Automatiserad testning av visuella defekter hos webbsidor

Kristian Lumme, 34065
Kandidatavhandling i datavetenskap
Handledare: Dragos Truscan
Institutionen för informationsteknologi
Åbo Akademi
2014

Referat

Medan webbsidor blir allt mer avancerade ökar mängden apparater och programvaror med vilka man kan besöka dessa. Subtila skillnader mellan implementationer medför ofta att en webbsida kan se ut och fungera olika beroende på i vilken miljö den visas. Att testa sidan med olika konfigurationer är alltså en viktig del av utvecklingen av varje webbsida. Medan man inom programvaruutvecklingen i allmänhet mer och mer drar nytta av automatiserad testning så fokuserar denna testning vanligen på funktionalitet och beteende snarare än på den visuella aspekten. Det existerar dock några olika tekniker också för denna sorts testning, vilka dessutom kan utvecklas för att skilja mellan obetydliga defekter som inte påverkar användarupplevelsen, och defekter som måste åtgärdas. I denna avhandling beskrivs de grundläggande teknikerna, samt några metoder med vilkas hjälp dessa kan förbättras.

Innehåll

Inledning

Programvaruutveckling har traditionellt innehållit en eller flera faser där programvaran testas. Under tidens gång har metoderna genom vilka testningen utförs, såväl som testningens roll i utvecklingen, förändrats. En tidig modell för programvaruutveckling, “vattenfallsmodellen” (Royce, 1970), beskrev en sekventiell process där programvaran designas, implementeras och slutligen testas. Idag förekommer testning ofta redan i ett tidigt skede av utvecklingen. Testdriven utveckling (Beck, 2003) är en metod som till och med förespråkar att testningen sker innan implementationen —utvecklare skriver först tester och sedan programvara som klarar dessa tester. På så vis är det helt klart vilken funktionalitet som krävs då man börjar implementera programvaran.

Testningen av vissa aspekter av ett system, till exempel hur snabbt en person lär sig använda det, är svår att automatisera. En stor del av testningen kan dock utföras automatiskt. Tester kan exempelvis kontrollera att det inte går att komma åt data i ett program utan att man loggat in med korrekta uppgifter, att programmet returnerar korrekta utdata vid vissa indata eller att programmet reagerar på stimuli inom en viss tid. Olika tester kan fokusera på olika aspekter av systemets operation. Utan att gå närmare in på de skilda ideologier som finns angående testning kan i alla fall konstateras att testning idag vanligen spelar en viktig roll i utvecklingen av programvara.

Inom webbutvecklingen har testningen dock ofta negligerats. Detta kan bero på att webbsidor vanligen fungerat som dokument snarare än som traditionella program, och att man därmed inte sett ett behov av testning. I och med webbens utveckling har situationen förändrats, och därmed har testning också tagits i bruk i större grad inom denna bransch. Istället för att utgöra dokument fungerar dagens webbsidor ofta som kompletta applikationer, och samma tekniker som används för att utveckla webbsidor används på många håll för att utveckla traditionell programvara, alltså programvara som inte körs i en webbläsare. I denna avhandling används ordet webbsida för att beskriva såväl traditionella dokumentliknande sidor som sidor som uppvisar mer applikationsliknande funktionalitet, eftersom det inte i detta fall finns anledning att behandla dem separat.

En webbsida kan vanligen sägas bestå av två delar, som båda kan testas. Serverdelen, eller back-enddelen, består av programvara som körs på en webbserver och genererar dokument som skickas till sidans användare. Klientdelen, eller front-enddelen, körs på klienten, t.ex. en webbläsare på användarens dator. Klientdelen står till exempel för funktionalitet som inte kräver kommunikation med servern. Även definitionerna som avgör sidans utseende tolkas på klienten. För de programmeringsspråk som vanligen används på servern, t.ex. PHP, Python och Ruby[1], finns idag ramverk som underlättar automatiserad testning. Också för JavaScript, som traditionellt

[1] Exempel på testramverk för respektive språk är PHPUnit (http://phpunit.de), unittest (http://docs.python.org/3.4/library/unittest.html) och test-unit (http://test-unit.rubyforge.org).

sett använts i klientdelen och numera även på servern[2] finns testramverk[3]. Testningen i detta sammanhang kan gå ut på att kontrollera att servern ger det väntade svaret på en förfrågan eller att funktioner på klienten fungerar som de ska.

Det finns dock en annan viktig aspekt av webbsidor, som inte rönt lika stor uppmärksamhet i testningen. De vanliga testerna kontrollerar egenskaper som indirekt påverkar upplevelsen för webbsidans användare. I någon mån testas också hur webbsidan reagerar på stimuli från användaren. Vad som saknas är testning av den mest direkta kontaktytan mellan användaren och webbsidan, det vill säga webbsidans utseende.

