Milla Henriksson

Testiautomaation työkalut sulautetuissa järjestelmissä

Tietotekniikan kandidaatintutkielma

28. huhtikuuta 2023

Jyväskylän yliopisto

Informaatioteknologian tiedekunta

Tekijä: Milla Henriksson

Yhteystiedot: milhe99@gmail.com

Ohjaaja: Annemari Auvinen

Työn nimi: Testiautomaation työkalut sulautetuissa järjestelmissä

Title in English: Automated software testing tools in embedded systems

Työ: Kandidaatintutkielma

Opintosuunta: Tietotekniikka

Sivumäärä: 32+0

Tiivistelmä: Tässä kirjallisuuskatsauksessa esitetään 7 erilaista testiautomaation työkalua, jotka on kehitetty vuoden 2010 jälkeen ja selvitetään, ovatko ne jollain tavalla hyödyllisiä testiautomaation näkökulmasta. Tulokset osoittavat, että valitut työkalut automatisoivat monia testitasoja ja testiaktiviteetteja. Toisaalta osa tutkimuksista ei voinut esittää suoraan tutkimuskysymyksiin vastaavia tuloksia, jolloin johtopäätökset jäivät vaillinaisiksi.

Avainsanat: testiautomaatio, ohjelmistotestaus, sulautetut järjestelmät, testityökalut

Abstract: In this literature review, 7 different automated software testing tools, that have been developed after the year 2010, are introduced. The study aims to investigate whether these tools are somehow beneficial in automating software testing processes. The results show that the chosen tools automated many test levels and test activities. On the other hand some of the studies were unable to answer some of the research questions. Hence, the conclusions of this literature review are incomplete.

Keywords: test automation, software testing, embedded systems, test tools

Termiluettelo

Mallintaminen	Todellisen järjestelmän esittämistä muulla tavalla kuin sillä itsellään.
Reaaliaikainen	Reaaliaikaisen järjestelmän tulee olla ajan tasalla tai reagoida oikea-aikaisesti (esim. sydämentahdistin).
Simulointi	Tekninen malli, jonka tarkoituksena on jäljitellä todellisen järjestelmän toimintaa.
Sulautettu järjestelmä	Laitteisto, joka on tehty tiettyä käyttötarkoitusta varten ja jota ohjaa tietokone (esim. älylaitteet).
Testiartefakti	Ohjelmistoa testatessa syntyvä sivutuote (esim. lokitiedosto).
Testiautomaatio	Ohjelmistotestauksen automatisointia.
Testioraakkeli	Tieto siitä, onko ohjelman ulostulo oikein vai väärin.
Testisviitti	Kokoelma testitapauksia, joita käytetään testaamaan ohjelmistoja.
Testityökalu	Tuote, joka tukee testauksen eri vaiheita automatisoidulla tavalla.
Validointi	Varmistetaan, että suunnitellut, aiemmin asetetut vaatimukset on saavutettu: onko oikea (engl. correct) tuote tuotettu?
Verifiointi	Varmistetaan, että tuotettu tuote vastaa suunniteltuja ja aiemmin asetettuja vaatimuksia: onko tuotettu se, mitä suunniteltiin?
Ympäristömalli	Kuvaus järjestelmän toiminnallisuuksista ja rakenteesta.

Kuviot

Kuvio 1. Kirjoittajien Broekman ja Notenboom (2003) kuvaukseen perustuva yksinkertainen esitys sulautetusta järjestelmästä. 9
Kuvio 2. Tutkijoiden Yoshida ym. (2017) esimerkki symbolisesta suorittamisesta. 17

Taulukot

Taulukko 1. Tutkijoiden Garousi ym. (2018) taulukosta rajausten perusteella valitut tutkimukset. 2
Taulukko 2. Yhteenveto tutkimukseen valituista työkaluista. 12

Sisällys

1 JOHDANTO 1

2 OHJELMISTOTESTAUS 4
2.1 Testityypit 4
2.2 Testitasot 5
2.3 Testitekniikat 5
2.4 Testiautomaatio 7
2.5 Testityökalut 8

3 OHJELMISTOTESTAUS SULAUTETUISSA JÄRJESTELMISSÄ 9
3.1 Simulaatio 10
3.2 Prototyyppi 11
3.3 Esi- ja jälkituotanto 11

4 TYÖKALUT OHJELMISTOTESTAUKSEN AUTOMATISOINNILLE SULAUTETUISSA JÄRJESTELMISSÄ 12
4.1 Ympäristön mallinnus sekä simulointi UML:n avulla 13
4.2 ATEMES 13
4.3 PLOOSE 14
4.4 KLOVER 15
4.5 AutoETF 16
4.6 radCHECK ja radCASE 18
4.7 Ubuild 19

5 MENETELMIEN HYÖDYLLISYYDESTÄ 21

6 YHTEENVETO 23

LÄHTEET 24

1 Johdanto

Nykypäivänä olemme riippuvaisia siitä, että erilaisia tietokonejärjestelmiä sisältävät laitteemme toimivat virheettömästi, sillä virheiden aiheuttamat seuraukset voivat olla tuhoisia. Näiden laitteiden ohjelmistotestaaminen on siksi erityisen tärkeää. Tunnettuna esimerkkinä voidaan pitää vahinkoa, jolloin NASAn luotain Mars Climate Observer lähestyi liian lähelle Marsin pintaa ja tuhoutui ohjelmistovirheen takia (Oberg 1999).

Yksi keino varmistaa ohjelmiston laatu ja virheettömyys on käyttää työkaluja, jotka automatisoivat ohjelmistotestausta. Kirjoittajien Broekman ja Notenboom (2003) mukaan testiautomaation työkaluja on järkevä käyttää vain silloin, kun se kannattaa rahallisesti, ajallisesti tai selvästi parantaa ohjelmiston laatua. Testaamisprosessissa voi tulla vastaan tilanteita, joissa täytyy suorittaa esimerkiksi useasti toistuvia testejä; helposti virheille altistuvia testejä; testejä, joilla on iso määrä syöttömuuttujia tai testejä, jotka vaativat erityisiä laitteita esimerkiksi oikean sisääntulosignaalin generoimiseksi (esimerkiksi CMU200, joka on yhtiön Rhode&Schwarz kehittämä laite radiokommunikaation testaamista varten). Tällöin testaamisen automatisoiminen voi olla hyödyllistä.

Tutkimuksessa on kiinnostuttu testiautomaation työkaluista sulautetuissa järjestelmissä. Ei ole olemassa yhtä oikeaa tapaa tai menetelmää testata sulautettuja järjestelmiä. "Sulautetut järjestelmät"on kattotermi ja sen alle mahtuu monenlaisia järjestelmiä, joihin ei voi millään soveltaa yhtä ainutta menetelmää tai työkalua. Käytetyt työkalut ovat jokaiselle järjestelmälle usein hyvin erilaisia. (Broekman ja Notenboom 2003.)

Kirjallisuuskatsauksen näkökulmaksi on valittu testiautomaation työkalut (engl. test tools), jotka ovat mahdollisesti yleisesti sovellettavissa. Tämä tarkoittaa sitä, että tutkimuksissa ei olla määritelty sovellusaluetta. Kirjallisuutta lähestytään seuraavilla kysymyksillä:

1. Minkälainen työkalu on kehitetty automatisoimaan ohjelmistotestausta?
2. Mihin ongelmaan se pyrkii mahdollisesti vastaamaan?
3. Onko työkalu hyödyllinen (ajallisesti, rahallisesti tai muulla tutkimuksen osoittamalla tavalla)?

Tutkimusstrategiseksi lähtökohdaksi on valittu kirjallisuuskatsaus, joka kokoaa aiheeseen liittyvää kirjallisuutta ja jäsentää sitä yhdessä kokonaisuudessa. Näkökulma on teknillinen, koska tutkitaan testiautomaation työkaluja. Tällaisen yleiskatsauksen tarkoituksena on tehdä hajautettu, olemassa oleva informaatio helposti saatavaksi.

