Tampereen yliopisto

Riku Autio

MALLIPOHJAISEN TESTAUKSEN HYÖDYT JA HAASTEET

Informaatioteknologian ja viestinnän tiedekunta
Pro gradu -tutkielma
Kesäkuu 2023

TIIVISTELMÄ

Riku Autio: Mallipohjaisen testauksen hyödyt ja haasteet
Pro gradu -tutkielma
Tampereen yliopisto
Tietojenkäsittelytieteiden tutkinto-ohjelma
Kesäkuu 2023

Mallipohjainen testaus on noussut viimeisten vuosikymmenten aikana merkittäväksi osaksi testauskulttuuria. Se markkinoi itseään taloudellisia säästöjä tuovana ja testauksen tehokkuutta lisäävänä testausmenetelmä. Mallipohjainen testaus onkin laajalti käytössä oleva testauksen muoto, mutta se ei kuitenkaan ole korvannut perinteisiä testausmenetelmiä.

Tässä tutkielmassa käydään ensin kattavasti läpi testauksen teoriapohjaa, jonka jälkeen tutustutaan mallipohjaiseen testaukseen menetelmänä tässä testauskehyksessä. Lisäksi etsitään, millaisia työkaluja mallipohjaisessa testauksessa on käytössä, ja esitellään niistä muutamia. Näiden jälkeen perehdytään kirjallisuuskatsauksen avulla mallipohjaisesta testauksesta viimeisen kymmenen vuoden aikana julkaistuun kirjallisuuteen ja etsitään haasteita, jotka rajoittavat mallipohjaisen testauksen käyttöä ja toisaalta hyötyjä, jotka tukevat mallipohjaisen testauksen soveltamista tulevaisuudessa yhä useammassa testausta sisältävässä hankkeessa.

Keskeisimpiä kirjallisuuskatsauksessa löytyneitä hyötyjä olivat mallin jo kehityksen varhaisessa vaiheessa antama selkeyttävä kuva testattavasta kohteesta, testauksen tehokkuuden ja virheiden havaitsemiskyvyn kasvu sekä testitapausten luonnin helpottuminen ja nopeutuminen. Haasteina puolestaan koettiin raportoitujen ohjeistusten ja konkreettisten käytännön esimerkkien vähyys kirjallisuudessa, mallipohjaisen testauksen työläys ja vaikeus varsinkin mallipohjaisen testauksen suhteen kokemattomille testaajille sekä mallinnuksen ja mallin luonnin vaativuus ja tietotaidon korkea vaatimustaso.

Avainsanat: Mallipohjainen testaus, testaus, tilamalli, aktiomalli, äärellinen tilakone

Sisällys

1 Johdanto

Ohjelmistokehityksen merkittävä ja tärkeä osa on testaus, jolla pyritään varmistumaan testauksen kohteena olevan ohjelmiston laadusta ja sujuvaa käyttöä haittaavien virheiden olemattomuudesta [Mohanty *et al.*, 2017]. Toisin sanoen testauksella tarkistetaan, vastaavatko ohjelmiston odotettu ja käytännön toiminta toisiaan vai löytyykö niistä eroja [Utting *et al.*, 2012]. Testaajan ydintehtävä on yrittää löytää testattavasta kohteesta virheitä ja osoittaa epätoimivuudet. Humoristisesti voidaankin todeta ohjelman olevan laadukas ja viimeistelty, jos ammattitaitoinen testaaja epäonnistuu päätehtävässään.

Teollisuudenala luo jatkuvasti lisää painetta testaukselle, kun vaaditaan ohjelmistojen entistä nopeampaa markkinoille saantia ja kustannustehokkaampaa toimintaa. Näihin vaatimuksiin mallipohjainen testaus pyrkii vastaamaan. Mallipohjaisen testauksen keskeisin idea on muodostaa ja automatisoida mahdollisimman moni testitapauksiin liittyvistä toiminnan vaiheista [Kramer ja Utting, 2016]. Mallipohjaisen testauksen syvin olemus pohjautuu testattavan kohteen vaatimusmäärittelyjen pohjalta luotavaan malliin. Malli pyrkii kuvaamaan ja selittämään ohjelmiston toimintaa riittävän tarkalla ja abstraktilla tasolla, jotta mallin pohjalta voidaan toteuttaa testitapauksien luonti. Mallien tarkoitus ei ole pelkästään kuvata ja luoda, vaan myös ylläpitää testausta [Kramer ja Utting, 2016].

Mallipohjaiseen testaukseen on kehitetty monia työkaluja, joiden tarkoitus on helpottaa ja auttaa testaajaa mallinnuksessa, testitapausten luonnissa ja valinnassa sekä testien toteutuksessa [Marinescu *et al.*, 2015]. Kirjon ollessa laaja ei sopivan työkalun valinta testaukseen ole helppoa. Työkalua valittaessa tulee pohtia, onko valittu työkalu sopiva kyseiseen ohjelmiston testaukseen ja toisaalta riittävän selkeä ja sujuvasti käytettävä helpottamaan testaajan työtä.

Tässä tutkielmassa perehdytään ensin testauksen ja mallipohjaisen testauksen teoriaan ja sen jälkeen mallipohjaisen testauksen työkaluihin sekä etsitään kirjallisuudesta mallipohjaiseen testaukseen liittyviä hyötyjä ja haasteita. Tarkoituksena on löytää vastauksia tutkimuskysymyksiin, millaisia työkaluja mallipohjaiseen testaukseen on olemassa, ja mitkä ovat mallipohjaisen testauksen haasteet ja toisaalta mitkä puolestaan hyödyt.

Mallipohjainen testaus valikoitui aiheeksi, koska halusin oppia siitä enemmän kuin vain pintaraapaisun verran parilla luennolla. Lisäksi koin kiinnostavana tutkia ja etsiä syitä eli hyötyjä ja haasteita, jotka lisäävät tai toisaalta rajoittavat mallipohjaisen testauksen käyttöä osana ohjelmistokehitystä. Olen myös pohtinut käsitystä, joka minulle on jäänyt, eli miksi testaus koetaan monesti niin vaikeaksi, vaivalloiseksi ja raskaaksi ohjelmistokehityksen vaiheeksi. Lisäksi olen miettinyt, toteutetaanko testausta vain, koska on

pakko, jotta välttämättömät ohjelmistot toimivat, vai voitaisiinko testausta ehkä mallipohjaisen testauksen avulla tehostaa, nopeuttaa ja parantaa niin, ettei se olisi vain pakollista, vaan useammin myös palkitsevaa ja hauskaa.

Melko vähäisen mallipohjaisen testauksen tuntemukseni vuoksi valitsin tutkimusmenetelmäksi kirjallisuuskatsauksen. Kirjallisuuteen perehtyminen on tärkeä osa ammattitaidon kehittämistä ja tiedon kartuttamista, ja niinpä tutkimusmenetelmän valinta tuntui luontevalta ja oikealta. Lisäksi kirjallisuuskatsaus tarjoaa myös muille aiheesta kiinnostuneille kootusti tietoa ja pohdinnan kautta uusia näkökulmia ja ajateltavaa aiheeseen liittyen. Hyötyjen ja haasteiden osalta kirjallisuuskatsaus on rajattu viimeisen kymmenen vuoden eli vuosien 2014–2023 aikana julkaistuun kirjallisuuteen.

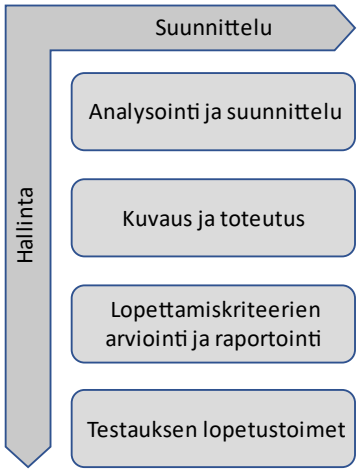
Tutkielman luvussa 2 käydään ensin läpi kattavasti testauksen teoriapohja, jonka jälkeen luvussa 3 puolestaan tutustutaan mallipohjaiseen testaukseen ja mallipohjaisen testauksen erilaisiin vaiheisiin. Luvussa 4 esitellään, millaisia mallipohjaiseen testaukseen soveltuvia työkaluja on tarjolla testaajien käyttöön. Näiden jälkeen luvussa 5 suoritetaan kirjallisuuskatsaus ja kootaan yhteen kirjallisuudesta löytyneet mallipohjaisen testauksen tarjoamat hyödyt ja haasteet. Lopuksi luvussa 6 pohditaan kirjallisuuskatsauksen myötä esiin nousseita hyötyjä ja haasteita sekä mahdollisia tulevaisuuden näkymiä ja kehityskohtia.

2 Testaus

Ohjelmistokehityksen yksi iso ja merkittävä osa on testaus, jolla pyritään saamaan vakuus toteutettujen ohjelmiston laadusta ja luotettavuudesta [Jena *et al.*, 2020]. Käytännössä testaus voidaan jakaa kahteen tapaan, joita ovat manuaalinen ja automaattinen testaus [Garousi ja Elberzhager, 2017]. Testaaja suorittaa testit itse manuaalisesti tai hyödyntää jotakin apuohjelmaa, joka suorittaa testit automaattisesti.

Testauksen tulee olla loputon osa ohjelmistojen kehitystä ja jatkua aina kehityksen alkuvaiheista ohjelmiston elinkaaren loppuun asti [Brar ja Kaur, 2015]. Ohjelmistokehityksen on tarkoitus tuottaa hyvin testattuja ohjelmia, joissa on mahdollisimman vähän virheitä. Testauksessa ei pelkästään tule keskittyä ohjelmistojen loogisesti oikeaan toimintaan, vaan myös tarkastella ohjelmistolle annettujen vaatimusten toteutumista. [Mohanty *et al.*, 2017]

Testaus ei ole vain testien kirjoittamista ja toteuttamista, vaan testaus on prosessi, johon kuuluvat tietyt perusteet [Kramer ja Legeard, 2016]. Testausprosessin perusteet on kuvattu kuvassa 1.



Kuva 1. Testausprosessi [Kramer ja Legeard, 2016].

Testausprosessin vaiheita ovat suunnittelu, testattavan kohteen analysointi ja testauksen suunnittelu, testitapausten kuvaaminen ja toteutus, saatujen testitulosten ja testauksen lopettamisehtojen arviointi ja raportointi sekä testauksen lopettaminen. Kun suunnitellut testit on suoritettu, tulee pohtia, tarvitaanko lisää testausta vai täyttyivätkö testaukselle annetut vaatimukset. Tämän jälkeen raportoidaan, missä mahdollisesti testausta vielä tarvitaan, miten testausta tulee muuttaa tai että testaus täyttää lopettamisehdot. Tämän jälkeen testausprosessi lopetetaan. Jos ilmeni, että lisätestausta tarvitaan, aloitetaan testausprosessi uudestaan ja toistetaan sitä niin kauan, kunnes lopettamiskriteerit

täyttyvät. Koko prosessin ajan tulee tarkkailla edistymistä ja arvioida, jatketaanko prosessia suunnitellusti vai onko jossain prosessinvaiheessa ilmennyt jotain, jonka vuoksi prosessi tulisi keskeyttää. Testausprosessia tulee hallita koko ajan.

Gopalaswamy ja Srinivasan [2009] toteavat kuluttajien ohjelmistoihin kohdistuvien vaatimusten nousseen merkittävästi viime vuosien aikana samalla, kun ohjelmista on tullut kaikkialla läsnä olevia ja joiden käytöltä ei enää nykymaailmassa voi välttyä. Näin ollen testauksen merkitys varmasti nousee ja uusien tehokkaampien ja nopeampien testausmenetelmien kehitys lisääntyy, jotta kuluttajien tiukkoihin vaatimuksiin pystytään vastaamaan entistä paremmin. Voidaan kuitenkin todeta, että mitä aikaisemmin ohjelmistokehityksessä testaus aloitetaan, sitä vähemmän virheet ehtivät monistua ja aiheuttaa seurannaisesti uusia virheitä [Brar ja Kaur, 2015]. Patton [2005] toteaa, että ohjelmistoprojektin koosta huolimatta eniten ohjelmointivirheitä muodostuu määrittelyvaiheessa. Aikanaan Kaner ja muut [1999] ovat todenneet, että jokaisessa ohjelmistossa on ja tulee olemaan virheitä, sillä täydellistä testausta ei pienillekään ohjelmille pystytä järkevässä ajassa ja järjellisillä resursseilla toteuttamaan. Tähän liittyen Umar [2019] tuo esiin, ettei nykypäivänkään laitteilla välttämättä pystytä löytämään kaikkia virheitä. Testauksella pyritään minimoimaan esille tulevat virheet, jotka estävät ohjelmiston oikeellisen toiminnan ja käyttämisen sujuvasti. Testaajan keskeisin tehtävä on etsiä ohjelmointivirheet mahdollisimman aikaisessa vaiheessa ja varmistua, että virheet tulee korjattua [Patton, 2005].

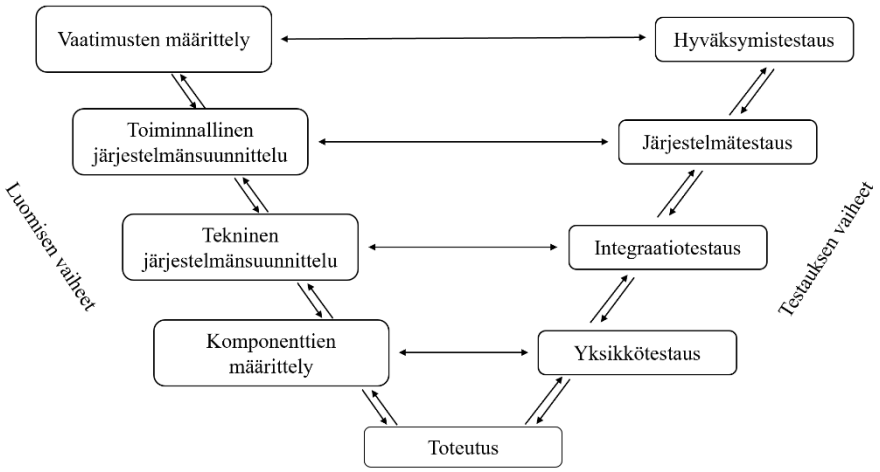
Vance [2013] mainitsee, ettei täydelliseen testaukseen ole olemassa varsinaista kaavaa tai yhtä oikeaa tapaa toteuttaa testaus, vaan testaajan on joskus jopa intuitiivisesti luotava testejä aiempien kokemusten pohjalta. Tästä huolimatta on olemassa tiettyjä periaatteita ja ohjeita, joita testaukseen liittyen on hyvä muistaa ja ottaa huomioon. Testauksen perusperiaatteet ovat [Gopalaswamy ja Srinivasan, 2009]:

1. Testauksen tavoitteena on löytää virheet ennen kuin varsinaiset käyttäjät löytävät ne.
2. Ohjelmistojen testauksella voidaan todentaa olemassa olevat virheet, muttei pystytä osoittamaan, ettei ohjelmistossa varmasti ole yhtään virhettä eli täydellinen testaus ei ole mahdollista.
3. Testaus kulkee mukana koko ohjelmiston elinkaaren ajan eikä ole ainoastaan kaaren viimeinen vaihe.
4. Testaajan tulee ymmärtää syyt suoritettavien testien taustalla.
5. Testi itsessään tulee testata ennen kuin se ajetaan varsinaisessa ohjelmassa.
6. Testejä tulee päivittää jatkuvasti, jottei virheet jää huomaamatta niille kehittyvän testejä vastustavan vastustuskyvyn vuoksi.
7. Virheet esiintyvät monesti tiedon siirtymiskohdissa ja osien yhdistymiskohdissa eli saattueissa ja klustereissa, joten testauksen tulisi keskittyä näihin kohtiin.

8. Testaamiseen sisältyy myös virheiden ennaltaehkäiseminen.
9. Testaus on tasapainottelua virheiden ennaltaehkäisemisen ja löytämisen kanssa.
10. Älykäs ja hyvin suunniteltu testausautomaatio on avain testauksen hyötyjen realisointiin.
11. Testaus vaatii taitavia ja sitoutuneita ihmisiä, jotka uskovat itseensä ja osaavat toimia yhdessä.

Myöhemmin tarkemmin esiteltävä mallipohjainen testaus pyrkii erityisesti tarttumaan kohtaan 10 ja luomaan testausta merkittävästi auttavaa automaatiota. Hyvästä automaatiosta huolimatta ei sovi unohtaa perusperiaatteiden kohtia 4 ja 5. Testaajan tulee ymmärtää mallia luodessaan, mitä ollaan testaamassa ja mitä testillä halutaan saavuttaa. Mallipohjaisetkin testit tulee testata ennen varsinaista ohjelmistolle tehtävää ajoa. Testauksen perusperiaatteet on hyvä muistaa testauksen tyypistä riippumatta.

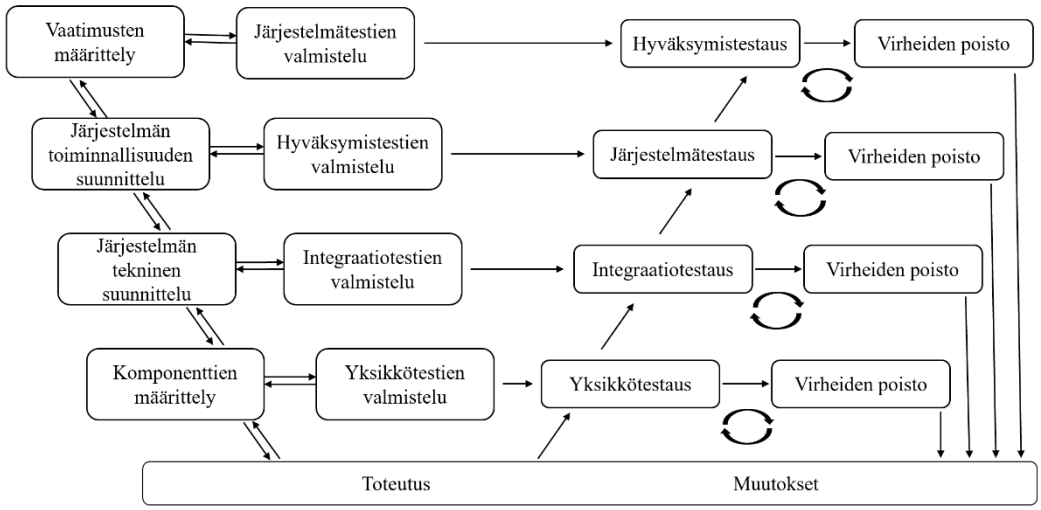
Testauksen osuutta ohjelmistokehityksessä kuvataan V-mallin avulla [Baker *et al.*, 2008]. V-malli on esitetty kuvassa 2.



Kuva 2. Testauksen V-malli [Baker *et al.*, 2008].

Testauksen V-mallissa vasemmalla puolella on kuvattu ohjelmiston kehityksen määrittelyjen vaiheet alimpana olevaan toteutukseen saakka. Luomisen vaiheita ovat vaatimusten määrittely, toiminnallinen järjestelmäsuunnittelu, tekninen järjestelmäsuunnittelu ja komponenttien määrittely. Oikealla puolestaan on kuvattu testauksen osat yhdistettynä määrittely- ja suunnitteluvaiheisiin. Testauksen vaiheita V-mallissa ovat yksikkötestaus, integraatiotestaus, järjestelmätestaus ja hyväksymistestaus. Määrittelyvaiheista saadaan testausvaiheiden vaatimukset, jotta tiedetään, mitä tulee testata ja millaisia tuloksia testeistä pitäisi tulla. Esimerkiksi hyväksymistestauksen halutut tulosteet tulevat suoraan vaatimusmäärittelyistä.

Testauksen V-mallin pohjalta on kehitetty tarkempi kuvaus W-malli, jossa on otettu mukaan testien suunnittelu ja virheidenpoistovaiheet [Baker *et al.*, 2008]. W-malli on esitetty kuvassa 3.



Kuva 3. Testauksen W-malli [Baker *et al.*, 2008].

Testauksen W-mallissa on laajennettu testauksen V-mallia tarkentamalla testauksen vaiheita. Testauksen määrittely- ja suunnitteluvaiheisiin on lisätty testien valmistelu ja puolestaan toteutukseen on lisätty toteutuksen muuttaminen. Tämä antaa itsessään jo tarkemman kuvan koko prosessista. Lisäksi uutena V-malliin on tuotu myös testauksessa löydettyjen virheiden poisto ennen siirtymistä testauksen seuraavaan vaiheeseen. Tästä käy ilmi paremmin, kuinka ohjelmaa muokataan koko ajan testauksen edetessä, eikä ainoastaan siinä vaiheessa, kun testaus on kokonaisuudessaan jo suoritettu. Virheet pyritään muokkaamaan pois yksitellen koko ajan ohjelmaa kehittäen.

