que éstas lo vean correctamente. Por ende, es muy importante que el sitio funcione igual en el mayor número de navegadores posibles [10]. Muchas veces no se sabe a qué causa atribuir el bajo número de visitantes a un sitio web y pocas veces se piensa y se cree que esto se relaciona con la visibilidad en los diferentes navegadores [12]; con frecuencia, sólo se piensa que, si se tiene una buena interfaz de usuario, se tendrá un elevado número de usuarios satisfechos. De esta manera, es importante la planificación y ejecución de pruebas de compatibilidad.

Sin embargo, el problema de las pruebas de compatibilidad tradicionales es la coexistencia de una amplia gama de navegadores y las diferentes configuraciones de plataformas subyacentes. Además, existen diferentes sistemas operativos: Windows, Mac OS, Linux, Solaris, etc.

Las pruebas de compatibilidad manuales requieren de un esfuerzo en abrir e inspeccionar las páginas web en cada navegador, combinando las diferentes configuraciones posibles (navegador/sistema operativo). Esta tarea es laboriosa y requiere un esfuerzo ocular. Así, resulta imposible incorporar pruebas de compatibilidad manuales a entornos de DSC. Por esta razón, emergen diferentes propuestas y herramientas para hacer frente a esta problemática.

Entre las técnicas más populares se encuentran las pruebas de compatibilidad web mediante análisis del modelo de objetos del documento (DOM), como por ejemplo Cross Browser Testing [13]. Los DOMs se producen mediante el renderizado de la página en diferentes configuraciones de navegadores y sistemas operativos. Luego estos objetos DOM son comparados. Cualquier diferencia es considerada como una incompatibilidad. Sin embargo, los navegadores tienden a diferir en la manera en que cada uno genera su DOM y esto puede producir falsos positivos.

Otra alternativa es la utilización de herramientas de capturas de pantalla automáticas como BrowserStack [14]. Las mismas, reducen parte del trabajo manual mediante la automatización del proceso de apertura de la página web en cada configuración del navegador y de la toma de capturas de pantalla una vez renderizada. Esto lo hace mediante una infraestructura en la nube compuesta por una gran variedad de dispositivos móviles y de sistemas operativos (a veces virtualizados). Sin embargo, el equipo encargado de las pruebas debe observar las capturas una por una en busca de incompatibilidades. En la Figura 3 se muestra un resultado de una comparación realizada con BrowserStack.

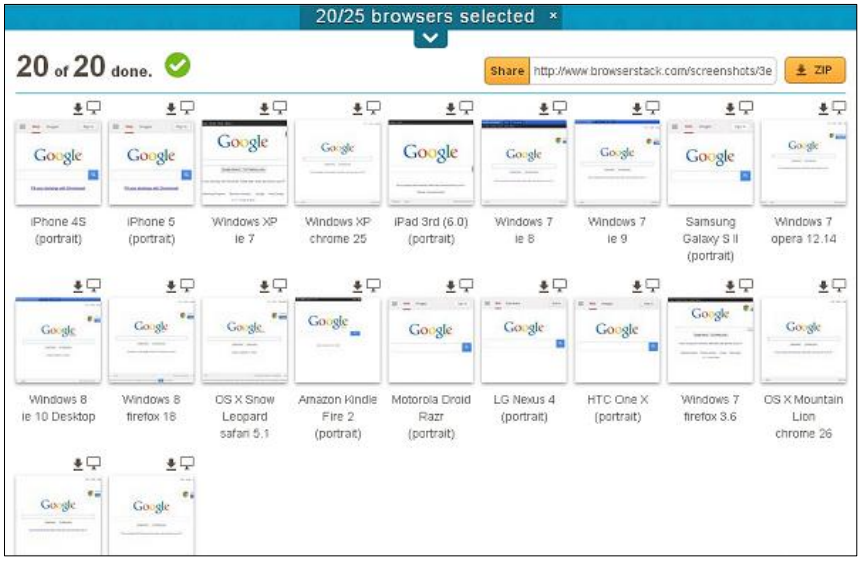

Fig 3. Prueba de compatibilidad ejecutada con BrowserStack [14]

En [15], se presenta una iniciativa para automatizar el proceso de pruebas de compatibilidad mediante un algoritmo de comparación de imágenes que puede ser integrado en un entorno de DSC. La técnica que implementa este algoritmo, se basa en tomar una captura de pantalla del sitio web bajo pruebas, y compararla (pixel por pixel) con otra captura con una configuración "correcta". Si ambas imágenes coinciden, entonces el resultado de la prueba será "éxito"; caso contrario, será "fallo". De esta manera, se determina la introducción (o no) de nuevos defectos de incompatibilidad. Además, la comparación genera una tercera imagen tipo "mapa de calor" que contiene las zonas en donde los pixeles difieren.