Garousi ym. (2018) ovat keränneet taulukkoon kokoavasti tutkimuksia liittyen ohjelmistotestaukseen sulautetuissa järjestelmissä. Taulukosta on valikoitu tutkimukset, jotka ovat ylätunnisteiden *Test automation*, *GENERIC* sekä *Proposed new tool* alla. Nämä on listattu taulukossa 1. Tutkimustunnus viittaa tutkijoiden taulukossa ylätunnisteeseen *Source #*.

Tutkimustunnus	**Tutkimuksen otsikko**
47	An automated approach to reducing test suites for testing retargeted c compilers for embedded systems
88	Automatic testing environment for multi-core embedded software - atemes
138	Environment modeling and simulation for automated testing of soft real-time embedded software
199	On the integration of model-driven design and dynamic assertion-based verification for embedded software
294	Tool support for automated traceability of test-code artifacts in embedded software systems
303	Ubuild: automated testing and performance evaluation of embedded linux systems

Taulukko 1: Tutkijoiden Garousi ym. (2018) taulukosta rajausten perusteella valitut tutkimukset.

Näistä on karsittu pois lähteet, jotka ei voitu todeta vertaisarvioiduiksi. Tämä on tarkistettu etsimällä julkaisijoiden nimikkeet Julkaisufoorumin tietokannasta tai, kun mahdollista, julkaisijoiden virallisilta nettisivuilta. Tämän lisäksi on karsittu pois kaikki tutkimukset, jotka on julkaistu ennen vuotta 2010.

Muussa tapauksessa aiheeseen liittyviä lähteitä on haettu JYKDOK-palvelusta tai Google Scholar -hakukoneesta eri yhdistelmillä avainsanoista *sulautetut järjestelmät*, *testiautomaatio*, *ohjelmistotestaus* ja *testityökalut* sekä englanniksi että suomeksi.

Aluksi luvuissa 2 ja 3 avataan erilaisia ilmiöitä ja käsitteitä, jotka liittyvät ohjelmistotestaukseen ja ohjelmistotestaukseen sulautetuissa järjestelmissä. Tämän jälkeen perehdytään tutkimusta varten valittuihin tutkimuksiin luvussa 4. Luvuissa 4 ja 5 pyritään analysoimaan näitä artikkeleita tutkimuskysymyksien näkökulmasta. Lopussa tutkimuksesta tehdään vielä yhteenveto.

2 Ohjelmistotestaus

Ohjelmistotestaus on prosessi, jonka aikana arvioidaan, onko tietty ohjelmisto tai ohjelmistotuote virheetön ja toimiiko se suunnitellulla tavalla. Tarkoituksena on tehdä tuotteesta hyvälaatuinen ja tunnistaa tuotteessa olevat virheet (engl. error) ennen kuin se viedään yleiseen käyttöön. Näin varmistetaan, että tuotteen käytön aikana ei esiinny odottamattomia virheitä käyttäjille ja että se vastaa käyttäjien odotuksia ja tarpeita. (Homès 2012.)

Kuitenkin on usein mahdotonta löytää kaikkia ohjelmistossa olevia virheitä. Ohjelmistotestausta tulee tällöin nähdä prosessina, jonka aikana aktiivisesti etsitään virheitä sen sijaan että varmistetaan, että niitä ei ole. (Myers, Sandler ja Badgett 2011.)

2.1 Testityypit

Testattavassa kohteessa on usein monia testattavia ominaisuuksia. Homès (2012) ottaa esimerkiksi auton: auton hintaa, hevosvoimaa, luotettavuutta, mukavuutta tai nopeutta voidaan arvioida. Mainitut ominaisuudet ovat esimerkkejä ei-funktionaalisista testikohteista.

Funktionaaliset testit testaavat kohteen toiminnallisuuksia tai sen tarjoamia palveluita. Tämä ominaisuus voi olla esimerkiksi tuotteen turvallisuus tai yhteensopivuus muiden laitteiden kanssa. (Homès 2012). Funktionaaliset testit liittyvät tyypillisesti mustalaatikkotestaukseen (Bertolino 2007). Mustalaatikkotestausta käsitellään alaluvussa 2.3.

Ei-funktionaaliset testit puolestaan testaavat ominaisuuksia, jotka eivät ole kriittisiä testattavan kohteen toiminnan kannalta (Homès 2012). Esimerkkejä ei-funktionaalisista ominaisuuksista ovat suorituskyky ja käytettävyys.

Rakenteelliset testit perustuvat ohjelman rakenteeseen. Esimerkiksi yksikkötasolla (testitasot käsitellään alaluvussa 2.2) testit tutkivat koodin haaroja, ehtoja sekä koodikäskyjä. (Homès 2012.) Toisaalta rakenteelliset testit voidaan nähdä valkolaatikkotesteinä (Bertolino 2007). Valkolaatikkotestausta käsitellään alaluvussa 2.3.

Kaikkia näitä testityyppejä on mahdollista suorittaa eri testitasoilla, jotka käsitellään seuraa-

vaksi.

2.2 Testitasot

Tässä luvussa määritellään neljä eri testitasoa, jolla testaamista voidaan suorittaa: yksikkötestaus, integraatiotestaus, järjestelmätestaus sekä hyväksymistestaus.

Yksikkötestaus (engl. module testing) kohdistuu ohjelman pienimpiin osiin, esimerkiksi komponentteihin, moduuleihin ja funktioihin (Myers, Sandler ja Badgett 2011). Integraatiotestaus (engl. integration testing) taas pyrkii löytämään mahdollisia virheitä järjestelmien ja komponenttien välillä (Homès 2012).

Järjestelmätestaus (engl. system testing) on koko järjestelmän, sisältäen siihen liittyvien dokumentaation, konfiguraatioiden ja komponenttien, testaamista (Homès 2012). Sen tarkoituksena on varmistaa, että järjestelmä on vaatimuksien sekä spesifikaatioiden mukainen: siksi testit ovat yleensä mustalaatikkotestejä (Sawant, Bari ja Chawan 2012). Spesifikaatiot ja vaatimukset käsitellään alaluvussa 2.3.

Puolestaan hyväksymistestauksen (engl. acceptance testing) tarkoituksena on testata, että järjestelmä on käyttäjien ja asiakkaiden vaatimuksien mukainen. Tässä vaiheessa ei enää priorisoida virheiden etsintää. (Homès 2012.) Beetatestaus (engl. beta testing) on eräs hyväksymistestauksen muoto, jolloin tuotetta testataan jo kohdekäyttäjillä (Sawant, Bari ja Chawan 2012).

2.3 Testitekniikat

Testaamista voidaan kuvata prosessina, jolloin järjestelmä verifioidaan ja validoidaan (Jamil ym. 2016). Tällöin varmistetaan, että järjestelmä on vaatimuksien ja spesifikaatioiden mukainen ja että vaatimukset ja spesifikaatiot on saavutettu.

IEEE määrittelee spesifikaatiot (engl. specifications) seuraavasti:
Määritelmä. *Spesifikaatio on dokumentti, joka määrittelee kattavasti, tarkasti sekä verifioitavalla tavalla järjestelmän tai komponentin vaatimukset, toiminnallisuudet tai muut omi-*

naisuudet ja usein myös menettelyt, joilla mainitut määräykset on saavutettu.

Toisaalta IEEE määrittelee ohjelmiston vaatimukset (engl. requirements) seuraavasti:

1. Käyttäjän vaatima ehto tai kyky jonkin ongelman ratkaisemiseksi.
2. Ehto tai kyky, jonka ohjelma tulee omata, jotta jonkin formaalisti esitetyn dokumentin, esimerkiksi sopimuksen, asettama spesifikaatio täyttyy.
3. Dokumentoitu representaatio ehdosta tai kyvystä niin kuin on määritelty aiemmissa kohdissa.