På grund av de tekniker som används för att definiera en webbsidas layout, färger och dylikt är det inte alltid enkelt att få den att se ut som planerat. Webbutveckling medför dessutom ett antal ovanliga utmaningar för en grafisk designer: en webbsida är inte statisk utan reagerar på stimuli och dess funktionalitet och utseende varierar enligt det system som används för att visa webbsidan. Att garantera att en webbsida ger majoriteten av dess besökare en konsistent upplevelse kan alltså vara svårt. Detta problem kan delvis avhjälpas med hjälp av automatiserad testning. På senare tid har en del forskning inom detta område gjorts, och ett antal kommersiella tjänster som automatiserar denna testning har startats. Denna avhandling beskriver vilka svårigheter denna testning medför och vilka tekniker som används för att genomföra den effektivt.

[2] Till exempel genom node.js, http://nodejs.org

[3] Bland annat QUnit, http://qunitjs.com

Bakgrund

För serverdelen av en webbsida kan i stort sett vilket programmeringsspråk som helst användas. Eftersom en webbsida består av text (som i och för sig hänvisar t.ex. till bilder) så är allt som krävs att serverdelen kan leverera text i korrekt format till klienten. På klienten tolkas denna text enligt några standarder, vanligen av en webbläsare, som sedan visar sidan för användaren.

Huvudsakligen bygger webbsidor på tre olika standarder: HTML (World Wide Web Consortium, 2014a), CSS (World Wide Web Consortium, 2011) och JavaScript (Ecma International, 2011). HTML är ett märkspråk som används för att ge webbsidans innehåll mening, till exempel genom att markera vilken del av texten som är en rubrik, vilken del som är en länk etcetera. CSS definierar webbsidans visuella aspekt, alltså dess layout, färger, typsnitt med mera. Programmeringsspråket JavaScript används vanligen för interaktiva funktioner hos webbsidan. Gränsen mellan domänerna för dessa olika språk är suddig snarare än skarp, och de samverkar med varandra för definiera webbsidans slutgiltiga form.

Dessa standarder är under ständig utveckling. HTML- och CSS-standarderna utvecklas av World Wide Web Consortium (vanligen kallat W3C) medan JavaScript utvecklas av Ecma International. Medan det i varje stund finns en officiell version av standarderna så är det inte helt klart vilken funktionalitet webbutvecklare kan använda. Standarderna är nämligen komplexa och i vissa fall mångtydiga, så det är praktiskt taget omöjligt för alla webbläsare att implementera standarderna exakt likadant. Webbläsare kan dessutom sakna stöd för viss funktionalitet definierad i standarden (ibland till förmån för egna, proprietära alternativ). Webbläsarnas utvecklare samarbetar dessutom ofta med standardernas utvecklare och börjar stöda ny funktionalitet redan innan dess slutgiltiga form fastslagits i en officiell standard. I praktiken skiljer sig alltså de verktyg webbutvecklare har till sitt förfogande från de som definieras i någon officiell standard.

Även om standarden skulle vara exakt likadant implementerad överallt skulle webbutvecklare vara tvungna att ta hänsyn till variationer. Det kommer nämligen ständigt nya versioner av webbläsare, och det är klart att t.ex. Internet Explorer 6, som släpptes för över tio år sedan och fortfarande används (Microsoft, 2014) inte stöder samma funktionalitet som den senaste versionen, Internet Explorer 11. Detta är så gott som oundvikligt, då synen på vad webben kan erbjuda och vad en webbsida bör vara kapabel till var en annan då än idag. I praktiken är webbutveckling en balansgång mellan att erbjuda den bästa användarupplevelsen och funktionaliteten, och att nå så stor del av marknaden som möjligt.

Variationerna i visningen av webbsidor tar inte heller slut i och med detta. Webbsidor kan nämligen visas på en mängd olika sorters apparater—från vanliga bordsdatorer till mobiltelefoner och TVn. Dessa apparater har varierande skärmstorlekar, upplösning, operativsystem, metoder för att interagera med webbsidan etcetera.

De aspekter som påverkar visningen av en webbsida, det vill säga webbläsaren, operativsystemet, skärmstorleken etc., kallas i denna avhandling för dess miljö. Med tanke på de variationer i miljöer som beskrivs ovan börjar det framstå som ett under att det alls förekommer någon konsistens mellan miljöerna. Det bör i alla fall stå klart, att ett effektiv testramverk som låter en webbutvecklare testa sin sida i de vanligast förekommande miljöerna automatiskt, och få rapporter om eventuella avvikelser mellan dessa, är önskvärt. Testning av sidors visuella aspekt är dock på många sätt svårt. Orsakerna till detta beskrivs i nästa del.