Si bien este enfoque resulta preciso al detectar cualquier diferencia existente entre dos capturas, produce una gran cantidad de falsos positivos, a causa de problemas mínimos de alineación o presentación de los componentes propios de cada navegador. En la Figura 4, se muestra un ejemplo de una comparación donde se aprecian los fallos, pero que no son percibidos a los ojos de los usuarios y por ende el resultado es considerado como un falso positivo.

Fig. 4. Resultado de una comparación de imágenes

En la actualidad, las propuestas y herramientas se basan en variaciones y combinaciones de los métodos mencionados anteriormente. En [16] se presenta una técnica para realizar análisis de objetos DOM mediante máquinas de estado de finito. Sin embargo, a pesar de que los resultados de la validación muestran que la propuesta es efectiva, la misma no contempla defectos causados por mala renderización de los navegadores. En [17], [18] y [19] se presentan herramientas (WebDiff y Crosscheck) que combinan el análisis de DOM con la comparación de imágenes, y los resultados de su implementación demuestran que el porcentaje de falsos positivos producidos por ellas se encuentra entre un 17% y 18%.

Finalmente, [20], [21], [22], [23], [24], [25] presentan algoritmos que utilizan técnicas del procesamiento digital de imágenes para realizar las comparaciones, obteniendo resultados más satisfactorios. Browserbite [20], por ejemplo, utiliza técnicas de segmentación para realizar comparaciones entre regiones de una determinada captura. Según los resultados que reporta Browserbite, el algoritmo reduce la tasa de falsos positivos considerablemente, pero la principal limitación de esta herramienta, es que solo puede utilizarse para realizar comparaciones de páginas en un determinado estado al que se accede solo a través de su URL.

En este trabajo, se presenta una nueva propuesta que utiliza las técnicas del procesamiento digital de imágenes similar a Browserbite, pero ampliando su alcance a cualquier sitio web, que además tenga estados a los que no pueden accederse mediante una URL.

3 Algoritmo de detección de incompatibilidades propuesto

Siguiendo con los objetivos presentados previamente en [15], se busca acelerar el proceso de pruebas en entornos de DSC, para asegurar la calidad del producto software a ser liberado en ciclos cortos de tiempo. En un ambiente de DSC, las pruebas funcionales automatizadas cumplen un papel fundamental, ya que permiten realizar regresiones de manera rápida en pequeños ciclos de desarrollo por cada cambio introducido en el producto software. Con la aparición de los servidores de integración continua, es posible detectar la introducción de defectos en el código fuente lo antes posible [26].

La técnica propuesta consiste, por tanto, en complementar el proceso de pruebas funcionales sobre un sitio web, con un algoritmo de detección de incompatibilidades entre navegadores. El algoritmo se basa en la comparación automatizada de las regiones de una imagen. Para ello, se utilizan técnicas del procesamiento digital de imágenes.

Para implementar este algoritmo, se utiliza el mismo entorno desarrollado en [15], con los siguientes componentes:

- Una herramienta para interactuar con los diferentes navegadores.
- Una herramienta para verificar el cumplimiento de condiciones (existencia de elementos, comportamientos esperados, etc.)

- Herramientas para procesar imágenes.
- El algoritmo de detección de incompatibilidades.
- Un mecanismo para ejecutar pruebas en paralelo sobre diferentes sistemas operativos y navegadores.
- Una herramienta de generación de reportes para mostrar los resultados de la ejecución de las pruebas.

El flujo de ejecución de los componentes del entorno se muestra en el Figura 5.

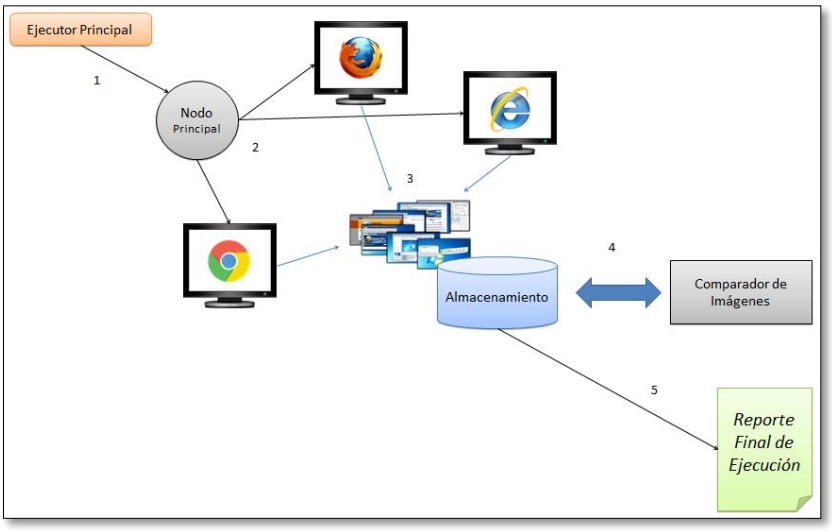

Fig. 5. Flujo de ejecución de las pruebas de compatibilidad [15]