Mustalaatikkotestaustekniikat perustuvat ohjelman vaatimuksiin ja spesifikaatioihin ja siihen, että ne on verifioitu (Homès 2012). Menetelmää kutsutaan mustalaatikoksi, koska ohjelman sisäistä rakennetta eli koodia ei huomioida (Khan ja Sadiq 2011). Tilakaaviotestaus on eräs mustalaatikkotestaustekniikka, jonka tavoitteena on tutkia ohjelman tiloja, siirtymiä ja niistä seuraavia tapahtumia (Homès 2012).

Lasilaatikkotestaustekniikat perustuvat ohjelman rakenteeseen, koodiriveihin, käskyihin sekä komponentteihin. Oletuksena on, että kun rakenteellisesti ohjelma verifioidaan ja validoidaan, ohjelma toimii oikein. (Homès 2012.) Menetelmää kutsutaan lasilaatikoksi, koska nimenomaan ohjelman sisäisiin ominaisuuksiin keskitytään (Khan ja Khan 2012). Mutaatiotestaus on eräs lasilaatikkotestaustekniikka, jolloin jotain koodin osaa muunnetaan aiheuttamaan virheitä ohjelmassa ja tutkitaan, havaitseeko testit muunnettuja osia (Homès 2012).

Mustalaatikkotestaus ja lasilaatikkotestaus testaavat koodia dynaamisesti. Siihen liittyy koodin ajaminen ja sen tarkastelu ajonaikaisesti. (Homès 2012.) Puolestaan staattiset tekniikat arvioivat koodia ilman, että koodia ajetaan (Fairley 1978).

Staattisia tekniikoita ovat esimerkiksi erilaiset katsaukset, läpikäynnit ja tarkastukset (Homès 2012). Staattisia tekniikoita voidaan soveltaa dokumentteihin ja koodin osiin, joita ei pysty ajamaan (Fairley 1978). Tästä syystä niitä voidaan käyttää ennen kuin käytetään dynaamisia tekniikoita, jotka taas vaativat toimivan järjestelmän. Staattiset tekniikat ovat yleensä myös halvempia verrattuna dynaamisiin tekniikoihin. (Homès 2012.)

2.4 Testiautomaatio

Ohjelmistotestauksen automaatisaatio voi olla hyvin hyödyllistä esimerkiksi ajallisesti ja rahallisesti (Broekman ja Notenboom 2003). Rafi ym. (2012) esittävät, että testiautomaatio voi tuoda seuraavia parannuksia:

1. Testiautomaatio tekee testattavasta kohteesta parempilaatuisen.
2. Testiautomaation avulla pystytään saavuttamaan korkea koodikattavuus.
3. Testaamiseen kuluva aika vähenee.
4. Testit ovat luotettavia.
5. Tuotteen laatuun luotetaan enemmän.
6. Testit ovat toistettavissa.
7. Testaamiseen kuluu vähemmän vaivaa.
8. Testaaminen on halvempaa.
9. Viat havaitaan entistä helpommin.

Kirjoittajien Fewster ja Graham (1999) mukaan testiautomaatio voi myös mahdollistaa entistä lyhyemmän markkinoilletuontiajan.

Toisaalta on olemassa tilanteita, jolloin on parasta tehdä testausta manuaalisesti tai jolloin testauksen automatisaatioon liittyy haittoja (Rafi ym. 2012):

1. Testiautomaatio ei voi täysin korvata manuaalista testaamista, koska kaikkea ei voida automatisoida.
2. Pysyviä hyötyjä ei välttämättä saavuteta.
3. Ylläpito on vaikeaa.
4. Automatisoidun prosessin kehittäminen vie aikaa.
5. Testiautomaatio kannustaa väärille tai liian korkeille odotuksille.
6. Strategia testausprosessin automatisoinnille on väärä.
7. Kehittäjillä ei ole riittävästi osaamista.

Kun testiautomaatio toteutetaan hyvin, on mahdollista välttää aiemmin mainitut ongelmat. Fewster ja Graham (1999) kirjoittavat, että huonosti toteutettu automatisaatio johtaa ajanhukkaan.

Automatisoitu testi ei ole aina parempi kuin manuaalisesti tehty testi. Automatisoidut testit eivät esimerkiksi usein löydä uusia virheitä, jotka on muuten jäänyt huomaamatta. Toisaalta kohonnut luottamus tuotteen laatuun voi olla virheellinen. On mahdollista, että automatisoitu testi on virheellinen tai vaillinainen. Testaamista automatisoivat työkalut eivät ole täydellisiä. (Fewster ja Graham 1999.)

Testaamisen automatisointi ei ole helppo prosessi ja sitä varten tarvitaan organisatorista tukea. Aikaa tulisi varata työkalujen valikoimiseen, työntekijöiden kouluttamiseen ja toisaalta myös eri vaihtoehtojen kokeilemiseen sekä tutkimiseen. (Fewster ja Graham 1999.)

2.5 Testityökalut

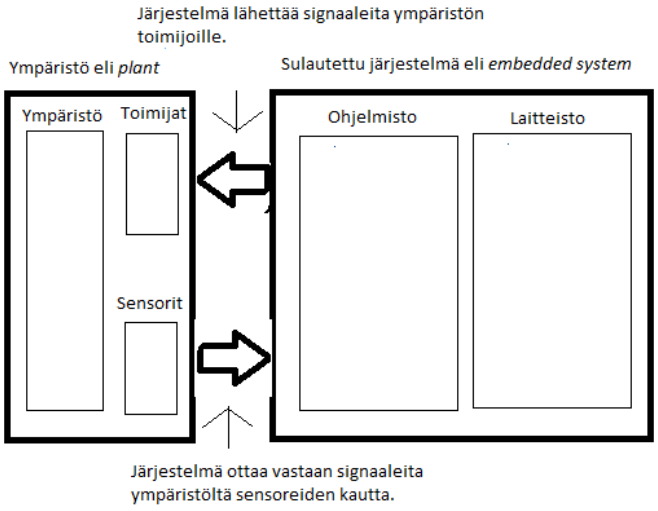
Testityökalujen avulla on mahdollista saavuttaa automatisaatio ohjelmistotestausprosessin aikana. Lähes mitä tahansa testiaktiviteettia voidaan tukea työkalulla: testien suorittamista, testisuunnittelua, testien evaluointia, testien hallintaa, jne. (Homès 2012.) Esimerkiksi Selenium on eräs kaupallinen testityökalu, joka tukee web-sovelluksien automatisointia.

3 Ohjelmistotestaus sulautetuissa järjestelmissä

Sulautettu järjestelmä on järjestelmä, joka koostuu ohjelmistosta ("softa", engl. software) ja laitteistosta ("rauta", engl. hardware) ja joka on toteutettu tiettyä käyttötarkoitusta varten (Barr 1999). Näitä ovat esimerkiksi kodissa olevat älylaitteet kuten mikro ja jääkaappi. Puolestaan perinteiset työpöytäkoneet ovat yleiskäyttöisiä (engl. general-purpose) eli niillä voi tehdä verrattain monenlaisia asioita (Barr 1999).

Kaikille sulautetuille järjestelmille on yhteistä se, että ne ovat vuorovaikutuksessa ympäristön kanssa. Tämä tapahtuu toimijoilla ja sensoreilla. Toimija viittaa moduuliin, joka ottaa vastaan sulautetulta järjestelmältä signaaleita ja muuttaa ympäristöä tarvittaessa niiden perusteella. Sensori taas viittaa moduuliin, joka vastaanottaa ympäristöltä ärsykkeitä ja muuntaa ne signaaleiksi, jotka syötetään sulautetulle järjestelmälle. (Broekman ja Notenboom 2003.)

Sulautetut järjestelmät ovat usein monimutkaisia järjestelmiä, joten tätä kirjallisuuskatsausta varten kuviossa 1 esitetään yksinkertaistettu kuvaus sulautetusta järjestelmästä.