Dagens webbsidor är vitt skilda från de statiska sidor som existerade i början av webbens livstid. Medan JavaScript möjliggör interaktiva webbsidor som inte behöver laddas om för varje förfrågan till servern erbjuder moderna tekniker som CSS3 avancerade verktyg för sidornas grafiska utformning. CSS3, den tredje versionen av specifikationen CSS-specifikationen (World Wide Web Consortium, 2014b), inbegriper olika moduler som till exempel specificerar webbläsarens stöd för genomskinliga element, gradienter och animationer. Medan denna specifikation revideras och utvecklas jobbar webbläsarnas tillverkare med att implementera stöd för denna, experimentera med ännu inte fastslagna rekommendationer samt utveckla nya funktioner som kanske småningom hittar sin väg in i den officiella specifikationen. Slutresultatet är en levande standard, där förändringar ständigt sker och där både mer och mindre subtila skillnader förekommer mellan de olika programmen.

På senare tid har dock ett antal tekniker för att testa även de visuella aspekterna av en webbsida automatiskt sett dagens ljus. Främst är man ute efter att automatiskt upptäcka väsentliga skillnader mellan webbsidans utseende i olika miljöer. Denna testning medför sina egna unika utmaningar, då ett program inte kan skilja mellan obetydliga skillnader och verkliga defekter på samma sätt som en människa kan.

Problem uppkommer förstås inte heller enbart på grund av inkompatibilitet mellan olika programvaror, eller andra aspekter av miljöer—den mänskliga faktorn kan också spela en roll. Till exempel kan misstag i sidans implementation göras, och om sidan bara ses i en webbläsare under utvecklingen kan dessa misstag upptäckas först då man testar sidan i andra miljöer. Även i dessa fall kan de tekniker som beskrivs i denna avhandling hjälpa—orsaken till felet påverkar inte teknikernas effektivitet.

Nyttan av denna testning är inte heller begränsad till upptäckande av skillnader som uppstår i olika miljöer. Många olika orsaker kan göra att en sida inte visas rätt i någon miljö, eller i alla testmiljöer. På en stor webbsida kan definitionerna som styr dess utseende lätt bli så komplicerade att ändringar kan medföra oväntade fel på till synes orelaterade delar av sidan. Sidor med dynamiskt innehåll kan fungera fint till att börja med, bara för att “gå sönder” då innehåll som inte motsvarar utvecklarens förväntningar förs in. Om ett element innehåller text så kan förändring av texten orsaka variationer av elements storlek, vilket i sin tur påverkar sidans layout. Som ett specialfall av

detta kan nämnas översättning av webbsidor—kanske texten på en sida är precis rätt längd på engelska, men ändrar längd och orsakar därmed problem då den översätts till svenska. Alla dessa exempel utgör potentiella användningsområden för dessa tekniker.

Grundläggande testningsmetoder

Ramverk för att automatisera webbläsarens funktioner, som till exempel att ladda en sida eller att följa länkar, existerar. Två exempel på populära ramverk är Selenium[4] och PhantomJS[5]. Med hjälp av dessa verktyg, virtuella maskiner samt andra automatiseringsprogram kan man relativt lätt automatisera en del av testningsprocessen—man kan skapa en tjänst som öppnar sidan i de olika miljöerna, fångar bilder av hur de ser ut samt presenterar dessa bilder för testaren. Vidare kan man, istället för att fånga statiska bilder, låta en testare interagera med sidorna i realtid. Denna typ av metoder ligger till grund för ett antal kommersiella tjänster, tillgängliga via internet, bland annat Sauce Labs[6] och BrowserStack[7]. Dessa automatiserar alltså produktionen av det material från olika miljöer som ska jämföras—de automatiserar inte själva jämförelsen.

Dessa tjänster underlättar alltså testarens uppgift, men kräver fortfarande att en människa bedömer resultatet—inspekterar bilderna och identifierar skillnader mellan dem—manuellt. Människor är dock inte speciellt effektiva när det gäller den här sortens jobb. Det handlar om att jämföra många bilder eller sidor då webbsidan som testas kan ha många undersidor eller många olika komponenter. Dessutom kan även viktiga skillnader vara svåra att märka, och de flesta jämförelser ger negativt resultat, det vill säga inga skillnader upptäcks. Koncentrationen tar snabbt slut och testarens precision och täckning minskar. Detta jämförelsejobb verkar vara just den sorts ensidigt jobb som en dator kan göra betydligt effektivare än en människa. Problemet är att en människa däremot är betydligt bättre på att känna igen mönster och på att avgöra vilka skillnader som kommer att påverka användarupplevelsen och alltså utgör defekter, och vilka skillnader som kan ignoreras. Dessutom är människor bra på att snabbt analysera situationen och till exempel notera att en rad olika fel har uppstått på grund av att ett element förskjutits, och att det alltså är denna förskjutning som är det ursprungliga problemet som måste åtgärdas.