En el flujo:

1. Las pruebas son ejecutadas a través de Maven. Esta herramienta se encarga de compilar el código y darle la orden al nodo principal para que comience su ejecución.
2. El nodo principal es un concentrador de Selenium Grid, un servidor que permite disparar hilos de ejecución de pruebas sobre múltiples nodos en paralelo. Este servidor instancia los navegadores configurados en cada nodo. Las pruebas son generadas utilizando Selenium Webdriver.
3. Durante la ejecución de las pruebas, se realizan capturas de pantalla y se almacenan en un disco duro al que tienen acceso todos los nodos de manera sincronizada. Las pruebas concluyen cuando todas las validaciones funcionales son completadas. Estas validaciones se realizan utilizando TestNG.
4. El algoritmo de detección de incompatibilidades toma las capturas de pantalla, y comienza a analizarlas en pares, obteniendo regiones,

comparándolas y generando resultados. Luego los resultados son almacenados en el mismo disco donde se encuentran las capturas.

5. Cuando finaliza el proceso, se genera el reporte final con los resultados de la ejecución de las pruebas en un archivo HTML.

3.1 Captura de pantallas

Para tomar las capturas de pantalla se utiliza el mismo proceso presentado en [15]. La interacción con el sitio web es responsabilidad de la herramienta Selenium Webdriver. El script de una prueba funcional determinada es ejecutado en distintos navegadores en paralelo, mediante una configuración de Selenium Grid que permite instanciar un hilo de ejecución por cada máquina virtual (nodo) del entorno. A medida que estas pruebas van avanzando de una página del sitio a otra, se va tomando capturas de pantalla.

El código fuente que genera las capturas de pantalla puede verse en la Figura 6 y consta de 3 pasos:

1. Se coloca el navegador en modo “pantalla completa”.
2. Se obtiene la dimensión total de la pantalla utilizando la API Toolkit de Java.
3. Se genera la captura de pantalla a través de la API Robot de Java.

```
//Paso 1
Actions actions = new Actions(navegador);
actions.sendKeys(Keys.F11).perform();

//Paso 2
Dimension dimension = Toolkit.getDefaultToolkit().getScreenSize();
Rectangle rectangle = new Rectangle(dimension);

//Paso 3
BufferedImage image = new Robot().createScreenCapture(rectangle);
ImageIO.write(image, "png", new File(path));
```

Fig. 6. Método para tomar captura de pantalla en un nodo Windows

Las imágenes tomadas son almacenadas en el disco duro de un computador, con una nomenclatura que representa: el identificador de la prueba; el nombre de la página en la que se tomó la captura; el navegador; y el nombre y versión del sistema operativo del nodo. Por ejemplo: Test1802-detalle_itinerario-chrome-win8.1.png; Test706-pagina_pago-ff_v44-ubuntu12.2.png.

La herramienta para realizar las verificaciones de los resultados esperados y condiciones (TestNG), presenta un mecanismo para ejecutar instrucciones al concluir las pruebas. Este mecanismo se define mediante las anotaciones de

@AfterSuite/@AfterTest/@AfterClass. Asimismo, permite la ejecución de instrucciones previa a la de las pruebas (utilizando los recíprocos @BeforeSuite/@BeforeTest/@BeforeClass).

Para ejecutar las pruebas de compatibilidad se utiliza la anotación @AfterSuite. Dentro del método que contiene esta anotación, se realiza la llamada al algoritmo de detección de incompatibilidades.