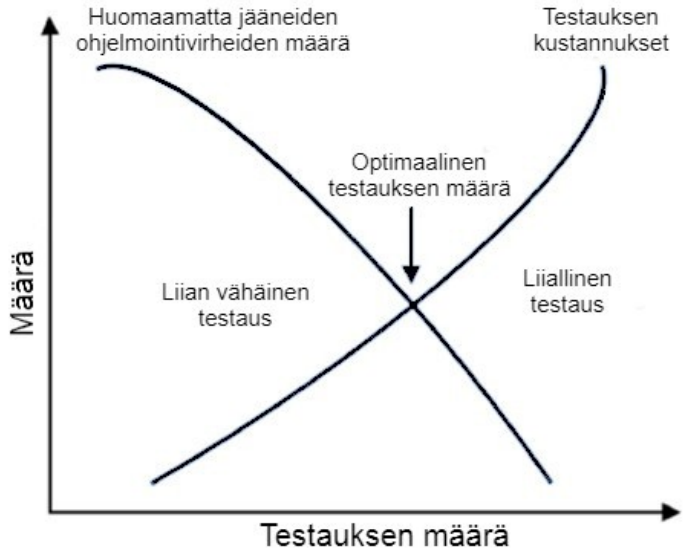
Pattonin [2005] mukaan testausta voidaan lähestyä kolmella erilaisella tavalla, jotka ovat valkolaatikko- (white-box testing), mustalaatikko- (black-box testing) ja harmaalaatikkotestaus (grey-box testing). Valkolaatikkotestauksessa testaajalla on pääsy testattavan kohteen koodiin ja rakenteeseen. Koodin tunteminen ja tarkastelu voivat auttaa tunnistamaan paremmin tilanteita, joissa virheitä saattaisi olla. Toisaalta tässä läpinäkyvyydessä piilee myös riski objektiivisuuden katoamiseen, jolloin testit saattavatkin alkaa vastaamaan ohjelmiston mahdollista virheellistä toimintaa. Valkolaatikkotestauksesta voidaan käyttää myös nimityksiä kirkaslaatikkotestaus (clear-box testing) tai lasilaatikkotestaus (glass-box testing). Vaihtoehtoiset nimitykset juontuvat juuri yllä mainitusta koodin ja rakenteen läpinäkyvyydestä testaajalle. Valkolaatikkotestausta voidaan käyttää testauksen kaikissa vaiheissa.

Mustalaatikkotestauksessa puolestaan ei tunneta lainkaan ohjelmiston sisäistä rakennetta tai varsinaista koodia ohjelman sisällä, vaan ainoastaan tiedetään, mitä ohjelmiston kuuluisi tehdä. Näin ollen mustalaatikkotestauksella testataan ohjelmiston toiminnallisuutta menemättä toteutuksen tasolle. Tarkoituksena on määritellä täyttääkö toiminnallisuus käyttäjän kanssa määritellyt vaatimukset. Pyritään selvittämään, antavatko tietyt syötteet halutut tulosteet, muttei tarkemmin tiedetä, mistä saadut tulosteet tulevat. Mustalaatikkotestauksesta saatetaan yllä mainituista syistä käyttää myös nimityksiä toiminnallisuustestaus (functional testing) tai käyttäytymistestaus (behavioral testing). Kuten valkolaatikkotestausta niin myös mustalaatikkotestausta voidaan toteuttaa kaikilla testauksen tasoilla.

Harmaalaatikkotestaus on noussut yhdeksi tekniikaksi valkolaatikko- ja mustalaatikkotestauksien rinnalle ja nimensäkin mukaisesti se on sekoitus näitä kahta tekniikkaa. Harmaalaatikkotestauksessa tunnetaan ohjelmiston sisäinen rakenne osittain eli paremmin kuin mustalaatikkotestauksessa, muttei yhtä hyvin kuin valkolaatikkotestauksessa. Testaus vastaa tyyliltään enemmän mustalaatikkotestausta, koska rakenteen ja toiminnan tuntemus on edelleen rajoitettua.

Valkolaatikko-, mustalaatikko- ja harmaalaatikkotestauksien lisäksi Patton [2005] mainitsee staattisen testauksen ja dynaamisen testauksen. Dynaaminen testaus on ohjelmiston ajamista ja käyttämistä, kun taas staattinen testaus puolestaan konkreettisen käytön sijaan testaa ohjelmistoa tarkastelemalla ja tutkimalla.

Ohjelmistokehityksessä tulee testata ohjelmistoa riittävästi, mutta liiallinen tai toisaalta liian vähäinenkään testaus eivät ole kestäviä ratkaisuja. Jokaiselle ohjelmistoprojektille on olemassa optimaalinen määrä testausta, mutta aina ei ole itsestään selvää, mikä optimaalinen määrä testausta on. Patton [2005] havainnollistaa tätä kuvassa 4.



Kuva 4. Optimaalinen testauksen määrä [Patton, 2005].

Kuvaajassa oleva optimaalisen testauksen määrää kuvaava piste on teoreettinen, mutta se kuvaa, että jokaisessa ohjelmistoprojektissa on löydettävissä optimaalinen testauksen määrä huomaamatta jääneiden ohjelmointivirheiden määrän ja testausten kustannusten tasapainopisteestä. Testauksen määrän ollessa vähäistä kustannukset pysyvät maltillisina, mutta toisaalta taas virheitä jää luvattoman paljon huomaamatta. Puolestaan testauksen määrän kasvaessa kohti täydellistä testausta löydettyjen virheiden kustannukset kasvavat kohtuuttoman suuriksi. Liikaa ei siis kannata testata, mutta liian vähäinenkään testaus ei ole hyvä ja voi tulla todella kalliiksi, kun huomaamattomia virheitä joudutaan jälkikäteen korjaamaan.

2.1 Yksikkötestaus

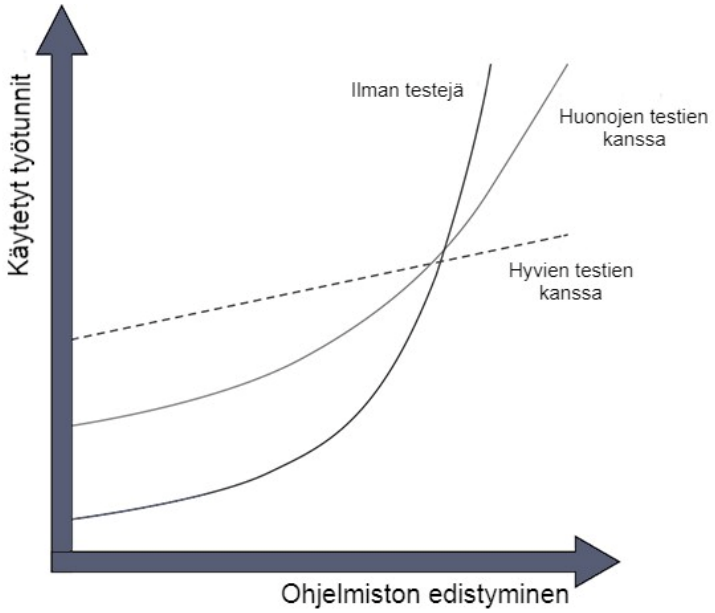
Yksikkötestaus toteutetaan yleensä varhaisessa vaiheessa ohjelmistojen testausta, mutta kyseistä testausta voi esiintyä ohjelmistojen kehityksessä lähes missä kohtaa vain. Aina kun uusi palanen tai pieni osa ohjelmistoa luodaan, tulee se testata erikseen yksikkötesteillä. Yksiköllä tarkoitetaan tarkkaan määritettyä ohjelmiston osaa, kuten esimerkiksi luokkaa tai funktiota. Yksikkötestauksen tarkoitus on varmistaa, että kyseinen osa toimii toivotusti annettujen vaatimusten mukaan. [Mathur, 2013]

Yksikkötestauksessa testitapausten luominen, testien toteuttaminen ja saatujen tulosten tulkinta vaativat hyvää teknistä osaamista Näin ollen on yleensä perusteltua, että yksikkötestauksen suorittaa ohjelmoija itse, sillä kyseisen yksikön luojalla on paras mahdollinen tuntemus yksikön koodista ja halutusta toiminnasta. Yksikkötestaus on myös kannattavaa suorittaa heti, kun yksi yksikkö on saatu ohjelmoitua, jolloin koodin sisältö ja haluttu toiminta ovat vielä hyvin muistissa. [Brar ja Kaur, 2015]

Testauksen tässä vaiheessa on syytä varmistua, ettei yksiköissä ole virheitä, sillä on selvää, että mahdolliset yksiköihin jäävät virheet tulevat moninkertaistumaan myöhemmissä kehityksen vaiheissa. Brarin ja Kaurin [2015] mukaan yksikkötestaukseen kuluva aika on suoraan riippuvainen yksikön koosta eli mitä isompi yksikkö on, sitä kauemmin testaukseen kuluu aikaa.

Khorikovin [2020] mukaan yksikkötestauksen keskeisin tavoite on mahdollistaa ohjelmistoprojektin kestävä kasvu ja ikään kuin sivutuotteena hyvän yksikkötestauksen seurauksena saadaan parempilaatuinen ja paremmin suunniteltu ohjelmisto. Kaikkia yksikkötestejä ei voida luoda samaan muottiin, vaan esimerkiksi osa testeistä keskittyy yleiseen ohjelmiston laatuun, ja toiset kenties kasvattavat virheidenhavaitsemiskykyä. Juuri ohjelmiston kestävän kasvun ja edistymisen kannalta on tärkeää, että testien tekoon käytetään alussa riittävästi aikaa ja varmistetaan testien korkea laatutaso. Näin varmistetaan pidemmällä aikavälillä ohjelmiston kasvun ja ohjelmistokehityksen eteneminen ilman, että siihen käytettävät työtunnit kasvaisivat niin paljon, että kasvu ja edistyminen pysähtyisivät

huonolaatuisten testien seurauksena. Tämä on esitetty kaaviona kuvassa 5. Yksikkötestien laatuun tulee siis keskittyä eikä ainoastaan määrään ja testien luomisen nopeuteen. [Khorikov, 2020]



Kuva 5. Testien laadun vaikutus käytettyihin työtunteihin ohjelmiston edistyessä [Khorikov, 2020].

Khorikov [2020] esittelee kaksi eri koulukuntaa yksikkötestaukseen: klassisen ja lontoolaisen koulukunnan. Koulukunnat eroavat toisistaan kolmella tavalla. Ensimmäinen eroavaisuus koulukuntien välillä on, että klassinen koulukunta pitää yksikkönä ohjelmiston luokkaa tai luokkien joukkoa, kun puolestaan lontoolaisessa koulukunnassa yksikkö on vain yksi ohjelmiston luokka eikä koskaan joukko luokkia. Toinen eroavaisuus on eristysominaisuudessa. Klassinen koulukunta lähtee siitä, että yksikkötestit itsessään tulee eristää toisistaan, kun lontoolaisessa koulukunnassa testattava järjestelmä tulee eristää muista sen kanssa yhteydessä olevista osasista yksikkötestauksen ajaksi. Kolmas ja samalla viimeinen eroavaisuus on testituplien käytössä. Testituplilla voidaan eristää testattavan koodin käyttäytyminen muiden koodin osien käyttäytymisestä, joista testattava koodi on riippuvainen. Testituplilla siis korvataan objekteja testaamisen ajaksi, jotta keskinäisestä riippuvuudesta seuraava käyttäytyminen saadaan rajattua testin ulkopuolelle. Klassisessa lähestymistavassa testituplia käytetään vain jaettujen riippuvuuksien osalta, joita esimerkiksi ovat tietokannat, jotka ovat riippuvaisia monesta ohjelmiston osasta. Lontoolaisessa lähestymistavassa testituplia tulee käyttää jaettujen riippuvuuksien lisäksi myös niissä yksityisissä riippuvuuksissa, jotka ovat muuttuvia. Näin ollen lontoolaisessa koulukunnassa testituplien ulkopuolelle rajataan vain muuttumattomat riippuvuudet. [Khorikov, 2020]

Yksikkötestauksen haasteeksi voidaan todeta yksiköiden tunnistaminen testejä varten, sillä testattavan kohteen pitäisi monesti olla yhdistynyt joukko yksiköitä eikä vain yksittäinen yksikkö [Runeson, 2006]. Lisäksi Runeson [2006] toteaa yksikkötestien automatisoinnin aiheuttavan ongelmia, sillä automatisoitujen yksikkötestien ylläpitäminen vaatii paljon vaivaa. Aina ei myöskään ole itsestään selvää, mitä testeillä tulisi tarkistaa ja testata [Daka ja Fraser, 2014].

2.2 Integraatiotestaus

Integraatiotestauksen pääasiallinen tarkoitus on tarkastella yksikkötestauksessa testattujen yksiköiden ja moduulien välisten rajapintojen toimivuutta [Leung ja White, 1990 ja 2002; Jan *et al.*, 2016]. Janin ja muiden [2016] mukaan integraatiotestauksessa pyritään kehittämään ohjelmiston rakennetta samalla, kun rajapintojen virheet pyritään poistamaan. Tässä vaiheessa ei enää kiinnitetä huomiota moduulien varsinaiseen toimintaan, vaan niiden oletetaan jo toimivan toivotulla tavalla yksikkötestauksen jäljiltä. On kuitenkin selvää, että koodin osasia saatetaan joutua yksikkötestaamaan uudestaan, jos integraatiotestauksessa havaitaan, etteivät yksiköt ja niiden käyttöliittymät toimi yhdessä halutulla tavalla. Basilin ja Perriconnen [1984] mukaan jopa 40 % ohjelmistojen sisältämistä virheistä voidaan liittää integraatiosta johtuviin ongelmiin. Yleisimmin havaitut virheet ovat seurausta jo määrittelyvaiheessa tulleista väärinymmärryksistä yksikköjen määritysten ja halutun toiminnan osalta.

Integraation ja integraatiotestauksen toteuttamiseen on kaksi erilaista pääasiallista lähestymistapaa. Ensinnäkin moduuleja voidaan lisätä yksi kerrallaan jo aiemmin testattuun ohjelmaan ja suorittaa testaus jokaisen lisäyksen jälkeen. Toisessa vaihtoehdossa kaikki moduulit lisätään kerralla ja suoritetaan testaus lopuksi. [Leung ja White, 1990]

Brarin ja Kaurin [2015] mukaan integraatiotestaus voidaan laajentaa lähestymistapojen pohjalta neljään erilaiseen toteutustapaan Top Down-, Bottom Up-, Big Bang- ja Sandwich-menetelmiin. Top Down -menetelmässä yksiköt yhdistetään toisiinsa hierarkiassa korkeimmasta yksiköstä alimmalle tasolle edeten. Bottom Up -menetelmässä puolestaan toimitaan päinvastoin eli lähdetään matalimmalta tasolta ja päädytään ylimmälle tasolle. Big Bang -toteutustapa on oikeastaan suora sovellus toisesta lähestymistavasta, jossa kaikki uudet yksiköt yhdistetään jo olemassa olevaan yksikköön kerralla. Yksiköitä ei siis Big Bang -toteutuksessa lisätä missään varsinaisessa järjestyksessä kuten Top Down- ja Bottom Up -toteutuksissa. Sandwich-menetelmä puolestaan yhdistää Top Down- ja Bottom Up -menetelmät. Sandwich-toteutuksessa yksiköitä lähdetään yhdistelemään molemmista suunnista eli ylimmältä ja alimmalta tasolta ja lopulta kohdataan jossain puolessa välissä ja yhdistetään nämä jo saadut integraatiot yhdeksi kokonaisuudeksi. [Brar ja Kaur, 2015]

Näiden neljän toteutustavan lisäksi Leung ja White [2002] ovat esittäneet vielä viidennen toteutustavan Build. Tässä toteutustavassa yksiköitä aletaan yhdistellä keskittymällä tärkeimpiin yksiköihin ja niiden yhdistämiseen ensin, jonka jälkeen laajennetaan muiden yksiköiden yhdistämiseen [Leung ja White, 2002]. Esimerkiksi voidaan yhdistää ensiksi jokin polku ylimmältä tasolta alimmalle tasolle saakka, jos nämä yksiköt koetaan tärkeimmiksi. Tärkein asia on siis aloittaa toteutettavan ohjelmiston kannalta kriittisimpiä toiminnallisuuksia sisältävistä yksiköistä ja edetä vähemmän kriittisiin, jolloin varmistutaan paremmin siitä, että ainakin tärkeimmät toiminnallisuudet ja yksiköt saadaan toimimaan integraatiossa keskenään mahdollisimman nopeasti. Tämä myös nopeuttaa ohjelmiston käyttöönottoa, jos se ollaan toteuttamassa porrastetusti niin, että aiempaan versioon lisätään uusia ominaisuuksia tietyissä sykleissä.

Integraatiotestauksen haasteena esimerkiksi yksikkötestaukseen verrattuna on se, ettei mahdollisia integraatiotestauksessa havaittuja virheitä pystytä niin helposti paikallistamaan ja sen myötä korjaamaan [Brar ja Kaur, 2015]. Havaitaan kyllä, että on virhe jossain ja yksiköt eivät toimi keskenään integraatiossa oikealla tavalla, muttei tiedetä, mikä virheen aiheuttaa. Tämä tekee integraatiotestauksesta niin haastavaa ja aikaa vievän prosessin. Yksikkötestauksessa voidaan aina olla varmoja, että virhe on kyseisessä yksikössä, jota testataan. Haaste luonnollisesti kasvaa ja vaikeutuu, mitä useampaa yksikköä integroidaan toisiinsa. Integraatiotestauksessa virhe saattaa myös olla laajemmalla alueella kuin vain yhdessä yksikössä [Brar ja Kaur, 2015].

2.3 Järjestelmätestaus

Järjestelmätestausta toteutetaan yleensä silloin, kun ohjelmisto alkaa olla valmis. Järjestelmätestauksella pyritään varmistamaan järjestelmän oikeellinen toiminta ja kunnollinen integraatio [Jan *et al.*, 2016]. Sommerville [2016] kuvailee järjestelmätestauksen tavoitteeksi samat ohjelmiston oikeellisen toiminnan varmistamisen ja komponenttien yhteensopivuuden integraatiossa, mutta lisäksi hän mainitsee myös rajapintojen kautta oikeaaikaisen oikeellisen tiedon liikkumisen. Järjestelmätestauksessa ei tarvitse tuntea lähdekoodia, vaan on tarkoitus suorittaa kaikenlaisia testejä ja näin kattaa ohjelmiston toiminnallisuus [Jan *et al.*, 2016].