För att automatiskt jämföra en webbsidas utseende i olika miljöer faller det sig naturligt att skriva programvara som besöker webbsidan i de olika miljöer som testas, tar en skärmdump eller på annat sätt spara en bild av hur sidan ser ut och slutligen jämför dessa bilder med hjälp av en passande algoritm och meddelar de upptäckta skillnaderna till testaren. För att göra de olika miljöerna tillgängliga för testet kan virtuella maskiner vara ett användbart verktyg—en virtuell maskin kan exempelvis köra Windows XP med Internet Explorer version 6, medan en annan som körs på samma fysiska maskin kan emulera webbläsaren på en Android-telefon. Testaren kan välja

---

[4] http://docs.seleniumhq.org

[5] http://phantomjs.org

[6] https://saucelabs.com

[7] http://www.browserstack.com

en version av webbsidan som den korrekta versionen, vilken de andra bilderna sedan jämförs med.

Denna procedur används som grund i många metoder. Den har dock sina begränsningar, som gör att den bör kompletteras med andra metoder för att vara praktisk. Problemen har sin grund i att många skillnader som på detta vis upptäcks är obetydliga och inte påverkar användarupplevelsen. Om ett element i en miljö till exempel visas två pixlar till höger jämfört med i en annan är detta antagligen inte en skillnad som påverkar användarupplevelsen nämnvärt. Ett annat exempel är då ett typsnitt används i en miljö och ett annat i en annan—om bilderna enbart jämförs pixel för pixel blir resultatet här en mängd rapporterade fel där tecknen i de båda typsnitten uppvisar små skillnader, trots att webbsidan kan fungera väl. Problemet är alltså att avgöra vilka skillnader som är väsentliga. Algoritmen som jämför bilderna kan förstås utrustas med en viss "tolerans"—skillnader kvantifieras och algoritmen rapporterar endast skillnader vars värde överstiger en viss gräns. I praktiken visar det sig dock att även en algoritm med tolerans antingen ger för många falska positiva—rapporterar oväsentliga skillnader—eller falska negativa—den missar verkliga defekter—beroende på hur toleransen konfigureras (Semenenko, Dumas och Saar, 2013).

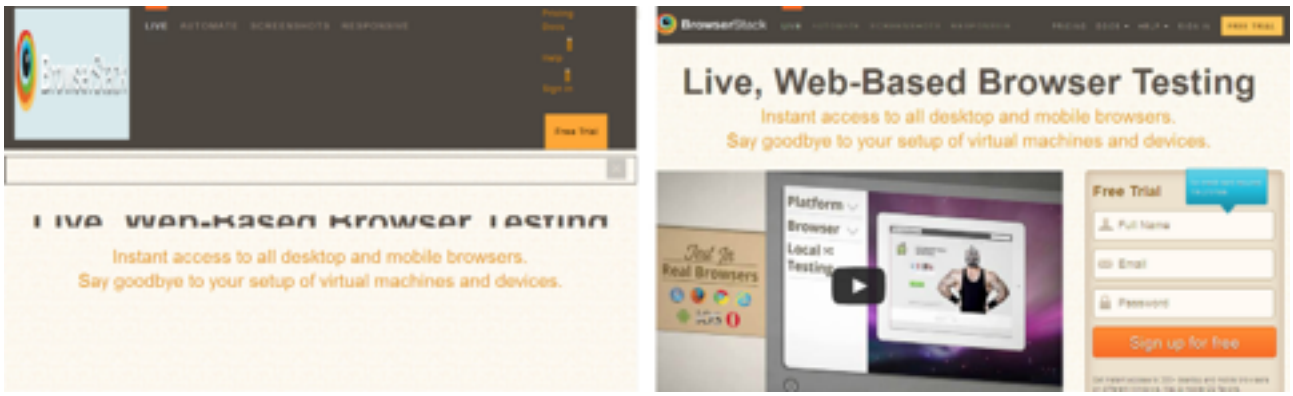
Figur 1 visar hur Svenska.yle.fi ser ut i två olika webbläsare. Man ser små skillnader mellan versionerna, bland annat kommer en radbrytning på olika plats. Dessa skillnader är dock så små att de inte nämnvärt påverkar användarupplevelsen. En optimal algoritm skulle alltså inte rapportera dessa skillnader som defekter på sidan. Figur 2 visar BrowserStacks webbsida i två olika webbläsare. Här ser man knappt att det är fråga om samma webbsida—algoritmen borde rapportera detta som en defekt. För båda dessa exempel användes BrowserStacks tjänst.

En annan utmaning uppstår då element på sidan påverkar varandra. Om ett förskjutet element också förskjuter de element som kommer efter det på sidan, men de efterföljande elementens inbördes förhållande förblir korrekt, skulle det bästa vara om algoritmen endast rapporterade om det första felet—elementet som förskjuts och påverkar de andra. En simpel jämförelse mellan bilderna skulle dock rapportera att alla dessa elements position är felaktig.