El primer paso en esta etapa consiste en obtener todas las imágenes correspondientes a un determinado nodo que contiene la forma correcta de visualización del sitio. Para ello, se utiliza el navegador y sistema operativo especificados en el nombre de las imágenes. Luego, se obtienen las imágenes correspondientes a un segundo nodo con el sitio bajo prueba (Figura 7). Posteriormente, las imágenes comienzan a analizarse de a pares (Figura 8).

```
List<BufferedImage> imagenesComp0 = getImagesMatchingWith("chrome45-win7");

//Primer Comparacion
List<BufferedImage> imagenesComp1 = getImagesMatchingWith("ff_v44-ubuntu12.2");
startComparison(imagenesComp0, imagenesComp1);

//Segunda Comparacion
List<BufferedImage> imagenesComp2 = getImagesMatchingWith("ff_v44-ubuntu12.2");
startComparison(imagenesComp0, imagenesComp2);

//Ultima Comparacion
startComparison(imagenesComp1, imagenesComp2);
```

Fig. 7. Proceso de toma de todas las imágenes de los diferentes nodos

```
private void startComparison(List<BufferedImage> base, List<BufferedImage> comp) {
    for (BufferedImage image: base) {

        //Se obtiene datos de la primer imagen de la lista
        String imgBase = Utils.getIDAndPagName(image);

        //Se busca la imagen correspondiente en la segunda lista
        BufferedImage corresp: Utils.getImageMatchingWith(imgBase);

        //Se realiza la comparación de ambas imágenes
        Compare.startComparison(image, corresp);

    }
}
```

Fig. 8. Obtención de pares de imágenes para ser comparadas

Finalmente, el algoritmo de detección de incompatibilidades se ejecuta.

3.2 Algoritmo de detección de incompatibilidades

El algoritmo utilizado para la detección de incompatibilidades, está compuesto por técnicas del procesamiento digital de imágenes. Como se menciona anteriormente, Browserbite [20], [27] utiliza este tipo de técnicas para realizar pruebas cruzadas entre

navegadores, mediante la URL de los sitios. Sin embargo, su principal limitación es que solo ciertos estados de una página web pueden ser verificados, ya que existen estados que no pueden ser accedidos mediante una URL, sino mediante la invocación de scripts internos, componentes HTML parciales, etc., que se ejecutan al presionar un botón o un enlace. Por esta razón, el alcance de este algoritmo ha sido extendido mediante su incorporación al framework de pruebas funcionales. De este modo, se pueden realizar detección de incompatibilidades en cualquier estado del sitio web.

El algoritmo se compone de dos etapas:

1. Segmentación de imágenes
2. Comparación de regiones

Durante la segmentación, se divide la imagen en pequeñas regiones. En el procesamiento digital de imágenes, estas regiones se denominan Imágenes de Interés o ROIs [28] (siglas en inglés de Regions of Interest). Para ello, se comienza convirtiendo la imagen a una escala de grises mediante una API de Java (incluida en el paquete awt) que utiliza un filtro llamado GrayScale y un generador de imágenes. La implementación de esta conversión puede verse en la Figura 9 y los resultados en la Figura 10.

```
ImageFilter filter = new GrayFilter(true, 50);
ImageProducer producer = new FilteredImageSource(colorImage.getSource(), filter);
Image mage = Toolkit.getDefaultToolkit().createImage(producer);
```

Fig. 9. Código de conversión de la imagen a escala de grises

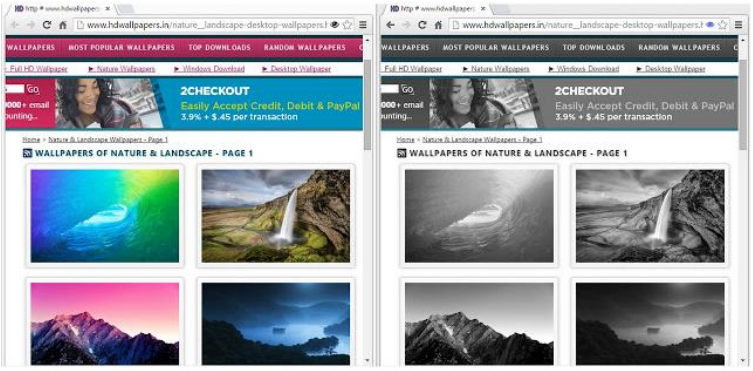

Fig. 10. Conversión a escala de grises de una captura del sitio http://www.hdwallpapers.in/

Una vez que se obtiene la imagen en escala de grises, se avanza con la detección de esquinas para encontrar los ROIs con detalles bien marcados. Para esta etapa, se utiliza la técnica de Harris y Stephens [29]. En este algoritmo, se utilizó una librería de Java denominada JFeatureLib [30], aplicada en el procesamiento digital de imágenes, especialmente para la detección de puntos y regiones. La implementación

del algoritmo de Harris por esta herramienta puede verse en [31]. El resultado de este procedimiento, es una imagen binaria donde los pixeles que representan esquinas tienen el valor 1 (blanco), y los pixeles que representan el fondo valen 0 (negro). Luego, la imagen binaria es procesada utilizando dilatación vertical y horizontal [32]. Este proceso es el encargado de conectar los puntos y generar una imagen con regiones bien marcadas. Finalmente, aplicando detección de objetos (blob analysis) [33], es posible separar los diferentes objetos de la imagen en ROIs. Para realizar esta última etapa, se utilizó una librería de OpenCV para Java, llamada JavaCV [34].