Janin ja muiden [2016] mukaan erilaisia järjestelmätestauksen tekniikoita ovat toipumistestaus, turvallisuustestaus, graafisen käyttöliittymän testaus ja yhteensopivuustestaus. Toipumistestauksessa tarkoituksena on saada ohjelmisto epäonnistumaan, jotta voidaan seurata, kuinka kauan ohjelmistolla menee toipua virhetilanteesta vai toipuuko se ollenkaan ja miten toipuminen suoritetaan. Toipumisen ollessa automatisoitua tarkastellaan ajan lisäksi myös rakenteellisia ominaisuuksia. Turvallisuustestauksessa tavoitteena on löytää ja tukkia järjestelmän mahdolliset porsaanreiät ja haavoittuvuudet, jottei niiden kautta synny tietovuotoja tai muuta haittaa käyttäjälle. Graafisen käyttöliittymän testauk-

sessa puolestaan testataan käyttöliittymän elementtien, kuten esimerkiksi valikkojen, kuvakkeiden ja painikkeiden, määritysten mukainen toiminta. Yhteensopivuustestaus poikkeaa muista siinä, ettei se ole varsinaisesti toiminnallisuuksien testausta, vaan testauksessa tarkastetaan järjestelmän yhteispelaamista esimerkiksi erilaisten käyttöjärjestelmien ja tietokantojen kanssa. [Jan *et al.*, 2016]

Osa järjestelmän toiminnallisuudesta tulee esiin vasta, kun komponentteja yhdistetään toisiinsa. Järjestelmätestauksessa tulee testata myös tämä niin sanottu kehittyvä käyttäytyminen. Jotkin kehittyvät toiminnallisuudet ovat ei-toivottuja ja testauksessa pitää oikeellisen toiminnan testauksen lisäksi pystyä tunnistamaan, mikä on toivottu ja mikä ei-toivottu toiminnallisuus. Järjestelmätestauksessa tulee siis keskittyä testaamaan komponenttien välistä vuorovaikutusta. Tällaisen testauksen avulla saadaan esiin ohjelmointivirheet, jotka ilmenevät vasta komponenttien välisen vuorovaikutuksen seurauksena. Tehokas lähestymistapa komponenttien vuorovaikutuksen testaukseen on käyttötapauspohjainen testaus. Yleensä käyttötapauksia ajettaessa tarvitaan useampaa komponenttia tai objektia, jolloin vuorovaikutuksen seuraukset tulevat esiin ja testattua. [Sommerville, 2016]

Kuten monessa muussakin testauksessa myös järjestelmätestauksen haasteena on tietämys siitä, milloin on testattu liikaa ja milloin liian vähän. Kaikkien tapauksien läpikäyminen eli täydellinen testaus ei ole mahdollista, joten on keskityttävä mahdollisten testitapausten osajoukkoihin. Osajoukkojen valinta voidaan toteuttaa esimerkiksi perustuen järjestelmän käyttökokemuksiin. [Sommerville, 2016]

2.4 Hyväksymistestaus

Hyväksymistestaus on testauksen vaihe, jossa kehittäjät ja asiakkaat kohtaavat. Asiakkaat määrittelevät hyväksymistestit, jotta varmistutaan koko järjestelmän oikeellisesta toiminnasta. Hyväksymistestaus ikään kuin edustaa asiakkaan vaatimuksia ja odotuksia järjestelmää ja ohjelmistoa kohtaan. Käytännössä hyväksymistestauksella pyritään varmistumaan oikeasta toiminnasta ennen ohjelmiston julkaisua tai luovuttamista asiakkaalle, muttei sovi ajatella, ettei hyväksymistestausta tulisi tehdä myös projektin aikana, kuten esimerkiksi yksikkötestaustakin. [Miller ja Collins, 2001]

Hyväksymistestaus antaa kolme merkittävää tulosta projektin tilasta. Ensinnäkin hyväksymistestit paljastavat ongelmat, jotka yksikkötestauksessa jäi havaitsematta. Toisekseen hyväksymistestit sisäistävät käyttäjän antamat vaatimukset ja mittaavat saatujen tulosten pohjalta, kuinka hyvin järjestelmä vastaa noita vaatimuksia. Lisäksi hyväksymistestit antavat myös kuvauksen siitä, kuinka valmis järjestelmä vaatimuksiin nähden todellisuudessa on. [Miller ja Collins, 2001]

Millerin ja Collinsin [2001] mukaan hyväksymistestaus on avain ohjelmistoprojektien onnistumiseen ja asiakkaiden tyytyväisyyteen. Manuaalinen hyväksymistestaus voi Haugsetin ja Hanssenin [2008] mukaan usein viedä paljon aikaa ja tulla melko kalliiksi,

joten automaatiota tulisi hyödyntää. Miller ja Collins [2001] ovat samaa mieltä ja uskovat automatisoidun hyväksymistestauksen tuovan merkittävästi lisäarvoa projektille. Haugset ja Hanssen [2008] huomauttavat, ettei automatisoitujen testien kirjoittaminen ja ylläpitäminen ole sekään ilmaista ja tulee tarkkaan harkita hyötyjen ja kustannusten tasapainoa. Miller ja Collins [2001] mielestä automaatio tulee kannattavaksi pitkässä juoksussa, jos pystytään kehittämään automatisoidun hyväksymistestauksen perusrunko, jota voidaan käyttää uudelleen projektin aikana ja mahdollisesti myös muissa myöhemmissä projekteissa.

Yksi hyväksymistestauksen muoto on käyttäjän hyväksymistestaus. Tämä on viimeinen testauksen vaihe ennen ohjelman varsinaista käyttöönottoa. Tähän hyväksymistestauksen muotoon voidaan viitata myös termillä loppukäyttäjätestaus [Hambling ja Goethem, 2013]. Ohjelma voidaan julkaista, kun käyttäjän hyväksymistestauksesta saadut tulokset vastaavat tuloksille annettuja hyväksymisvaatimuksia ja ohjelmiston todetaan niiltä osin toimivan oikealla tavalla. Hyväksymisvaatimuksia voivat olla esimerkiksi järjestelmän tai joidenkin komponenttien tietynlainen haluttu toiminta. Yleensä käyttäjän hyväksymistestauksessa nimenomaan käyttäjä valitsee testitapaukset, kun taas hyväksymistestauksessa yleensä testitapaukset voi valita monesti myös asiakas tai kehittäjä. [Leung ja Wong, 1997] Hamblingin ja Goethemin [2013] mukaan käyttäjän hyväksymistestauksesta tekee uniikin se, että järjestelmää testaa käyttäjä ja tuloksia verrataan käyttäjän tarpeisiin eikä teknisiin vaatimuksiin.

Leung ja Wong [1997] esittelevät kolme erilaista strategiaa toteuttaa käyttäjän hyväksymistestausta. Nämä ovat käyttäytymisperusteinen hyväksymistestaus, operaatioperusteinen hyväksymistestaus ja mustalaatikkotestauksellinen lähestymistapa käyttäjän hyväksymistestaukseen. Käyttäytymisperusteisessa strategiassa testitapaukset suunnitellaan testaamaan järjestelmää käyttäjien näkökulmasta keskittyen järjestelmän ulkoiseen käyttäytymiseen eli järjestelmän käyttäjälle antamiin tulosteisiin. Operaatioperusteisessa hyväksymistestauksessa puolestaan keskitytään järjestelmän toimintaprofiileihin, joiden pohjalta testit luodaan. Kolmannessa strategiassa mustalaatikkotestauksen menetelmässä puolestaan testitapaukset luodaan ulkoisten tai toiminnallisten ohjelmistojärjestelmän vaatimus erittelyn pohjalta. Testitapaukset luodaan yleensä käyttäjän ja kehittäjän yhteistyössä. [Leung ja Wong, 1997]

Käyttäjän hyväksymistestaus on lähellä käyttäjätestausta ja sen ehdoton vahvuus on suora kontakti ohjelmiston tuleviin käyttäjiin, jolloin todennäköisemmin ohjelmisto toimii halutulla tavalla. Joissain tilanteissa virheet saattavat olla seurausta siitä, ettei määritellyt ja kehittäjien näin olettama ohjelmiston oikeellinen toiminta vastaakaan käyttäjän todellista ajatusta oikeellisesta toiminnasta [Davis ja Venkatesh, 2004]. Vaikka yleensä mahdollisten virheiden korjaaminen tässä vaiheessa on melko kallista, on sekin parempi

vaihtoehto kuin kokonaan hukkaan menevä ohjelmisto, jota kukaan ei halua käyttää. Hyväksymistestaus on koko ohjelmistokehitysprosessin kannalta yhtä tärkeä testauksen vaihe kuin muutkin testauksen osa-alueet [Hambling ja Goethem, 2013].

2.5 Regressiotestaus

Ohjelmiston elinkaaressa regressiotestaus suoritetaan yleensä ylläpitovaiheessa eli vasta siinä kohtaa, kun ohjelmistosta esimerkiksi julkaistaan uusi versio tai alkuperäistä jo testattua toteutusta on muokattu. Regressiotestauksessa testataan siis kohteena olevaa ohjelmistoa uudestaan ainakin osaltaan jo aiemmin käytetyillä testeillä, kun ohjelmistoa on muutettu jollain tavalla tai siihen on lisätty joitain uusia ominaisuuksia tai palasia. Uudelleentestauksella pyritään estämään uusien virheiden päätyminen jo testattuun ja toimivaksi todettuun koodiin. [Duggal ja Suri, 2008] Agrawal ja muut [1993] toteavat regressiotestauksen tarkoituksen olevan varmistuminen siitä, etteivät tehdyt virheiden korjaukset tai uudet ohjelmistoon lisättävät toiminnallisuudet vaikuta haitallisesti aiemmilta versioilta periytyviin jo ennestään toimiviin toiminnallisuuksiin. Regressiotestaus pyrkii siis varmistamaan uudelleen, ettei mitään aiemmin testattua ole hajonnut tai toiminnallisesti muuttunut uusien lisäysten tai muutosten myötä [Wong *et al.*, 1997]. Niinpä regressiotestausta voidaan pitää jopa osittaisena vaatimuksena, mikä tulee tehdä ohjelmiston tai systeemin uusille versioille [Agrawal *et al.*, 1993].

Wahl ja Li [1999] korostavat regressiotestauksen merkitystä osana kehitettävän ohjelmiston elinkaarta ja toisaalta sitä, kuinka regressiotestauksen parantaminen laskee ja hillitsee kehitettävän ohjelmiston hintalapun kasvua ja kertyviä kustannuksia. Engströmin ja muiden [2010] mukaan regressiotestaus ei ole pelkästään tärkeää vaan jopa välttämätöntä organisaatioille, joiden kuluista suuri osuus menee ohjelmistojen kehittämiseen. Regressiotestaukseen ei kuitenkaan panosteta pelkästään taloudellisista syistä vaan, muita tärkeitä syitä regressiotestauksen kehitykseen ovat ohjelmiston luotettavuuden ja laadun parantaminen entisestään [Wahl ja Li, 1999].

Leung ja White [1989] jakavat regressiotestauksen kahteen tyyppiin, jotka ovat progressiivinen ja korjaava regressiotestaus. Jaottelu tehdään sen mukaan, onko ohjelman määrittelyä muutettu aiemmasta ohjelmiston määrittelystä. Progressiivisessa regressiotestauksen muodossa määrittelyä on muutettu, josta seuraa se, ettei samoja jo käytettyjä testejä voida suuressa määrin käyttää uudestaan, vaan myös uusia testejä on luotava muuttuneiden määrityksien pohjalta. Puolestaan korjaavassa regressiotestauksessa muutoksia määrittelyyn ei ole tehty, jolloin testitapauksiakin voidaan käyttää enemmän uudestaan ilman suurempia muutoksia. [Leung ja White, 1989]

Regressiotestauksessa on useita erilaisia tekniikoita. Duggalin ja Surin [2008] mukaan pääasiallisia tekniikoita on neljä, jotka ovat kaiken uudelleentestaus, regressiotestien valinta, testitapausten priorisointi ja hybridilähestymistapa. Kaiken uudelleen testauksella

tarkoitetaan kaikkien jo olemassa olevien testien uudelleen ajamista muunnettuun ohjelmistoon. Aggarwal ja Singh [2005] toteavat tämän tekniikan olevan paljon aikaa vievä toimintatapa, jolloin kulutkin nousevat ja testaus kallistuu.

Regressiotestien valinnalla puolestaan tarkoitetaan testien rajaamista niin, ettei kaikkia testitapauksia käydä läpi. Regressiotestien valinnassa olemassa oleva testisarja jaetaan uudelleenkäytettäviin ja -testattaviin sekä vanhentuneisiin testitapauksiin, joiden lisäksi voidaan myös luoda uusia testitapauksia tarvittaessa kattamaan testaamattomia osia. Tässä menetelmässä testitapausten valinta toteutetaan kolmen kategorian pohjalta. Kategoriat ovat kattavuus, minimalisointi ja turvallisuus. Kattavuuden pohjalta valittaessa etsitään katettavat ohjelman osat, joita on muokattu, ja valitaan testit, jotka soveltuvat näihin kohtiin. Minimalisoinnissa testitapausten valinnat tehdään muuten samalla tavalla kuin kattavuusmenetelmässäkin, mutta valitaan pienin mahdollinen määrä testitapauksia. Aggarwal ja Singh [2005] huomauttavat, että minimalisoinnissa on suurempi riski sulkea pois virheen tuottava testitapaus. Turvallisessa valinnassa sen sijaan ei keskitytä kattavuuteen, vaan valitaan ne testitapaukset, jotka antavat eri tulosteen muokatussa ohjelmassa kuin ennen muokkausta ohjelma olisi näillä testitapauksilla antanut. [Duggal ja Surin, 2008]

Kolmannella tekniikalla eli testitapausten priorisoinnilla yritetään laskea regressiotestauksen kustannuksia valitsemalla tiettyjen mittauksien perusteella tärkeimpiä testitapauksia testattavaksi mahdollisimman aikaisin [Elbaum *et al.*, 2002]. Tavoitteena testitapausten priorisoinnissa on testitapauksia järjestämällä kasvattaa testisarjojen kykyä havaita virheitä testattavasta ohjelmistosta aikaisemmin ja samalla nopeammin. Testitapausten priorisointi voidaan jakaa kahteen tyyppiin eli yleiseen ja versiokohtaiseen testitapausten priorisointiin. Yleisessä tyypissä testitapaukset pyritään järjestämään testisarjan sisällä niin, että testisarjaa voitaisiin hyödyntää mahdollisimman hyvin virheiden löytämiseen ohjelmiston myöhemmissä versioissa. Versiokohtaisessa tyypissä puolestaan pyritään järjestämään testitapaukset niin, että testisarja tuottaisi parhaan mahdollisen tuloksen testauksen osalta tietyissä ohjelmiston versioissa. [Rothermel *et al.*, 2001]

Näiden kahden tyypin lisäksi Elbaum ja muut [2002] ovat esittäneet varsinaisia tekniikoita, joiden avulla testitapausten priorisointi voitaisiin tehdä näiden kahden tyypityksen sisällä. Näitä tekniikoita on kaikestaan 18 ja ne on jaoteltu kolmeen erilaiseen ryhmään [Elbaum *et al.*, 2002]. Ei kuitenkaan ole tämän tutkielman kannalta oleellista lähteä avaamaan näitä kaikkia, joten tyydyn vain esittelemään nämä kolme ryhmää ja antamaan niistä lyhyen kuvauksen jokaisesta. Testitapausten priorisoinnin tekniikat voidaan jakaa vertaileviin, lausetason ja toiminnallisuustason tekniikoihin [Elbaum *et al.*, 2002]. Vertailevissa tekniikoissa keskitytään vertailemaan testitapauksien järjestyksiä testisarjojen sisällä. Lausetason tekniikoissa puolestaan testitapaukset järjestetään koodin lauseisiin

pohjautuvan kattavuuden perusteella. Sen sijaan toiminnallisuustason tekniikoissa testitapausten järjestys pohjautuu puolestaan ohjelmiston toimintoihin ja niiden kattavuuksiin.

Mulkahainen [2019] toteaa, että testitapausten priorisointiin voidaan käyttää myös koneoppimista. Koneoppiminen on tekoälyn osa-alue, jossa järjestelmät voivat kokemuksen ja ajan myötä oppia ennustamaan haluttuja tuloksia ja suoriutumaan itsenäisesti toivotulla tavalla [Bell, 2020]. Mulkahaisen mukaan havaittujen vikojen määrän painotetulla keskiarvolla mitattuna testitapauksia priorisoitaessa koneoppimisen avulla saadaan parempia tuloksia kuin muilla priorisointimenetelmillä. Testitapausten priorisoinnin lisäksi koneoppimista voidaan hyödyntää myös testitapausten valinnassa [Mulkahainen *et al*., 2022].

Viimeinen eli neljäs regressiotestauksen tekniikka on hybriditekniikka, jossa käytännössä yhdistellään ja sovelletaan toisiinsa aiemmin esitettyjä tekniikoita eli regressiotestien valintaa ja testitapausten priorisointia [Duggal ja Surin, 2008]. Testitapauksien järjestys pyritään optimoimaan toteuttamalla priorisointia ja valintaa ristiin esimerkiksi ensiksi priorisoimalla testisarjan sisällä olevat testitapaukset, jonka jälkeen suoritetaan valinta testitapauksista ja priorisoidaan uudestaan valintojen pohjalta [Gupta ja Chauhan, 2011]. Zhang [2018] puolestaan esittää rakeisuuksien eli ohjelmiston testien tarkkuuden pohjalle rakentuvan hybridivaihtoehdon regressiotestauksien valintaan liittyen, missä yhdistetään menetelmän ja tiedoston testien tarkkuudet perinteisestä regressiotestauksen valinnasta. Tällä hybridimenetelmällä voidaan voittaa perinteinen regressiotestauksen valintamenetelmä testaukseen käytettävässä ajassa sekä offline- että online-tapauksissa. [Zhang, 2018]

2.6 Testiautomaatio

Yksinkertaisesti ilmaistuna testiautomaatio on testauksen muoto, jossa testit suoritetaan pääasiassa automaattisesti ilman varsinaisia testaajan toimenpiteitä. Nykyään testiautomaatiota voidaan hyödyntää lähes kaikissa testauksen vaiheissa, kuten esimerkiksi testien luonnissa ja virheiden raportoinnissa, eikä pelkästään testien suorittamisessa [Garousi ja Elberzhager, 2017]. Automaatiotestaus vaatii toimiakseen tiettyjä siihen suunnattuja apuvälineitä ja työkaluja kontrolloimaan ja suorittamaan testejä. Automaatiotestaus on kehittyvä ala ja uusia menetelmiä, tekniikoita ja työkaluja kehitetään nopealla tahdilla koko ajan lisää. [Oliinyk ja Olesiuk, 2019]

Onnistuneeseen testiautomaatioon tarvitaan järkevä testausprosessi, mutta tärkeintä ja jopa välttämätöntä on täsmällinen strategia testiautomaation suorittamiseksi oikealla tavalla ja oikeissa kohdissa testausprosessia. Järkevällä testiautomaatiostrategialla ei pelkästään pystytä välttämään monia ongelmia, vaan se antaa myös mahdollisuuden arvioida kustannustehokkuutta etukäteen, jolloin voitaisiin välttää turhia kuluja testauksen osalta. [Berner *et al.*, 2005]

Berner ja muut [2005] antavat neljä vaihetta, joiden pohjalta testiautomaatiota kannattaisi organisaatioissa ja projekteissa lähteä toteuttamaan. Ensimmäiseksi tulee määritellä testausstrategia, joka ottaa huomioon testattavan kohteen erityispiirteet ja omat laatuvaatimukset. Toimintoja tulee integroida jatkuvasti ylläpitääkseen ja parantaakseen testitapauksia. Toiseksi tulee määritellä testiautomaatiolle tavoitteet, jotka voivat esimerkiksi olla laadullisia tai kustannuksellisia. Kolmanneksi tulee valita lähestymistapa, jossa on monipuolisesti yhdistelty erilaisia testiautomaation lähestymistapoja, jotta päästään parhaaseen mahdolliseen tehokkuuteen projektin kannalta. Neljänneksi testiautomaatiota ja lähestymistapaa tulee jatkuvasti arvioida valittuihin tavoitteisiin peilaten ja näiden arvioiden myötä kehittää edelleen, jotta tavoitteet saavutettaisiin vieläkin paremmin ja tarkemmin. [Berner *et al.*, 2005]

Pocatilu [2002] ja Sneha ja Malle [2017] kuvaavat testiautomaation prosessin, jossa suunnitellaan testaus, mietitään testien ulkoasu, luodaan testit, toteutetaan testit ja lopuksi arvioidaan testien tulokset oraakkeleiden avulla vertaamalla saatuja tuloksia odotettuihin tuloksiin. Tällä prosessilla voidaan vähentää testaukseen liittyviä kuluja vähentämällä testauksen toteutukseen ja luomiseen käytettävää aikaa [Pocatilu, 2002]. Automaatiotestauksella voidaan suorittaa testejä merkittävästi nopeammin kuin testaaja itse pystyisi niitä manuaalisesti suorittamaan [Sharma ja Angmo, 2014]. Hayesin [2004] mukaan automaatiotestauksesta on kolme merkittävää hyötyä, jotka ovat ajan myötä kasvava kattavuus virheiden havaitsemiseksi ja vioista aiheutuvien kustannusten vähentämiseksi, testien helpompi toistettavuus ajan säästämiseksi ja markkinakustannusten vähentämiseksi sekä vipuvaikutus omien resurssien tuottavuuden parantamiseksi. Testiautomaatio mahdollistaa testisarjojen paremman toistettavuuden ja uudelleenkäytettävyyden myöhemmille ohjelmiston versioille, minkä lisäksi automaatiotestaus tekee testauksesta luotettavamman, sillä se toistaa sille annetut operaatiot täsmälleen samalla tavalla jokaisella ajokerralla [Sharma, 2014]. Kuten Wiklund ja muut [2017] toteavat, automaatiotestauksella pyritään vähentämään riskiä ihmisen eli testaajien tekemille virheille automatisoimalla toimintoja ja testejä.

Testiautomaatiosta on useita merkittäviä hyötyjä, mutta on syytä muistaa, ettei automaatiokaan sovi kaikkiin tilanteisiin. Automaatiotestaus tulee jatkossa korostumaan enemmän ja vähentämään manuaalista testausta, muttei koskaan täysin poistamaan manuaalisen testauksen tarvetta [Oliinyk ja Olesiuk, 2019]. Berner ja muut [2005] toteavat, että usein uudet virheet löydetään nimenomaan manuaalisella testauksella eikä automaatiotestauksella. Automaatiotestaus on luonteeltaan rakentava eikä niinkään tuhoava testauksen muoto. Kokenut testaaja pystyy kohdistamaan testauksen automaatiota paremmin inhimillisiin tekijöihin, kuten kehitystiimin ja teknologioiden heikkoihin kohtiin, virheiden havaitsemiseksi ja esiin nostamiseksi [Berner *et al.*, 2005]. Hayesin [2004] mukaan automaatiotestausta tulisi käyttää tilanteissa, joissa halutaan varmistua ennustettavissa

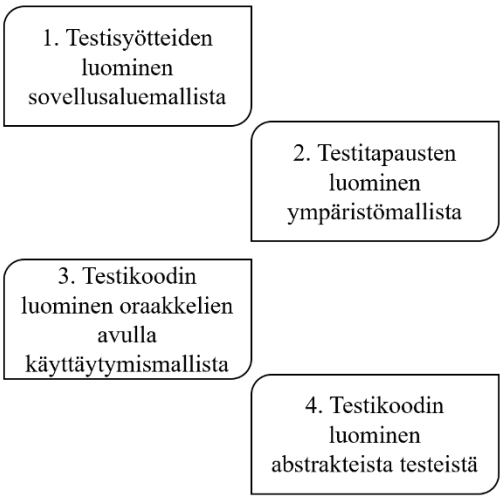
olevan toiminnallisuuden oikeellisuudesta ja manuaalista testausta puolestaan ennustamattomissa olevissa toiminnoissa.

