Jukka Määttälä

AUTOMATISOITU REGRESSIOTESTAUS SULAUTETUN JÄRJESTELMÄN KEHITYSTYÖSSÄ

Tietotekniikan pro gradu -tutkielma

Ohjelmistotekniikan linja

27.6.2005

Jyväskylän yliopisto

Tietotekniikan laitos

Tekijä: Jukka Määttälä.

Yhteystiedot: jukka.maattala@chydenius.fi.

Työn nimi: Automatisoitu regressiotestaus sulautetun järjestelmän kehitystyössä.

Title in English: Automated regression testing in the development of an embedded system.

Työ: Pro gradu -tutkielma.

Sivumäärä: 95+12.

Linja: Ohjelmistotekniikka.

Teettäjä: Jyväskylän yliopisto, tietotekniikan laitos.

Avainsanat: Testaus, virheiden hallinta, regressiotestaus, testauksen automatisointi, sulautetun järjestelmän testaus, V-malli, TTCN-3.

Keywords: Testing, defect management, regression testing, test automation, testing embedded system, V-model, TTCN-3.

Tiivistelmä: Tutkielman keskeisenä tehtävänä on pohtia automatisoidun regressiotestauksen merkitystä sulautetun järjestelmän kehitystyössä. Tutkielman alussa on käsitelty testausta erilaisten ohjelmistotuotannon vaihemallien avulla selventämään testauksen sijoittumista ohjelmiston elinkaaressa. Testaamisen yksityiskohtaisemmassa selvittämisessä on paneuduttu virheiden hallintaan, erilaisiin testaustekniikoihin ja -tasoihin. Testauksen automatisoinnissa on käyty läpi myös automatisoitavissa olevat testaustoiminnot. Sulautettujen järjestelmien yhteydessä on ohjelmistotestauksen perinteisiä ja yleisiä tapoja sovellettu sulautettujen järjestelmien käyttöön yhdessä johdatuksen tietoliikenteeseen kanssa.

Tutkielman case-tapauksessa on tutkittu LOPO-projektin tuotoksena syntyneen lyhyen kantaman ja alhaista tiedonsiirtonopeutta tarjoavan langattoman LR-WPAN -laiteen (Low-Rate Wireless Personal Area Network) kehitystyötä. Tutkimuksessa on keskitytty enimmäkseen järjestelmätestauksen tasolla tapahtuvaan automatisoituun regressiotestaukseen. Tutkimuksen pohjalta yhteenvedossa on pohdittu yleisemmällä tasolla automatisoidun regressiotestauksen merkitystä sulautetun järjestelmän kehitystyössä sekä arvioidaan teorian ja käytännönkokemuksen pohjalta, kuinka testausta voitaisiin kehittää ylläpidettävyys huomioonottaen.

Abstract: The main task of this thesis is to consider the meaning of automated regression testing in the development of an embedded system. The beginning of the thesis covers testing in different phase models of software engineering to clarify the placement of testing in the life cycle of software. Emphasis has been placed on defect management, different kinds of testing techniques and testing levels. All the testing functions that are possible to automatize, have been gone through, during the automatization of the testing. In the context of embedded systems, the traditional and common software testing has been utilized in embedded systems.

In the case section of the thesis a short rate wireless LR-WPAN device (Low-Rate Wireless Personal Area Network) was used. The device was constructed in the LOPO project. During the research the focus has mainly been on automated regression testing at the system level. Based on the research automated regression testing has been considered regarding the development of embedded systems and an estimate was made based on theory and experience regarding how testing could be developed while keeping maintenance in mind.

Esipuhe

Ensimmäiseksi haluan kiittää kaikkia niitä henkilöitä, jotka ovat olleet tukemassa ja auttamassa minua tämän pro gradu -tutkielman tekemisessä. Erityiskiitokset Merja Tikkakoskelle ja Jukka Ihalaiselle, jotka auttoivat ja opastivat minua työni aikana. Professoreille Ismo Hakalalle sekä Markku Sakkiselle haluan osoittaa kiitokset tutkielmani ohjaamisesta ja tarkastamisesta. Unohtaa ei myöskään sovi Chydenius-instituutti – Kokkolan Yliopistokeskusta, joka tarjosi minulle mahdollisuuden tutkielman tekemisen Tietoliikennelaboratorion suunnittelijana.

Lopuksi haluan kiittää kaikkia niitä henkilöitä, jotka ovat edesauttaneet minua tämän tutkielman tekemisessä ja muutenkin opinnoissani. Erityiskiitoksen haluan osoittaa Minnalle, jonka kannustuksella olen jaksanut eteenpäin opinnoissani.

Jukka Määttälä

Termiluettelo

ASN.1	Abstract Syntax Notation One
CH	Component Handler
CORBA	Common Object Request Broker Architecture
ECD	External CoDecs, ulkoinen kooderi-dekooderi
EDS	Encoding/Decoding System
ETS	Executable Test Suite
ETSI	The European Telecommunications Standards Institute
FFD	Full Function Device
HR-WPAN	High-Rate Wireless Personal Area Network
IEEE	The Institute of Electrical and Electronics Engineers
IP	Internet Protocol
IUT	Implementation Under Test
IXIT	Implementation Extra Information for Testing
LR-WPAN	Low-Rate Wireless Personal Area Network
MAC	Medium Access Control, siirtoyhteyskerros
MR-WPAN	Medium-Rate Wireless Personal Area Network
OSI	Open System Interconnection
PA	Platform Adapter

PCO	Point of Control and Observation
QoS	Quality of Service
RFD	Reduced Function Device
SA	SUT Adapter
SUT	System Under Test
SLIP	Serial Line IP
TC	Test Control
TCI	TTCN-3 Control Interface
TE	TTCN-3 executable
TID	Timer Identification
TL	Test Logging
TM	Test Management
TRI	TTCN-3 Runtime Interface
TTCN-3	The Testing and Test Control Notation version 3
T3RTS	TTCN-3 Runtime System
UDP	User Datagram Protocol
V&V -prosessi	Validointi ja verifiointi -prosessi
WLAN	Wireless Local Area Network
WPAN	Wireless Personal Area Network
WSN	Wireless Sensor Network

Sisältö

1 Johdanto

Tänä päivänä yhä kiristyvät ohjelmistoaikataulut aiheuttavat paineita ohjelmistojen elinkaarien lyhentämiselle. Vaikka elinkaaret lyhenevätkin, ei nykyaikana kuitenkaan enää ohjelmistojen elinkaaria lyhennetä testauksen kustannuksella. Kiristyvät ohjelmistomarkkinat ovat pakottaneet ohjelmistoyrityksiä panostamaan myös ohjelmistotestaukseen laadunvarmistuksen takia.

Nykyään testaus nähdään tärkeäksi ohjelmistotuotannon vaiheeksi, joka jatkuu läpi ohjelmiston elinkaaren. Testaus ei ole enää vain juuri ennen käyttöönottoa tehtävä toiminto, vaan keskeisessä osassa ohjelmiston suunnittelua ja kehitystä oleva toiminto. Ohjelmistovirheiden hinnan nousu, sitä korkeammaksi mitä myöhäisemmässä vaiheessa elinkaarta virhe löydetään, on lisännyt halua etsiä virheitä aikaisemmassa vaiheessa. Tämän vuoksi perinteisen ajonaikaisen testaamisen rinnalla tarvitaan verifiointia, joka mahdollistaa virheisiin käsiksi pääsyn jo ennen kuin riviäkään koodia on kirjoitettu. Toisaalta virheitäkään ei tarvitsisi tehdä, mikäli ne voitaisiin estää esimerkiksi formaalien menetelmien avulla.

Erityisesti ohjelmiston elinkaaren loppuvaiheessa tehtävä testaus on kriittinen aikataulun suhteen, jonka vuoksi sen nopeuttamiseksi on apua haettu testauksen automatisoinnilla. Elinkaaren lyhentäminen ei toisaalta ole ainoa hyöty mitä automatisoinnilla voidaan saavuttaa, vaan säästetty aika on yhtä tärkeätä kohdistaa testausta enemmän vaativiin kohteisiin, jolloin testauksesta saadaan kattavampi. Automatisointia ei kuitenkaan kannata pitää minään itsestäänselvyytenä, sillä huonosti toteutettu automatisointi aiheuttaa vain enemmän haittaa kuin hyötyä.

Testauksessa kannattaa erityisesti automatisoida usein toistettavat toiminnot. Näin ollen jokaisen ohjelmistoon tehdyn muutoksen tai lisäyksen jälkeen suoritettava regressiotestaus on erityisen kannattava automatisoinnin kohde. Perinteisestä ohjelmiston testauksesta hieman poikkeavassa sulautetun järjestelmän testaamisessa regressiotestaus kuuluu osaksi elinkaarta. Sulautetun järjestelmän kehitys ja samalla testaaminen eroavat perinteisen ohjelmiston testaamisesta, sen rinnakkain kehitettävien laitteiston ja ohjelmiston takia.