Kuvio 1. Kirjoittajien Broekman ja Notenboom (2003) kuvaukseen perustuva yksinkertainen esitys sulautetusta järjestelmästä.

Seuraavissa alaluvuissa kuvataan yksinkertaistetulla tavalla ohjelmistotestausprosessi sulautetuissa järjestelmissä. Broekman ja Notenboom (2003) tunnistavat neljä eri vaihetta: simulaatio, prototyyppi, esituotanto ja jälkituotanto.

3.1 Simulaatio

Tyypillisesti ensimmäisenä järjestelmästä luodaan malli ja sen puuttuvia fyysisiä osia simuloidaan, koska ne eivät ole vielä saatavilla (Broekman ja Notenboom 2003). Malli on kuvaus järjestelmästä ja sen toiminnasta. Puolestaan järjestelmän simulointi on järjestelmän mallin toiminnallistamista. (Maria 1997.) Esimerkiksi MATLABin mallintamisohjelman Simulinkin avulla on mahdollista luoda tällaisia malleja ja simuloida niitä.

Simulaatiota käytetään ennen oikeaa järjestelmää, koska tätä voidaan muokata helposti. Tämä olisi liian kallista tai jopa mahdotonta oikean järjestelmän kohdalla. Toisaalta sen avulla järjestelmää ja sen resurssien käyttöä voidaan optimoida. (Maria 1997.)

Aluksi järjestelmää testataan simulaatiovaiheessa, jossa Broekman ja Notenboom (2003) tunnistavat kolme vaihetta:

1. *Yksisuuntainen simulaatio*, jolloin sulautetun järjestelmän malli testataan eristyksessä. Syötettä ohjataan simuloituun järjestelmään, jonka jälkeen ulostuleva data analysoidaan. Vuorovaikutus ympäristön kanssa ei ole tässä vaiheessa oleellista.
2. *Palautteen simulointi*. Tässä vaiheessa testataan simuloidun järjestelmän ja ympäristön välistä vuorovaikutusta. Ympäristömalli generoi syötettä sulautetulle järjestelmälle. Ulostulo syötetään takaisin ympäristömallille, minkä seurauksena uutta syötettä generoidaan järjestelmälle, ja niin edelleen.
3. *Pikavalmistusvaiheessa* (engl. rapid prototyping) simuloitua järjestelmää testataan sen oikeassa ympäristössä.

Simulointivaiheen testejä voidaan kutsua myös model-in-the-loop ("MiL") -testeiksi, jotka testaavat simuloitua mallia järjestelmästä simuloidussa ympäristössä. (Broekman ja Notenboom 2003.)

3.2 Prototyyppi

Simulaatiovaiheen jälkeen siirrytään prototyyppivaiheeseen, jossa simuloidut komponentit vähitellen vaihdetaan oikeiksi komponenteiksi. Tämän prosessin kautta voidaan luoda useampia prototyyppejä ja jokainen uusi prototyyppi on yhä enemmän lopputuotteen näköinen. (Broekman ja Notenboom 2003.)

Sekä software-in-the-loop ("SiL") että hard-ware-in-the-loop ("HiL") -testit ovat osa prototyyppivaihetta (Broekman ja Notenboom 2003). SiL-testausta varten malli muutetaan koodiksi joko manuaalisesti tai automaattisesti. Kuitenkin testausta suoritetaan vielä virtuaalisessa ympäristössä. (Matinnejad ym. 2013.) Joitain fyysisiä komponentteja saatetaan sisällyttää kokeilevasti. Puolestaan HiL-testit testaavat järjestelmän oikeaa laitteistoa simuloidussa ympäristössä. (Broekman ja Notenboom 2003.)

3.3 Esi- ja jälkituotanto

Esituotantovaiheessa tarkistetaan, että tuotettu tuote vastaa vaatimuksia ja on alan standardien mukainen. Tässä vaiheessa tuote näyttää enimmäkseen lopputuotteelta. Näin ollen on mahdollista suorittaa järjestelmätestausta (engl. system testing), jolloin oikeaa järjestelmää testataan oikeassa ympäristössä. (Broekman ja Notenboom 2003.)

Jälkituotantovaiheessa oleva tuote on tarpeeksi laadukas, että käyttäjät voivat ottaa sen käyttöön. Toisaalta ei voida olla varmoja esimerkiksi siitä, että tuotetut lopputuotteet ovat yhtä laadukkaita. Siksi on syytä vielä tehdä tarkistuksia (engl. inspections). (Broekman ja Notenboom 2003.) Nämä ovat menetelmiä, joiden avulla tuotetta arvioidaan systemaattisesti ja määrätyllä tavalla (Parnas ja Lawford 2003).

4 Työkalut ohjelmistotestauksen automatisoinnille sulautetuissa järjestelmissä

Tässä luvussa testityökaluja käsitellään tutkimuskysymyksien 1 ja 2 näkökulmasta, niin kuin on määritelty aiemmin luvussa 1. Luvussa 5 työkaluja käsitellään vielä tutkimuskysymyksen 3 näkökulmasta. Taulukkoon 2 on kerätty työkalut yhteenvetona.

Työkalun nimi	Testitaso	Toiminta
-	Järjestelmätestaus	Luo automaattisen ympäristösimulaattorin UML-mallin perusteella.
ATEMES	Yksikkötestaus	Testausympäristö, joka tuo automaattisia ratkaisuja yksikkötestaukseen ja analysoi testikattavuutta.
PLOOSE	Järjestelmätestaus	Automaattisesti pienentää testisviittejä uudelleenkohdistetuissa kääntäjissä.
KLOVER	Yksikkötestaus	Luo automaattisia testejä ja analysoi testikattavuutta.
AutoETF	Yksikkötestaus, integraatiotestaus ja järjestelmätestaus	Automaattisesti analysoi ja havainnollistaa jäljitettävyyttä.
radCHECK, radCASE	Järjestelmätestaus	Suorittaa automaattista verifiointia hyödyntämällä mallivetoista suunnittelua ja assertiopohjaista verifiointia.
Ubuild	Järjestelmätestaus	Suorittaa testejä automaattisesti.

Taulukko 2: Yhteenveto tutkimukseen valituista työkaluista.

4.1 Ympäristön mallinnus sekä simulointi UML:n avulla

Iqbal, Arcuri ja Briand (2015) esittävät artikkelissaan uuden menetelmän ympäristön mallintamista ja simulointia varten. Ympäristömallit kuvaavat ympäristön sekä rakenteellisia että toiminnallisia ominaisuuksia.

Tutkijat kirjoittavat, että ympäristöjen mallintaminen mahdollistaa kolmen erilaisen tehtävän automatisoimisen:

1. Ympäristösimulaattorin luomisen ja näin testauksen ilman varsinaista laitteistoa.
2. Testitapauksien valinnan. Testitapaukset sisältävät tarvittavat vaiheet, datan, edellytykset sekä jälkiehdot jonkun testattavan ominaisuuden verifioimiseksi (Homès 2012, 26).
3. Tuloksien evaluoinnin. Testien evaluoinnin toinen määritelmä on testioraakkelit, jotka vertaavat testien odotettuja tuloksia niiden toteutuneisiin tuloksiin. Niiden avulla määritellään, onko testi läpäissyt vai epäonnistunut. (Homès 2012.)

Menetelmää varten tarvitaan UML-kieltä ja sen lisäosia MARTE (Modeling and Analysis of Real Time and Embedded systems) sekä OCL (Object Constraint Language). Ensimmäisenä mallintaja loisi ympäristömallin UML:n luokkakaavion (engl. class diagram) sekä tilakoneen (engl. state machine) avulla. Sen tulisi kuvata ympäristön rakennetta ja toiminnallisuuksia tarpeeksi kattavasti, jotta siitä on mahdollista luoda ympäristösimulaattori. (Iqbal, Arcuri ja Briand 2015.)