Wissink ja Amaro [2006] nostavat esiin automaatiotestauksen toteutukseen käytettävien työkalujen kehittymisen myötä nousseet automaatiotestauksen menetelmät eli mallipohjaisen ja avainsanaperusteisen automaatiotestauksen. Avainsanaperusteisessa lähestymistavassa toiminnot koodataan avainsanoina käytettävälle apuohjelmalle, joka pyörittää testejä automaattisesti avainsanojen pohjalta [Wissinak ja Amaro, 2006]. Muuhun automaatiotestaukseen nähden avainsanoihin perustuvalla menetelmällä voitaisiin vähentää testiskriptien määrää ja näin ollen myös ylläpitotoimia testiskriptien osalta. Lisäksi Wisnak ja Amaro [2006] toteavat avainsanaperusteisen menetelmän vaativan vähemmän ohjelmointitaitoja testaajalta, jolloin menetelmän käyttäminen on helpompaa. Mallipohjaisessa testauksessa puolestaan luodaan malli, joka kuvaa testauksen kohdetta mahdollisimman tarkasti ja yksityiskohtaisesti. Mallipohjaisen testauksen etuna on varsinkin se, että mallista pystytään apuohjelmien avulla systemaattisesti muodostamaan kaikki mahdolliset kombinaatiot testitapauksista vaatimusmäärittelyyn perustuen [Blackburn *et al.*, 2004]. Mallipohjainen testaus on siis yksi automaatiotestauksen muoto, joka esitellään ja käydään tarkemmin läpi luvussa 3.

3 Mallipohjainen testaus

Mallipohjainen testaus on rakenteellinen testausmenetelmä, jossa malleja käytetään ohjaamaan halutun kohteen testausta [Aichernig *et al.*, 2018]. Mallit luodaan testattavan kohteen vaatimusten pohjalta [Dalal *et al.*, 1999]. Mallipohjaisen testauksen voidaan sanoa olevan automaatiotestausta kehittyneemmässä muodossa [Schieferdecker, 2012]. Automaatiotestauksessa testien suorittaminen automatisoidaan, kun puolestaan mallipohjaisessa testauksessa automatisoidaan ainakin testien luominen ja mahdollisesti myös toteutus. Malli luo automaattisesti vaatimusten pohjalta testitapaukset, minkä jälkeen ne voidaan suorittaa halutulle kohteelle halutussa ympäristössä. Mallin luomat testitapaukset voidaan suorittaa suoraan testattavana olevalle kohteelle tai testiympäristössä.

Mallipohjaisesta testauksesta on tullut viimeisten kahden vuosikymmenen aikana eräänlainen testauksen muoti-ilmiö ja sanaa käytetään yhä useamman testimenetelmän yhteydessä. Jorgensenin [2017] mukaan mallipohjainen testaus vaatii taitoa ja osaamista, mutta lisäksi tarvitaan myös hyvät apuohjelmistot ja työkalut. Taitava mallipohjaisen testauksen osaaja voi saada aikaan kohtuullisen hyväksyttävän testauksen huonoillakin työkaluilla, mutta vähän heikompi osaaja ei parhaillakaan työkaluilla saa hyvää mallipohjaista testausta aikaan. Mallipohjaisena testauksena voidaan pitää neljää erilaista lähestymistapaa, jotka on esitetty kuvassa 6 [Utting ja Legeard, 2007].



Kuva 6. Neljä pääasiallista lähestymistapaa, jotka tunnetaan mallipohjaisena testauksena [Utting ja Legeard, 2007].

Ensimmäisessä lähestymistavassa malli on luotu sisältämään tiedon sovellusalueen mahdollisista syötearvoista. Näin ollen mallin pohjalta luotavat testisyötteet sisältävät erilaisia yksittäisten syötearvojen osajoukkoja. Tässä mallipohjaisen testauksen muodossa ei kuitenkaan ole tietoa mahdollisista testattavan kohteen käyttäjälle antamista tulosteista, vaan mallilla testataan ainoastaan, kuuluvatko syötteet sovellusalueen syötteisiin ja kuinka testattavana oleva kohde reagoi annettuihin syötteisiin.

Ympäristömalliin pohjautuvassa lähestymistavassa eli toisessa mallipohjaisen testauksen pääasiallisessa tavassa malli luodaan kuvaamaan oletettua ympäristöä testattavana olevasta kohteesta. Ympäristömallin avulla voidaan altistaa testauksen kohde ulkopuolelta tuleviin erilaisiin ja yllättäviinkin tilanteisiin, kuten esimerkiksi erilaisten käyttäjien ja muiden järjestelmien antamiin komentoihin [Siavashi ja Truscan, 2015]. Tällaisilla malleilla voidaan siis esimerkiksi testata kohteen selviytymistä näistä yllättävistä tilanteista. Toisaalta on selvää, ettei varmuudella voida sanoa, selvisikö testauksen kohde halutulla tavalla epänormaaleista tilanteista, jos kohteen käyttäytymistä ei tunneta. Ympäristömalli usein sisältää vain tiedot siitä, millainen kohteen ympäristö on, muttei sisällä suoraan kohteen käyttäytymistä ja reagointia erilaisiin syötteisiin.

Muista kohdista poiketen kolmannessa lähestymistavassa on lähtökohtana, että malli tuntee, kuinka testattavan kohteen tulisi käyttäytyä ja reagoida annettuihin syötteisiin ja syötesarjoihin. Käyttäytymismallissa pyritään kuvaamaan tarkasti, mitä ohjelman kuuluisi antaa tulosteena, kun sille annetaan jokin tietty käsky tai toiminto. Käyttäytymismallissa lähtökohtana ovat siis oraakkelit, joiden avulla malli tietää jo etukäteen, mitä tulosteita ohjelman pitäisi kullakin syötteellä antaa. Näin pystytään tarkistamaan, suoriuduttiinko toteutetuista testeistä halutulla tavalla vertaamalla mallissa kuvattuja ja kohteen tulosteita keskenään.

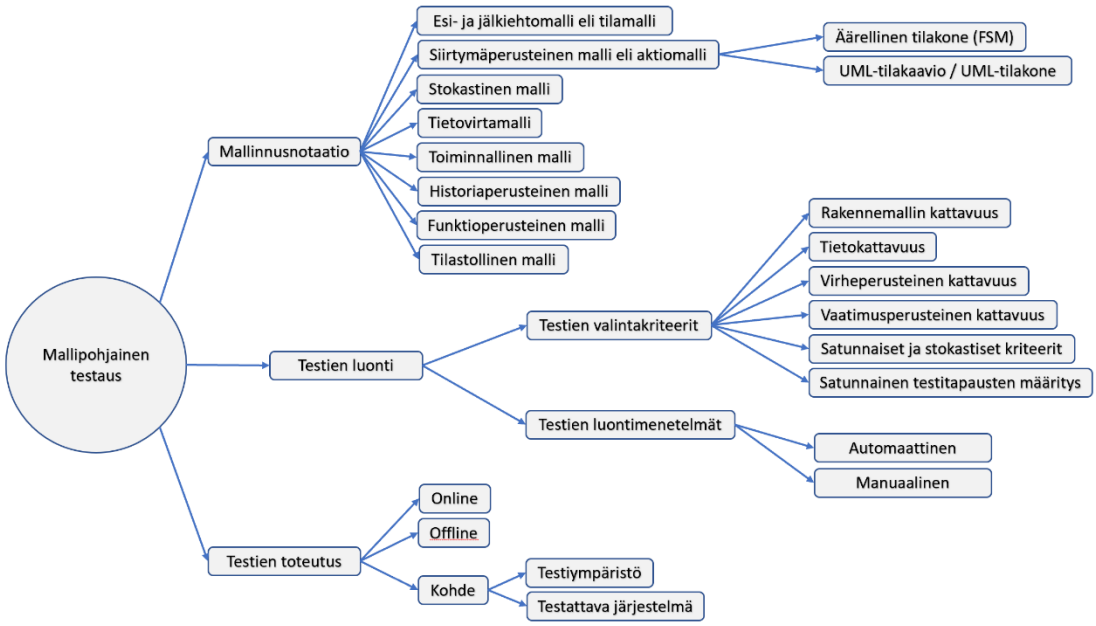
Neljännessä mallipohjaisen testauksen lähestymistavassa lähtökohtana on jokin korkeamman tason abstrakti kuvaus testistä ja itse malli puolestaan on kuvaus testattavan kohteen rakenteesta ja rajapinnoista sekä yksityiskohdista, joilla nämä korkeamman tason kuvaukset saadaan muutettua matalamman tason testisyötteiksi. Esimerkiksi tilanne voisi olla sellainen, että on olemassa piirretty kaavio, kuinka jokin testi kohteelle halutaan suorittaa, muttei varsinaisia käskyjä. Tällöin malli sisältäisi tarkat ohjeet, jotta kaavion testit osataan muuttaa konkreettisiksi käskyiksi testattavalle kohteelle. Malli toimii näin eräänlaisena tulkkausrajapintana.

Uttingin ja Legeardin [2007] mukaan mallipohjaista testausta voidaan pitää automatisoituna mustalaatikkotestauksena. Myös Marinescu ja muut [2015] näkevät mallipohjaisen testauksen mustalaatikkotestauksen tekniikkana, joka abstraktin järjestelmämallin perusteella pyrkii luomaan ja toteuttamaan testitapaukset automaattisesti. Mallipohjainen testaus ei kuitenkaan korvaa klassisia mustalaatikkotestauksen tekniikoita, mutta täydentää niitä [Kramer ja Legeard, 2016]. Malli luodaan vastaamaan testattavan kohteen odotettua käyttäytymistä, jolloin otetaan testattavaksi tiettyjä haluttuja kohtia suoraan vaatimuksista mustalaatikkotestauksen tapaan. Eron perinteiseen mustalaatikkotestaukseen antaa luotu malli, joka mahdollistaa mallipohjaisen testauksen ohjelmistoja käyttäen automaattisen testien luomisen suoraan mallista.

Kramer ja Legeard [2016] muistuttavat, ettei järjestelmää kuvaava malli ole sama kuin mallipohjaiseen testaukseen käytettävä malli. Mallit eroavat abstraktiotasolla toisistaan. Järjestelmää kuvaava malli on laajempi, kun taas testausmalli on yksityiskohtaisempi [Kramer ja Legeard, 2016].

Malli ei ole vain yksi diagrammi ohjelmiston käyttäytymisestä vaan kokoelma useammasta. Kuvaus erilaisten osien toiminnasta järjestelmän ollessa testauksessa on yksi graafinen esitys ja osa mallia, mutta malliin kuuluu paljon muutakin. Muita mahdollisia malliin sisältyviä graafisia esityksiä ovat esimerkiksi järjestelmän aikakäyttäytyminen, kuvaus järjestelmän sisäisistä tiloista sekä järjestelmän virtauksen hallinta. Malli siis koostuu useammasta palasesta ja on osiensa summa. [Kramer ja Legeard, 2016]

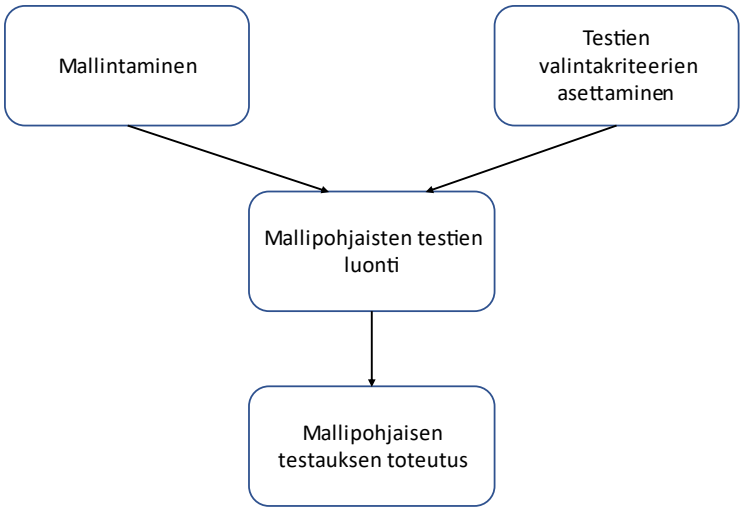
Mallipohjainen testaus voidaan luokitella moniin luokkiin ja alaluokkiin, joista muodostuu yhtenäinen kuvaus mallipohjaisen testauksen yleisistä vaiheista. Omanlaisen kaavion luokittelusta ovat esittäneet sekä Utting ja muut [2012] että Marinescu ja muut [2015]. Näiden yhdistelmänä on koottu luokittelukaavio, joka on esitetty kuvassa 7.



Kuva 7. Mallipohjaisen testauksen luokittelukaavio [Utting *et al.*, 2012; Marinescu *et al.*, 2015].

Mallipohjainen testaus muodostuu kuvan 7 luokittelukaaviossa kuvatuista osista ja näin ollen sisältää osa-alueet, jotka ovat mallinnusnotaation valinta, testien luontiprosessi ja testien toteutus. Lisäksi testien luontiprosessiin sisältyy testitapausten valintakriteerien määrittäminen. Kramer ja Legeard [2016] ovat esittäneet melko samantyyppisiin vaiheisiin perustuvan pelkistetymmän ja ainoastaan prosessin toimintoihin ja kulkuun keskittyvän kaavion, joka on esitetty kuvassa 8.

Kramer ja Legeard [2016] toteavat, että mallipohjaisen testauksen prosessi varmasti poikkeaa aiemmin tässä työssä kuvassa 1 esitetystä testausprosessista, mutta pääpiirteet pysyvät samoina. Suurin ero heidän mukaansa syntyy mallinnuksessa ja vaadituissa prosessin vaiheissa toteutettavien testien hankkimisessa mallista.



Kuva 8. Tyypilliset mallipohjaiselle testaukselle ominaiset toiminnot ja testauksen kulku [Kramer ja Legeard, 2016].

Mallipohjaisen testauksen kulussa ensimmäisenä tehtävät vaiheet ovat mallin luonti ja testien valintakriteerien laatiminen. Mallin tulee olla riittävän tarkka ja valmiiksi viimeistelty, jotta testit pystytään siitä automaattisesti luomaan. Abstraktion taso on yksi avaintekijä mallin toimivuuden kannalta. Monet uudet testaajat sortuvat laittamaan liikaa tai liian vähän yksityiskohtia malliin tai yrittävät saada kaiken mahtumaan yhteen malliin. Ennen varsinaista mallin luontia on syytä määrittää, mitä ollaan mallintamassa ja mikä mallinnuskieli valitaan. [Kramer ja Legeard, 2016]

Ensimmäisen vaiheen mallin luontia kuvaa kuvan 8 kohta mallintaminen ja tähän liittyy tiiviisti kuvan 7 mallintamisnotaatiot. Mallinnusnotaatioita ovat kuvassa 7 esitetyt tila-, aktio- ja tietovirtamalli sekä stokastinen, toiminnallinen, historiaperusteinen ja funktioperusteinen malli. Lisäksi jokaisella notaatiolla on vielä erikseen sille ominaisia ja käytössä olevia mallinnusmenetelmiä ja -kieliä, joista kaaviossa on mainittu esimerkin omaisesti vain kaksi melko yleisesti käytössä olevaa mallia: äärellinen tilakone ja UML-tilakone. Mallinnuskielien avulla malli ilmaistaan mallipohjaisten työkalujen ymmärtämässä muodossa. Mallinnusnotaatioita ja mallinnusvaihetta kuvataan tarkemmin kohdassa 3.1.

Testien luontia varten tulee määrittää testitapausten valintakriteerit, joilla testejä tullaan rajaamaan. Valintakriteereinä toimivia testien kattavuuksia ovat esimerkiksi rakennemallin kattavuus, vaatimusperusteinen ja virheperusteinen kattavuus. Valintakriteereinä toimivia kattavuuksia esitellään yksityiskohtaisemmin kohdassa 3.2.

Mallin luonnin ja testien valintakriteerien valinnan jälkeen suoritetaan itse testien luonti, kuten kuvasta 8 voidaan todeta. Yleensä testien luonti tapahtuu automaattisesti mallista käyttäen apuna jotain työkalua, mutta manuaalinenkin testitapausten luonti on mahdollista, kuten kuvasta 7 nähdään. Lopuksi kuvien 8 ja 7 mukaisesti testit suoritetaan mallipohjaisen testauksen kohteena olevalle järjestelmälle tai testiympäristössä. Testaus voidaan suorittaa joko online- tai offline-menetelmällä, riippuen testattavasta kohteesta, halutusta menettelytavasta ja käytettävissä olevista työkaluista. Testien luontia ja toteutusta tarkastellaan tarkemmin kohdassa 3.3.

3.1 Mallintaminen, mallinnusnotaatiot ja mallin luonti

Testattavan systeemin mallinnus aloitetaan tärkeimmästä vaiheesta eli abstraktiotason valinnasta. Aluksi määritellään, mitä asioita malliin sisällytetään ja mitä kannattaa jättää suosiolla pois. Mallia käytetään lähtökohtaisesti vain testaukseen, joten sen ei tarvitse sisältää kaikkea systeemin toimintaa. [Utting ja Legeard, 2007] Ei ole edes kovin viisasta sisällyttää kaikkea systeemin toimintaa yhteen malliin, sillä malli kasvaisi monesti niin isoksi, ettei sitä pystyisi enää hallitsemaan tai ylläpitämään. Utting ja Legeard [2007] toteavatkin, että mallinnus kannattaa toteuttaa pienten osamallien kautta ennen kuin tekee varsinaisen ylimmän tason mallin koko systeemille.

Kun on päätetty, mitä aiotaan mallintaa, siirrytään mallintamisen toiseen vaiheeseen. Mallinnuksen toisessa vaiheessa tulee miettiä, mitä kaikkea mallissa tulee tapahtumaan. Tulee olla selvillä data, joka mallin tulee hallita, operaatiot, joita malli suorittaa, ja alisysteemit, joiden kanssa malli kommunikoi. Jos on olemassa vaatimusmäärittelyjen pohjalta toteutettu valmis luokkadiagrammi testattavan systeemin muodosta, sitä voidaan tässä käyttää apuna. Luokkadiagrammia tehdessä on syytä pitää mielessä, että testausta varten luokkadiagrammin tulee olla paljon yksinkertaisempi kuin systeemin muotoilua varten tarvitaan. Luokkadiagrammia tulee yksinkertaistaa testauksen vaatimalle tasolle tai tehdä se kokonaan uudestaan. [Utting ja Legeard, 2007]

Utting ja Legeard [2007] mainitsevat muutamia tyypillisimpiä yleistyksiä, jotka kannattaa mallinnusta tehdessä ottaa huomioon liittyen dataan, luokkiin, operaatioihin ja alisysteemeihin. Mallinnusta tehdessä tulee keskittyminen olla testattavassa kohteessa. Mallinnetaan siis vain luokat tai alisysteemit, jotka liittyvät testattavaan kohteeseen tai joiden arvoja tarvitaan testidatassa, ja sisällytetään vain operaatiot, joita on tarkoitus testata ja tietokentät, jotka ovat hyödyllisiä testattavien operaatioiden käyttäytymistä mallinnettaessa. Lisäksi korvataan monitahoiset tietokentät ja luokat yksinkertaisella listauksella. Nämä mahdollistavat testidatan rajoittamisen valittuihin yksinkertaisempiin arvoihin. [Utting ja Legeard, 2007]

Jokaisen mallinnettavan operaation kohdalla tulee miettiä tarvittavat syöte- ja tulosteparametrit. Syöteparametrit kannattaa jättää yksinkertaistamisen vuoksi mallinnuksen ulkopuolelle, jos ne eivät vaikuta operaation toimintaan tai tätä operaation toimintaa ei

ole tarkoitus testata kyseisellä mallilla. Mahdollisia syötteitä tai syötealueita on usein niin paljon, että se aiheuttaa testausvaiheessa haasteita. Tulee miettiä todella tarkkaan, ovatko syötteet tarvittavia ja relevantteja mallintaa testausta ajatellen. Tulosteparametrit puolestaan tulee mallintaa vain, jos ne toimivat oraakkeleina testauksessa eli kuvaavat, mitä testistä pitäisi tulla tulosteena ulos, kun operaatio on suoritettu. Operaatioiden ei tarvitse olla mallissa täysin samanlaisia kuin testattavassa systeemissä, vaan monimutkaiset operaatiot voidaan hajottaa useampaan operaatioon tai vastaavasti operaatioita voidaan yhdistää. [Utting ja Legeard, 2007]

Ennen mallin kirjoittamista tulee valita mallinnuksessa käytettävä notaatio eli mallinnustyyppi. Notaation valintaan vaikuttaa työkalut, joita testaukseen on käytössä, mutta tulee myös miettiä, mikä notaatio sopii parhaiten testauksen kohteeseen.