Målet är alltså att minimera både falska positiva och falska negativa. Problemet är att om man ökar toleransen för att minska antalet falska positiva så ökar antalet falska negativa eftersom den högre toleransen också släpper genom fler verkliga defekter. Minskar man toleransen ökar antalet falska positiva. Då man söker efter en balans mellan dessa är man vanligen villig att acceptera falska positiva mer än falska negativa. Falska positiva utgör bara en olägenhet, medan falska negativa utgör ett oupptäckt fel. Som nämnts visar det sig att teknikerna som beskrivs ovan inte är tillräckligt effektiva för att användas i verkliga situationer. På något sätt måste alltså dessa metoder förbättras för att få ett verktyg som går att använda inom industrin.

Här kan nämnas en annan metod, som exempelvis används av programvarubiblioteket Fighting Layout Bugs (Tamm, 2009) för att hitta visuella defekter. Till skillnad från att jämföra sidor i olika miljöer går denna ut på att definiera ett antal vanliga fel som webbsidor ofta uppvisar, och automatiskt söka efter dessa. Denna metod jämför alltså inte någon testversion med en korrekt version, utan går igenom en sida i en miljö, utan att ta hänsyn till andra miljöer, och söker efter vanliga tecken på att något är fel. Dessa fel kan till exempel vara text som går utanför ramarna för det element texten finns i, eller element som överlappar varandra. Sådana metoders är dock mycket begränsade. Det finns nämligen situationer där noder överlappar varandra utan att något för den skull är fel, och samma sak gäller för de flesta allmänna regler man kan ställa upp för vad som utgör en defekt hos en webbsida—undantagen är för många och för svårdefinierade för att reglerna ska kunna tillämpas i praktiken. Dessutom finns det många fel för vilka det är svårt att ens formulera några regler överhuvudtaget, än mindre allmängiltiga.

Figur 1: Svenska.yle.fi, visad i Safari (till vänster) och i Mozilla Firefox (till höger) (URL: http://svenska.yle.fi, hämtad 1.4.2014).



Figur 2: BrowserStacks webbsida, visad i två olika webbläsare med hjälp av BrowserStacks egen tjänst. Till vänster syns sidan i Internet Explorer 6, till höger ses hur sidan ska se ut (här i en modern version av Mozilla Firefox) (URL: http://www.browserstack.com, hämtad 1.4.2014).

Utvecklade metoder

Att bara jämföra bilder, pixel för pixel, ger alltså inte tillräckligt goda resultat. En användbar metod måste på något vis ta mera information, så som webbsidans struktur, i beaktande. Ett sätt på vilket man bättre kan avgöra de upptäckta skillnadernas betydelsefullhet är att resonera utgående från elementen sidorna är uppbyggda av, som stycken, rubiker, ramar etcetera, istället för pixlar. Två olika sätt på vilka detta har gjorts baserar sig på DOM-träd och på analys av en bild av webbsidan.

Metoder baserade på DOM

Då en webbläsare laddar en sida för att visa den för användaren bygger den upp en intern representation av sidan och dess komponenter. Denna representation utgörs av ett s.k. DOM-träd. DOM står för Document Object Model eller dokumentobjektmodell, och är en standard för att representera och interagera med objekt i till exempel en webbsida (World Wide Web Consortium, 2004). Exempelvis sker JavaScripts interaktion med en webbsida via dess DOM-träd. I denna trädstruktur utgörs noderna av element som länkar, bilder, listor, samt av textnoder, sidans egentliga “innehåll”. DOM-trädet kan användas för att effektivare analysera skillnader mellan sidans utseende i olika miljöer.

Choudhary, Versee och Orso (2010) beskriver en teknik, använd i deras program WebDiff, där testprogramvaran sparar denna trädstruktur för varje miljö där webbsidan testas och utnyttjar denna vid den visuella analysen. Deras algoritm noterar först vilka noder som förändras då sidan laddas om. Dessa variabla noder kan till exempel vara reklam, som varierar varje gång sidan laddas. Dessa element ignoreras i analysen, eftersom de annars med stor sannolikhet skulle leda till falska positiva.

Algoritmen parar sedan ihop noderna från webbsidans DOM-träd i varje testad miljö med noderna från referensversionens DOM-träd. I de fall där trädens strukturer inte helt stämmer överens försöker algoritmen para ihop noder utgående från vissa kriterier. Algoritmen använder sedan denna data vid den visuella analysen. Nodernas position på sidan jämförs utgående från deras skiftning i förhållande till nodens förälder i DOM-trädet. Detta på grund av att om nodernas inbördes förhållande och position i förhållande till föräldern är korrekt, medan positionen i förhållande till sidan är fel, så är det föräldraelements position som måste korrigeras. Nodernas synlighet, storlek och utseende jämförs sedan oberoende av föräldraelementet, eftersom detta här inte har någon inverkan.

Med denna teknik fick man 17 % falska positiva—17 % av de rapporterade felen var alltså oväsentliga. Statistik om hur många falska negativa man fick, alltså hur många verkliga defekter som missades, ges inte.