El proceso completo de segmentación puede verse en la Figura 11.

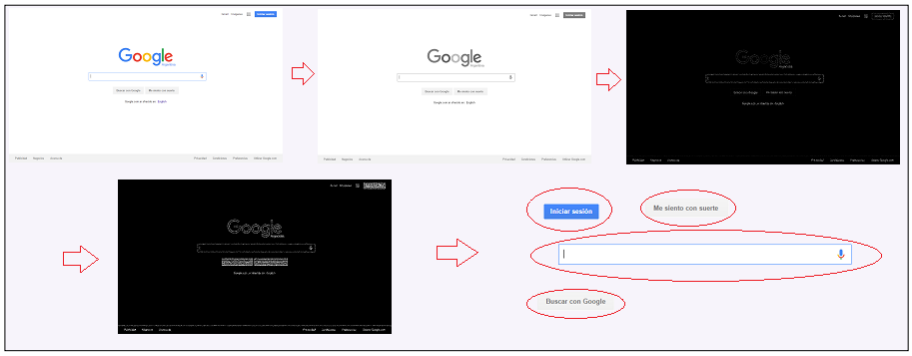

Fig. 11. Proceso completo de segmentación en ROIs utilizando una captura de https://www.google.com

El proceso de segmentación es aplicado a una captura base del sitio web, y a una captura del sitio web bajo prueba. Cada ROI generado es del tipo BufferedImage en Java, y con ellos se generan dos objetos de tipo `ImageToCompare`, uno que representa la captura base y otro la captura bajo prueba. Los objetos `ImageToCompare` tienen dos atributos: una lista de ROIs y una variable resultado. Ambas listas a su vez están compuestas por objetos que contienen el ROI en sí, un identificador y las coordenadas del ROI. Esta estructura puede verse en la Figura 12.

Fig. 12. Objetos ImageToCompare y ROI

Los dos objetos ImageToCompare son sometidos al algoritmo de detección de incompatibilidades (Figura 13). El procedimiento es el siguiente:

1. Se obtienen las listas de ROIs de los objetos ImageToCompare y se verifica que ambas listas tengan el mismo tamaño. Si difieren, se busca cuáles son los ROIs faltantes en la lista de ROIs bajo prueba (mediante los identificadores de la lista base). Finalmente, se agrega a la lista a comparar, un equivalente a los ROIs faltantes con todos los pixeles en color rojo. Antes de pasar al siguiente paso, se agrega una marca de "Fail" al objeto ImageToCompare del sitio bajo prueba, mediante la variable resultado = false.
2. Se recorren ambas listas, obteniendo por cada ROI una matriz de pixeles. Por cada matriz de pixeles, se comienza a recorrerlos uno por uno, comparándolos con su recíproco en el otro ROI. En caso de ser diferentes, se convierten a rojo los pixeles del ROI bajo prueba, y se agrega la marca de "Fail" al objeto ImageToCompare. Para esta etapa se utiliza el mismo algoritmo de comparación de imágenes propuesto en [15].
3. Al finalizar, se genera una nueva imagen con los ROIs (modificados o no), y luego se reemplaza con ella la captura del sitio bajo prueba. Para la generación de la nueva imagen, se utilizan las coordenadas de los ROIs.

```
private boolean compare(BufferedImage base, BufferedImage test)
    throws IOException {

    List<ROI> baseList = segment(base);
    List<ROI> testList = segment(test);

    return compareROIs(new ImageToCompare(baseList), new ImageToCompare(testList));

}

private boolean compareROIs(ImageToCompare sc1, ImageToCompare sc2)
    throws IOException {

    List<ROI> baseList = sc1.getROIs();
    List<ROI> testList = sc2.getROIs();

    if (baseList.size() != testList.size())
        sc2.setResult(Utils.fillTestListWithMissingROIs(baseList, testList));

    boolean result;

    for (int i = 0; i < baseList.size(); i++) {
        ROI baseROI = baseList.get(i);
        ROI testROI = testList.get(i);
        if(!bufferedImagesEqual(baseROI.getROI(), testROI.getROI()))
            result = false;
    }

    if (sc2.getResult() && !result) {
        sc2.setResult(false);
    }

    return  sc2.getResult();
}
```

Fig. 13. Algoritmo de detección de incompatibilidades.