Mallipohjaisen testauksen mallit voidaan jakaa rakenteellisiin ja käyttäytymisperusteisiin malleihin [Jorgensen, 2017]. Uttingin ja Legeardin [2007] mukaan varsinaisia mallinnustyyppejä on useita, joita onkin esitetty tässä työssä aikaisemmin kuvassa 7. Kuvassa olevista notaatioista yleisimmin käytetään kahta tyyppiä, jotka ovat aktio- ja tilamalli. Näitä mallinnustyyppejä käytetään varsinkin silloin, kun testattavalle kohteelle kehitetään käyttäytymisperusteista mallia. Esi- ja jälkiehtoihin perustuva mallinnustyyppi eli tilamalli on mallinnustyypeistä käyttökelpoisin testattavan kohteen ollessa dataorientoitunut. Puolestaan siirtymäperusteinen mallinnustyyppi eli aktiomalli on parempi vaihtoehto, jos kohde on enemmän kontrolliorientoitunut. [Utting ja Legeard, 2007]

Muita mahdollisia, mutta harvemmin käytettäviä, mallinnustyyppejä ovat historiaperusteinen, funktioperusteinen, toiminnallinen ja tilastollinen malli sekä stokastinen notaatio ja tietovirtamalli. Historiaan perustuvassa mallinnustyypissä malli luodaan testattavan kohteen pitkällä aikavälillä tapahtuneiden muutosten ja käyttäytymisen pohjalta. Funktioperusteinen malli pohjautuu matemaattisiin funktioihin. Muihin mallinnustapoihin verrattuna funktioperusteinen mallinnus sen sijaan on haastava ja käytön mahdollisuuskin rajallista. Usein matemaattisen funktion löytäminen ja systeemiin yhdistäminen ovat niin vaikeita, että funktionaalisen mallin käyttö jää vähemmälle. [Utting *et al.*, 2012]

Toiminnallinen malli kuvaa kohteen erilaisten samanaikaisesti suoritettavien toimintojen kokoelmana. Sen sijaan tilastollinen malli kuvaa mallinnettavan kohteen tapahtumien ja syötearvojen todennäköisyyksillä. Tämän mallinnustavan heikkoutena on, ettei tulosteista ole tietoa lainkaan, ellei luoda toista mallia, joka sisältää oraakkelimaisesti oletettavat tulosteet. Stokastinen notaatio käyttää tapahtumien ja syötteiden todennäköisyysmallia järjestelmän kuvaamiseen. [Utting *et al.*, 2012] Tietovirtaukseen perustuvassa mallinnustyypissä taas keskitytään enemmän testikohteen läpi kulkevaan tietoon kuin kohteen ohjaukseen ja ohjausvirtaan. Tämän takia tämä mallinnustyyppi sopiikin lähinnä vain jatkuviin järjestelmiin. [Utting ja Legeard, 2007]

Mallinnusnotaatioilla on erikseen niitä ilmentäviä mallinnuskieliä ja menetelmiä, joiden avulla mallit ilmennetään ja joihin mallipohjaisen testauksen työkalut pohjautuvat. Tällaisia ovat esimerkiksi kuvassa 7 esitetyt äärellinen tilakone ja UML-tilakone.

Monet kaupallisista ja vapaasti saatavilla olevista mallipohjaisen testauksen työkaluista ja ohjelmista on suunniteltu äärellisen tilakoneen käyttämiseen mallinnuksessa. Äärellinen tilakone voidaan määritellä nelikkona, johon kuuluvat tilat, siirtymät, sarja syötteitä, jotka aiheuttavat siirtymät, ja sarja tulosteita, joita voi esiintyä siirtymissä. Tilakoneessa solut kuvaavat tiloja ja reunat siirtymiä, ja näistä muodostuu suunnattu graafi. Tilakoneessa testattavat polut voidaan määrittää tilojen jonoina, tilojen siirtymien jonoina tai syötteiden aiheuttamien siirtymien jonoina. [Jorgensen, 2017] Äärellinen tilakone on yksi siirtymäperusteisen mallinnusnotaation muoto, jonka pohjalta varsinainen malli voidaan luoda [Marinescu *et al.*, 2012].

Kun notaatio ja mallinnustapa on valittu sekä malli saatu luotua, tulee malli itsessään testata ennen testien luonnin ja järjestelmän testauksen aloittamista. Näin pyritään varmistamaan, ettei mallissa ole virheitä. Virheet mallissa saattavat johtaa virheellisiin tulkintoihin järjestelmän virheellisestä toiminnasta, jolloin mallin virheet päätyvät itse testattavaan järjestelmään. [Utting ja Legeard, 2007]

3.2 Mallipohjaisen testauksen testitapausten valintakriteerit

Mallista pystytään yleensä luomaan todella monia testitapauksia, joten niitä on syytä valintakriteereillä rajoittaa. Valintakriteerit tehdään halutun kattavuuden perusteella. Testien valintakriteereitä on kaikestaan tarjottu kuusi erilaista vaihtoehtoa aiemmin esitetyssä kuvassa 7. Nämä valintakriteerit ovat rakennemallin kattavuus, tietokattavuus, virheperusteinen ja vaatimusperusteinen kattavuus, satunnaiset ja stokastiset kriteerit sekä satunnainen testitapausten määritys. [Utting *et al.*, 2012; Marinescu *et al.*, 2015]

Rakennemallin kattavuus -valintakriteeri käyttää ehtoihin mallien rakenteellisia osia, kuten esimerkiksi solmuja, kaaria tai mallin ehtorakenteita. Tätä valintakriteeriä voidaan käyttää kaikkien muuttujia sisältävien mallinnusnotaatioiden kanssa. Tietokattavuus on puolestaan dataan perustuva valintakriteeri. Tässä menetelmässä pyritään hallitsemaan isoja datamassoja jakamalla niitä luokkiin. Jokaisesta luokasta valitaan yksi luokan edustaja ja nämä edustajat toimivat valintakriteereinä.

Virheperusteinen kattavuus perustuu aiemmin löydettyihin virheisiin. Valintakriteerinä pyritään valitsemaan testitapaukset, joista on seurannut virheitä. Sen sijaan vaatimusperusteinen kattavuus puolestaan määritellään testattavana olevan kohteen vaatimusten mukaan. Vaatimuksista voidaan valita tietty joukko tai kaikki valintakriteeriksi.

Satunnaiset ja stokastiset kriteerit perustuvat osittaiseen yllätyksellisyyteen. Ei voida varmuudella tietää, mikä tulee olemaan seuraava toiminto tai mistä siihen päädytään. Parhaiten näitä kriteereitä voidaan hyödyntää käyttäjäprofiilien ja ympäristömallien kanssa toimittaessa. Valintakriteerit siis valitaan käyttäjäprofiilien mukaan. Satunnaisten

testitapausten määritys -menetelmässä puolestaan testaaja luo testitapausten määritelmän, jonka pohjalta testitapaukset tulee luoda ja valita.

Kramerin ja Legeardin [2016] mukaan yleensä kattavuudeksi valitaan vaatimuksiin pohjautuva, mallin elementteihin eli rakenteisiin kohdistuva tai dataan eli tietoon yhdistetty kattavuus. Valintakriteerien valinta ei kuitenkaan välttämättä ole niin yksinkertaista, sillä valintakriteerejä voidaan joutua yhdistelemään, jotta testaustavoitteet täyttyvät. Toisaalta voidaan myös haluta valita tiettyjä testejä kattamaan tiettyä tilannetta.

3.3 Testien luonti ja ajaminen mallista

Mallipohjainen testaus voidaan toteuttaa automaattisesti tai manuaalisesti. Testit ja testitapaukset luodaan lähes poikkeuksetta työkaluja hyödyntäen automaattisesti testaajan luoman mallin ja testien valintakriteerien pohjalta. Tämä ei kuitenkaan poissulje mahdollisuutta manuaaliseen luontiin [Kramer ja Legeard, 2016].

Manuaalisessa testien generoinnissa testaaja kirjoittaa testiskriptit ja testitapaukset itse. Automaattinen testien luonti tapahtuu puolestaan täysin koneellisesti. Automaattisia testien luontimenetelmiä ovat esimerkiksi mallin tarkistus, satunnainen luonti ja polunetsintäalgoritmi. Mallin tarkistus -menetelmän periaate on pyrkiä kaikilla toteutettavissa olevilla mallin ajoilla osoittamaan mallin ominaisuudet joko oikeiksi tai vaihtoehtoisesti vääriksi. Näin voidaan saada testitapaukset luotua automaattisesti tätä menetelmää hyödyntäen. Puolestaan satunnainen luonti -menetelmässä hyödynnetään järjestelmän syöteavaruutta. Syötteitä valitaan satunnaisesti luomaan testitapauksia. Polunetsintäalgoritmimenetelmä yksinkertaisesti perustuu nimensä mukaan järjestelmän läpi kulkemiseen eri polkuja pitkin. Polut muodostavat testitapaukset automaattisesti. [Marinescu *et al.*, 2015]

Kramer ja Legeard [2016] toteavat mallipohjaista testausta monesti ajateltavan vain testauksen automaationa. Monet mallipohjaisen testauksen hyödyt voidaan kuitenkin saavuttaa manuaalisellakin testien toteutuksella. Vaikka testaus toteutettaisiin manuaalisesti, tapahtuu testien luonti mallista kuitenkin automaattisesti. Manuaalisen testauksen kannalta on oleellista, että käytettävä mallipohjaisen testauksen työkalu mahdollistaa konkreettisten testien tekstimuotoisen kuvauksen tulostamisen. Tyypillisesti tämä on mahdollista useammassakin muodossa, kuten esimerkiksi tavallisena tekstinä, työarkkina tai HTML-dokumenttina. Tämän jälkeen testit viedään testienhallintaan käytettävälle työkalulle tai testaaja suorittaa testit manuaalisesti ja kirjaa ylös testien tulokset läpimenon osalta. Tekstitiedostona tulostetut testitapaukset ovat riittävän konkreettisia testaajan toteutettavaksi. [Kramer ja Legeard, 2016]

Automaattisessa mallipohjaisen testauksen muodossa testianalyytikko toteuttaa mallipohjaisen testauksen työkalun avulla mallin, asettaa testien valintakriteerit ja luo testiskriptin, jonka jälkeen tämä viedään testausautomaatiokehykseen. Testausautomaatioinsinööri linkittää testauksen kohteen ja testausautomaatiokehyksen, jossa aiemmin luotu

testitapaukset sisältävä testiskripti on. Kun linkitys on suoritettu onnistuneesti, testit voidaan suorittaa automaattisesti. Joissain työkaluissa linkitys testattavaan kohteeseen voi olla valmiina, eikä erillistä rajapintaa tarvita. [Kramer ja Legeard, 2016]

Testien suorittaminen testauksen kohteelle on jaettu kahteen muotoon: online- ja offline-ajoon [Utting *et al.*, 2012]. Online-menetelmässä testit toteutetaan heti luonnin jälkeen testattavalle kohteelle ja toteutetut testit voivat vaikuttaa saatujen tulosteiden kautta seuraaviin testitapauksiin, joita luodaan. Testaus tapahtuu siis suoraan lennosta luonnin ja toteutuksen ollessa limittäin keskenään. Uttingin ja muiden [2012] mukaan online-menetelmä on hyvä varsinkin toimittaessa epädeterministisien järjestelmien kanssa. Offline-menetelmässä puolestaan luodaan ensin etukäteen suunniteltu testisarja testitapauksia ja vasta kaikkien luonnin jälkeen toteutetaan testisarja testattavalle kohteelle. Testejä ei siis luoda samalla tavalla lennosta ja ajeta heti, vaan luonti ja toteutus toimivat erikseen isompina peräkkäisinä vaiheina. [Marinescu *et al.*, 2015]

4 Mallipohjaisen testauksen työkaluja

Mallipohjaisen testauksen kyky hallita ja suorittaa testaus halvemmin ja nopeammin perinteisiin testausmenetelmiin verrattuna on lisännyt mallipohjaista testausta tukevien työkalujen määrää. Li ja muut [2017] toteavat työkalujen kirjon olevan laaja ja niiden saattavan erota paljonkin toisistaan. Lisäksi heidän mukaansa oikean ja sopivimman työkalun valinta voi olla haastavaa ilman aikaisempaa tietotaitoa ja osaamista.

Tässä luvussa vastataan ensimmäiseen tutkimuskysymykseen, jossa pohdittiin, millaisia työkaluja mallipohjaiseen testaukseen on olemassa. Työkaluja löydettiin melko paljon ja näistä valittiin käsiteltäväksi 12. Tarkoituksena on lähinnä pintapuolisesti esitellä, millaisia työkaluja mallipohjaiseen testaukseen on yleisesti kehitetty.

4.1 fMBT

Intel:n kehittämä fMBT on vapaasti saatavilla oleva työkalu, jolla voidaan suorittaa niin online- kuin offline-testausta [Li *et al.*, 2017]. Työkalu tarjoaa mallieditorin, testigeneraattorin, työkalut lokien analysointiin ja välikappaleet, joilla suorittaa testaus useiden rajapintojen läpi. Mallinnuskielenä fMBT:ssä käytetään AAL/Python-kieltä. Kielen avulla luodaan mallit käyttäen esi- ja jälkiehtoja. Esiehtoina toimivat vartijat (guard) ja jälkiehtoina runko-osat (body). [Jorgensen, 2017]

Lin ja muiden [2017] mukaan fMBT tarjoaa testaukseen tarvittavat rajapinnat laajallekin otannalle testattavia kohteita. Heidän ja myös Jorgensenin [2017] mukaan testausta fMBT:llä voidaan toteuttaa niin yksittäisille luokille kuin kokonaisille graafisen käyttöliittymän omaaville sovelluksillekin.

Jorgensenin [2017] mukaan virtuaalikoneen asettaminen on välttämätöntä, sillä fMBT on Linux-käyttöjärjestelmälle pohjautuva sovellus. Esimerkiksi Ubuntua voidaan käyttää alustana, jolle fMBT asennetaan. Sovellus käynnistetään pääteohjelman kautta komennolla fmbt-editor, jonka jälkeen mallia voidaan alkaa luomaan fMBT:n käyttöliittymän kautta.

Avoimen saatavuuden lisäksi fMBT:n vahvuus Jorgensenin [2017] mukaan on työkalun kyky mahdollistaa todella nopea testitapausten luonti. Toisaalta Jorgensen painottaa Python-kielen osaamista merkittävänä tekijänä työkalun käytölle ja lisäksi toteaa, ettei työkalu välttämättä sovellu aloitteleville testaajille johtuen varsinkin vähäisestä ohjeistusdokumentaatiosta.

4.2 AGEDIS

AGEDIS on joukko yhdistettyjä työkaluja hajautettujen sovellusten käyttäytymisen mallintamiseen. Työkalu pohjautuu UML-malliin ja käyttää mallinnusnotaationa siirtymäperusteista mallia [Marinescu *et al*., 2015]. Mallinnuksen lisäksi AGEDIS pystyy luomaan ja toteuttamaan testit sekä analysoimaan testien tulokset. AGEDIS voi myös toimia tes-

tauskehyksenä, johon voidaan yhdistää muita mallipohjaisen testauksen työkaluja ja hyödyntää AGEDIS-työkalun tarjoamia tiloja ja toimintoja. AGEDIS-työkalun vahvuus on kyky yhdistyä erilaisten toimittajien työkalujen kanssa omanlaisista vahvuuksista ja vaatimuksista riippumatta. [Hartman ja Nagi, 2004]

Shafiquen ja Labichen [2010] toteavat AGEDIS-testauskehyksen soveltuvan komponenttipohjaisten hajautettujen järjestelmien mallipohjaiselle testaukselle. Kuten monet muutkin työkalut myös AGEDIS käyttää omaa mallinnuskieltään. Työkalun mallinnuskielenä toimii AML, joka on lyhenne sanoista AGEDIS Modeling Language.

AGEDIS syntyi projektissa, jonka tarkoitus oli lisätä mallipohjaisen testauksen käyttämistä ja yleisyyttä pyrkimyksellä poistaa muutamia haasteita, jotka vaikuttivat mallipohjaisen testauksen vähäisempään käyttöön [Hartman ja Nagi, 2004]. Haasteita, joita pyrittiin taklaamaan, olivat kunnollisten työkalujen ja riittävien automaatiotestauksen mittareiden puute sekä toisaalta kunnollisten ohjeiden ja harjoittelumateriaalien vähyys [Robinson, 2003].

4.3 TestOptimal

Shafiquen ja Labichen [2010] mukaan TestOptimal on verkkopohjainen asiakaspalvelintyökalu mallipohjaiseen testaukseen. Mallinnuksen pohjana käytetään laajennettua tilakonetta ja notaationa siirtymäperusteista mallia [Marinescu *et al*., 2015]. Työkalulla voidaan testata tavallisten työpöytäsovellusten lisäksi monitasoisia yrityssovelluksia. Mallin ja testattavan järjestelmän yhdistämisen hoitaa XML-pohjainen mScript-skriptikieli. TestOptimal mahdollistaa mallin vaatimusten vahvistamisen, simuloinnin ja korjaamisen. Testitapausten luontiin on tarjolla useita erilaisia algoritmeja, ja työkalua voidaan käyttää niin regressio-, kuormitus- kuin sietokykytestaukseenkin.

TestOptimal yhdistää testien suunnittelun ja automatisoinnin samaan työkaluun ja sitä voidaan siis käyttää niin toiminnallisuuksien kuin suorituskyvynkin testaukseen [Jorgensen, 2017; Rodrigues *et al.*, 2015]. Rodriguesin ja muiden [2015] mukaan TestOptimal on tehokas ja yksi valmiimmista kokonaisuuksista mallipohjaiseen testaukseen, mutta he myös toteavat työkalun olevan kaupallisena ohjelmistona kohtuu kallis investointi yrityksille lisenssimaksujen ja koulutuksien myötä.

TestOptimal työkalusta on saatavilla useita erilaisia versioita, jotka sisältävät hieman toisistaan poikkeavia ominaisuuksia. Näitä versioita ovat esimerkiksi BasicMBT, ProMBT ja EnterpriseMBT. [Jorgensen, 2017]

4.4 Gotcha-TCBeans

GOTCHA-TCBeans-työkalu muodostuu kahdesta osasta. Ensimmäinen osa eli GOTCHA luo testitapaukset tilakoneen mallin pohjalta. Toinen osa eli TCBeans puolestaan tuo mukaan Java-luokat konkretisoimaan ja toteuttamaan testitapaukset testauksessa olevalle järjestelmälle. GOTCHA:n muodostama tilakoneen testimalli on puolestaan luotu

käyttäen työkalun omaa GOTCHA Definition language -kieltä. Testimalli määrittelee testien odotetut tulokset, mallissa käytettävät funktiot, tilamuuttujat ja tilat. Lisäksi testimallin on tarkoitus valvoa ja pyrkiä varmistamaan testitapausten luonnin aikana, ettei tila-avaruusräjähdystä pääse syntymään. Työkalun käyttäjällä on myös mahdollisuus määrittää testiskenaariot. GOTCHA:n tuottaessa testitapaukset XML-muodossa, joita voidaan ajaa sekä online- että offline-testauksessa. [Shafique ja Labiche, 2010]

Uttingin ja Legeardin [2007] mukaan IBM-yritys on onnistuneesti käyttänyt GOTCHA-TCBeans-työkalua raportoiden virheiden löytymiskyvyn olleen hyvä ja sen onnistuneensa samalla laskemaan kustannuksia. Hartmanin ja muiden [2002] mukaan IBM on myös alun perin kehittänyt GOTCHA-TCBeans-työkalun mallipohjaiseen testaukseen.

4.5 mbt

Avoimena lähdekoodina saatavilla oleva mbt-työkalu toimii komentoikkunan kautta, eikä sillä ole varsinaista käyttöliittymää. Tilakonemallin ja sen mahdolliset alimallit mbt tuottaa GraphML-muodossa hyödyntäen jotakin apuohjelmistoa, kuten yED, jonka Shafique ja Labiche [2010] mainitsevat esimerkkinä.