Tässä pro gradussa tulen käsittelemään sulautetun järjestelmän kehitystyötä automatisoidun regressiotestauksen näkökulmasta. Luvussa 2 käyn läpi ohjelmistotuotannon vaiheita ja vaihemalleja sekä ohjelmistotestauksen perusmallin. Luku 3 sisältää puolestaan testaamisen perusmääritteitä ja virheiden hallinnan jakamisen. Luvussa selvitän myös virheiden tunnistukseen ja testaamiseen liittyviä erilaisia testaustekniikoita ja -tasoja sekä käyn läpi regressiotestauksen perusajatusta. Testauksen automatisointiin olen keskittynyt luvussa 4, jossa on selvitetty myös testaustoimintoja. Luku 5 sisältää tietoliikenteeseen liittyviä perusasioita sekä sulautettujen järjestelmien läpikäymisen vaihemallien, testauksen perusmallin ja sulautettujen järjestelmien regressiotestauksen avulla. Luvussa 6 olen esitellyt case-tapauksessa käyttämiäni teknologioita, kuten LOPO-projektin tuotoksena tutkimustyöhön saadun laitteen sekä LR-WPAN -tekniikan (Low-Rate Wireless Personal Area Network) että TTCN-3 -testauskielen (The Testing and Test Control Notation version 3) ja -arkkitehtuurin. Luvassa 7 esittämässäni case-tapauksessa olen keskittynyt LR-WPAN -laitteen testaamiseen liittyviin asioihin esimerkin avulla ja pohtinut omien tekemisieni ja valintojeni onnistumisia. Lopuksi luvussa 8 olen tehnyt yhteenvedon koko automatisoidun regressiotestauksen soveltuvuudesta sulautetun järjestelmän kehitystyöhön ja esitellyt esiinnousseita jatkotutkimusaiheita.

2 Ohjelmistotuotanto

Ohjelmistotuotanto (engl. *software engineering*) on ohjelmistotyötä, jonka tavoitteena on tuottaa käyttäjille heidän kohtuulliset toiveensa ja odotuksensa täyttävä järjestelmä. Ohjelmistotyölle laaditaan ennalta suunnitelma, jolle reunaehdot asettaa kustannusarvio ja aikataulu. Ohjelmiston toteuttamisessa edetään jotakin ennalta valittua vaihemallia käyttäen, joka voi olla esimerkiksi vesiputousmalli tai inkrementaalinen malli.

Vaihemallit koostuvat erilaisista ohjelmistotuotannon vaiheista alkaen esitutkimuksesta ja päättyen käyttöönottoon ja ylläpitoon. Vaikka erilaiset vaihemallit tunnistavatkin testauksen yhtenä ohjelmistotuotannon aktiviteettina, eivät ne kuitenkaan anna testaajalle riittävää apua työnsä jäsentämiseen. Tämän takia ohjelmistotestaukseen on kehitetty oma V-malli, joka auttaa tunnistamaan eri tasoilla tehtävää testausta.

2.1 Ohjelmistotuotannon vaiheet

Ohjelmiston kehittymistä kuvataan ohjelmiston elinkaaren (engl. *life cycle*) vaiheilla, jotka käyvät ilmi vaihejakomallista. Riippuen vaihejakomallista alkaa elinkaari joko esitutkimuksesta tai määrittelyvaiheesta päättyen käyttöönottoon ja ylläpitoon. Sulautetun järjestelmän kehittämisen vaihejakomalli on hieman erilainen verrattuna tavalliseen ohjelmistotuotantoon, koska järjestelmässä kehitetään yhtä aikaa ohjelmistoa ja laitteistoa. Seuraavassa alaluvussa käsiteltävät ohjelmistotuotannon vaiheisiin pohjautuvat asiat perustuvat pääasiassa Haikalan ja Märijärven [14] teokseen.

2.1.1 Esitutkimus

Ohjelmistotuotannon elinkaaren ensimmäisen vaiheen, esitutkimuksen (engl. *feasibility study*), tehtävänä on asettaa yleiset järjestelmätason vaatimukset. Usein järjestelmätason vaatimuksia kutsutaankin asiakasvaatimuksiksi, koska ne määrittelevät asiakkaan tarpeita. Asiakastarpeiden analysointi ja tarkentaminen jatkuu yleensä koko määrittelyvaiheen ajan. Tämän vuoksi esitutkimusta ei aina nähdä omana vaiheena, vaan se voi olla myös liitettynä osaksi määrittelyvaihetta.

Toisinaan asiakasvaatimusten kerääminen saattaa kestää läpi ohjelmiston elinkaaren silmälläpitäen seuraavia versioita. Isot ohjelmistot sisältävät paljon erilaisia asiakkaiden tarpeita, jolloin ne myös tulevat muuttumaan projektin aikana. Vaikka tarpeet tulisivatkin muuttumaan, on niiden määrittämiseen panostaminen tärkeätä, koska ne luovat perustan koko ohjelmistolle. Mitä myöhäisemmässä vaiheessa virheet huomataan, sitä enemmän ne tuottavat lisätyötä ja kasvattavat kustannuksia.

2.1.2 Määrittely

Esitutkimuksen jälkeisessä määrittelyvaiheessa (engl. *requirements analysis*, *requirements specification*) asiakasvaatimukset analysoidaan ja niistä määritellään toteutettavan järjestelmän täsmälliset ohjelmistovaatimukset. Määrittelyprosessin tuloksena saadaan dokumentoitu toiminnallinen määrittely (engl. *functional specification*). Toiminnallisesta määrittelystä selviää ohjelmiston toiminnot sekä toteutuksen ei-toiminnalliset vaatimukset ja rajoitteet.

Mikäli esitutkimusvaihetta ei ole mielletty ohjelmistotuotannon elinkaaren omaksi vaiheeksi, jaetaan määrittelyvaihe usein asiakasvaatimusten kartoittamiseen ja toteutettavan järjestelmän määrittelyyn. Tällöin asiakasvaatimusten kartoittamisella tarkoitetaan esitutkimusvaihetta ja toteutettavan järjestelmän määrittelyllä määrittelyvaihetta. Osittain päällekkäisyyksistä johtuen suoritetaan määrittelyvaiheessa vielä projektin tarpeellisuuden ja toteuttamiskelpoisuuden selvittämistä, tavoitteiden ja vaatimusten asettamista sekä ratkaisumallin laatimista.

2.1.3 Suunnittelu

Suunnitteluvaiheessa (engl. *design*) asiakkaiden tarpeiden pohjalta tehty toiminnallinen määrittely muunnetaan tekniselle kielelle. Suunnittelu on mahdollista jakaa vielä kahteen eri osaan: arkkitehtuuri- ja moduulisuunnitteluun. Järjestelmä jaetaan arkkitehtuurisuunnittelussa pienempiin osiin eli moduuleihin määrittäen samalla niiden rajapinnat. Moduulien sisäisen rakenteen suunnittelu tehdään puolestaan moduulisuunnittelussa.

Arkkitehtuurisuunnittelun (engl. *architectural design*) tavoitteena on suunnitella mahdollisimman vähän toisistaan riippuvia moduuleita, jolloin yksittäisen moduulin sisällä tehtävät muutokset eivät vaikuta koko järjestelmään. Arkkitehtuurisuunnittelun tuloksena syntyy ohjelmistosta dokumentoitu tekninen määrittely. Moduulisuunnittelussa (engl. *module design*, *detailed design*) voi olla useampiakin tasoja, jotta järjestelmä voidaan pilkkoa mahdollisimman pieniin osiin. Pienien osien etuna on se, että ne voidaan antaa yksittäisten suunnittelijoiden työstettäväksi.

2.1.4 Toteutus

Yleensä toteutusvaihe (engl. *implementation*) ymmärretään ohjelmointivaiheeksi (engl. *programming*), jossa suoritetaan ohjelman kirjoittaminen aikaisemmissa vaiheissa tehtyjen suunnitelmien pohjalta. Selvän rajan vetäminen eri vaiheiden välille saattaa kuitenkin olla vaikeata varsinkin toteutusvaiheessa, sillä esimerkiksi moduulin suunnittelu ja ohjelmointi sekä testaus yhdistyvät vaihejakomallissa yhdeksi vaiheeksi. Jotkut vielä liittävät mukaan myös integroinnin ja sitä seuraavat testausvaiheet. Tämän vuoksi toteutusta voidaan käyttää myös yleisnimenä kaikille näille määrittelyvaihetta seuraaville vaiheille.

2.1.5 Testaus

Testausvaiheen (engl. *testing*) tarkoituksena on löytää ohjelmistosta mahdollisimman monta virhettä. Erilaisten ohjelmistokehitysmallien ongelmana on, että ne tunnistavat testauksen keskeisenä aktiviteettinä, mutta eivät kuitenkaan tarjoa testaajalle riittäviä työkaluja oman työn jäsentämiseksi. Tämän vuoksi testaajat jäsentävät työtään käyttäen V-mallia, jonka avulla tunnistetaan eri tasoilla tarvittavat erilaiset testaukset.