Por último, el resultado de las comparaciones puede ser visualizado en el reporte final. El reporte final es generado con ReportNG, como un complemento de TestNG. Para la sección de pruebas de compatibilidad se ha desarrollado un reporte HTML que tiene el formato mostrado en la Figura 14, al cual se accede haciendo click en cada prueba.

Arriba de cada imagen, una leyenda indica si la captura contiene defectos de incompatibilidad (FAIL) o no (SUCCESS). En caso que los tenga, se observan los segmentos con errores en color rojo. Además, se puede ampliar el tamaño de la imagen mediante un click sobre la miniatura.

Fig. 14. Reporte con resultados de la prueba de compatibilidad.

4 Resultados obtenidos

Para validar el algoritmo propuesto, ha sido implementado en una empresa que desarrolla aplicaciones de software para relaciones públicas y que utiliza el enfoque de DSC para hacerlo. Previo a la implementación, la empresa utilizaba el algoritmo de comparación de imágenes pixel a pixel, presentado en [15]. Los resultados muestran que la nueva propuesta es más eficiente que la anterior.

Por un lado, al igual que en [15] la implementación del algoritmo fue una tarea muy sencilla de llevar a cabo por el equipo de desarrollo. Esto se debe principalmente a que ya se contaba con la infraestructura y los recursos, y que además solo debía reemplazarse la capa donde se encontraba el algoritmo de comparación de imágenes anterior.

Si bien el algoritmo de comparación píxel a píxel [15] había acelerado un 92% el tiempo de ejecución de las pruebas de compatibilidad y por ende toda la etapa de pruebas, el algoritmo propuesto aceleró un 35% más la ejecución de las pruebas (ver Figura 15). Una prueba de compatibilidad realizada con el algoritmo de comparación pixel a pixel llevaba aproximadamente 1 minuto, requerido para observar el mapa de calor. Al implementar la nueva herramienta, el tiempo pasó a ser de aproximadamente 45 segundos. Esto se debe a que los reportes del algoritmo de detección de

incompatibilidades propuesto, contiene las capturas de todos los dispositivos en los que se realizó la comparación en una misma sección, en lugar de mostrarse de a pares.

Por otro lado, el algoritmo de detección de incompatibilidades redujo la cantidad de falsos positivos obtenidos de 40% a 12% (Figura 16), ya que el algoritmo omite problemas mínimos de alineación o pequeñas diferencias en los componentes producidas por la manera en la que cada navegador hace su propio renderizado.

Finalmente, se obtuvieron comentarios y sugerencias de los equipos encargados de realizar las pruebas. Si bien se redujeron el número de falsos positivos, todavía se detectan falsos fallos en las comparaciones de páginas que contienen anuncios publicitarios cambiantes o ventanas emergentes localizadas en diferentes secciones.

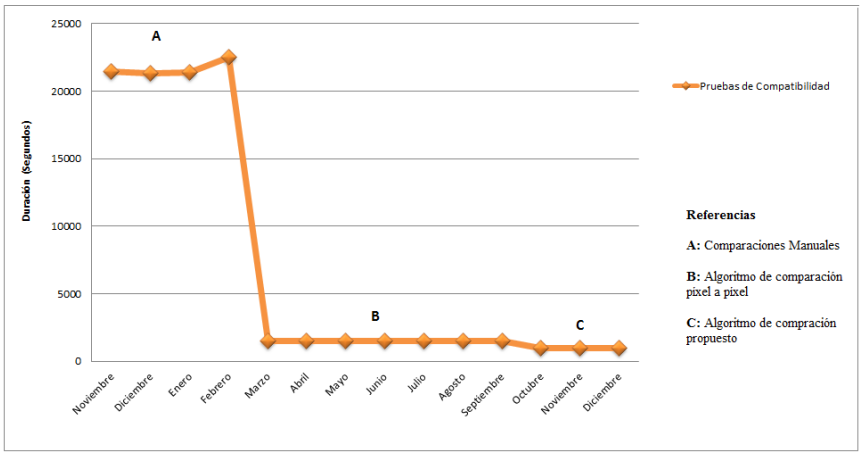

Fig. 15. Tiempo de ejecución de las pruebas en los últimos meses (en segundos)