Shafiquen ja Labichen [2010] mukaan mbt tukee molempia, online- ja offline-testausta, mutta online-testaus on vain osaksi automatisoitu. Testaajan tulee itse antaa syötearvot, kun peräkkäisiä testejä suoritetaan. He myös toteavat, että mbt:llä voidaan luoda peräkkäisiä testejä kattamaan vain tiettyjä käyttäjän määrittämiä tiloja, siirtymiä tai vaatimuksia. On kuitenkin hyvä huomata, ettei testitapausten suorittaminen kuulu mbt:n tukemiin ominaisuuksiin. Tästä huolimatta mbt pystyy useita erilaisia algoritmeja hyödyntämällä kuitenkin tuottamaan testitapaukset.

4.6 MOTES

MOTES on Eclipse-ohjelmiston liitännäinen. MOTES käyttää laajennettua tilakonetta testitapausten luontiin. XMI-muotoinen testimalli, testattavan datan kuvaus TTCN-3-testikielellä ja asetustiedot laajennetun tilakoneen tiloille ja syöte- ja tulosteporteille ovat kolme syötetiedostoa, jotka MOTES tarvitsee testitapausten luomiseen. Mallin luomiseen XMI-muodossa MOTES tarvitsee kolmannen osapuolen työkalun, kuten esimerkiksi Poseidonin. MOTES tukee sekä online- että offline-testausta. [Shafique ja Labiche, 2010]

Ali ja muut [2010] toteavat artikkelissaan MOTES-työkalun hyväksyvän laajennetun tilakoneen syötteenä, mutta vaativan testidatan manuaalisen valmistelun ennen testitapausten luontia. Heidän mukaansa MOTES kuitenkin tarjoaa laajennettuja mahdollisuuksia tulosteena saataville malleille vakioidun syöterajapinnan myötä. MOTES tarjoaa muunneltavissa olevat testauksen kattavuuskriteerit. Ali ja muut [2010] mainitsevat esimerkkeinä mahdollisuuden tilojen tai siirtymien valintaan tai mahdollisuuden valita kaikki siirtymät.

4.7 MISTA

MISTA on kaupallisena tuotteena ja ilman veloitusta avoimena versiona akateemiseen käyttöön saatavilla oleva mallipohjaisen testauksen työkalu [Jorgensen, 2017]. MISTA sisältää graafisen käyttöliittymän, jonka kautta työkalua käytetään. Li ja muut [2017] toteavat MISTA:n luovan testitapaukset UML-tilakoneen malleista tai vaihtoehtoisesti toimintoverkosta. Lisäksi he tuovat esille, että MISTA pystyy rakentamaan sekä tieto-orientoituneita että kontrolliorientoituneita malleja. Mallinnusnotaationa on siirtymäperusteinen malli [Marinescu *et al*., 2015]. Visuaaliseen mallinnukseen MISTA käyttää korkean tason Petri net -mallinnuskieltä [Jorgensen, 2017].

Jorgensen [2017] toteaa mallin tarkistuksen olevan yksi MISTA:n hyödyllisistä ominaisuuksista. MISTA itsessään tarkistaa, että mallissa jokainen siirtymä ja tila ovat saavutettavissa. Tämä antaa työkalun käyttäjälle turvaa siitä, että malli olisi oikein luotu. Mallin odotettu toiminta voidaan tarkistaa Petri net:n avulla simuloimalla [Jorgensen, 2017].

MISTA:n käyttäminen edellyttää Petri net:n ja äärellisen tilakoneen tuntemusta. Itse testien luonti ei Jorgensenin [2017] mukaan ole vaikeaa, kun vain on valinnut testien kattavuusvaatimukset ensin. MISTA tarjoaa laajan ja moninaisen joukon vaihtoehtoja kattavuudelle ja vaatimuksille. Vaihtoehtoja ovat esimerkiksi siirtymä- tai syvyyskattavuus. Siirtymäkattavuudessa testit luodaan niin, että jokainen siirtymä tulee käytyä läpi. Syvyyskattavuudessa puolestaan MISTA luo testit niin, että puurakenteen solmut tulee käytyä annettuun syvyyteen saakka. MISTA ei käytä mitään omaa muokattua mallinnuskieltä, mikä tekee työkalusta helpommin saavutettavan moniin muihin kaupallisiin työkaluihin verrattuna. [Jorgensen, 2017]

4.8 ParTeG

Alkujaan Partition Test Generator eli ParTeG on suunniteltu muuttamaan uusia algoritmeja prototyypeiksi konseptin toimivuuden todistamiseksi, mutta nykyään sitä käytetään monipuolisemmin mallipohjaiseen testaukseen [Weißleder ja Sokenou, 2010]. ParTeG on mallipohjaisen testauksen työkalu, joka on avoimesti saatavilla oleva Eclipse-ohjelmiston liitännäinen, jossa testitapaukset luodaan TopCased-liitännäisen avulla testausmallin pohjalta [Shafique ja Labiche, 2010]. ParTeG luo automaattisesti testitapaukset UML-tilakoneen ja OCL-ilmauksin merkityn luokkadiagrammin mukaan [Weißleder ja Sokenou, 2010]. ParTeG käyttää siirtymäperusteista mallinnusnotaatiota [Marinescu *et al*., 2015].

ParTeG rakentaa testimallista siirtymäpuun, jossa jokainen polku on yksi testitapaus. OCL-ilmauksia käytetään määrittämään tilamallille tyypillisten vartijoiden ehdot. Näitä ilmauksia tarvitaan, kun siirtymäpuun polku muutetaan syötearvojen ehdoiksi. Syötear-

vojen ehdot määrittävät, kuinka syötteet jaotellaan syötteiden osajoukkoihin. Testitapaukset koodataan ParTeG:ssa Java-kielellä ja ne toteutetaan testattavana olevalle järjestelmälle JUnit:n avulla. [Shafique ja Labiche, 2010]

ParTeG mahdollistaa kokonaisen testisyklin testimallien luonnista testisarjojen tuottamiseen yhdistettynä kattavuuskriteereihin ja myöhempään testien toteutukseen. Lisäksi yhteenvetona voidaan todeta, että automaattinen syöteosioiden luonti, testauksen tavoitteiden priorisointi, testimallin mukauttaminen ja tilakoneiden johtaminen luokkien välisten periytymisten mukaan ovat testauksen kannalta oleellisia ParTeG:n tukemia ominaisuuksia. [Weißleder ja Sokenou, 2010]

4.9 Qtronic

Qtronic on kaupallisesti saatavilla oleva mallipohjaisen testauksen työkalu. Qtronic luo testitapaukset ohjelmiston tavoitellun käyttäytymisen tila-avaruusanalyysin perusteella. Käyttäytyminen merkitään suunnittelumallilla hierarkkisen UML-tilakartan suhteen syötteenä työkalulle. Suunnittelumalli voidaan ilmaista tekstitiedostojen kokoelmana käyttäen Qtronicin omaa QML-tekstinotaatiota. QML on Java-kielen ylijoukko. Vaihtoehtoisesti suunnittelumallin ilmaisu voidaan tehdä graafisten mallien avulla. [Sarma *et al.*, 2010]

Mallipohjainen testaus Qtronicilla perustuu kolmeen merkittävään osaan, jotka ovat Qtronic Computation Server, Qtronic Modeler ja Qtronic Client. Testitapauksien luontiin ja toteutukseen mallin mukaan sekä testauksella saatujen tulosten tulkintaan hyödynnetään Qtronic Computation Serveriä. Qtronic Modelerilla puolestaan luodaan testimalli UML-tilakoneina käyttäen QML-kieltä. Näiden osien ja ominaisuuksien käytön mahdollistaa Qtronic Client, joka tarjoaa ympäristön ja tilat testien kattavuuskriteerien valinnalle, testimallien luomiselle sekä testitapausten toteutuksen tulosten ja mallin analysoinnille. [Shafique ja Labiche, 2010]

Pohjimmiltaan Qtronic käyttää mallin symbolista toteutusta testitapausten luomiseen. Luoduilla testitapauksilla arvioidaan testattavana olevaa kohdetta ulkoisten rajapintojen kautta. Qtronicissa toteutetaan kohteelle ikään kuin mustalaatikkotestausta mallinnusta ja malleja apuna käyttäen. [Sarma *et al.*, 2010]

4.10 MoMuT

MoMuT on sarja mallipohjaisen testauksen työkaluja. Se toimii UML-tilakoneen, vaatimusrajapintojen, toimintajärjestelmien ja ajastetun automaation avulla. MoMuT sisältää myös mekanismin virheiden paikantamiseen, joka suorittaa korjauksen, kun testitapaus ei mene onnistuneesti läpi. [Li *et al.*, 2017]

MoMuT pohjautuu mallipohjaiseen mutaatiotestaukseen (MBMT), joka puolestaan on merkittävä esimerkki virheperusteisesta testauksesta. Mallipohjainen mutaatiotestaus

toimii ikään kuin mustalaatikkotestauksena, niin ettei varsinaisesti perehdytä toteutukseen, vaan ainoastaan keskitytään testattavana olevan järjestelmän malliin, joka luodaan suoraan vaatimusten perusteella toteutetun UML-tilakoneen pohjalta. Tämän takia malli on yleensä toteutusta abstraktimpi. [Krenn *et al.*, 2015]

MoMuT sisältää virheisiin perustuvan testienluontistrategian, joka sallii muutosten tekemisen malleihin. Näin ollen pystytään luomaan laadukkaampia testitapauksia hyödyntäen sekä alkuperäisiä että muuttuneita malleja [Li *et al.*, 2017]. Karlssonin ja muiden [2022] mukaan MoMuT-työkalulla voidaan mallintaa monitahoisiakin järjestelmiä.

4.11 Spec Explorer

Spec Explorer on Visual Studio -kehitysalustaan liitetty mallipohjaisen testauksen työkalu. Alkujaan Spec Explorer on Microsoftin luoma. Spec Explorer tuo Visual Studion ohjelmointiympäristöön ohjelmiston käyttäytymisen mallintamisen, ohjelmiston käyttäytymisen graafisen kuvauksen ja analysoinnin sekä muusta koodista erillään olevan testikoodin luonnin. Keskeisin ajatus Spec Explorer -työkalussa on koodata testattavan järjestelmän haluttu toiminta mallina. Mallin luonnin jälkeen tutkitaan määritysten mukaisen ohjelmiston mahdolliset ajot, joiden avulla tuotetaan testitapaukset systemaattisesti. [Sarma *et al.*, 2010]

Spec Explorer -työkalussa testimalli on yleistetty äärellinen tilakone eli abstrakti tilakone. Esi- ja jälkiehtoinen malli eli tilamalli toimii tämän työkalun mallin notaationa, ja mallinnuskielenä puolestaan toimii Spec#, joka on C#-kielen lisäosa [Marinescu *et al*., 2015]. Testimalli kirjoitetaan pseudokoodina toimimaan satunnaisissa tietorakenteissa. Tilojen väliset siirtymät määritellään aktioiden ja sallittujen siirtymien määrittämistä ehdoista. Spec Explorer käytännössä muuttaa käyttäjän tekemän abstraktin mallin sisäiseksi malliksi, jolloin abstraktit tilat muuttuvat konkreettisiksi tiloiksi ja konkreettiset toiminnot annetaan varsinaisina syötearvoina. Vaarana on, että tiloja syntyy liikaa ja seuraa tilaräjähdys. Tätä voidaan estää antamalla ehtoja, joilla tilojen luontia rajoitetaan. [Shafique ja Labiche, 2010]

Jorgensenin [2017] mukaan Spec Explorer työkalua on helppo käyttää, mutta työkalun opiskelu ja abstraktin tason ymmärtäminen vievät aikaa. Lisäksi työkalun käyttö vaatii .NET-ohjelmointikielien ja Visual Studion ympäristön osaamista ja tuntemusta.

4.12 Graphwalker

Graphwalker on avoimesti saatavilla oleva työkalu, jossa testit luodaan GraphML-muodossa olevan mallin mukaan. Kuten moni muukin työkalu myös GraphWalker luo testisarjan tilakoneen pohjalta. Mallinnusnotaationa GraphWalker käyttää siirtymäperusteista mallia [Marinescu *et al*., 2015]. GraphWalkerissa on seitsemän sisäänrakennettua kattavuuskriteeriä, joista testaaja valitsee yhden. Kattavuuskriteeri määrittelee, koska testien luonti pysäytetään. [Jorgensen, 2017]

Akpinar ja muut [2020] kuvaavat GraphWalker-työkalun realisoivan järjestelmän suunnatut graafiset mallit testausmetodeiksi. Suunnatut graafit koostuvat tiloja kuvaavista solmuista ja toimintoja esittävistä reunoista. Järjestelmämalli on siis mahdollista ilmaista graafina. GraphWalker luo testiluokkarajapinnan käyttäen suunnattuna graafina ilmaistun järjestelmämallin solmuja ja reunoja. Tämän jälkeen GraphML-tyypin mallista luodut testitapaukset suoritetaan kyseisessä rajapinnassa. [Akpinar *et al.*, 2020]

Jorgensenin [2017] mukaan monen muun avoimesti saatavilla olevan mallipohjaisen testauksen työkalun tapaan GraphWalker on monimutkainen asentaa, vaatii paljon tietotaitoa ja työkalusta olemassa oleva dokumentaatio on melko heikkoa. Jorgensen toteaa GraphWalker-työkalun käytön vaativan useita liitännäisiä ja Windows-ympäristössä toimiessa muuttujien ja turvallisuusasetusten muuttamista, jotta työkalua voidaan käyttää testauksen suorittamiseen. Mallien luontiin tarvittavien kulkukaavioiden tekemiseen tarvittava yEd-työkalu puolestaan on huomattavasti suoraviivaisempi asentaa. Alun haasteista huolimatta Jorgensen näkee työkalun olevan tehokas ja omaavan potentiaalia yleistyäkseen.

GraphWalker omaa suoraviivaisen suunnittelumenetelmän. Mallinnuksen aikana työkalu ei tarjoa mahdollisuutta pysäyttää mallinnusta, vaan pysäytys täytyy hoitaa erillisillä keinoin tietyin pysäytyskriteerein [Sivanadan ja Yogeesha, 2014]. Tämän takia GraphWalker soveltuu paremmin pienempien järjestelmien mallinnukseen. [Karlssonin *et al.*, 2022]

5 Mallipohjaisen testauksen hyödyt ja haasteet

Mallipohjaisessa testauksessa on omat hyötynsä, mutta siitä voidaan löytää myös haasteita. Tutkielman tutkimuskysymykseen mallipohjaisen testauksen hyödyistä ja haasteista haettiin vastauksia kertovan kirjallisuuskatsauksen avulla. Vielä tarkemmin määriteltynä toteutetaan tietämyksen tila -tyyppinen kirjallisuuskatsaus. Tässä kirjallisuuskatsauksen muodossa pyritään kokoamaan yleiskuva tutkittavasta aiheesta olemassa olevan kirjallisuuden pohjalta [Baumeister ja Leary, 1997].

Kiinnostuksen myötä tarkoituksena oli perehtyä mahdollisimman ajankohtaiseen kirjallisuuteen. Lyhyen kirjallisuuteen tutustumisen jälkeen päätin rajata tarkasteltavan aikavälin viimeiseen kymmeneen vuoteen eli 2014–2023. Tämä lähinnä siitä syystä, ettei hyötyjä ja haasteita tuntunut löytyvän kovin paljoa, vaikka kirjallisuutta kyllä löytyi. Tässä luvussa esitellään sekä kootaan yhteen kirjallisuuskatsauksen tuloksena löytyneitä hyötyjä ja haasteita.

5.1 Hyödyt

Mallipohjaisen testauksen mallit auttavat hallitsemaan testiprojektien monimutkaisuutta ja pitämään yllä käsityksen siitä, mitä ollaan testaamassa ja mitä halutaan testata. Kramerin ja Legeardin [2016] mukaan mallin luominen auttaa vahvistamaan testattavan ohjelmiston vaatimukset aikaisessa vaiheessa kehitystä. Kuten jo aiemminkin on todettu, mitä aikaisemmin virheet löydetään, olivat ne sitten ohjelmistossa tai vaatimusmäärittelyssä tai jossain muussa osassa, sitä vähemmän korjaaminen laskennallisesti tulee aiheuttamaan kustannuksia. Myös Wagner ja muut [2017] sekä Herpel ja muut [2016] mainitsevat varhaisen vaatimusten varmistumisen mallipohjaisen testauksen tuomaksi hyödyksi.

Graafinen esitys mallin myötä helpottaa myös kommunikaatiota projektin sidosryhmien välillä ja antaa konkreettisemman kuvan kohteesta. Mallipohjaisen testauksen mukana tuleva testien automaattinen luonti helpottaa ja nopeuttaa testien luontivaihetta ja lisäksi auttaa ylläpitämään dokumentoitua testivarastoa. Lisäksi mallipohjaisen testauksen myötä tuleva testausautomaatio mahdollistaa testauksen ympäri vuorokauden tai yleisesti ottaen myös aikaan, jolloin testausta ei manuaalisesti suoritettaisi. Tämä lisää käytettävissä olevaa testausaikaa. Tehtyjä malleja voidaan myös mahdollisesti käyttää uudestaan myöhemmissä projekteissa tai hyödyntää niitä tietämyspohjana liittyen toimialueeseen, johon malli on toteutettu. [Kramer ja Legeard, 2016] Samaa sanovat Wagner ja muut [2017] ja he aikovatkin hyödyntää kehitettyä malliarkkitehtuuria pohjana malleille myöhemmissäkin projekteissa.

Garousin ja muiden [2021] mukaan mallipohjainen testaus nostaa testauksen tehokkuutta havaita todelliset virheet testattavasta järjestelmästä. Tästä esimerkkinä he raportoivat vielä vuosien perusteellisen testauksen jälkeenkin löytäneensä ohjelmistostaan merkittäviäkin vikoja mallipohjaisen testauksen avulla. Lisäksi Garousi ja muut sanovat

johdannaisesti mallipohjaisen testauksen myötä testitapausten suunnittelukäytäntöjen parantuneen. Mallipohjainen testaus antaa myös mahdollisuuden systemaattiseen vaatimusten kattavuuden arviointiin ja niiden jäljitettävyyden parantamiseen [Garousi *et al.*, 2021].

Peleska ja muut [2018] toteavat omassa käytännön projektissaan mallipohjaisen testauksen hyödyksi automaattisen vaatimusten jäljityksen. Mallipohjainen testaus mahdollistaa automaattisen jäljitystietojen luonnin linkittämällä. Toinen havaittu hyöty on automaattinen epäsuorasti katettujen testitapausten tunnistaminen. Mallipohjaisessa testauksessa testitapauksia suoritettaessa saatetaan automaattisesti tulla kattaneeksi joku toinen samaan tilaan liittyvä testitapaus, kun testattavana oleva järjestelmä kulkee tilan läpi tarkistaessaan testattavaksi tarkoitettua testitapausta. Kolmas hyöty liittyy regressiotestaukseen. Kun testattavasta järjestelmästä julkaistaan uusi versio, tulee regressiotestauksen toimenpiteet analysoida kokonaan uudestaan. Mallipohjaisen testauksen avulla tässä päästään helpommalla, kun ei tarvitse muuttaa kuin testausmallia uusien ominaisuuksien mukaan, minkä jälkeen testit voidaan automaattisesti luoda ja toteuttaa uudistetulle järjestelmälle. [Peleska *et al*., 2018; Peleska, 2018]

Mallipohjainen testaus mahdollistaa testisarjojen validoinnin jo ennen kuin varsinainen testauksen kohde on testattavissa. Mallipohjaisen testauksen työkaluilla voidaan vaatimusmäärittelyjen pohjalta simuloida testattavaa järjestelmää. Hyötynä on, että pystytään arvioimaan testisarjojen toimivuutta ja vahvuutta järjestelmässä, vaikkei järjestelmä olisi vielä testattavassa vaiheessa. Edistyneet mallipohjaisen testauksen työkalut mahdollistavat myös tehokkaan testien epäonnistumisten analysoinnin. Kattavien analyysien perusteella on helpompaa tunnistaa, mitä tulisi korjata, jotta järjestelmä toimisi halutulla tavalla. Hyvät analyysit mahdollistavat varmemmin myös toimivamman järjestelmän. [Peleska *et al*., 2018; Peleska, 2018]

Herpel ja muut [2016] toteavat mallipohjaisen testauksen parantavan vaatimusmäärittelyä tarkentamalla ja tuomalla siihen lisää yksityiskohtia. Lisäksi Herpel ja muut kokevat mallipohjaisen testauksen vähentävän testitapauksien valintaan ja luontiin tarvittavaa ponnistelua. Mallipohjaisen testauksen myötä kattavuus nousee, kun käydään useampia polkuja ja polkuvaihtoehtoja läpi. Näin varmistetaan, ettei mikään polku jää huomaamatta, ellei mallissa ole vikaa.