Esimerkiksi järjestelmätestaaminen suunnitellaan osana ohjelmiston määrittelyä, jossa syntynyttä määritelmädokumentaatiota verrataan valmiiseen järjestelmään. Testaaminen nähdään samoin myös integrointivaiheessa, jossa integrointitestaus suunnitellaan arkkitehtuurisuunnittelun aikana, ja moduulitestauksessa, jonka suunnittelu tapahtuu moduulisuunnittelun yhteydessä. Testauksella on kriittinen osa kehitysprosessissa, kun tavoitellaan tuotteen korkeaa käyttövarmuutta [16].

2.1.6 Käyttöönotto ja ylläpito

Käyttöönoton jälkeinen ylläpitovaihe (engl. *maintenance*) sisältää enimmäkseen asiakkaan ongelmien ratkomista, virheiden korjaamista, ohjelman muuttamista vaatimusten muuttuessa sekä uusien piirteiden lisäämistä. Tosin ohjelmistotuotteilla ylläpitovaihe ei välttämättä ole samanlainen kuin muilla tuotteilla, koska korjaukset, muutokset ja lisäykset toteutetaan usein vasta uudessa seuraavan version projektissa. Toisaalta nykyteknologian avulla yrityksillä on mahdollisuus antaa ylläpitoapua internet-verkon välityksellä. Asiakkaalla on mahdollisuus käydä lataamassa yrityksen WWW-palvelimelta uusin päivitetty versio ohjelmistostaan.

Sommerville [30] jakaa ohjelmiston ylläpidon kolmeksi tyypiksi: korjaavaksi (engl. *corrective maintenance*), mukauttavaksi (engl. *adaptive maintenance*) ja täydentäväksi ylläpidoksi (engl. *perfective maintenance*). Korjaavassa ylläpidossa keskitytään tavallisesti melko halpojen koodausvirheiden korjaamiseen. Suunnitteluvirheet ovat huomattavasti kalliimpia ja määrittelyssä tehdyt virheet ovat kaikista kalleimpia, koska ne voivat vaatia jopa koko järjestelmän uudelleensuunnittelua. Mukauttavassa ylläpidossa vaihdetaan ohjelmisto uudempaan ympäristöön, esimerkiksi erilaiselle laitteistoalustalle tai otetaan käyttöön uusi käyttöjärjestelmä. Täydentävä ylläpito sisältää puolestaan uusien funktionaalisten tai ei-funktionaalisten vaatimusten toteuttamisen.

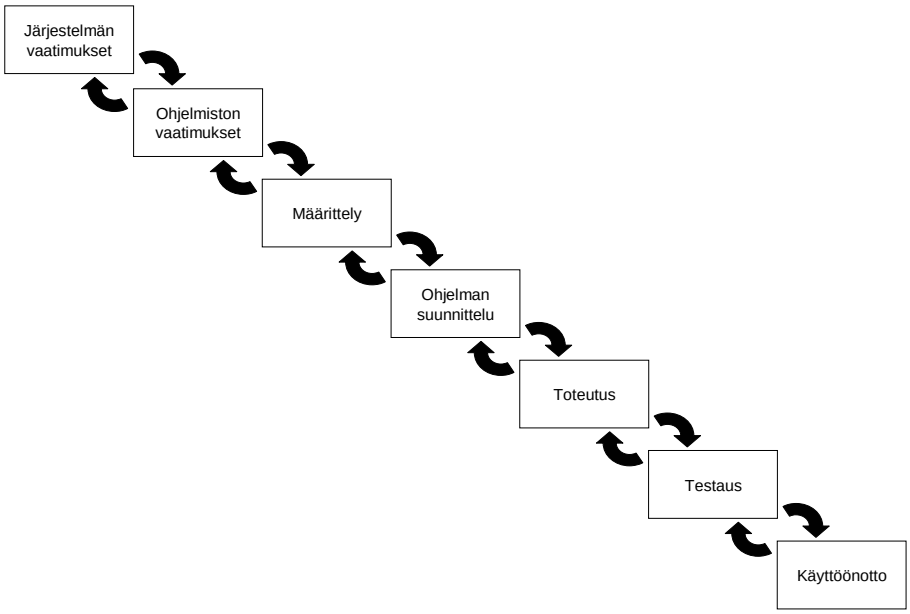
2.2 Ohjelmistotuotannon vaihemallit

Ohjelmistotuotantoa kuvaamaan voidaan käyttää useita erilaisia vaihejakomalleja. Seuraavassa käydään läpi vesiputousmalli ja inkrementaalinen malli. Näistä vesiputousmallia pidetään kaikkien vaihejakomallien isänä, ja seuraavassa alaluvussa on käytetty lähteenä Roycen [27] alkuperäistä artikkelia. Inkrementaalinen malli sopii puolestaan hyvin sulautettujen järjestelmien kehitystyöhön ja sitä käsittelevä alaluku perustuu pääosin Pressmanin [24] teokseen.

2.2.1 Vesiputousmalli

Yksi ensimmäisistä ja parhaiten tunnetuista ohjelmistotuotannon kehittämisen vaihemalleista on vesiputousmalli (engl. *waterfall model*), jonka Royce vuonna 1970 julkaisi. Roycen mukaan tietokoneohjelman kehittämisen kaksi keskeisintä askelta (engl. *step*) ovat määrittely (engl. *analysis*) ja toteutus (engl. *coding*). Tämän yksinkertaisen toteutuskonseptin avulla on mahdollista toteuttaa ainakin pieniä ohjelmia omaan käyttöön.

Mutta yksistään näillä kahdella askeleella ei kuitenkaan voida rakentaa suurempia ohjelmistoja, vaan tarvitaan useampia askeleita avuksi. Laajemmassa ja paremmassa lähestymistavassa ovat määrittely ja toteutus yhä mukana, mutta erotettuna toisistaan. Ensinnäkin määrittelyä ja toteutusta edeltävät määrittelyvaiheen (engl. *requirements analysis*) kaksi tasoa, järjestelmän ja ohjelmiston vaatimukset. Tämän jälkeen tulevan määrittelyvaiheen erottaa toteutuksesta vielä ohjelmiston suunnitteluvaihe. Toteutusvaihetta seuraa vielä testaus- ja käyttöönottovaihe.



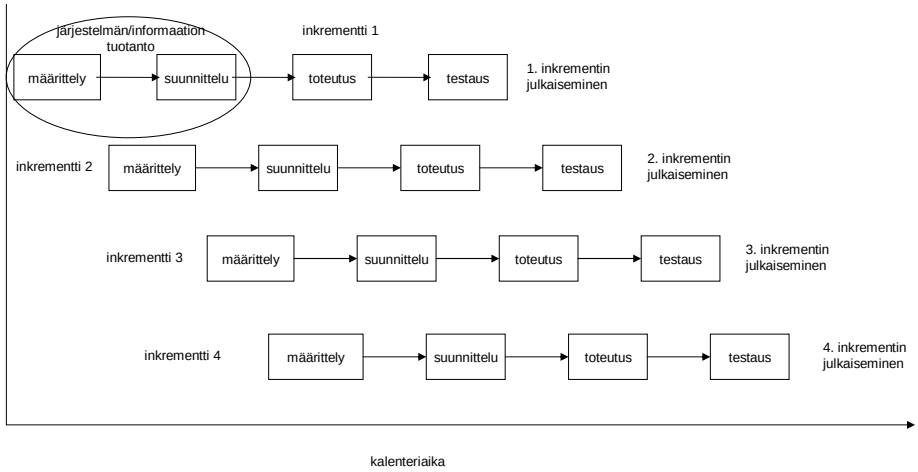
Kuva 1: Roycen alkuperäinen vesiputousmalli [27].

Vesiputousmallille on ominaista iteratiivisuus perättäisten kehitysvaiheiden välillä. Ennen seuraavaan vaiheeseen siirtymistä edellytetään edellisen vaiheen valmiiksi suorittamista erillisine katselmointeineen, joissa käydään läpi mahdollisuutta seuraavaan vaiheeseen

siirtymisestä. Royce itse näkee tämän mallin ongelmaksi sen, että testaus tulee vasta kehityselinkaaren loppuvaiheessa, jolloin virheiden löytäminen vaikuttaa pitkälle taaksepäin elinkaaressa. Löydetty virhe saattaa olla peräisin suunnittelusta, jolloin se voi vaikuttaa aina järjestelmän määrityksiin asti. Näin tehokkaana pidetty kehitysprosessi saattaa aiheuttaakin isoja aikataulumuutoksia ja kalliita taloudellisia tappioita.