Kevin Menard beskriver en mycket liknande metod i sin presentation av vad han kallar för Web Consistency Testing (Menard, 2011). Hans kommersiella tjänst Mogotest[8] grundar sig på just denna metod. Han noterar att DOM-trädet utgör en hierarki: vid jämförelse av noders position kan en avvikelse beräknas som sedan förs vidare till nodernas avkomma för att normalisera deras positioner i förhållande till föräldern. I praktiken motsvarar detta metoden som Choudhary m. fl. (2010) använde där positionen beräknas i förhållande till nodens föräldrer. Menard ger ingen statistik för hur väl hans verktyg fungerar men han noterar att sådana falska negativa som inte upptäcks vid den första testrundan ofta upptäcks vid tester som körs efter att de fel som först upptäcktes korrigerats.

Menard nämner också att DOM-strukturen bör filtreras innan den verkliga analysen. Man kan till exempel avlägsna noder som inte ritas ut på sidan, eller noder som är helt dolda av andra element eftersom dessa inte påverkar sidans utseende.

Även den kommersiella tjänsten Browsera[9] använder en liknande metod där DOM-strukturen används för effektivare jämförelse av bilder.

Metoder baserade på ROI

En annan, besläktad teknik för att få en uppfattning om webbsidans struktur beskrivs av Semenenko m. fl. (2013). Istället för att utnyttja DOM-träd analyserar deras algoritm bilderna som tas av webbsidan i varje miljö, och identifierar så kallade Regions of Interest, ROI, eller betydelsefulla områden. Dessa områden identifieras på basis av färger, ramar, etcetera. I likhet med den DOM-baserade algoritmen som beskrivs ovan paras elementen i varje version sedan ihop med elementen i den korrekta versionen. Varje områdespar får ett värde baserat på hur väl de stämmer överens, och om detta är under en viss gräns rapporteras ett fel. En första version av denna algoritm gav endast 10 % falska positiva men missade däremot 44 % av felen. När algoritmen justerats så att den fångade 98 % av alla fel så var mängden falska positiva 34 %, vilket konstaterades vara oacceptabelt. Semenenko m. fl. (2013) fortsätter sin artikel med att beskriva hur man utvecklade denna metod för att åstadkomma ett användbart verktyg. Teknikerna de beskriver ligger till grund för den kommersiella tjänsten BrowserBite[10].

Choudhary, Versee och Orso (2010) ansåg inte heller att resultaten de fick var tillräckligt bra. I artikeln CrossCheck: Combining Crawling and Differencing To Better Detect Cross-browser Incompatibilities in Web Applications (Choudhary, Prasad och Orso, 2012) beskriver de en uppdaterad metod för att hitta defekter.

---

[8] http://mogotest.com

[9] http://www.browsera.com

[10] http://browserbite.com

Såväl Semenenko m. fl. (2013) som Choudhary, Versee och Orso (2010) använder maskininlärning för att öka verktygens tillförlitlighet. Deras metoder beskrivs i nästa sektion.

Klassifikation av testresultaten

Hittills har jag beskrivit hur en webbsidas struktur kan utnyttjas för att förbättra resultaten från jämförelser mellan bilder av sidan tagna i olika miljöer, antingen genom att ta denna information från sidans DOM-träd eller genom att analysera bilderna själva. Eftersom ingen av dessa tekniker i sig ännu gett en tillräckligt "träffsäker" algoritm har de supplementerats med metoder som klassificerar resultaten och uppskattar en sannolikhet för att de utgör verkliga fel genom maskininlärning.

Choudhary m. fl. (2012) beskriver CrossCheck, ett verktyg som bygger på WebDiff (Choudhary m. fl., 2010), men som dessutom använder bland annat maskininlärning för att förbättra verktygets träffsäkerhet. Metoden de beskriver bygger på en beslutsträdsklassificerare (Quinlan, 1993). Klassificeraren tar bland annat noders storlek, position samt skillnaderna mellan bildernas histogram i beaktande. Denna klassificerare tränades genom att man genererade data—potentiella defekter—med en metod som motsvarar den som användes i verktyget WebDiff och sedan sorterade dem manuellt—man markerade dem antingen som verkliga defekter eller som fel som var så små att de kunde ignoreras. När klassificeraren tränats enligt dessa data, kan den förutsäga om det är troligt att en viss skillnad mellan hur webbsidan visas i två olika miljöer utgör en defekt, utgående från skillnadens egenskaper av den typen som klassificeraren beaktar. Klassificeraren opererar alltså på basis av tidigare "erfarenhet".

Choudhary m. fl. (2012) kom fram till att verktyget CrossCheck, utrustat med denna klassificerare, presterade bättre än WebDiff (Choudhary m.fl., 2010). De utförde ett test där CrossCheck sammanlagt hittade 314 verkliga defekter jämfört med 119 för WebDiff. Täckningen för CrossCheck var alltså betydligt bättre. CrossCheck hade en precision på 36 % jämfört med 21 % för WebDiff. Det är inte klart hur detta resultat relaterar till den precision för WebDiff på 83 % som Choudhary m. fl (2010) rapporterade om, men klart är i alla fall att CrossCheck (med maskininlärning) presterade bättre än WebDiff (utan maskininlärning).