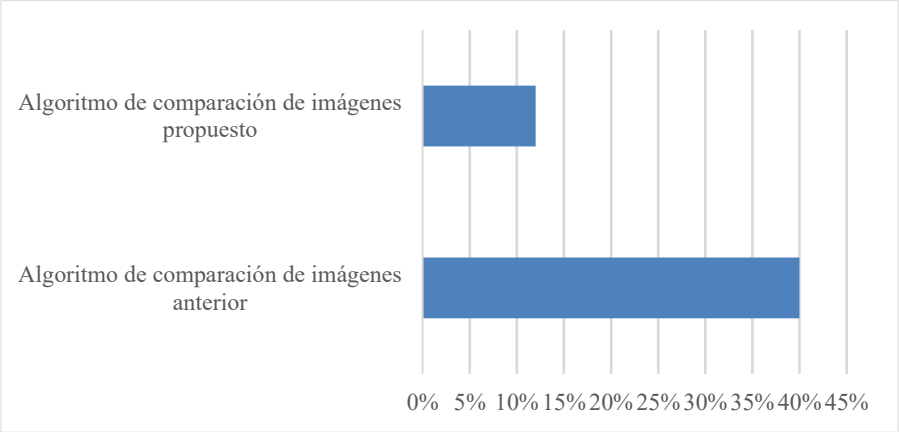

Fig. 16. Porcentaje de falsos positivos según la herramienta utilizada

5 Conclusiones

Uno de los enfoques más actuales del desarrollo de software, es el desarrollo de software continuo, en donde el tiempo es un factor fundamental. Las pruebas cobran relevancia, ya que al lanzar versiones del producto con mayor frecuencia, emergen más defectos en el mismo. En este trabajo se presentó un algoritmo que permite automatizar las pruebas de compatibilidad web e introducirlas en un proceso de desarrollo continuo de software. La implementación de este algoritmo fue posible gracias a las técnicas del procesamiento digital de imágenes.

Para validar la propuesta, la misma ha sido implementada en una empresa que trabaja con desarrollo continuo de software. Los resultados demuestran que el algoritmo es eficiente, ya que permite la visualización total de los fallos de una prueba en un solo reporte. Además, los tiempos de ejecución de las pruebas disminuyeron un 35%, y esto también redujo el tiempo total del proceso de liberación de las versiones del sitio. Con la implementación de este enfoque, se disminuyó la cantidad de falsos positivos a un 12% sobre el total de fallos. Sin embargo, se encontraron inconvenientes en la comparación de sitios que contienen anuncios publicitarios y/o elementos emergentes, produciendo falsos fallos en las pruebas.

Como trabajo futuro, se propone seguir mejorando el algoritmo para solucionar los falsos fallos mencionados. Una alternativa es la utilización de nuevas herramientas del procesamiento digital de imágenes. Una técnica de segmentación que será evaluada es la técnica VIPS publicada por Microsoft Research. Además, se busca mejorar la visualización de los reportes a través de algoritmos de clasificación. De esta manera, podrán agruparse los verdaderos fallos por un lado, y el resto por otro.

Se buscará ampliar el alcance de las pruebas de compatibilidad a dispositivos móviles, a través de técnicas de emulación. Finalmente, se comenzarán a evaluar el uso de contenedores con herramientas como Docker, para reemplazar a Selenium Grid.

Referencias

1. B. Fitzgerald and K. J. & Stol, "Continuous software engineering and beyond: trends and challenges," in Proceedings of the 1st International Workshop on Rapid Continuous Software Engineering, 2014, pp. 1-9.
2. L. Chen, "Continuous delivery: Huge benefits, but challenges too.," Software, IEEE, vol. 32, no. 2, pp. 50-54, 2015.
3. G. G. Claps, R. B. Svensson, and A. Aurum, "On the journey to continuous deployment: Technical and social challenges along the way.," Information and Software technology, vol. 57, pp. 21-31, 2015.
4. H. H. Olsson and J. Bosch, "Towards agile and beyond: an empirical account on the challenges involved when advancing software development practices," in Agile Processes in Software Engineering and Extreme Programming. Roma, Italia: Springer International Publishing, 2014, pp. 327-335.
5. E. Dustin, T. Garrett, and B. Gauf, Implementing automated software testing: How to save time and lower costs while raising quality.: Pearson Education, 2009.