2.2.2 Inkrementaalinen malli

Inkrementaalinen malli (engl. *incremental model*) yhdistää lineaarisesta sarjasta koostuvan mallin elementtejä iteratiivisella tavalla. Jokainen lineaarinen sarja tuottaa toimitettavaan ohjelmistoon inkrementin. Usein inkrementeistä ensimmäisenä julkaistu toimii tuotteen ytimenä, jonka ympärille voidaan lisätä muita inkrementtejä.



Kuva 2: Inkrementaalinen malli [24].

Ensimmäisen inkrementin aikana keskitytään perusmäärittelyihin jättäen monet lisäpiirteet huomioimatta. Tämän jälkeen tehtävät inkrementit täydentävät suunnitelman tuloksia. Jokaisen inkrementtikierroksen jälkeen suunnitelma keskittyy aikaisemmin kehitetyn tuotteen parantamiseen, jotta se vastaisi paremmin asiakkaan tarpeita lisättävien ominaispiirteiden että toiminnallisuuksien toimittamisen näkökulmasta. Prosessia jatketaan niin kauan, että jokainen inkrementti on toimitettu ja tuote on valmis.

Inkrementaalinen prosessimalli, kuten protoilu ja muut kehitysratkaisumallit, on iteratiivinen. Mutta toisin kuin protoilu, inkrementaalinen tapa tuottaa jokaisella

inkrementillä toiminnallisen tuotteen. Aikaisemmat inkrementit tuottavat niin sanottuja riisuttuja versioita lopullisesta tuotteesta tarjoten käyttäjilleen hyvän alustan kehitystyölle. Lisäksi kilpailevat tuotteet saattavat pakottaa organisaatioita merkittäviin muutoksiin kesken kehitysprojektin, jolloin inkrementaalinen kehittäminen ja julkaisu ovat keskeisessä asemassa. Merkittävät muutokset eivät ole ainoa syy, miksi inkrementaalista mallia käytetään. Toiminnallisten tuotteiden halutaan ilmestyvän aikaisemmin ja kehitysaikataulu halutaan minimoida, mikä usein vaatii inkrementaalista ohjelmiston rakentamista [8].

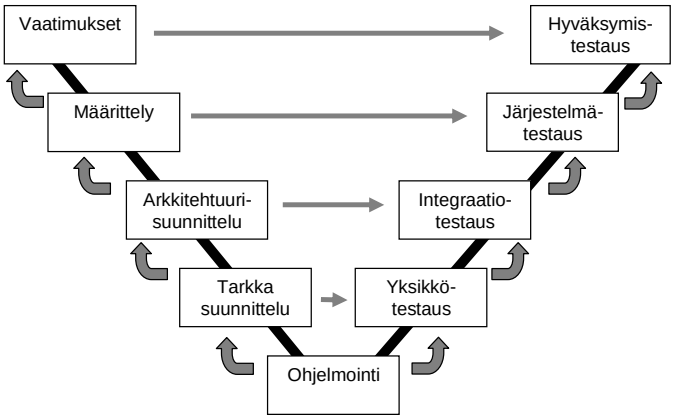
2.3 Ohjelmistotestauksen perusmalli

Koska vesiputousmallissa testaus on vasta toteutuksen jälkeen juuri ennen tuotteen julkaisua, ei se mallina ole kovinkaan hyvä testaajan näkökulmasta. Tällöin testaus tulee liian myöhäisessä vaiheessa ohjelmistoprosessia. Vesiputousmallissa testauksen aiheuttamat korjaustyöt jäävät epäselviksi ja haittana on myös testausajan lyhentäminen tai testauksen kokonaan poisjättäminen, mikäli aikataulun kanssa tulee ongelmia [31].

Muutenkaan tavalliset vaihemallit eivät anna testaajille työkaluja työnsä jäsentämisessä. Tämän vuoksi testaajien avuksi on kehitetty V-malli, jonka ensimmäisenä julkaisi Boehm [5] vuonna 1979. Vesiputousmallin laajennuksena toimiva V-malli tunnistaa eri tasoilla tarvittavat erityyppiset testaukset. V-mallia käsittelevä alaluku perustuu pääosin Spillnerin [31] artikkeliin.

2.3.1 V-malli

Monet prosessimallit, joita nykyään käytetään, voidaan yleensä yhdistää V-malliin. Malli on erittäin yksinkertainen ja helppo ymmärtää, sillä siinä toiminnot on järjestetty aikajärjestykseen ja abstraktiotasoilla on selvennetty sekä kehitys- että testaustoimintojen välistä yhteyttä. V-mallin vastakkaisella puolella oleva kehitystoiminto luo pohjan samalla tasolla olevalle testaustoiminnolle.



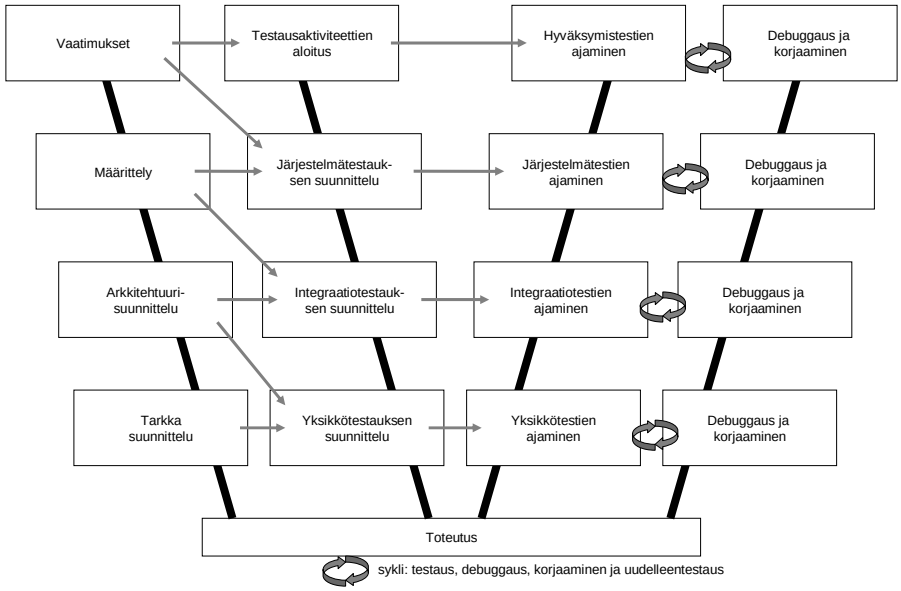
Kuva 3: V-malli [31]

V-mallissa esimerkiksi arkkitehtuurisuunnittelu antaa pohjan integraatiotestaukselle. Karkea näkymä mallista voi helposti jättää sellaisen mielikuvan, että testaustoiminnot aloitetaan vasta toteutuksen jälkeen. Näin ei kuitenkaan ole, sillä esimerkiksi testaussuunnitelma ja -strategia on tehtävä heti vaatimusten määrittämisen jälkeen, jolloin ne voivat myös auttaa ohjelmiston kehittämismallin jäsentämisessä.

Vaikka V-malli auttaakin testaajia työnsä jäsentämisessä, näkee Spillner siinä kuitenkin heikkouksia. Mallissa vasemmalla puolella on rakentavat työt, mukaan lukien toteutus, ja oikealla puolella enemmänkin ohjelmistoa tuhoavia töitä (engl. *destructive tasks*). V-mallista voi jäädä mielikuva, että toteutusvaiheen jälkeen valmis tuote voidaan julkaista. Mallin ongelmana on pitkälti se, että siitä puuttuu täysin regressiotestaus ja suunnitelmallinen virheiden poisto. Regressiotestauksen tarkoituksena on selvittää, että muunneltu ohjelma vastaa yhä määrityksiä ja että uusia virheitä ei ole päässyt ohjelmaan lisäysten ja muutosten jälkeen.

Monet prosessimallit ovat testauksen näkökulmasta riittämättömiä. Saattaa olla, että jos testaustoiminto aloitetaan vasta toteutusvaiheen jälkeen, testaustasojen ja testauksen välinen yhteys on epäselvää tai testauksen, virheiden etsimisen eli debuggauksen ja muutostöiden välinen yhteys testausvaiheessa on epäselvä. Tämän vuoksi Spillner on laajentanut V-mallia W-malliksi, jossa mukaan on otettu myös syklimäisesti toteutettava testaus, debuggaus, virheiden korjaaminen ja uudelleentestaus. On syytä huomata, että testaus ja debuggaus eli vian paikantaminen ovat kaksi eri asiaa. Debuggausprosessi

aloitetaan vasta testauksen jälkeen, kun on huomattu, ettei ohjelmisto käyttäydy määritysten mukaisesti [7].