Ympäristömalli muutettaisiin sitten Java-pohjaiseksi simulaattoriksi käyttäen tutkijoiden itsetekemiä sääntöjä. Nämä säännöt toteutettiin käyttämällä MOFScript-työkalua, joka kykenee muuttamaan mallit tekstiksi. (Iqbal, Arcuri ja Briand 2015.)

4.2 ATEMES

Ohjelmistotestauksen yksikkötestaamista ja testikattavuuden analysointia voidaan automatisoida tutkijoiden Koong ym. (2012) kehittämän ATEMES-työkalun (Automatic Testing Environment for Multi-core Embedded Software) avulla. Testikattavuus on prosenttiluku, joka kuvaa kuinka kattavasti testisviitti testaa olemassa olevaa koodia. Testisviitti on taas kokoel-

ma testitapauksia. (Homès 2012.)

Tutkijat ovat pyrkineet vastamaan seuraaviin ongelmiin, jotka liittyvät moniytimisten sulautettujen järjestelmien ohjelmistotestaamiseen:

1. Miten voidaan vähentää työntekijän osalta vaativaa ja toistuvaa työtä?
2. Miten resurssirajoitteet voidaan käsitellä, kuten muistiin liittyvät rajoitteet?
3. Miten optimoidaan rinnakkaisuus niin, että saadaan maksimaalinen laskentateho?
4. Miten varmistetaan riittävä testikattavuus analyysin avulla?
5. Miten käsitellään datan synkronointi, jotta voidaan välttää esimerkiksi kilpailutilanteet?
6. Miten saadaan aikaiseksi tasokasta ja luotettavaa testaamista asiakastyytyväisyyden takaamiseksi?

ATEMES kykenee (Koong ym. 2012):

1. Automaattisesti luomaan C/C++-kielistä syöttödataa.
2. Automaattisesti luomaan CppUnit-pohjaisia testitapauksia.
3. Automaattisesti luomaan CppUnit-pohjaisen testiajurin, joka ohjaa testien sujuvuutta ja ottaa talteen ohjelmissa tapahtuvat ajonaikaiset virheet.
4. Toteuttamaan testaamista automaattisesti sekä toistuvasti.
5. Valitsemaan automaattisesti sopivan parametrin Intel TBB -kirjaston funktiolle, joka kontrolloi rinnakkaisuutta. Rinnakkaisuus viittaa useamman prosessoriyksikön käyttämiseen samaan aikaan (Allan ym. 1995).
6. Automaattisesti analysoi haarakattavuutta (engl. branch coverage) sekä rivikattavuutta (engl. line coverage). Sekä haarakattavuus että rivikattavuus ovat prosenttilukuja. Haarakattavuus kertoo, kuinka kattavasti testisviitti testaa ohjelmassa olevia haaroja. Rivikattavuus puolestaan kuvaa testisviitin testaamien rivien määrää. (Homès 2012.)

4.3 PLOOSE

Chae ym. (2011) ovat kehittäneet PLOOSE-työkalun, joka kykenee vähentämään testisviittien määrää, kun testataan uudelleenkohdistettuja kääntäjiä. Uudelleenkohdistetut kääntäjät

tehdään uusia ja tehokkaampia prosessorimalleja varten. Tällöin uusia kääntäjiä ei kehitetä täysin tyhjästä, vaan ne konstruoidaan vain vanhojen kääntäjien back endiä muuttamalla. Back end viittaa siihen, miten kääntäjä generoi koodia kohdekielelle (engl. target language). Tämä kohdekieli voi olla esimerkiksi Assembly-koodia tai binäärikoodia (Ganapathi, Fischer ja Hennessy 1982).

Tutkijoiden esittämä työkalu perustuu välikieleen (engl. intermediate language). Heidän mukaan testisviitit, jotka luodaan välikielelle, ovat tehokkaita, koska konekieli riippuu suoraan välikielestä. Tästä syystä välikieli pitää testata täysin. Lähdekielen testit eivät välttämättä kata kaikkia välikielen ominaisuuksia.

Välikielelle luodut testisviitit voidaan luoda perustamalla testitapaukset johonkin valittuun kielioppiin. Kieliopin sääntöihin pohjautuva kattavuus tarkoittaa, että on testattu kaikki kieliopin säännöt. Testisviitin katsotaan olevan kattavuudeltaan riittävä, jos jokaiselle kielioppisäännölle on vähintään yksi testitapaus. (Chae ym. 2011.)

Tutkijat käyttivät esimerkkinä standardia C-kielioppia, joka on määritelty asiakirjassa ISO 9899. Testisviitistä poistettiin tarvittaessa joitakin C-ohjelmia sen jälkeen, kun testit käännettiin C:n RTL-kielelle (Register Transfer Language), joka on välikieli GCC:ssä. Työkalussa on analysaattori, joka tutkii jokaisen pätkän RTL-koodia ja selvittää, kattaako se sellaisen kielioppisäännön, jota ei olla vielä käsitelty. Jos kielioppisääntöä ei olla vielä käsitelty, koodi lisätään pienennettyyn testisviittiin.

4.4 KLOVER

KLOVER on tutkijoiden Yoshida ym. (2017) kehittämä työkalu, joka soveltuu C- ja C++-ohjelmiin. Sen keskiössä on symbolinen suorittaminen (engl. symbolic execution), joka näkee kaiken syöttödatan symbolisena.

Symbolista suorittamista voidaan havainnollistaa esimerkkifunktiolla *f*, joka ottaa parametrina int-tyyppisen muuttujan.

```
int f(int x){
    if (x==123)
```

```
        x++;
    if (x<0)
        x=-x;
    return;
}
```

Muuttujan x arvon ollessa -1, koodi etenisi seuraavasti: ensimmäinen ehto ei menisi läpi, mutta toinen ehto menisi läpi. Muuttujaan x sijoitettaisiin sitten luku 1, jonka jälkeen funktiosta poistuttaisiin. Symbolisessa suorittamisessa muuttujalle x ei anneta alkuarvoa, vaan sen arvo on symbolinen, esimerkiksi X. Tämän jälkeen ohjelman suoritusta seurataan jokainen haara ja ehto kerrallaan niin, että lopputuloksena on kokoelma polkuja (vrt. kuvio 2), joita ohjelma voi seurata muuttujan x:n arvosta riippumatta. Lopuksi polkujen asettamien ehtojen avulla voidaan esittää ratkaisuja. (Yoshida ym. 2017.)