Också Semenenko m. fl. (2013) använder maskininlärning för att förbättra sina resultat. Efter att ha beskrivit sin metod, som utnyttjade områden som identifierats på bilder av webbsidorna istället för information från sidornas DOM-träd, fortsätter de med att beskriva hur de utökat sitt verktyg med en modul för klassifikation av resultaten. Ett mängd testdata skaffades genom att köra deras tidigare metod på en stor mängd webbsidor vilket gav en mängd möjliga defekter. Även i detta fall identifierades bland dessa sedan verkliga defekter genom manuell kontroll. I likhet med metoden som Choudhary m. fl. (2012) använde så beaktar klassificeraren som används bland annat områdens storlek och position, samt skillnader i bildernas histogram. Två metoder för att klassificera resultaten testades: beslutsträd, som Choudhary m. fl. (2012) också använde, och artificiella neuronnät (eller neurala nätverk) (Zhang, 2000).

Ett framåtmatande neuronnät som bestod av tre lager användes. I lagret för indata fanns 17 neuroner, vilket motsvarade antalet egenskaper klassificeraren tog i beaktande. I det mellersta lagret fanns 11 neuroner, ett antal som avgjordes experimentellt. I lagret för utdata fanns två neuroner, eftersom nätverkets indata skulle sorteras i två kategorier. Nätverket tränades med hjälp av den tidigare nämnda datamängden.

För att beskriva de olika metodernas effektivitet kan man tala om falska negativa och falska positiva som ovan. Man kan också tala om deras precision och täckning (Powers, 2011). Med precision menas hur stor del av de rapporterade resultaten som verkligen är relevanta, det vill säga som *inte* är falska positiva. Med täckning menas hur stor del av de relevanta resultaten som rapporteras. Med andra ord utgör täckningen det totala antalet relevanta resultat minus de falska negativa—de relevanta resultat som verktyget missade.

Med beslutsträd uppnådde man en precision på 0,84, dvs. 16 % av felen som verktyget rapporterade var oväsentliga, medan täckningen, eller mängden av defekter som verktyget hittade, jämfört med den totala mängden existerande defekter var 0,79, vilket betyder att verktyget missade 21 % av de verkliga defekterna. Resultaten visade att neuronnät var den mest effektiva metoden för att identifiera defekter: endast 4 % av de rapporterade felen var oväsentliga och 11 % av de verkliga defekterna missades.

Framtida utveckling

Som tidigare nämnts så är det först på senare tid som verktygen för webbutveckling börjat uppnå lika hög sofistikering som motsvarande verktyg för traditionell programmering. Detta beror säkert på att webben i sig utvecklats, att man börjat tänja på gränserna för vad som går att göra på webben och att gränsen mellan mellan webbapplikationer och traditionella program inte mera är lika tydlig. Därmed har det också stått klart att mer sofistikerade verktyg behövs för att kunna leverera system som är pålitliga, högpresterande och ger en god användarupplevelse i majoriteten av miljöerna där systemet används.

Automatiserad testning är ett av dessa verktyg och kan, förutom att visa att sidans användarupplevelse är konsekvent i olika miljöer, också visa att man inte skapar buggar i webbsidan då man utveckar den och tar i bruk nya versioner. Med tanke på den enorma variationen av miljöer är automatiserad testning oumbärlig om man vill få en heltäckande bild över hur sidan fungerar i dessa. Eftersom webben fortfarande utvecklas i snabb takt verkar det troligt att också verktygen för testning kommer att utvecklas. Medan mängden forskning inom det fält som beskrivs i denna avhandling är ganska liten kan metoderna som beskrivits dra nytta av framsteg inom andra fält. Maskininlärning, mönsterigenkännande och datorseende hör till grenar av datavetenskapen inom vilka framsteg kan leda till bättre testprogramvara. Choudhary m. fl. (2012) lyckades förbättra precisionen hos sitt verktyg genom användningen av beslutsträd, och när Semenenko m. fl. (2013) jämförde beslutsträd med artificiella neuronnät fann de att de med hjälp av neuronnät lyckades öka precisionen ytterligare. Framtiden kan mycket väl föra med sig nya framsteg som ytterligare ökar verktygens effektivitet.

Å andra sidan kan man konstatera att om alla webbläsare skulle implementera standarden exakt likadant skulle behovet av denna typ av testning minska drastiskt. I och med webbens framsteg har de mest populära webbläsarnas utvecklare allt mer börjat samarbeta för att erbjuda användarna en så konsekvent upplevelse som möjligt. Om både standarderna samt processen genom vilka dessa implementeras fortsätter utvecklas i framtiden, och automatiska uppdateringar i webbläsarna ser till att de flesta användare har relativt nya versioner av sin webbläsare, är det möjligt att problemen med inkonsekvens mellan olika miljöer minskar betydligt. Behovet av att testa till exempel olika skärmstorlekar och typer av apparater kvarstår, men eftersom webbutvecklaren vanligen anpassar sidan till dessa olika miljöer manuellt, blir det svårt, om inte omöjligt, för ett program att veta vad som är den korrekta versionen av sidan—vad som är en avsiktlig ändring och vad som är ett fel.