Kuva 4: W-malli [31]

W-mallissa testauksien tärkeys ja yksittäisten toimintojen järjestys on testaukselle selvä. W-mallissa on toteutettu testausprosessi rinnakkain suunnittelun kanssa eikä ensimmäistä kertaa vasta, kun kehitystyö on saatu suoritettua. Tarkka jakaminen rakentavien ja tuhoavien töiden välillä on myös poistettu W-mallissa. W-mallissa töiden jakaminen ei ole mielekästä, sillä kehittämis- ja testaustöiden välillä täytyy olla läheistä yhteistyötä. Projektin alusta lähtien testaajien ja kehittäjien on työskenneltävä yhdessä ja nähtävä toisensa tasavertaisina työtovereina. Testausvaiheen aikana kehittäjä on vastuussa virheiden poistamisesta ja toteutuksen korjaamisesta. Ryhmien välillä vaaditaan tiukkaa yhteistyötä.

Mitä suurempi osuus testauksella on ohjelmiston resursseista, sitä enemmän W-mallia tarvitaan selventämään sitä tosiasiaa, että testaus on enemmän kuin vain testitapausten rakentamista, suorittamista ja arviointia. Vaikka toisaalta W-malli yksinkertaistaa tosiasioita selventäen käytännössä kehitysprosessin eri osien välisiä suhteita, on tarvetta kuitenkin vielä yksinkertaisemmalle mallille. Tämän vuoksi yksinkertaista V-mallia käytetään useammin kuin vielä melko uutta ja eikä kovin tunnettua W-mallia.

3 Virheiden tunnistus ja testaus

Koska testattavat ohjelmistot ovat yleensä laajoja kokonaisuuksia, on niiden täydellinen testaaminen käytännössä mahdotonta. Tämän vuoksi testaamisella ei voida todistaa ohjelmiston virheettömyyttä, vaan ainoastaan löytää ohjelmistossa olevia virheitä. Ohjelmistotuotannossa testaus määritelläänkin usein suunnitelmalliseksi virheiden etsimiseksi ohjelmaa tai sen osaa suorittamalla [14].

Toisaalta perinteinen ohjelmistoprojektin loppuvaiheessa oleva dynaaminen testaus ei kuitenkaan ole riittävä toimenpide virheiden poistamiseksi. Koska virheiden korjaamisesta aiheutuu erisuuruisia kustannuksia riippuen siitä, missä vaiheessa ohjelmistoprojektin elinkaarta ollaan, on tärkeätä keskittyä tarkemmin myös virheiden hallintaan. Virheiden hallinnassa on kyse virheiden estämisestä ja olemassa olevien virheiden tunnistamisesta.

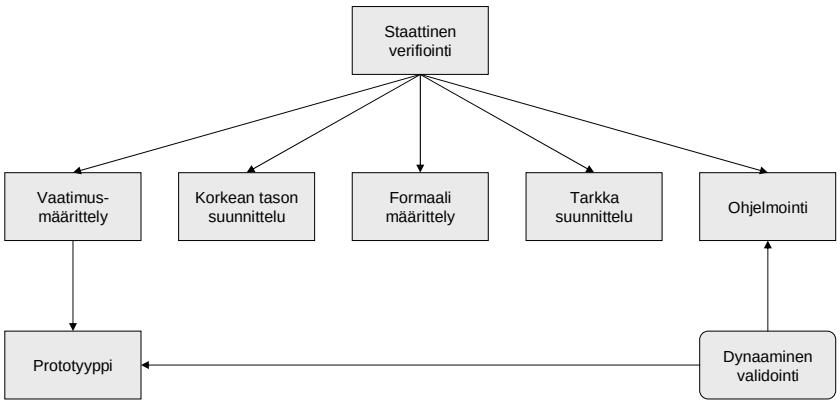
Kun virheitä hallitaan, käytetään olemassa olevien virheiden tunnistamiseen monenlaisia tekniikoita. Tärkeintä on testaajan näkökulmasta sisäistää vaihe, jolloin testausta tarvitaan. Juuri testausajankohdasta riippuen valitaan eri testaustasoilla tapahtuva testaustekniikka, jonka avulla pyritään olemassa olevia virheitä tunnistamaan. Ohjelmistoprojektin edetessä testaustekniikat vaihtuvat uusilla testaustasoilla ohjelmistokoon kasvaessa integrointien ansiosta. Erityisesti testaustekniikoiden valintaan vaikuttaa juuri ohjelmiston koko, mutta myös ohjelmiston elinkaaren vaihe, jolloin testaamista suoritetaan.

3.1 Verifiointi ja validointi

Verifioinnilla eli todennuksella (engl. *verification*) ja validoinnilla eli kelpuutuksella (engl. *validation*) tarkoitetaan yleisesti tarkastusprosessia, jossa varmistetaan, että ohjelmiston määritykset ja asiakkaan asettamat tarpeet täyttyvät [30]. Järjestelmä täytyy verifioida ja validoida joka ohjelmistoprosessin vaiheessa käyttäen aikaisemmassa vaiheesta syntynyttä dokumentoitua tuotetta hyväksi. Verifiointi ja validointi käynnistetään vaatimusten tarkastelulla jatkaen läpi suunnittelun ja koodauksen aina tarkastettavan tuotteen testaukseen saakka.

Sommerville [30] selvittää validoinnin ja verifioinnin seuraavalla tavalla. Validoinnin tarkoituksena on vastata kysymykseen: ”Olemmeko rakentamassa oikeaa tuotetta?”, kun taas verifiointi puolestaan vastaa kysymykseen: ”Rakennammeko tuotetta oikein?”. Verifioinnissa siis keskitytään tarkastamaan, että ohjelma täyttää sille annetut määritykset. Validointi puolestaan tarkastaa, että ohjelma on toteutettu vastaamaan ohjelmistoasiakkaan toiveita.

Täyttääkseen V&V -prosessin (validointi ja verifiointi -prosessi) tavoitteet järjestelmän tarkistuksessa on käytettävä sekä staattista että dynaamista tekniikkaa [30]. Staattiset tekniikat keskittyvät järjestelmän esitysmuodon analysointiin ja tarkistamiseen, jotta ne vastaavat määrittelydokumentteja ja suunnittelukaavioita. Staattisia tarkastuksia voidaan tehdä kaikilla tasoilla läpi prosessin, alkaen vaatimusmäärittelyvaiheesta. Dynaamisia tekniikoita voidaan puolestaan käyttää vasta, kun prototyyppi tai suoritettava ohjelma on käytettävissä.



Kuva 5: Staattinen verifiointi ja dynaaminen validointi [30].

3.2 Virhe

Haikala ja Märijärvi [14] määrittelevät seuraavalla tavalla virheen (engl. *error*, *bug*). Testauksessa virhe on koodissa oleva poikkeama sovelluksen toiminnallisesta tai teknisestä määrittelystä eli spesifikaatiosta. Tämän vuoksi testaaminen ilman spesifikaatiota on mahdotonta, sillä lopputulosta ei voida todeta ilman spesifikaatiota. Yleensä testauksessa käytetään spesifikaatioina toiminnallisia ja teknisiä määrittelyitä. Virhettä voi olla vaikeata

tulkita, sillä asiakkaan näkemä virhe saattaa olla toimittajan mielestä ominaisuus (engl. *feature*). Käytännössä testauksen suurin ongelma on siinä, että paraskin spesifikaatio on puutteellinen, jolloin ristiriitatilanteiden selvittely on vaikeata.

Koska ohjelmistojen koot ovat niin suuria, on niiden täydellinen testaus mahdotonta. Tämän vuoksi ohjelmistoon jää havaitsemattomia virheitä. Sovelluksessa olevan virheellisen kohdan suorittaminen ei aina aiheuta virhetoimintoa, vaan se voi aiheuttaa vain vian (engl. *fault*). Vika voi korjaantua itsestään toisen toiminnan seurauksena, tai toisen virheen toiminta voi myös korjata sen. Vika voi pahimmassa tapauksessa aiheuttaa häiriön (engl. *failure*), joka näkyy järjestelmän ulkoisessa toiminnassa. [14]

3.3 Virheen aiheuttaman muutoksen hinta

Testausprosessin tarkoituksena on virheiden löytämisen avulla todistaa tuotteen laatua ja vähentää virheiden etsimiseen tarvittavaa aikaa ja vaivannäköä. Jotta nämä edut saavutettaisiin, on testaustoiminto ja -suunnittelu laitettava alulle aikaisessa vaiheessa projektia. Lisäksi testaussuunnittelijoiden tulee olla mukana sekä analyysi- ja määrittelyvaiheen toiminnoissa että määrittelyn ja suunnittelun tarkastustoiminnoissa.

Erilaiset tarkastukset, katselmoinnit ja läpikäynnit voivat tarjota tehokkaimman testaustekniikan määrittely- tai suunnitteluvirheiden estämisessä. Testausryhmä saa sitä paremman ymmärryksen asiakkaiden tuen tarpeista, mitä aikaisemmassa vaiheessa testausryhmän mukanaolo sallitaan. Tämä auttaa myös kehittämään testausympäristön arkkitehtuuria ja testaussuunnittelua perusteellisemmaksi.