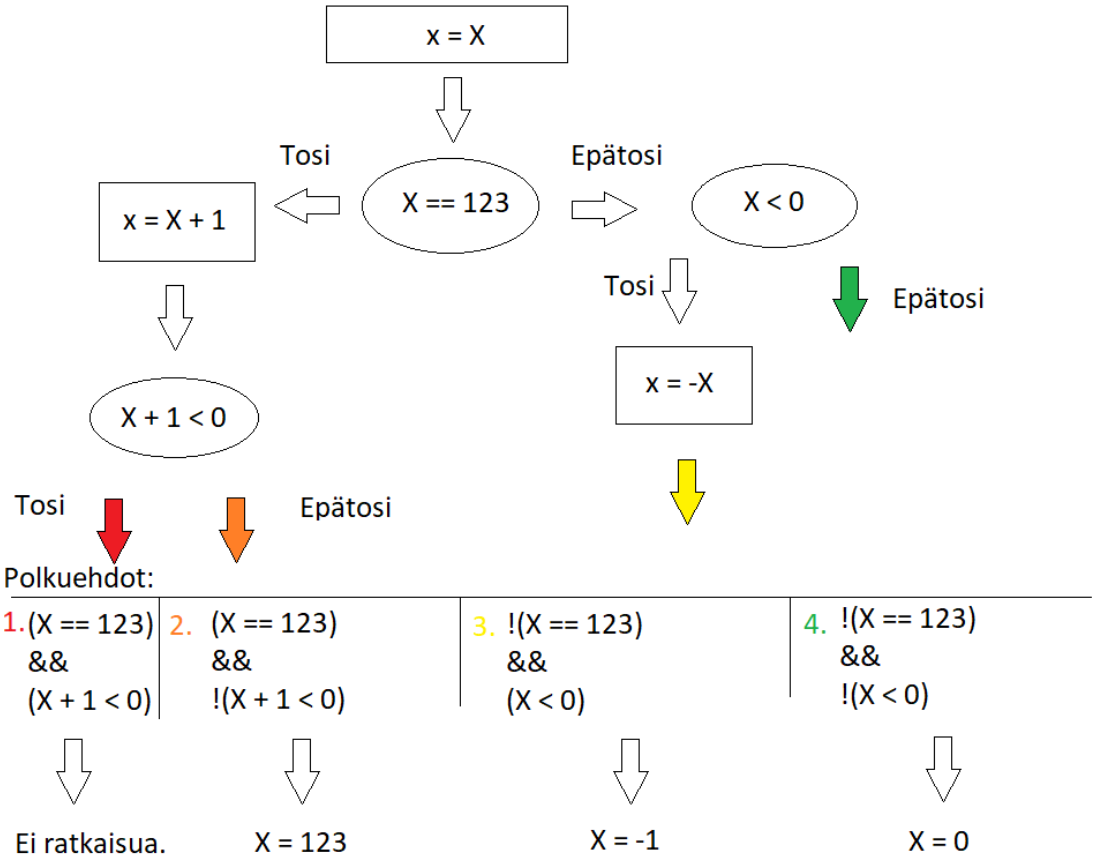
Käyttämällä symbolisen suorittamisen avulla saatuja ratkaisuja, KLOVER-työkalu kykenee luomaan testejä ja tutkimaan testikattavuutta. Kun ratkaisut asetetaan testien syöttödataksi, voidaan varmistaa, että testattavissa oleva ohjelma käy läpi ratkaisua vastaavan polun. (Yoshida ym. 2017.)

4.5 AutoETF

Tutkijoiden Wiederseiner, Garousi ja Smith (2011) kehittämä AutoETF vastaa yleiseen puutteelliseen määrään työkaluja, jotka automatisoivat jäljitettävyyden analysointia sulautetuissa järjestelmissä. Jäljitettävyys voidaan määritellä ohjelmiston kyvyksi nivoa tunnistettavat artefaktit toisiinsa ja ylläpitää ohjelmistossa olevia yhteyksiä ajan kuluessa (Cleland-Huang ym. 2014). Testiartefaktit ovat puolestaan ohjelmistotestauksen aikana syntyviä sivutuotteita, kuten lokitiedostot (Homès 2012).

Tutkijat perustivat työkalun toimimaan osittain työkalun E-Unit päälle. E-Unit-työkalun tarkoituksena on tehokkaasti analysoida koodikattavuutta sulautetuissa järjestelmissä. Se myös kykenee luomaan, kääntämään ja suorittamaan testisviittejä sekä raportoimaan tulokset.

AutoETF koostuu kahdesta moduulista (Wiederseiner, Garousi ja Smith 2011):



Kuvio 2. Tutkijoiden Yoshida ym. (2017) esimerkki symbolisesta suorittamisesta.

1. Moduuli, joka havaitsee jäljitettävyyden automaattisesti. Tämä viittaa prosessiin, jossa tunnistetaan linkit lähdekoodin ja testiartefaktien välillä.
2. Moduuli jäljitettävyyden visualisoimista varten.

Ensimmäisen moduulin generoidut linkit nähdään XML-tiedostossa, jonka jälkeen visualisaatiomoduuli analysoi tulokset ja luo niistä jäljitettävyyskaavion (engl. traceability graph, "TRG") (Wiederseiner, Garousi ja Smith 2011).

Tutkijoiden mukaan työkalun tuomat hyödyt ovat seuraavat:

1. Testikattavuus on parempi.
2. Testisviittien ylläpito on helpompaa.
3. Virheet voidaan paikallistaa.
4. On mahdollista tunnistaa, mitkä testit ovat tarpeettomia.

4.6 radCHECK ja radCASE

Sulautettujen järjestelmien ohjelmistojen verifiointi sekä toteuttaminen tunnistetaan olevan aikaa vievä ja virheille helposti altistuva prosessi tutkijoiden Di Guglielmo ym. (2013) artikkelissa. He ovat kehittäneet viitekehyksen (engl. framework), jossa yhdistyvät mallivetoinen suunnittelu (engl. model-driven design, "MDD") sekä assertiopohjainen verifiointi (engl. assertion-based verification, "ABV"). Näin voidaan yhdistää kaksi konseptia, joihin liittyy ongelmia erikseen ja tukea koodin suunnittelua sekä verifiointia.

Mallivetoisen suunnittelun menetelmät pyrkivät tekemään kehitettävän järjestelmän suunnittelusta mahdollisimman abstraktia käyttämällä malleja jokaisessa kehitysvaiheessa. Puolestaan assertiopohjainen verifiointi on menetelmä, jolla voidaan tehostaa verifiointia. Se tarjoaa tavan, jolla spesifikaatiot voidaan tarkistaa ja varmistaa, että ne ovat yhteensopivia sulautetun järjestelmän ohjelmiston kanssa. Menetelmä luo väliaikaisia assertioita (engl. temporal assertion) esimerkiksi PSL-kielellä (Property Specification Language), jolloin vältytään luonnollisen kielen aiheuttamalta moniselitteisyydeltä. (Di Guglielmo ym. 2013.) Assertioita on mahdollista toteuttaa myös esimerkiksi kielellä SystemVerilog. Alla on tutkijoiden Chang ja Foster (2023) esimerkki yksinkertaisesta assertiosta PSL-kielellä.

```
assert always (!(en1 & en2));
```

Suunnittelua ja verifiointia varten luotu viitekehys koostuu kahdesta osasta: radCASE sekä radCHECK. radCASE on UML-pohjainen mallintamis- ja kehitysympäristö sulautettujen järjestelmien ohjelmistojen mallivetoista suunnittelua varten. radCHECK on dynaamisen assertiopohjaisen verifioinnin ympäristö. Dynaamisuus viittaa siihen, että väliaikaiset assertiot verifioidaan simuloimalla. radCHECK tukee myös automaattisia tarkistuksia ja luo järjestelmälle syötettäviä ärsykkeitä (engl. stimuli). (Di Guglielmo ym. 2013.)

Suunnittelija voi radCASEn ja UML:n avulla luoda mallin ohjelmistosta. radCHECKin editorilla voidaan sitten luoda väliaikaisia assertioita, jotka ohjelmiston täytyy toteuttaa. radCASE automaattisesti kääntää UML-spesifikaatiot C-koodiksi ja ottaa laajennetun äärellisen tilakoneen (engl. extended finite state machine) verifioinnin tueksi. Laajennettu äärellinen tilakone on yleistys äärellisestä tilakoneesta, joka on eräs tapa mallintaa järjestelmiä. (Di Guglielmo ym. 2013.)

radCHECK generoi automaattisesti komponentteja, jotka monitoroivat sulautetun ohjelmiston kehitystä dynaamisen assertiopohjaisen verifioinnin aikana. Tätä ohjaavat ärsykkeet. Jos huomataan assertioiden olevaan epätosia, tätä tietoa voidaan käyttää parantamaan UML-spesifikaatioita inkrementaalisesti ja iteratiivisesti. (Di Guglielmo ym. 2013.)

4.7 Ubuild

Tutkijoiden Erculiani, Abeni ja Palopoli (2014) Ubuild-työkalu on suunniteltu tukemaan toistuvien sekä kontrolloitujen testien automaattista suorittamista Linux-käyttöympäristössä sulautetuissa järjestelmissä.