Avslutning

Det står klart att automatiserad testning av webbsidor i olika miljöer fyller en viktig roll. Medan de olika standarderna som används på webben får mer spridning och webbläsarnas implementationer av dessa blir mer homogena är samlingen miljöer i vilka webbsidorna används allt mer fragmenterad. Inte bara persondatorer, utan också TVn, telefoner och bilar kan nuförtiden ha utrustning som möjliggör användadet av webben. Manuell testning blir ohållbart i denna situation.

Erfarenheten visar också att de tekniker som finns idag klarar att avhjälpa problemet i någon mån. Den bästa grunden för testning har visat sig vara en jämförelse mellan en korrekt och en testad version av en sida, med hjälp av strukturell information om sidan. Denna jämförelse i sig räcker dock inte till för att ge tillräcklig precision. Därför har dessa metoder kompletterats med tekniker baserade på maskininlärning för att bättre kunna skilja mellan obetydliga skillnader och defekter. En metod baserad på artificiella neurala nätverk har hittills visat sig vara den mest effektiva.

Framsteg inom övriga vetenskaper kan medföra ännu effektivare verktyg i framtiden. I dagsläget kan dock konstateras att de existerande metoderna uppnår en effektivitet som gör dem användbara också i verkliga situationer. Detta styrks också av att flera kommersiella tjänster, baserade på metoder som motsvarar de som här beskrivits, startats inom de senaste åren.

Källor

Choudhary, Shauvik Roy, Husayn Versee & Alessandro Orso, 2010. *WebDiff: Automated Identification of Cross-browser Issues in Web Applications*, i Proceedings of the 2010 IEEE International Conference on Software Maintenance (ICSM), s. 1-10.

Choudhary, Shauvik Roy, Mukul R. Prasad & Alessandro Orso, 2012. *CrossCheck: Combining Crawling and Differencing To Better Detect Cross-browser Incompatibilities in Web Applications*, i Proceedings of the 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation (ICST), s 171-180.

Semenenko, Nataliia, Marlon Dumas & Tõnis Saar, 2013. *Browserbite: Accurate Cross-browser Testing via Machine Learning Over Image Features*, i Proceedings of the 2013 29th IEEE International Conference on Software Maintenance (ICSM), s. 528-531.

Royce, Winston W., 1970. *Managing The Development of Large Software Systems*, i Proceedings of IEEE WESCON, s 1-9.

World Wide Web Consortium, 2014a. *HTML5, A vocabulary and associated APIs for HTML and XHTML, W3C Candidate Recommendation 04 February 2014*. URL: *http://www.w3.org/TR/html5/* (hämtad 20.3.2014).

World Wide Web Consortium, 2011. *Cascading Style Sheets Level 2 Revision 1 (CSS 2.1) Specification, W3C Recommendation 07 June 2011*. URL: *http://www.w3.org/TR/CSS2/* (hämtad 20.3.2014).

World Wide Web Consortium, 2014b. *CSS Current Status*. URL: *http://www.w3.org/standards/techs/css#w3c_all* (hämtad 1.4.2014).

Ecma International, 2011. *ECMAScript Language Specification, 5.1 Edition*. URL: *http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-262.pdf* (hämtad 20.3.2014).

Microsoft, 2014. *IE 6 Countdown*. URL: *http://www.modern.ie/ie6countdown* (hämtad 27.3.2014).

Beck, Kent, 2003. *Test-driven Development by Example*. Addison Wesley.

Michael Tamm, 2009. *Fighting Layout Bugs*. URL: *https://code.google.com/p/fighting-layout-bugs/* (hämtad 1.4.2014).

World Wide Web Consortium, 2004. *Document Object Model (DOM) Level 3 Core Specification*. URL: *http://www.w3.org/TR/2004/REC-DOM-Level-3-Core-20040407/* (hämtad 1.4.2014).

Menard, Kevin, 2011. *Web Consistency Testing*. URL: *http://webconsistencytesting.com* (hämtad 1.4.2014).

Powers, D. M. W., 2011. *Evaluation: From Precision, Recall and F-Measure to ROC, Informedness, Markedness & Correlation*, i Journal of Machine Learning Technologies vol. 2.

Quinlan, John Ross, 1993. *C4.5: Programs for Machine Learning*. Morgan Kaufmann.

Zhang, Guoqiang Peter, 2000. *Neural Networks for Classification: A Survey*, i IEEE Transactions on Systems, Man, and Cybernetics, Part C: Applications and Reviews.