Testauksen aikainen osallistuminen ei tue ainoastaan testaussuunnittelun tehokkuutta, joka on kriittisesti tärkeä toiminto automaattisten testaustyökalujen hyödyntämisessä, vaan se tarjoaa myös aikaisen virheiden havaitsemisen estäen virheiden pääsemistä vaatimusmäärittelystä suunnitteluun ja sen jälkeen suunnittelusta koodiin [8]. Tämän tapainen virheiden estäminen vähentää kustannuksia, minimoi korjaamisia sekä säästää aikaa.

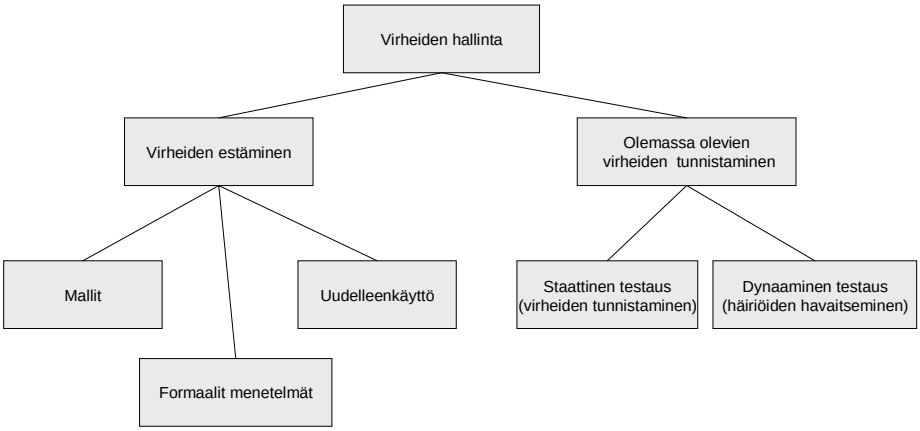
Mitä aikaisemmassa vaiheessa elinkaarta virheet pystytään estämään, sitä helpompaa ja vähemmän kustannuksia vaativaa niiden korjaaminen on, kuten taulukosta 1 käy ilmi [8]. Virheen korjaamisesta aiheutuvia kustannuksia voidaan mitata resurssivaatimuksina, joihin kuuluu myös säästetty aika. Aikaisessa vaiheessa löydetty virhe on suhteellisen helppo korjata, koska sillä ei ole toiminnallista vaikutusta eikä se näin vaadi niin paljoa resursseja. Päinvastoin on virheellä, joka löydetään toiminnallisessa vaiheessa. Tällainen virhe voi sekoittaa useita organisaatioita, vaatia laaja-alaista uudelleentestausta ja aiheuttaa toiminnallisen seisokin.

Vaihe	Kustannus
Määrittely	1
Korkean tason suunnittelu	2
Alemman tason suunnittelu	5
Koodaus	10
Yksikkötestaus	15
Integrointitestaus	22
Järjestelmätestaus	50
Julkistuksen jälkeen	100->

Taulukko 1: Ohjelmistotuotannon elinkaaren aikaisten virheiden korjaamisen hinnat [8].

3.4 Virheiden hallinta

Koska virheitä on monenlaisia, on tärkeätä olla selvillä niiden hallinnasta. Aina ei kannata laittaa kaikkia voimia yksistään olemassa olevien virheiden tunnistamiseen, sillä virheiden hallinnalla voidaan vaikuttaa myös virheiden estämiseen [38]. Virheiden estämisellä voidaan päästä käsiksi mahdollisiin tuleviin virheisiin ennen kuin ne pääsevät edes ohjelmistoon asti.



Kuva 6: Virheiden hallinnan jakaminen virheiden estämiseen ja olemassa olevien virheiden tunnistamiseen [38].

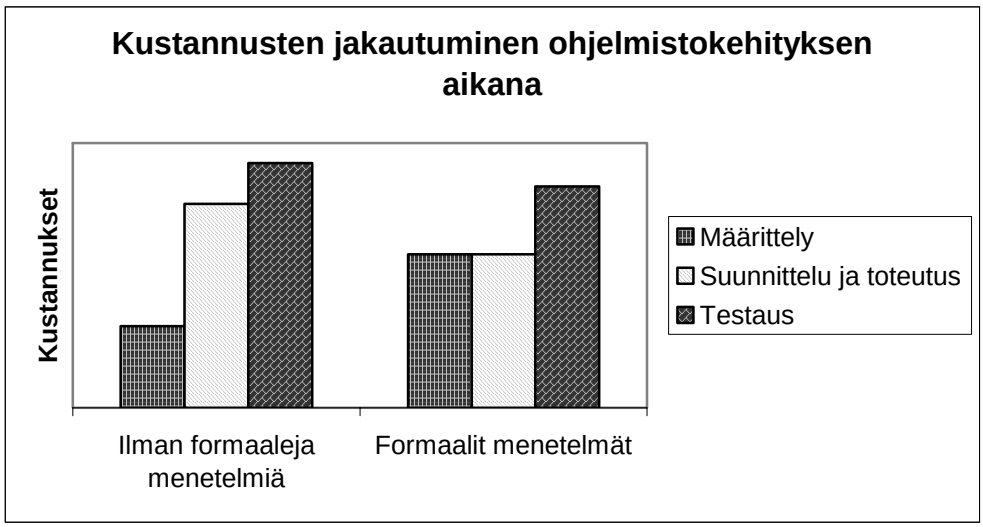
Kuten kuvasta 6 ilmenee, jakaa Younessi [38] virheiden hallinnan juuri virheiden estämiseen ja olemassa olevien virheiden tunnistamiseen. Perinteisen testaamisen avulla pyritään olemassa olevien virheiden tunnistamiseen. Mutta koska virheistä aiheutuvat kustannukset kasvavat edetessä elinkaarta eteenpäin, ei yksistään testaaminen riitä, vaan myös virheiden estämiselle on annettava painoarvoa. Kuten esimerkiksi virheiden estämisessä formaalien menetelmien avulla on mahdollista poistaa virheet järjestelmän määritelmien pohjalta ennen kuin on kirjoitettu riviäkään koodia [19].

3.4.1 Virheiden estäminen

Younessi [38] näkee virheiden tapahtumisen estämisen yhtenä tehokkaana, vaikka ei välttämättä tehokkaimpana virheiden hallitsemistapana. Perinteisesti näihin tarkkoihin ja täsmällisiin vaatimuksiin pääsemiseen on liitetty matemaattinen formaalius, johon myös formaalit menetelmät (engl. *formal methods*) perustuvat. Ohjelmistotuotannossa formaaleja menetelmiä käytetään luomaan määrittelyjä, jotka ovat täydellisempiä, yhdenmukaisempia ja yksikäsitteisempiä kuin tavanomaisin tai oliokeskeisin menetelmien tuotetut määrittelyt [24]. Formaalia määrittelyä ei ole perinteisesti pidetty osana validointia ja verifiointia tai virheiden hallintaa, kuten Younessi asian näkee. Kuitenkin formaaleja menetelmiä voidaan pitää virheiden estämisen näkökulmasta tärkeänä toimintona virheettömyyden varmistamisessa. Formaaleja menetelmiä käytetäänkin tämän vuoksi erityisesti

turvallisuuskriittisissä järjestelmissä, joissa järjestelmän tai sen osan häiriöllä saattaa olla kallis hinta.

Määrittelyvaiheessa analysoitujen asiakasvaatimusten ja niiden pohjalta määritellyn toteutettavan järjestelmän täsmällisten ohjelmistovaatimuksien toteuttaminen formaaleilla menetelmillä muuttaa kustannuksien jakautumista ohjelmiston kehittämisessä. Sommervillen [30] mukaan on vaikeata tarkastella formaalien menetelmien hyvyyttä kriteerien valinnan vaikeuden takia. Hän on kuitenkin tarkastellut kustannusten jakautumista formaalien määritysten käyttämisen ja käyttämättä jättämisen välillä.



Kuva 7: Ohjelmistokehityksen kustannukset formaalein määrityksin [30].

Formaalit menetelmät vaativat paljon resursseja, ja näin ollen kustannuksia, varsinkin määrittelyvaiheessa. Mutta tämän jälkeen kustannukset pienenevät suunnittelu- ja toteutusvaiheissa sekä testausvaiheessa, johtuen virheiden onnistuneesta karsimisesta ohjelmiston elinkaaren aikaisessa vaiheessa. Ilman formaalin menetelmän käyttöä määrittelyvaiheen kustannukset jäävät huomattavasti matalammalle tasolle, mutta suunnittelu- ja toteutusvaiheen sekä testausvaiheen kustannukset nousevat puolestaan korkeammiksi kuin formaaleja menetelmiä käyttäen.