Työkalu koostuu kolmesta pääkomponentista (Erculiani, Abeni ja Palopoli 2014):

1. Kokoaja (engl. builer), joka on vastuussa komentorivin argumenttien jäsentämisestä, spesifikaatiotiedostojen välittämisestä jäsentäjälle, kohdevälimuistin hallitsemisesta sekä koontisuunnitelman suorittamisesta kohdekontrollerin kautta.
2. Välimuistin haltija, joka hoitaa välimuistiin kirjoittamisen ja sieltä lukemisen.
3. Skriptit, joita käytetään kääntämään koodia.

Ubuild-työkalu on tutkijoiden mukaan hyödyllinen jatkuvan integraation näkökulmasta, ja sen avulla voidaan tutkia, miten eri toteutusoptiot vaikuttavat järjestelmän suorituskykyyn.

5 Menetelmien hyödyllisyydestä

Tähän mennessä on esitelty erilaisia työkaluja ja selvitetty, minkälaisia ne ovat ja mihin ongelmiin ne mahdollisesti vastaavat. Tässä luvussa tarkoituksena on pohtia edellä mainittuja työkaluja tutkimuskysymyksen 3 näkökulmasta. Tutkimuskysymykseen 3 liittyy luvussa 2.4 käydyt ilmiöt: testiautomaation tuomat mahdolliset hyödyt ja haitat. On kuitenkin mahdollista, että jotkut artikkelit eivät anna vastausta tutkimuskysymykseen 3, jos tutkijat eivät ole pystyneet osoittamaan tutkimuksissaan siihen vastaavia tuloksia.

Esimerkiksi Ubuildin kehittäjien Erculiani, Abeni ja Palopoli (2014) ja AutoETF:n kehittäjien Wiederseiner, Garousi ja Smith (2011) artikkeleissa ei varsinaisesti tutkittu työkalun hyödyllisyyttä testiautomaation näkökulmasta, jonka takia näistä on vaikea tehdä johtopäätöksiä ja työkalujen lupaamiin parannuksiin on syytä suhtautua kriittisesti.

Puolestaan tutkijat Yoshida ym. (2017) pystyivät osoittamaan, että KLOVER-työkalun käyttäminen suhteessa manuaalitestaamiseen oli 750 kertaa nopeampaa, kun luotiin testejä SQLLitelle, ja 40 kertaa nopeampaa, kun luotiin testejä verkkokytkintuotteelle.

Tämän lisäksi tutkijat Iqbal, Arcuri ja Briand (2015) testasivat omaa menetelmää tapaustutkimuksessa viiteen eri reaaliaikaiseen järjestelmään ja onnistuivat osoittamaan, että:

1. Säännöt ovat riittävät ja näin kykenemät muuttamaan erikokoisia ympäristömalleja.
2. Toteutuneet simulaattorit kykenivät tunnistamaan virheitä järjestelmissä. Yksi kriittinen ongelma havaittiin asiakkaan järjestelmässä.

Kokonaisuutena tutkijoiden Iqbal, Arcuri ja Briand (2015) nimetön työkalu näyttäisi luovan luotettavia testejä ja tekee testaamisesta vaivatonta.

Tutkijoiden Koong ym. (2012) ATEMES-työkalun aiemmin mainitut toiminnallisuudet pystyttiin myös osoittamaan erilaisten kokeiden avulla. Toiminnallisuudet testattiin moniytimisessä ARM11-alustassa. Tämän lisäksi tutkijat testasivat työkalua 30 jatko-opiskelijalla, jotka kokivat sen olevan hyödyllinen automatisoimaan testausprosessia.

Toisaalta tutkimukset onnistuivat myös näyttämään tuloksia testiautomaatiota vastaan. Esi-

merkiksi tutkijoiden Chae ym. (2011) PLOOSE-työkalu ei tunnistanut virheitä yhtä hyvin verrattuna manuaalitestaukseen. Tämä voi olla ongelma turvallisuuskriittisissä järjestelmissä.

Tutkimuskysymyksestä poiketen on hyvä huomata, että PLOOSEn kehittäjien Chae ym. (2011) ja KLOVERin kehittäjien Yoshida ym. (2017) tutkimuksissa tutkimusta tehtiin tapaustutkimuksen muodossa tai otos jäi pieneksi. Niinpä tuloksia on nähtävä suuntaa antavina. Tapaustutkimukset käsittelevät aina yhtä tapausta tai kohdetta, jolloin ei voida olla varmoja, että tulokset pätevät laajalti. Ideaalitapauksessa edellä mainittuja työkaluja olisi kokeiltu laajemmassa mittakaavassa.

6 Yhteenveto

Katsauksen mukaan monia eri testitasoja, kuten yksikkötestausta (Koong ym. 2012) (Yoshida ym. 2017) ja järjestelmätestausta (Iqbal, Arcuri ja Briand 2015), ja testiaktiviteetteja, kuten testitapauksien suunnittelua (Chae ym. 2011), testien suorittamista (Di Guglielmo ym. 2013) (Erculiani, Abeni ja Palopoli 2014) ja testien evaluointia (Wiederseiner, Garousi ja Smith 2011), voidaan automatisoida työkaluilla sulautetuissa järjestelmissä.

Tämänkaltaiseen tutkimukseen liittyy myös käytännön hyötyjä. On tärkeää, että yritykset ovat tietoisia menetelmistä, joita on jo kehitetty, jotta resursseja voidaan suunnatta paremmin. Monet yritykset käyttävät liian paljon resursseja heidän mielestänsä uusien, mutta jo olemassa olevien testimenetelmien kehittämiseen (Garousi ym. 2018).

Vaikka artikkelit, jotka julkaistiin ennen 2010, on jätetty pois, on tässä tutkimuksessa silti heikkous se, että osa tutkimuksessa käytettävistä lähteistä on jopa yli 10 vuotta vanhoja mikä voi tehdä tutkimustuloksista merkityksettömiä nykyajassa.

Tämän lisäksi tietotekniikan tutkimuksissa on hyvä olla kriittinen siinä tapauksessa, kun tutkimus on kytköksissä jonkun yrityksen kanssa, koska on mahdollisuus, että tutkimus ei ole täysin puolueeton. Käsitellyistä tutkimuksista tutkijoiden Di Guglielmo ym. (2013) tutkimuksessa oli jonkinlainen kytkös yritykseen.

Toisaalta tämä katsaus voidaan nähdä pelkästään pintaraapaisuna vallalla olevaan tutkimuksen määrään, jota on olemassa aiheeseen liittyen, joten kokonaisvaltaista ja täysin kattavaa yleiskuvausta olisi ollut mahdotonta tehdä. Tämän lisäksi kaikista työkaluista ei olla välttämättä tehty tieteellistä tutkimusta tai niitä ei olla esitelty tieteellisessä kontekstissa, jolloin ne eivät luonnollisesti päässyt tämän artikkelin piiriin. Niinpä ei voida sanoa, että tässä on kaikki mahdollinen, vaan enemmänkin tämä on pelkkä pieni otos siitä, mitä on olemassa.

Lähteet

Allan, Vicki H., Reese B. Jones, Randall M. Lee ja Stephen J. Allan. 1995. “Software Pipelining”. *ACM Comput. Surv.* (New York, NY, USA) 27, numero 3 (syyskuu): 367–432. ISSN: 0360-0300. https://doi.org/10.1145/212094.212131.

Barr, Michael. 1999. *Programming embedded systems in C and C++*. "O’Reilly Media, Inc.".

Bertolino, Antonia. 2007. “Software Testing Research: Achievements, Challenges, Dreams”. Teoksessa *Future of Software Engineering (FOSE ’07),* 85–103. https://doi.org/10.1109/FOSE.2007.25.

Broekman, Bart, ja Edwin Notenboom. 2003. *Testing embedded software.* Pearson Education.