Mallipohjaisella testauksella voidaan havaita virheitä paremmalla tarkkuudella ja enemmän, mutta tämä vaatii myös ahkeruutta ja panostusta. Voidaan myös todeta, että mallipohjainen testaus soveltuu hyvin toiminnallisuuden testaukseen, muttei niin hyvin käyttöliittymän graafisen ulkoasun testaukseen. [Schulze *et al.*, 2014]

Schulzen ja muiden näkemyksestä huolimatta käyttöliittymien testaukseen on pystytty myös onnistuneesti soveltamaan mallipohjaista testausta ja testauksesta on koettu

olevan hyötyä. Iqbal ja muut [2019] raportoivat käyttäneensä mallipohjaista testausta lentokoneen perusmittarinäytön (primary flight display) testaukseen. Testauksessa käytettiin apuna käyttöliittymän kuvan tekstintunnistukseen (optical character recognition) avoimesti saatavilla olevaa työkalua. Koska tekstintunnistustyökalun toiminta ei ollut sataprosenttista, käytettiin apuna lisämenetelmiä, kuten kuvan esikäsittelytekniikoita, jotta virheelliset tulkinnat pystyttiin välttämään. Igbal ja muut toteavat virheitä löytyneen merkittävästi mallipohjaista testausta hyödyntämällä. Samalla he kokevat yleisesti mallipohjaisen testauksen olevan hyödyllinen lähestymistapa käyttöliittymien testaukseen.

Akipinarin ja muiden [2020] näkemyksenä projektinsa pohjalta on, että mallipohjainen testaus lyhentää testien ajoaikaa perinteisiin modulaarisiin testeihin verrattuna. Mallipohjaisella testausmenetelmällä voidaan välttää toistuvia operaatioita testien sisällössä ja perinteisiin testimoduuleihin verrattuna koodirivejä on vähemmän. Akpinar ja muut [2020] toteavat näiden pohjalta, että mallipohjainen testaus vaatii testinkehittäjältä vähemmän työtä.

Weißleder ja Schlingloff [2014] havaitsevat testitapausten luonnin olevan nopeampaa ja toisaalta kokevat myös hyödyksi mahdollisuuden suorittaa testit sekä automaattisesti että manuaalisesti. Weißleder ja Schlingloff ovat samaa mieltä Peleskan ja muiden [2018] kanssa, että mallipohjaisesta testauksesta on merkittävää hyötyä regressiotestauksessa. Kun vaatimusmäärittelyä muutetaan ja testitapaukset joudutaan luomaan ja toteuttamaan uudestaan, käy se helpommin mallia apuna käyttäen kuin ilman mallia.

Mallipohjaisen testauksen mallin avulla voidaan käydä läpi kaikki mahdolliset vaihtoehdot, mikä vähentää virheiden määrää [Sivakumar *et al.*, 2020]. Ei kuitenkaan voida olettaa, että kaikki mahdolliset vaihtoehdot tulisi testattua, sillä jo melko pienelläkin järjestelmällä mahdollisten kombinaatioiden määrä on liian suuri. Sivakumar ja muut [2020] kertovat lisäksi mallipohjaisen testauksen tuottavan heille säästöjä niin kustannusten kuin ajankin suhteen.

Mohacsin ja muiden [2015] mukaan testauksen työmäärä ja erityisesti testien ylläpito vähenee mallipohjaisen testauksen myötä. Lisäksi he toteavat muitakin hyötyjä, joita ovat vaatimusten laadun paraneminen, jo alkuvaiheessa löydettyjen virheiden määrän kasvu, testitapausten laadun parantaminen sekä päällekkäisten testitapausten poistuminen. He kuitenkin mainitsevat, että näitä tuloksia on paljon vaikeampi määrittää ja tarkasti todentaa. Mohacsi ja muut [2015] toteavat vielä, että mallipohjaisen testauksen avulla voidaan jatkossa tuottaa parempia ja laadukkaampia järjestelmiä testauksen tehokkuuden ja nopeuden nousun myötä.

Drave ja muut [2018] sekä Kriebel ja muut [2018] ovat todenneet hyödyksi mallipohjaisen testauksen mukana tulevan mahdollisuuden automaatioon. Molemmat toteavat testitapausten luonnin helpottuvan automaattisen testien luonnin myötä. Ahmad ja muut

[2018] toteavat, kuinka mallipohjainen testaus parantaa ohjelmointirajapintojen yhteentoimivuutta. Mallipohjainen testaus on manuaalista testausta parempi arvioimaan järjestelmän dokumentaatiota ja lisäksi se soveltuu manuaalista testausta paremmin monitahoisiin käyttötapauksiin. [Marques *et al.*, 2014]

Tässä luvussa mainitut kirjallisuuskatsauksen myötä esiin nousseet hyödyt on kuvattu vielä kootusti seuraavan luvun lopussa yhdessä haasteiden kanssa. Tämä koonti on esitetty taulukossa 1.

5.2 Haasteet

Wagner ja muut [2017] toteavat mallipohjaisen testauksen hyödyillä olevan hintansa. Heidän mukaansa hyötyjen vaatimukset ovat kovat, sillä mallipohjaisen testauksen myötä testaus tarvitsee enemmän toimeksiantoa, laajemman työkaluketjun ja tietämystä useista testauksen strategioista. Lisäksi Wagner ja muut sanovat mallipohjaisen testauksen työkalujen ketjun aiheuttavan rajoitteita, kun testauksen kohteena on tarkka toteutuksen kohta.

Garousin ja muiden [2021] näkemyksen mukaan mallipohjaisen testauksen haasteena on tietämyksen ja lähteiden yleinen vähyys liittyen parhaisiin käytäntöihin ja suunnittelumalleihin mallipohjaisen testauksen mallien luomiseksi. Mallipohjaisen testauksen hyötyjen virallinen määrittäminen teollisuudenalalle on akateemisesta näkökulmasta haastavaa, sillä monesti tehdyt tutkimukset toimivat teoreettisissa konteksteissa [Garousi *et al.*, 2021].

Peleska ja muut [2018] kokevat mallipohjaisen testauksen haasteena tehokkaan ja oikein toimivan mallin luomisen. Mallinnus vaatii paljon vaivaa, koska mallin tulee olla riittävän yksityiskohtainen kattaakseen sekä testauksen että odotettujen testitulosten vahvistamisen. Mallin laatu on todella merkittävä testisarjojen laatuun vaikuttava tekijä. Peleska ja muut toteavat laadukkaan mallin luomisen vaativankin enemmän ammattitaitoa kuin perinteiset menetelmät.

Toinen Peleskan ja muiden [2018] havaitsema haaste liittyy rajapintoihin. Testauksen alla olevan järjestelmän hallinnoidessa suurta määrää samalla tavalla toimivia, mutta keskenään toisistaan erillisiä osia, tulee esittää testimallille nämä kaikki rajapinnat erikseen kuvattuina, mikä on aikaa vievää ja vaivalloista. Lisäksi Peleska ja muut kokevat myös hybridijärjestelmän ja mallipohjaisen testauksen yhteensovittamisen haastavaksi, sillä oraakkeleille tulee tällöin määrittää useampi muuttujan arvoalue yhden sijaan. Lisäksi haastetta tuottaa testien asetusten määrittäminen niin, että voitaisiin rajata testien luontia tilanteissa, joissa parametrien arvoalue on niin laaja, ettei kaikkia mahdollisia kombinaatiota voida käydä läpi. [Peleska *et al.*, 2018]

Kolchin ja muut [2019] puolestaan kokevat ongelmiksi mallipohjaisen testauksen hyötysuhteen sekä testisarjojen väärin kohdistuminen. He perustelevat väitettään sillä,

että jo pieni malli kaksoisattribuuteilla ja äärellisellä määrällä prosesseja voi tuottaa kohtuuttoman määrän tiloja tutkittavaksi. Mallipohjaisilla testauksen menetelmillä luodut testit ja testisarjat eivät ole myöskään aina tarpeeksi laadukkaita, vaan Kolchinin ja muiden mukaan luontivaiheessa työkalu keskittyy määritellyn kattavuuden saavuttamiseen käyttämällä joskus epäloogisiakin polkuja nostaakseen kattavuutta. Tästä seuraa yleensä myöhemmässä vaiheessa testauksellisia haasteita ja kulut kasvavat. Mallipohjaisen testauksen haasteena nähdään myös turhan selkeä riippuvuus testisarjojen koon ja virheiden havaitsemiskyvyn välillä. Virheiden havaitseminen vähenee selvästi, kun testisarjaa pienennetään. Lisäksi Kolchin ja muut kokevat raja-arvojen konkretisoinnin haastavaksi, kun testitapausten luonti tapahtuu symbolisen lähestymistavan kautta. Symbolit tulee määritellä, jotta testit voidaan suorittaa ja virheiden löytämisen edellytykset olisivat parhaat mahdolliset.

Weißleder ja Schlingloff [2014] mukaan mallipohjainen testaus tulee kannattavaksi vasta muutaman iteraation jälkeen, sillä mallipohjaisen testauksen toteutus ja työkalut tulee olla hyvin koulutettuna testienkehittäjille, jotta hyödyt saadaan esiin. Herpel ja muut [2016] puolestaan toteavat mallin luonnin olevan haaste mallipohjaisessa testauksessa varsinkin, jos mallin luonti ei ole ennestään tuttua. Tätä ei kuitenkaan voida pitää harvinaisena tai yllättävänä haasteena, sillä mallin luonnin haasteellisuuden mainitsevat kohdanneensa myös Ali ja Hemmati [2014] sekä de Cleva Farto ja Endo [2015]. Tähän liittyen de Cleva Farto ja Endo [2015] toteavat mallipohjaisen testauksen vaativan paljon kohdistettua osaamista ja kohteen vaatimusten tuntemusta.

Testattavaan kohteeseen sopivan työkalun valinta ja käyttö voivat olla haastavia. Tämän ohella Ali ja Hemmati [2014] kokevat haasteena skaalattavuusongelmat mallinnettaessa ei-toiminnallista käyttäytymistä. Tässä tapauksessa malli karkaa helposti liian suureksi. Lisäksi he kokevat testitapausten luonnin monitahoisten mallien kanssa haasteeksi. Testitapausten luonnin mallista kokevat haastavaksi myös Khan ja muut [2019]. Lisäksi he mainitsevat, että testaukseen käytettävän panostuksen minimointi on haastavaa eli toisin sanoen mallipohjaisen testauksen hyötysuhteen nostaminen koetaan haastavaksi. Mallipohjaiseen testaukseen panostetaan joko paljon tai ei ollenkaan.

Mallipohjaista testausta tehdessään Ali ja Hemmati [2014] huomasivat myös suurten testisarjojen suorittamisen vaativan odotettua enemmän aikaa ja fyysistä tekemistä. Lisäksi heidän mukaansa kestävyystestien toteutus testiympäristössä tulee kalliiksi, sillä virheellistä toimintaa jäljittelemään tarvitaan erikoisvälineitä. Testiympäristöön liittyen Ali ja Hemmati [2014] havaitsevat myös toisen haasteen, joka ilmenee testidatan luonnissa liittyen virheellisten tilanteiden jäljittelyyn verratakseen testattavan kohteen toimintaa suhteessa virheelliseen toimintaan. Myös testitapausten luonti mallista koetaan vaikeana.

Kun mallipohjaista testausta lähdetään tekemään teollisuuden isoissa projekteissa, aiheutuu monimutkaistuvan testimallien käsittelystä haasteita. Mallipohjaisen testauksen työkalut eivät myöskään välttämättä pysty tuottamaan oikein kohdistuneita testejä. Näitten kohtaamiensa haasteitten takia Petrenko ja muut [2015] toivoisivat jatkossa sujuvampaa testi-insinöörien ja työkalujen välistä yhteistyötä. Heidän mukaansa testaaja pystyy yhä edelleen kohdistamaan testauksen paremmin koneen puolestaan keskittyessä enemmän testien määrään.

Sivakumar ja muut [2020] mainitsevat kolme haastetta, joihin ovat törmänneet. Nämä haasteet ovat vaatimus erityiskoulutetusta osaajasta suorittamaan testaus, mallien luonnin ja ylläpidon vaatima merkittävä panostus sekä testitapausten laadun, kattavuuden ja toteutuksen haasteet. Mallipohjaista testausta ei voida suorittaa ilman osaamista, ja toteutus vaatii osaavaltakin toteuttajalta aikaa. Laadun ja kattavuuden määrittäminen valintakriteerien avulla ei ole välttämättä aina niin yksinkertaista.

Taulukossa 1 on koottuna yhteen kirjallisuuskatsauksen tuloksena löydetyt hyödyt ja haasteet. Nämä ovat kummatkin esitetty omassa sarakkeessaan listattuna.

Hyödyt	Haasteet
• Vaatimusten vahvistaminen aikaisessa vaiheessa kehitystä • Laadukkaampi vaatimusmäärittely • Laadukkaampi testaus • Parempi testattavan kohteen kattavuus • Parempi virheiden havaitsemiskyky • Hyvä toiminnallisuuden testaukseen • Soveltuu käyttöliittymien testaukseen • Lyhyempi testien ajoaika • Voidaan välttää toistuvia operaatioita testeissä • Vähäisempi koodirivien määrä • Testit ja ylläpito vaativat vähemmän työtä • Testitapausten luonti nopeampaa • Mahdollisuus suorittaa testit sekä automaattisesti että manuaalisesti • Säästää kustannuksissa • Vähemmän virheitä, josta seuraa laadukkaampi järjestelmä • Automaattinen testitapausten luonti ja luonnin helpottuminen • Parantaa ohjelmointirajapintojen yhteentoimivuutta • Järjestelmän dokumentaation arvioinnin parantuminen • Auttaa hallitsemaan testiprojektin monimutkaisuutta • Auttaa pitämään yllä käsitystä, mitä ollaan testaamassa • Graafisen esityksen myötä sidosryhmien kommunikaatio paranee • Auttaa ylläpitämään testivarastoa • Mahdollistaa testauksen kellon ympäri • Mallien uudelleenkäytettävyys • Vaatimusten jäljitettävyyden parantuminen • Automaattinen vaatimusten jäljitys • Automaattinen epäsuorien testitapausten tunnistaminen • Regressiotestauksen helpottuminen • Testisarjojen validointi ennen kuin testattava kohde on valmis • Työkalut mahdollistavat testattavan kohteen simuloinnin • Jotkin työkalut mahdollistavat epäonnistumisten analysoinnin, joka helpottaa virheiden korjausta	• Vaatii aikaa ja panostusta • Mallin luominen haastavaa • Työkalujen valinta ja käyttö • Skaalattavuusongelmat mallinnettaessa ei-toiminnallista käyttäytymistä • Testitapausten luonti mallista • Testidatan luonnissa virheellisten tilanteiden jäljittely testiympäristössä • Mallinnus haasteellista • Suurien testisarjojen suorittaminen • Testitapausten laadun arviointi • Kestävyystestien toteutus testiympäristössä kallista • Mallipohjaiseen testaukseen vaaditun ponnistelun minimointi • Testimallien käsittely teollisuuden kokoluokassa • Työkalujen kykenemättömyys luoda oikein kohdistettuja testejä • Kannattavaa vasta muutaman iteraation jälkeen, kun on opittu soveltamaan • Vaatii erityiskoulutetun osaajan • Testitapausten laatu • Vaatii tietämystä vaikea aloitteleville • Työkalujen ketjutuksen haasteet • Tietämyksen ja lähteiden vähyys • Hyötyjen määrittämisen vaikeus • Samalla tavalla toimivien rajapintojen jokaisen esittäminen erikseen, jos ovat erillisiä osia • Ei sovellu hybriditestaukseen • Ei sovellu käyttöliittymän graafisen ulkoasun testaukseen kovin hyvin • Hyötysuhteen arviointi, hyötyjä vai haittoja enemmän • Testisarjan pienetessä virheiden havaitsemiskyky laskee • Raja-arvojen konkretisointi symboleja käytettäessä

Taulukko 1. Kirjallisuuskatsauksen tuloksena löydetyt mallipohjaisen testauksen hyödyt ja haasteet.

6 Yhteenveto ja pohdinta

Tässä tutkielmassa esiteltiin ensin testausta ja mallipohjaista testausta yleisellä tasolla. Tämän jälkeen pyrittiin vastaamaan tutkimuskysymyksiin, mitä työkaluja mallipohjaisessa testauksessa käytetään ja millaisia hyötyjä ja haasteita mallipohjaiseen testaukseen liittyy. Mallipohjaisen testauksen hyötyjä ja haasteita etsittiin viimeisen kymmenen vuoden aikana julkaistuun kirjallisuuteen tutustuen.

Työkalujen kirjo osoittautui kirjallisuuden perusteella todella laajaksi, mikä saattaakin aiheuttaa vaikeuksia kuhunkin tilanteeseen sopivan työkalun valinnassa. Lisäksi haasteita tuottaa se, että työkaluista tulee olla riittävästi etukäteistietoa tai niihin tulee perehtyä ennen testauksen aloittamista, sillä kaikkia työkaluja ei voida käyttää kaikkien mallinnusnotaatioiden kanssa.

Kirjallisuuskatsauksen pohjalta nousi esiin kolme kiinnostavaa mallipohjaisen testauksen tuomaa hyötyä. Ensinnäkin löydettiin, että malli antaa selkeämmän kuvan testattavasta kohteesta jo varhaisessa vaiheessa kehitystä. Tämä on hyvin loogista, sillä esimerkiksi graafisesti piirretystä mallista on huomattavasti helpompi havaita kokonaisuus kuin pelkästään vaatimuksista lukemalla. Toisena hyötynä havaittiin konkreettiset hyödyt testauksien tuloksien laadussa. Mallipohjaisen testauksen myötä virheiden havaittavuus parani merkittävästi. Kolmas hyöty puolestaan oli testitapausten luonnin helpottuminen ja nopeutuminen. On melko odotettavaa, että testitapausten luonti on helpompaa, jos ne voidaan työkalulla tuottaa automaattisesti verrattuna siihen, että testaaja tuottaisi testitapaukset manuaalisesti itse.

Merkittäviä haasteita puolestaan olivat dokumentoitujen ohjeistusten puute ja mallinnuksen vaikeus. Mallin luonnin todettiin vaativan melko paljon osaamista niin käytettävästä työkalusta kuin itse mallintamisestakin. Tuloksissa nousi myös esiin, etteivät hyödyt tule ilmaiseksi. Mallipohjaisen testauksen käyttöönottaminen vaati useamman mielestä paljon taloudellista ja ajallista panostusta, jotta mallipohjaisen testauksen hyötyihin voidaan päästä käsiksi. On myös hyvin todennäköistä, että hyödyt näkyisivät vasta myöhemmissä projekteissa, kun koneisto on saatu kuntoon. Vaikka mallipohjainen testaus pitkällä aikavälillä toisi säästöjä, eivät kaikki ole joko halukkaita tai joissain tapauksissa jopa kykeneviä ottamaan mallipohjaista testausta käyttöön. Lisäksi haastavuuden tunne saattaa estää uusia testaajia tarttumasta mallipohjaisen testauksen menetelmiin. Toisaalta taas kokeneemmatkin käyttäjät saattavat pysyä mieluummin vanhassa itselle tutussa ja turvallisessa testauksen menetelmässä, varsinkin jos mallipohjaisen testauksen hyödyt eivät ole konkreettisesti ja selvästi nähtävissä. Kuten monessa muussakin asiassa niin myös tässä vanhasta toimivasta tavasta voi olla hankalaa vaihtaa uuteen.

Garousin ja muiden [2021] mielestä mallipohjaisen testauksen pariin pitää jatkossa saada lisää tutkijoita ja sen myötä tutkimuksia, jotta tietämys ja osaaminen aiheeseen kas-

vaa. Heidän näkemyksensä mukaan tarvitaan lisää mallipohjaisen testauksen suunnittelumalleja auttamaan uusia testaajia, joille mallipohjainen testaus ei ole ennestään tuttua. Näin voidaan mahdollistaa, että myös uudet tekijät pääsevät helpommin ja nopeammin alkuun ja pystyisivät tuottamaan korkealaatuisia malleja ilman, että tarvitaan koko ajan kokenut testaaja opastamaan. Uusia tekijöitä ja tutkijoita on kuitenkin haastavaa houkutella alalle, jos mallipohjaisen testauksen hyötyjä ei pystytä virallisesti määrittämään [Garousi *et al.*, 2021]. Khan ja muut [2018] ovat samoilla linjoilla Garousin ja muiden kanssa ja painottavat, että jatkossa myös raportoinnin laatuun tulee panostaa nykyistä enemmän. Aikaa tulee jatkossa käyttää enemmän mallipohjaisen testauksen toteutuksen vaatimien yksityiskohtien ja ohjeistusten raportointiin [Khan *et al*., 2018].