Formaalit menetelmät eivät ole kuitenkaan ainoa virheiden estämisen keino. Muita mahdollisuuksia virheiden estämiselle ovat mallit ja uudelleenkäyttö. Malleilla on

keskeinen osa ohjelmistonkehityksessä, sillä niiden avulla pyritään yksinkertaistamaan ohjelmiston ymmärtämystä ja näin helpottamaan virheiden estämistä. Myös uudelleenkäytöllä voidaan estää virheitä, sillä uudelleenkäytettävät komponentit on jo valmiiksi testattu ennen ohjelmistoon liittämistä. Näin ollen komponentit ovat jo itsenäisesti testattuja, mutta niiden rajapinnat vaativat testausta ollessaan osana isompaa ohjelmistoa.

3.4.2 Olemassa olevien virheiden tunnistaminen

Testattaessa ohjelmaa tarvitsee tunnistaa kahdenlaisia virheitä: toiminnallisia, jotka johtuvat ohjelmiston määrittelyjen poikkeamista, ja loogisia virheitä, jotka on alustettu ohjelmistosuunnittelun toteuttamisen aikana. Kaikkia virheitä ei kuitenkaan voida estää pääsemästä suunnitteluvaiheeseen, vaan virheiden estämisen lisäksi virheiden hallinnassa tarvitaan olemassa olevien virheiden tunnistamista.

Olemassa olevien virheiden tunnistaminen voidaan jakaa kahteen eri testausmenetelmään: staattiseen ja dynaamiseen testaukseen. Testausmenetelmistä dynaaminen ymmärretään perinteisen tavan mukaan varsinaisena testauksena, jossa ajon aikana pyritään selvittämään häiriö tai vika, joka suorituksessa esiintyy. Staattisessa testaamisessa pyritään puolestaan tunnistamaan virhe, ennen kuin se suorittamisen jälkeen muuttuu viaksi tai häiriöksi. Olemassa olevien virheiden testaamista käsitellään enemmän seuraavassa testaustekniikat -kohdassa.

3.5 Testaustekniikat

Testausta voidaan jakaa testaustekniikoiden puolesta pienempiin osiin. Eräs tapa jakaa testaustekniikoita, on jakaa ne staattiseen ja dynaamiseen testaukseen. Staattista testausta suoritetaan ihmisvoimin, kun taas dynaamisessa käytetään yleensä tietokonetta avuksi. Dynaamisen testauksen avulla löydetään virheitä, kun taas staattisella testauksella voidaan todentaa myös ohjelmiston laatua.

Testaustekniikat voidaan jakaa myös musta- ja lasilaatikkotestaukseen. Mustalaatikkotestausta (engl. *black-box testing*) kutsutaan myös toimintatestaukseksi, sillä

se keskittyy ohjelmiston funktionaalisiin vaatimuksiin [24]. Mustalaatikkotestauksessa testaaja näkee ohjelman mustana laatikkona, kun taas lasilaatikkotestauksessa (engl. *white-box testing, glass-box testing*) testaaja tutkii ohjelman sisäistä rakennetta koodia läpikäymällä [18].

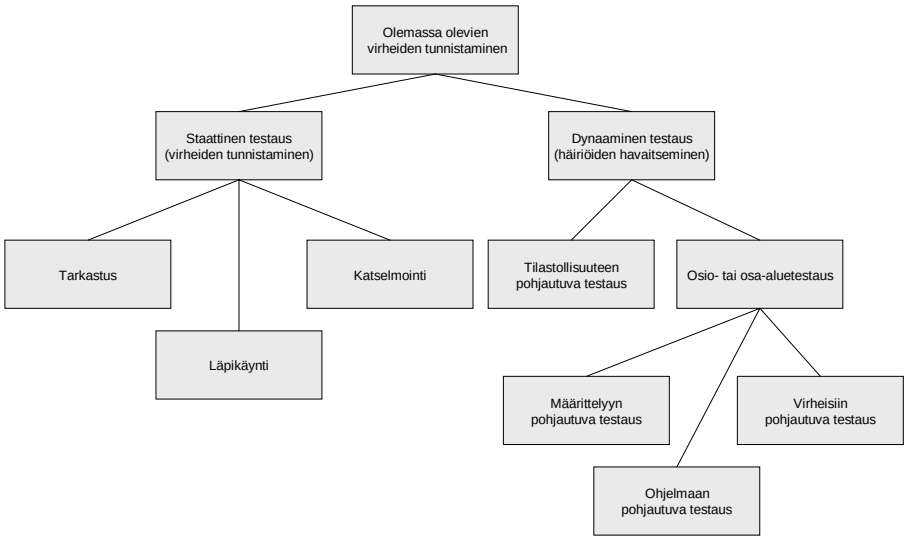
3.5.1 Staattinen ja dynaaminen testaus

Sommerville [30] määrittää seuraavalla tavalla staattisen verifioinnin, joka ei vaadi ohjelman suorittamista. Verifioinnin tarkoituksena on ohjelmiston lähdekoodin tai suunnittelun läpikäynti ja virheiden tunnistaminen ennen suorittamista, jolloin jokaista virhettä voidaan tarkastella yksinään. Virheiden vuorovaikutukset eivät ole huomattavia ja koko komponentti voidaan kelpuuttaa yksittäisissä istunnoissa. Tämän vuoksi tarvitaan vähemmän aikaa jokaisen virheen löytämiseen ja näin ollen verifiointi on taloudellisesti kannattavampaa kuin virheiden testaaminen eli dynaaminen validointi, jonka tarkoituksena on paljastaa piilevät virheet suoritettavasta järjestelmästä ennen sen julkaisemista. Ei-formaalilla staattisella verifioinnillakaan ei kuitenkaan päästä yhtä hyvään tulokseen virheiden estämisessä kuin formaalilla lähestymistavalla, jonka yksityiskohtaisella ja tarkalla mallintamisella voidaan estää lähes kaikkien virheiden esiintyminen.

Sommervillen [30] mukaan tämä ei kuitenkaan tarkoita sitä, että staattinen verifiointi voisi täydellisesti korvata dynaamisen validoinnin. Mieluummin sitä voidaan käyttää enimpien ohjelmistovirheiden löytämisessä alustavassa verifiointiprosessissa. Staattisen verifioinnin avulla voidaan tarkastaa määritysten yhdenmukaisuus, mutta sen avulla ei toisaalta voida ennustaa dynaamista käyttäytymistä. Testaus on tarpeellista luotettavuuden arvioinnissa, suorituskyvyn analysoinnissa, käyttöliittymän validoinnissa ja ohjelmistomääritysten varmistamisessa, jotta voidaan varmistaa ohjelmiston täyttävän asiakkaan asettamat vaatimukset.

Tässä kappaleessa käydään läpi Younessin [38] esittämiä asioita dynaamisesta testaamisesta eli häiriöiden havaitsemisesta, jonka jako näkyy myös kuvassa 8. Kokonaisuudessaan Younessin [38] virheiden hallinnan jäsentämisen -kuvio on nähtävissä liitteissä (Liite 5). Younessi jakaa dynaamisen testauksen tilastollisuuteen pohjautuvaan testaukseen ja osio- tai osa-aluetestaukseen. Tilastollisuuteen pohjautuvassa testauksessa

(engl. *statistically based testing*) käytetään niin sanottua virheiden kylvämistä (engl. *error seeding*), jossa testaaja laittaa ohjelmistoon virheitä. Kylvettyjä virheitä pidetään tyypillisinä virheinä, jonka vuoksi testauksen jälkeen voidaan tutkia, montako virhettä jäi löytämättä. Löytämättä jääneiden virheiden määrä on samassa suhteessa tuntemattomien virheiden kanssa. Tämän lisäksi virheiden kylvämisellä saadaan selville myös testauksen tehokkuus, sillä jo ennalta tiedetään, montako virhettä testauksen ainakin tulisi löytää. Toinen tapa, jota voidaan käyttää tilastollisuuteen pohjautuvassa testauksessa, on satunnaistestaus (engl. *random testing*), jossa testaaja valitsee testien syötteet satunnaisotannalla.



Kuva 8: Olemassa olevien virheiden tunnistamisen jakaminen staattiseen ja dynaamiseen testaamiseen [38].

Muunnostestauksen (engl. *transformational testing*) tarkoituksena on varmistaa, että annetut rutiinit voivat tuottaa vaadittuja vasteita tietyillä syötteillä [38]. Koska yritys hallita yhdistellen saatua suurta ongelmaa muunnostestauksessa on vaikeaa, on suosituksi lähestymistavaksi noussut osio- eli osa-aluetestaus (engl. *partition or subdomain testing*). Osio- tai alitasotestauksessa jaetaan syötealue edelleen loogisiin osajoukkoihin, joita kutsutaan osa-alueiksi. Tällöin testaaminen on helpommin hallittavissa pienemmissä osissa, joita voidaan tarkastella määrittelyn, ohjelman tai virheisiin pohjautuvan testauksen avulla.