6. S. Elbaum, G. Rothermel, and J. Penix, "Techniques for improving regression testing in continuous integration development environments," in Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, Hong Kong, China, 2014, pp. 235-245.
7. M. Cohn, Succeeding with agile: software development using Scrum.: Pearson Education, 2010.
8. A. Bagmar, "Enabling Continuous Delivery in Enterprises with Testing," in Agile2015, India, 2015.
9. G. N. Dapozo et al., "Características del desarrollo de software en la ciudad de Corrientes," in XXI Congreso Argentino de Ciencias de la Computación, Junin, 2015.
10. J. Liberty and D. Maharry, Programming ASP.NET 3.5, 4th ed. Massachusetts, USA: O'Reilly Media, 2008.
11. J. R. Aranda Córdoba, Desarrollo y reutilización de componentes software y multimedia mediante lenguajes de guión. Málaga, España: IC Editorial, 2014.
12. B. Shneiderman, Designing the user interface: strategies for effective human-computer interaction. Pearson Education India, 2009.
13. Cross Browser Testing. Real mobile devices & browsers! [Online]. https://crossbrowsertesting.com/ [Online]
14. Browserstack. Browser Screenshots for Quick Testing. 300+ Real Browsers, Internet Explorer 6-10, Local Testing, API, Resolution Options. [Online]. http://browserstack.com/screenshots [Online]
15. M. Mascheroni, M. Cogliolo, and E. Irrazábal, "Automatización de pruebas de compatibilidad web en un entorno de desarrollo continuo de software," in Jornadas Argentinas de Informática, 2016.
16. A. Mesbah and M. R. Prasad, "Automated cross-browser compatibility testing," in Proceedings of the 33rd International Conference on Software Engineering, 2011, pp. 561-570.
17. S. R. Choudhary, H. Versee, and A. Orso, "WEBDIFF: Automated identification of cross-browser issues in web applications," in Software Maintenance (ICSM), 2010 IEEE International Conference, 2010, pp. 1-10.
18. S. R. Choudhary, "Detecting cross-browser issues in web applications," in 2011 33rd International Conference on Software Engineering (ICSE), 2011, pp. 1146-1148.
19. S. R. Choudhary, M. R. Prasad, and A. Orso, "Crosscheck: Combining crawling and differencing to better detect cross-browser incompatibilities in web applications," in In 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation, 2012, pp. 171-180.
20. T. Saar, M. Dumas, M. Kaljuve, and N. Semenenko, "Cross-browser testing in browserbite," in International Conference on Web Engineering, 2014, pp. 503-506.
21. S. Roy Choudhary, M. R. Prasad, and A. Orso, "X-PERT: a web application testing tool for cross-browser inconsistency detection," in Proceedings of the 2014 International Symposium on Software Testing and Analysis, 2014, pp. 417-420.
22. S. Mahajan and W. G. Halfond, "Finding html presentation failures using image comparison techniques," in Proceedings of the 29th ACM/IEEE international conference on Automated software engineering, 2014, pp. 91-96.
23. E. Selay, Z. Q. Zhou, and J. Zou, "Adaptive random testing for image comparison in regression web testing," in Digital lmage Computing: Techniques and Applications (DlCTA), 2014 International Conference, 2014, pp. 1-7.
24. A. Hori, S. Takada, H. Tanno, and M. Oinuma, "An Oracle based on Image Comparison for Regression Testing of Web Applications," in The Twenty-Seventh International Conference on Software Engineering and Knowledge Engineering, 2015, pp. 639-645.

25. A. Ramakrishnan and R. Manjula, "Perceptual Difference for Safer Continuous Delivery," International Research Journal of Engineering and Technology (IRJET), vol. 3, no. 11, pp. 793-798, Nov 2016.
26. P. M. Duvall, S. Matyas, and A. Glover, Continuous integration: improving software quality and reducing risk.: Pearson Education, 2007.
27. T. Saar, M. Dumas, M. Kaljuve, and N. Semenenko, "Browserbite: cross-browser testing via image processing," Software: Practice and Experience, 2015.
28. J. R. Parker, Algorithms for image processing and computer vision.: John Wiley & Sons, 2010.
29. C. Harris and M. Stephens, "A combined corner and edge detector," in Alvey vision conference, 50, 1988, p. 50.
30. Sona Type. JFeatureLib. [Online]. https://github.com/locked-fg/JFeatureLib
31. Program Creek. Java API Examples. Harris Method with JFeatureLib. http://www.programcreek.com/java-api-examples/index.php?source_dir=JFeatureLib-master/src/main/java/de/lmu/ifi/dbs/jfeaturelib/pointDetector/Harris.java [Online]
32. F. Y. Shih, Image processing and mathematical morphology: fundamentals and applications.: CRC press, 2009.
33. R. J. Schalkoff, Digital image processing and computer vision. Vol 256.: New York: Wiley, 1989, vol. 286.
34. JavaCV. Java interface to OpenCV and more. [Online]. https://github.com/bytedeco/javacv
35. J. Aracil. (2015) Globe Testing. [Online]. http://www.globetesting.com/2012/07/pruebas-de-compatibilidad/
36. G. Smith. (2014, Febrero) Mashable. [Online]. http://mashable.com/2014/02/26/browser-testing-tools/#ZenWlmiAYGqa
37. E Selay, Z. Q. Zhou, and J Zou, "Adaptive random testing for image comparison in regression web testing," in Digital lmage Computing: Techniques and Applications (DlCTA). International Conference. IEEE, 2014, pp. 1-7.