Näihin edellä esitettyihin näkemyksiin liittyen huomasin itsekin tutkielmaa tehdessäni, ettei konkreettisten ohjeistusten löytäminen siitä, kuinka malli varsinaisesti luodaan tai testitapaukset generoidaan mallista, ollut kovin helppoa. Lisäksi käytännön esimerkkejä löytyi muutenkin yllättävän vähän. Uskon kuitenkin, että mallipohjaisen testauksen projekteja on tehty, muttei niitä vain ole raportoitu ainakaan kaikkien nähtäville. Tämä toki varmasti osaltaan selittyy sillä, ettei ole riittävästi aikaa kunnolliseen raportointiin, kun muutenkin testauksen osalta aikataulu tahtoo yleensä olla tosi tiukka. Tai toisaalta voi olla, ettei haluta antaa kilpaileville yrityksille tietoja omasta hyväksi havaitusta menetelmästä. Olisin silti näistäkin huolimatta uskonut löytyvän edes muutamia tapauksia enemmän. Jos haasteisiin löydetään ratkaisuja, voi mallipohjainen testaus tulevaisuudessa yleistyä entisestään.

Hyötyjä ja haasteita pohdittaessa tulee ottaa myös huomioon, että haaste osalle testien kehittäjistä voi olla toisille kehittäjille hyöty tai päin vastoin. Tämänkin kirjallisuuskatsauksen tuloksista voidaan nähdä joitain samoja piirteitä kummallakin puolella. Tästä mielenkiintoinen esimerkki liittyy käyttöliittymiin, kun Schulze ja muut [2014] kokivat mallipohjaisen testauksen soveltuvan melko huonosti käyttöliittymien ulkoasun testaukseen ja puolestaan Igbal ja muut [2019] totesivat mallipohjaisen testauksen soveltuvan hyvin käyttöliittymien testaukseen. Tähän saattaa toki vaikuttaa ajan myötä tapahtunut mallipohjaisen testauksen kehittyminen. Toisaalta jo Schulzea ja muita aiemmin Jääskeläinen [2011] väitöskirjassaan sekä Takala ja muut [2011] artikkelissaan raportoivat käyttöliittymiin kohdistuneen aktiomalliin pohjautuvan mallipohjaisen testauksen onnistuneen hyvin.

Lisäksi projektilla ja monella muullakin asialla, kuten testaajien taitotasolla, on vaikutusta, mikä koetaan haasteeksi ja mikä ei. Tästä huolimatta sekä hyötyjä että haasteita varmasti löytyy ja tulee löytymään jatkossakin, vaikka haasteita yritetäänkin poistaa tai ainakin helpottaa.

7 Viiteluettelo

Aggarwal, K. K. & Yogesh Singh. 2005. *Software Engineering*. 2nd edition. New Age International Publishers.

Agrawal, Hiralal, Joseph R. Horgan, Edward W. Krauser & Saul A. London. 1993. Incremental regression testing. In: *Proceedings of the 1993 Conference on Software Maintenance,* 348–357.

Ahmad, Abbas, Fabrice Bouquet, Elizabeta Fourneret & Bruno Legeard. 2018. Model-based testing for internet of things systems. *Advances in Computers*, 108, 1–58.

Aichernig, Bernhard K., Wojciech Mostowski, Mohammed R. Mousavi, Martin Tappler & Masoumeh Taromirad. 2018. Model learning and model-based testing. In: Bennaceur, Amel, Rainer Hähnle & Karl Meinke (eds.), *Machine Learning for Dynamic Software Analysis: Potentials and Limits,* 11026, 74–100. Springer.

Akpinar, Pelin, Mehmet S. Aktas, Alper B. Keles, Yunus Balaman, Zeynep O. Guler & Oya Kalipsiz. 2020. Web application testing with model based testing method: case study. In: *Proceedings of the 2020 International Conference on Electrical, Communication, and Computer Engineering*, 1–6.

Ali, Shaukat & Hadi Hemmati. 2014. Model-based testing of video conferencing systems: challenges, lessons learnt, and results. In: *Proceedings of the 7th International Conference on Software Testing, Verification and Validation,* 353–362.

Ali, Shaukat, Hadi Hemmati, N. E. Holt, E. Arisholm & L. C. Briand. 2010. *Model Transformations as a Strategy to Automate Model-Based Testing-A Tool and Industrial Case Studies*. Technical Report, Simula Research Laboratory.

Basili, Victor R. & Barry T. Perricone. 1984. Software errors and complexity: an empirical investigation. *Communications of the ACM*, 27(1), 42–52.

Baker, Paul, Zhen R. Dai, Jens Grabowski, Øystein Haugen, Ina Schieferdecker & Clay Williams. 2008. *Model-Driven Testing Using the UML Testing Profile*. Springer.

Baumeister, Roy F. & Mark R. Leary. 1997. Writing Narrative Literature Reviews. *Review of General Psychology*, 1(3), 311–320.

Bell, Jason. 2020. *Machine Learning*. 2nd edition. Wiley.

Berner, Stefan, Roland Weber & Rudolf K. Keller. 2005. Observations and lessons learned from automated testing. In: *Proceedings of the 27th International Conference on Software Engineering*, 571–579.

Blackburn, Mark, Robert Busser & Aaron Nauman. 2004. Why model-based test automation is different and what you should know to get started. In: *Proceedings of the International Conference on Practical Software Quality and Testing*, 212–232.

Brar, Hanmeet K. & Puneet J. Kaur. 2015. Differentiating integration testing and unit testing. In*: Proceedings of the 2nd International Conference on Computing for Sustainable Global Development*, 796–798.

Daka, Ermira & Gordon Fraser. 2014. A survey on unit testing practices and problems. In: *Proceedings of the 25th International Symposium on Software Reliability Engineering*, 201–211.

Davis, Fred D. & Viswanath Venkatesh. 2004. Toward preprototype user acceptance testing of new information systems: implications for software project management. *IEEE Transactions on Engineering Management*, 51(1), 31–46.

Dalal, Siddharta R., Ashish Jain, Nachimuthu Karunanithi, J. M. Leaton, Christophe M. Lott, Gardner C. Patton & Bruce M. Horowitz. 1999. Model-based testing in practice. In: *Proceedings of the 21st International Conference on Software Engineering*, 285–294.

de Cleva Farto, Guillermo & Andre T. Endo. 2015. Evaluating the model-based testing approach in the context of mobile applications. *Electronic Notes in Theoretical Computer Science*, 314, 3–21.

Drave, Imke, Steffen Hillemacher, Timo Greifenberg, Bernhard Rumpe, Andreas Wortmann, Matthias Markthaler & Stefan Kriebel. 2018. Model-based testing of software-based system functions. In: *Proceedings of the 44th Euromicro Conference on Software Engineering and Advanced Applications,* 146–153.

Duggal, Gaurav & Bharti Suri. 2008. Understanding regression testing techniques. In: *Proceedings of the 2nd National Conference on Challenges and Opportunities in Information Technology*, 1–6.

Elbaum, Sebastian, Alexey G. Malishevsky & Gregg Rothermel. 2002. Test case prioritization: a family of empirical studies. *IEEE Transactions on Software Engineering*, 28(2), 159–182.

Engström, Emelie, Per Runeson & Mats Skoglund. 2010. A systematic review on regression test selection techniques. *Information and Software Technology*, 52(1), 14–30.

Garousi, Vahid, Alper B. Keleş, Yunus Balaman, Zeynep Ö. Güler & Andrea Arcuri. 2021. Model-based testing in practice: an experience report from the web applications domain. *Journal of Systems and Software*, 180, 1–28.

Garousi, Vahid & Frank Elberzhager. 2017. Test automation: not just for test execution. *IEEE Software*, 34(2), 90–96.

Gopalaswamy, Ramesh & Srinivasan Desikan. 2009. *Software Testing: Principles and Practice*. 1st edition. Pearson Education Canada.

Gupta, Varun & D. S. Chauhan. 2011. Hybrid regression testing technique: a multi layered approach. In: *Proceedings of the 2011 Annual IEEE India Conference*. 1–5.

Hambling, Brian & Pauline van Goethem. 2013. *User Acceptance Testing a Step-by-Step Guide*. BCS.

Hartman, Alan, A. Kirshin & K. Nagin. 2002. A test execution environment running abstract tests for distributed software. In: *Proceedings of Software Engineering and Applications*, 1–6.

Hartman, Alan & Kenneth Nagin. 2004. The AGEDIS tools for model based testing. *ACM SIGSOFT Software Engineering Notes*, 29(4), 129–132.

Haugset, Børge & Geir Kjetil Hanssen. 2008. Automated acceptance testing: A literature review and an industrial case study. In: *Proceedings of the Agile 2008 Conference*, 27–38.

Hayes, Linda G. 2004. *The automated testing handbook*. Software Testing Institute.

Herpel, H. J., M. Kerep, J. Li, J. Xie, B. Johansen, K. Kvinnesland, S. Krueger & P. Barrios. 2016. Model based testing of satellite on-board software - An industrial use case. In: *Proceedings of the Aerospace Conference*, 1–9.

Iqbal, Muhammad Z., Hassan Sartaj, Muhammad U. Khan, Fitash U. Haq & Ifrah Qaisar. 2019. A model-based testing approach for cockpit display systems of avionics. In: *Proceedings of the 22nd International Conference on Model Driven Engineering Languages and Systems,* 67–77.

Jan, Syed R., Syed T. U. Shah, Zia U. Johar, Yasin Shah & Fazlullah Khan. 2016. An innovative approach to investigate various software testing techniques and strategies. *International Journal of Scientific Research in Science, Engineering and Technology*, 2(2), 682–689.

Jena, Ajay Kumar, Himansu Das & Durga Prasad Mohapatra. 2020. *Automated Software Testing: Foundations, Applications and Challenges*. Springer.

Jorgensen, Paul C. 2017. *The Craft of Model-Based Testing*. CRC Press.

Jääskeläinen, Antti. 2011. *Design, Implementation and Use of a Test Model Library for GUI Testing of Smartphone Applications*. Doctoral dissertation, Tampere University of Technology.

Kaner, Cem, Jack Falk & Hung Q. Nguyen. 1999. *Testing Computer Software*. John Wiley & Sons.

Karlsson, Victor. A., Ahmed Almasri, Eduard P. Enoiu, Wasif Afzal & Peter Charbachi. 2022. Automation of the creation and execution of system level hardware-in-loop tests through model-based testing. In: *Proceedings of the 13th International Workshop on Automating Test Case Design, Selection and Evaluation*, 9–16.

Khan, Muhammad A., Ayesha Jadoon, Kazi M. S. Haq, Shahid Mumtaz & Jonathan Rodriques. 2019. An overview of resilient and automatic model-based testing approaches for automotive industry. In: *Proceedings of the International Conference on Communications Workshops,* 1–6.

Khan, Muhammad U., Sidra Iftikhar, Muhammad Z. Iqbal & Salman Sherin. 2018. Empirical studies omit reporting necessary details: a systematic literature review of reporting quality in model based testing. *Computer Standards and Interfaces*, 55, 156–170.

Khorikov, Vladimir. 2020. *Unit Testing Principles, Practices, and Patterns*. 1st edition. Manning Publications.

Kolchin, Alexander, Stepan Potiyenko & Thomas Weigert. 2019. Challenges for automated, model-based test scenario generation. In: Damasevicius, Robertas & Giedre Vasiljeviene (eds.), *Information and Software Technologies*, 1078, 182–194. Springer.

Kramer, Anne & Bruno Legeard. 2016. *Model-Based Testing Essentials - Guide to the ISTQB Certified Model-Based Tester Foundation Level*. 1st edition. John Wiley & Sons.

Krenn Willibald, Rubert Schlick, Stefan Tiran, Bernhard Aichernig, Elisabeth Jöbstl & Harald Brandl. 2015. MoMut::UML model-based mutation testing for UML. In: *Proceedings of the 8th International Conference on Software Testing, Verification and Validation*, 1–8.

Kriebel, Stefan, Matthias Markthaler, Karin S. Salman, Timo Greifenberg, Steffan Hillemacher, Bernhard Rumpe, Christoph Schulze, Andreas Wortmann, Philipp Orth & Johannes Richenhagen. 2018. Improving model-based testing in automotive software engineering. In: *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice,* 172–180.

Leung, Hareton K. N. & Lee White. 1989. Insights into regression testing. In: *Proceedings of the Conference on Software Maintenance-1989*, 60–69.

Leung, Hareton K. N. & Lee J. White. 1990. A study of integration testing and software regression at the integration level. In: *Proceedings of the Conference on Software Maintenance 1990*, 290–301.

Leung, Hareton K. N. & Lee J. White. 2002. Integration testing. In: Marciniak, John J. (ed.), *Encyclopedia of Software Engineering*. 2nd edition. John Wiley & Sons.

Leung, Hareton K. N. & Peter W. L. Wong. 1997. A study of user acceptance tests. *Software Quality Journal*, 6(2), 137–149.

Li, Wenbin, Franck Le Gall & Naum Spaseski. 2017. A survey on model-based testing tools for test case generation. In: Itsykson, Vladimir, Andre Scedrov & Victor Zakharov (eds.), *Tools and Methods of Program Analysis*, 779, 77–89. Springer.

Marinescu, Raluca, Cristina Seceleanu, Helene Le Guen & Paul Pettersson. 2015. A Reasearch overview of tool-supported model-based testing of requirements-based desings. In: Hurson, Ali R. (ed.), *Advances in Computers*, 98, 89–140. Academic Press.

Marques, Arthur, Franklin Ramalho & Wilkerson L. Andrade. 2014. Comparing model-based testing with traditional testing strategies: an empirical study. In: *Proceedings of the 7th International Conference on Software Testing, Verification and Validation Workshops,* 264–273.

Mathur, Aditya. 2013. *Foundations of Software Testing*. 2nd edition. Pearson Education India.

Miller, Roy & Christopher T. Collins. 2001. Acceptance testing. In: *Proceedings of the XPUniverse*, 1–7.

Mohacsi, Stefan, Michael Felderer & Armin Beer. 2015. Estimating the cost and benefit of model-based testing: a decision support procedure for the application of model-based testing in industry. In: *Proceedings of the 41st Euromicro Conference on Software Engineering and Advanced Applications*, 382–389.

Mohanty, Hrushikesha., J. R. Mohanty & Arunkumar Balakrishnan. 2017. *Trends in Software Testing*. Springer.

Mulkahainen, Markus. 2019. *Test Case Selection and Prioritization in Continuous Integration Environment.* Master of Science Thesis, Tampere University.

Mulkahainen, Markus, Kari Systä & Hannu-Matti Järvinen. 2022. Test case selection with incremental ML. In: Taibi, Davide, Marco Kuhrmann, Tommi Mikkonen, Jil Klünder & Pekka Abrahamsson (eds.), *Product-Focused Software Process Improvement: 23rd International Conference, PROFES 2022, Proceedings,* 401–417. *Lecture Notes in Computer Science*, 13709.

Oliinyk, Bohdan & Vasyl Oleksiuk. 2019. Automation in software testing, can we automate anything we want? In: *Proceedings of the 2nd Student Workshop on Computer Science & Software Engineering*, 224–234.

Patton, Ron. 2005. *Software Testing*. 2nd edition. Sams.

Peleska, Jan. 2018. Model-based avionic systems testing for the airbus family. In: *Proceedings of the 23rd European Test Symposium*, 1–10.

Peleska, Jan, Jörg Brauer & Wen-ling Huang. 2018. Model-based testing for avionic systems proven benefits and further challenges. In: Margaria, Tiziana & Bernhard Steffen (eds.), *Leveraging Applications of Formal Methods, Verification and Validation*, 11247, 82–103.

Petrenko, Alexandre, Omer Nguena Timo & S. Ramesh. 2015. Model-based testing of automotive software: some challenges and solutions. In: *Proceedings of the 52nd Annual Design Automation Conference,* 1–6.

Pocatilu, Paul. 2002. Automated software testing process. *Economy Informatics*, 1, 97–99.

Robinson Harry. 2003. Obstacles and opportunities for model-based testing in an industrial software environment. In: *Proceedings of the 1st European Conference on Model Driven Software Engineering*, 118–127.

Rodrigues, Elder M., Maicon Bernardino, Leandro T. Costa, Avelino Zorzo & Flavio M. Oliveira. 2015. PLeTsPerf - A model-based performance testing tool. In: *Proceedings of the 8th International Conference on Software Testing, Verification and Validation*, 1–8.

Rothermel, Gregg, Roland H. Untch, Chengyun Chu & Mary J. Harrold. 2001. Prioritizing test cases for regression testing. *IEEE Transactions on Software Engineering*, 27(10), 929–948.

Runeson, Per. 2006. A survey of unit testing practices. *IEEE Software*, 23(4), 22–29.

Sarma, Monalisa, P. V. R. Murthy, Sylvia Jell & Andreas Ulrich. 2010. Model-based testing in industry: a case study with two MBT tools. In: *Proceedings of the 5th Workshop on Automation of Software Test*, 87–90.

Schieferdecker, Ina. 2012. Model-based testing. *IEEE Software*, 29(1), 14–18.

Schulze Christoph, Dharmalingam Ganesan, Mikael Lindvall, Rance Cleaveland & Daniel Goldman. 2014. Assessing model-based testing: an empirical study conducted in industry. In: *Companion Proceedings of the 36th International Conference on Software Engineering*, 135–144.

Shafique, Muhammad & Yvan Labiche. 2010. *A Systematic Review of Model Based Testing Tool Support*. Technical Report, Carleton University.

Sharma, R. M. 2014. Quantitative analysis of automation and manual testing. *International Journal of Engineering and Innovative Technology*, 4(1), 252–257.

Sharma, Monika & Rigzin Angmo. 2014. Web based automation testing and tools. *International Journal of Computer Science and Information Technologies*, 5(1), 908–912.

Siavashi, Faezeh & Dragos Truscan. 2015. Environment modeling in model-based testing: concepts, prospects and research challenges. In: *Proceedings of the 19th International Conference on Evaluation and Assessment in Software Engineering*, 1–6.

Sivakumar, P., R. S. Sandhya Devi, A. D. Buwanesswaran, B. Vinoth Kumar, R. Raguram & M. Ranjithkumar. 2020. Model-based testing of car engine start/stop button debouncer model. In: *Proceedings of the 2nd International Conference on Inventive Research in Computing Applications,* 1077–1082.

Sivanandan, Sandeep & C. B. Yogeesha. 2014. Agile development cycle: approach to design an effective model based testing with behaviour driven automation framework. In: *Proceedings of the 20th Annual International Conference on Advanced Computing and Communications*, 22–25.

Sneha, Karuturi & Gowda M. Malle. 2017. Research on software testing techniques and software automation testing tools. In: *Proceedings of the 2017 International Conference on Energy, Communication, Data Analytics and Soft Computing*, 77–81.

Sommerville, Ian. 2016. *Software engineering*. 10th edition. Pearson.

Takala, Tommi, Mika Katara & Julian Harty. 2011. Experiences of system-level model-based GUI testing of an Android application. In: *Proceedings of the 4th International Conference on Software Testing, Verification and Validation,* 377–386.

Umar, Mubarak A. 2019. Comprehensive study of software testing: categories, levels, techniques, and types. *International Journal of Advance Research, Ideas and Innovations in Technology*, 5(6), 32–40.

Utting, Mark & Bruno Legeard. 2007. *Practical Model-Based Testing: A Tools Approach*. Elsevier Science & Technology.

Utting, Mark, Alexander Pretschner & Bruno Legeard. 2012. A taxonomy of model-based testing approaches. *Software: Testing, Verification and Reliability*, 22(5), 297–312.

Vance, Stephen. 2013. *Quality Code: Software Testing Principles, Practices, and Patterns*. Addison-Wesley.

Wagner, Florian, Sean W. Dalton & Arno Bergmann. 2017. Benefits and drawbacks of target specific model-based testing. In: *Proceedings of the 2017 International Conference on Research and Education in Mechatronics*, 1–5.

Wahl, Nancy J. & Yuejian Li. 1999. An overview of regression testing. *ACM SIGSOFT Software Engineering Notes*, 24(1), 69–73.

Weißleder, Stephan & Dehla Sokenou. 2010. ParTeG - a model-based testing tool. *Softwaretechnik-Trends*, 30(2), 1–2.

Weißleder, Stephan & Holger Schlingloff. 2014. An evaluation on model-based testing in embedded applications. In: *Proceedings of the 7th International Conference on Software Testing, Verification and Validation,* 223–232.

Wiklund, Kristian, Sigrid Eldh, Daniel Sundmark & Kristina Lundqvist. 2017. Impediments for software test automation: A systematic literature review. *Software: Testing, Verification and Reliability*, 27(8), 1–20.

Wissink, Tom & Carlos Amaro. 2006. Successful test automation for software maintenance. In: *Proceedings of the 22nd International Conference on Software Maintenance*, 265–266.

Wong, W. Eric, Joseph R. Horgan, Saul London & Hira Agrawal. 1997. A study of effective regression testing in practice. In: *Proceedings of the 8th International Symposium on Software Reliability Engineering*, 264–274.