Chae, Heung Seok, Gyun Woo, Tae Yeon Kim, Jung Ho Bae ja Won-Young Kim. 2011. “An automated approach to reducing test suites for testing retargeted C compilers for embedded systems”. *Journal of Systems and Software* 84 (12): 2053–2064. ISSN: 0164-1212. https://doi.org/https://doi.org/10.1016/j.jss.2011.04.023.

Chang, C., ja Harry Foster. 2023. “Property Specification: The key to an Assertion-Based Verification Platform” (huhtikuu).

Cleland-Huang, Jane, Orlena C. Z. Gotel, Jane Huffman Hayes, Patrick Mäder ja Andrea Zisman. 2014. “Software Traceability: Trends and Future Directions”. Teoksessa *Future of Software Engineering Proceedings,* 55–69. FOSE 2014. Hyderabad, India: Association for Computing Machinery. ISBN: 9781450328654. https://doi.org/10.1145/2593882.2593891.

Di Guglielmo, Giuseppe, Luigi Di Guglielmo, Andreas Foltinek, Masahiro Fujita, Franco Fummi, Cristina Marconcini ja Graziano Pravadelli. 2013. “On the integration of model-driven design and dynamic assertion-based verification for embedded software”. *Journal of Systems and Software* 86 (8): 2013–2033. ISSN: 0164-1212. https://doi.org/https://doi.org/10.1016/j.jss.2012.08.061.

Erculiani, Fabio, Luca Abeni ja Luigi Palopoli. 2014. “uBuild: Automated Testing and Performance Evaluation of Embedded Linux Systems”. Teoksessa *Architecture of Computing Systems – ARCS 2014,* toimittanut Erik Maehle, Kay Römer, Wolfgang Karl ja Eduardo Tovar, 123–134. Cham: Springer International Publishing. ISBN: 978-3-319-04891-8.

Fairley, R.E. 1978. “Tutorial: Static Analysis and Dynamic Testing of Computer Software”. *Computer* 11 (4): 14–23. https://doi.org/10.1109/C-M.1978.218132.

Fewster, Mark, ja Dorothy Graham. 1999. *Software test automation.* Addison-Wesley Reading.

Ganapathi, Mahadevan, Charles N. Fischer ja John L. Hennessy. 1982. “Retargetable Compiler Code Generation”. *ACM Comput. Surv.* (New York, NY, USA) 14, numero 4 (joulukuu): 573–592. ISSN: 0360-0300. https://doi.org/10.1145/356893.356897.

Garousi, Vahid, Michael Felderer, Çağrı Murat Karapıçak ja Uğur Yılmaz. 2018. “What We Know about Testing Embedded Software”. *IEEE Software* 35 (4): 62–69. https://doi.org/10.1109/MS.2018.2801541.

Homès, Bernard. 2012. *Fundamentals of Software Testing.* Hoboken, UNITED STATES: John Wiley & Sons, Incorporated. ISBN: 9781118603093.

IEEE. 1990. “IEEE Standard Glossary of Software Engineering Terminology”. *IEEE Std 610.12-1990,* 1–84. https://doi.org/10.1109/IEEESTD.1990.101064.

Iqbal, Muhammad Zohaib, Andrea Arcuri ja Lionel Briand. 2015. “Environment Modeling and Simulation for Automated Testing of Soft Real-Time Embedded Software”. *Softw. Syst. Model.* (Berlin, Heidelberg) 14, numero 1 (helmikuu): 483–524. ISSN: 1619-1366. https://doi.org/10.1007/s10270-013-0328-6.

Jamil, Muhammad Abid, Muhammad Arif, Normi Sham Awang Abubakar ja Akhlaq Ahmad. 2016. “Software Testing Techniques: A Literature Review”. Teoksessa *2016 6th International Conference on Information and Communication Technology for The Muslim World (ICT4M),* 177–182. https://doi.org/10.1109/ICT4M.2016.045.

Khan, Mohd. Ehmer, ja Farmeena Khan. 2012. “A Comparative Study of White Box, Black Box and Grey Box Testing Techniques”. *International Journal of Advanced Computer Science and Applications* 3:12–15.

Khan, Mumtaz Ahmad, ja Mohd. Sadiq. 2011. “Analysis of black box software testing techniques: A case study”. Teoksessa *The 2011 International Conference and Workshop on Current Trends in Information Technology (CTIT 11),* 1–5. https://doi.org/10.1109/CTIT.2011.6107931.

Koong, Chorng-Shiuh, Chihhsiong Shih, Pao-Ann Hsiung, Hung-Jui Lai, Chih-Hung Chang, William C. Chu, Nien-Lin Hsueh ja Chao-Tung Yang. 2012. “Automatic testing environment for multi-core embedded software—ATEMES”. Dynamic Analysis and Testing of Embedded Software, *Journal of Systems and Software* 85 (1): 43–60. ISSN: 0164-1212. https://doi.org/https://doi.org/10.1016/j.jss.2011.08.030.

Maria, Anu. 1997. “Introduction to Modeling and Simulation”. Teoksessa *Proceedings of the 29th Conference on Winter Simulation,* 7–13. WSC ’97. Atlanta, Georgia, USA: IEEE Computer Society. ISBN: 078034278X. https://doi.org/10.1145/268437.268440.

Matinnejad, Reza, Shiva Nejati, Lionel Briand, Thomas Bruckmann ja Claude Poull. 2013. “Automated Model-in-the-Loop Testing of Continuous Controllers Using Search”. Teoksessa *Search Based Software Engineering,* toimittanut Günther Ruhe ja Yuanyuan Zhang, 141–157. Berlin, Heidelberg: Springer Berlin Heidelberg. ISBN: 978-3-642-39742-4.

Myers, Glenford J., Corey Sandler ja Tom Badgett. 2011. *The Art of Software Testing.* John Wiley & Sons, Incorporated.

Oberg, J. 1999. “Why the Mars probe went off course [accident investigation]”. *IEEE Spectrum* 36 (12): 34–39. https://doi.org/10.1109/6.809121.

Parnas, D.L., ja M. Lawford. 2003. “The role of inspection in software quality assurance”. *IEEE Transactions on Software Engineering* 29 (8): 674–676. https://doi.org/10.1109/TSE.2003.1223642.

Rafi, Dudekula Mohammad, Katam Reddy Kiran Moses, Kai Petersen ja Mika V. Mäntylä. 2012. “Benefits and limitations of automated software testing: Systematic literature review and practitioner survey”. Teoksessa *2012 7th International Workshop on Automation of Software Test (AST),* 36–42. https://doi.org/10.1109/IWAST.2012.6228988.

Rhode&Schwarz. n.d. “R&S®CMU200 Universal Radio Communication Tester Specifications”. (accessed: 05.04.2023). https://scdn.rohde-schwarz.com/ur/pws/dl_downloads/dl_common_library/dl_brochures_and_datasheets/pdf_1/CMU200_dat-sw_en.pdf.

Sawant, Abhijit A, Pranit H Bari ja PM Chawan. 2012. “Software testing techniques and strategies”. *International Journal of Engineering Research and Applications (IJERA)* 2 (3): 980–986.

Selenium. n.d. “The Selenium Browser Automation Project”. (accessed: 08.04.2023). https://www.selenium.dev/documentation/.

Wiederseiner, Christian, Vahid Garousi ja Michael Smith. 2011. “Tool Support for Automated Traceability of Test/Code Artifacts in Embedded Software Systems”. Teoksessa *2011IEEE 10th International Conference on Trust, Security and Privacy in Computing and Communications,* 1109–1117. https://doi.org/10.1109/TrustCom.2011.151.

Yoshida, Hiroaki, Guodong Li, Takuki Kamiya, Indradeep Ghosh, Sreeranga Rajan, Susumu Tokumoto, Kazuki Munakata ja Tadahiro Uehara. 2017. “KLOVER: Automatic Test Generation for C and C++ Programs, Using Symbolic Execution”. *IEEE Software* 34 (5): 30–37. https://doi.org/10.1109/MS.2017.3571576.