3.5.2 Lasi- ja mustalaatikkotestaus

Pressman [24] ei näe lasi- ja mustalaatikkotestauksia vaihtoehtoisina tekniikoina toisilleen, vaan pikemminkin toisiaan täydentävinä, sillä mustalaatikkotestaus täydentää lasilaatikkotestausta löytäen erilaisia virheitä kuin lasilaatikkotestausmetodit. Mustalaatikkotestauksen tarkoituksena on löytää virheellisiä ja puuttuvia funktioita, rajapintavirheitä, virheitä tietorakenteista tai ulkoisista tietokannoista, käyttäytymis- tai suoritusvirheitä sekä alustus- ja päättämisvirheitä. Lasilaatikkotestauksen avulla ohjelmistotestaaja puolestaan voi johtaa testitapauksia varmistaakseen, että jokainen polku moduulissa on suoritettu vähintään kerran. Lasilaatikkotestaus keskittyy siis menettelytapojen yksityiskohtiin. Kaikkien ohjelmiston loogisten polkujen läpikäynti tarkoittaisi ohjelmiston testauksen perusteellista suorittamista. Ongelmaksi voi muodostua rajoittamaton silmukka, jolloin polkujen määrä kasvaa äärettömäksi, ja kattavaa testaamista on mahdotonta suorittaa.

Beizerin [2] oppien mukaan täydellistä toiminnallista testausta on käytännössä mahdotonta suorittaa, vaikka jokainen ohjelma toimii äärellisellä määrällä syötteitä. Toisaalta testausta vaikeuttaa myös se, että syötearvojen pituuksiakaan ei ole yleensä rajoitettu. Täydellinen toiminnallinen testaus muodostettaisiin kaikista mahdollisista syötevirroista, joiden läpikäyminen on hidasta testattavien toiminnallisuuksien kasvaessa suureksi. Jokaisella syötteellä on hyväksyttävä virta, joka tuottaa oikean vasteen, hyväksyttävä virta, joka tuottaa virheellisen vasteen tai hylätty virta, jonka tulisi kertoa miksi niin kävi.

Mustalaatikkotestauksessa testitapaukset on johdettu ohjelmiston määrityksistä, kun taas lasilaatikkotestauksessa testitapaukset on johdettu ohjelmiston rakenne ja toteutus tietäen. Näiden kahden yleisimmin mainitun virheentestausmenetelmien välille sijoittuu lisäksi rajapintatestaus (engl. *interface testing*), jota kutsutaan myös harmaalaatikkotestaukseksi (engl. *gray-box testing*) [30]. Rajapintatestauksessa testitapaukset johdetaan ohjelmiston määrityksistä tietäen sen sisäiset rajapinnat.

3.6 Testaustasot

Jotta testaamisen havainnollistaminen ja sisäistäminen olisi helpompaa, on testaamista jaettu eri testaustasoilla tehtäviksi pienemmiksi osakokonaisuuksiksi. Testaustasoja voidaan tarkastella V-mallin avulla alkaen sen oikean puolen alaosasta. Ensimmäisenä testaustasona on yksikkötestaus, jonka jälkeen tulee pienempien yksiköiden ja komponenttien integroimisen testaamiseen keskittyvä integrointitestaus. Näiden kahden ensimmäisen tason toteuttamisen jälkeen päästään järjestelmätestaukseen, jonka jälkeen onkin enää lopulliselle tuotteelle vuorossa hyväksymistestaus. Eli ohjelmiston kasvaessa suuremmaksi ja suuremmaksi siirrytään eri tasoilla suoritettavissa testauksissa korkeammalle V-mallin oikealla puolella. Myös tekniikassa tapahtuu muutos, kun lasilaatikkotestaus vaihtuu ohjelmiston kasvaessa mustalaatikkotestaukseksi.

3.6.1 Yksikkötestaus

Yksikkö on pienin testattava ohjelmiston osa, joka koostuu korkeintaan muutamasta sadasta lähdekoodirivistä. Yksikkötestaus (engl. *unit testing*) keskittyy juuri näiden pienimmille ohjelmiston komponenteille tai moduuleille suoritettavaan testaukseen. Sommerville [30] jakaa yksikkötestauksen vielä kahdeksi pienemmäksi testausosaksi: yksikkö- ja moduulitestaukseksi (engl. *module testing*).

Yksikkötestauksessa Sommervillen [30] mukaan yksittäiset komponentit testataan varmistuakseen, että ne operoivat oikein. Jokainen komponentti testataan yksin ilman järjestelmän komponentteja. Moduulitestauksessa moduuli on riippuvien komponenttien kokoelma, olioluokka, abstrakti tietotyyppi tai jokin löyhä proseduurien ja funktioiden kokoelma. Moduulissa tiivistetään yhteenkuuluvat komponentit, jolloin niitä voidaan testata ilman muita järjestelmän moduuleja. Yksikkötestauksessa on tarkoituksena näyttää, että yksikkö/moduuli ei täytä toiminnallisia määrityksiään ja/tai sen toteutettu rakenne ei ole yhteensopiva tavoitellun suunnitelmarakenteen kanssa [2].

Yksikkötestauksessa käytetään yleensä lasilaatikkotestausta, yksikön tai moduulin pienen koon takia. Pienestä koosta johtuen yksikkötestaukseen sopivat hyvin lasilaatikkotestauksen menettelytavat, joissa on keskeistä nähdä yksikön sisältö, jonka

pohjalta testitapaukset laaditaan. Pienen koon ansiosta ohjelmiston yksikköä on helppo hallita, jolloin sen kattava testaaminenkin on mahdollista suorittaa läpikäymällä ohjelmiston yksikön kaikki loogiset polut. Mutta mikäli silmukan kokoa tai syötearvojen pituuksia ei ole rajoitettu, on pienienkin moduulien kattava testaaminen mahdotonta.

3.6.2 Integrointitestaus

Integrointiprosessissa kootaan yhteen komponentit, joista sitten luodaan yhtenäinen isompi komponentti. Integrointitestauksen (engl. *integration testing*) tarkoituksena on näyttää, että aiemmin itsenäisesti hyväksytyt komponentit voivat olla yhdistettynä komponenttina virheellisiä tai yhteensopimattomia [2].

Sommerville [30] kutsuu integrointitestausta myös alijärjestelmätestaukseksi (engl. *sub-system testing*). Tämä taso sisältää testauksen keruuta niistä moduuleista, jotka on integroitu itsenäisesti suunniteltuun ja toteutettuun alijärjestelmään. Yleisimmin ongelmia ilmenee laajoissa ohjelmistojärjestelmissä juuri alijärjestelmärajapintojen yhteensopimattomuuksissa. Integrointitestausprosessi keskittyy siksi tarkastelemaan rajapintojen aiheuttamia rajapintavirheitä.

Integrointitestauksessa voidaan käyttää sekä lasi- että mustalaatikkotestausta. Lasilaatikkotestaus soveltuu paremmin pienien moduulien integroimiseen kuin mustalaatikkotestaus, joka tulee ajankohtaiseksi integroitavan järjestelmän kasvaessa suuremmaksi ja vaikeammin hallittavaksi.

3.6.3 Järjestelmätestaus

Järjestelmätestaus (engl. *system testing*) aloitetaan vasta, kun yksikkö- ja integrointitestaus on suoritettu. Järjestelmätestauksessa on tarkoituksena testata järjestelmää, jonka Beizer [2] kuvaa isoksi komponentiksi, sen käyttötarkoitusta vastaavassa ympäristössä. Testauksen avulla pyritään löytämään virheet, jotka johtuvat arvaamattomista vuorovaikutuksista alijärjestelmän ja järjestelmäkomponenttien välillä. Järjestelmätestaus keskittyy validoimaan, että järjestelmä vastaa funktionaalisia ja ei-funktionaalisia vaatimuksiaan [30].

Järjestelmätestauksessa käytetään lähes aina mustalaatikkotestausta, koska järjestelmän koko on kasvanut jo liian suureksi lasilaatikkotestausta ajatellen. Järjestelmätestauksessa testitapaukset suunnitellaan määrittelyiden pohjalta, jolloin syötteelle odotettava vaste on tarkkailun alla. Yleensä järjestelmätestaus pyritään suorittamaan jo varsinaisessa käyttöympäristössä.

3.6.4 Hyväksymistestaus

Testausprosessin viimeisenä vaiheena, ennen kuin järjestelmä on hyväksytty toiminnalliseen käyttöön, on hyväksymistestaus (engl. *acceptance testing*), jonka Sommerville [30] määrittelee seuraavasti. Järjestelmä on testattava mieluummin järjestelmän datalla kuin simuloidulla testidatalla. Sommerville muistuttaa myös, että hyväksymistestauksella pyritään paljastamaan virheet ja laiminlyönnit järjestelmän vaatimusmäärittelyissä, sillä järjestelmä käyttäytyy todellisella datalla eri tavalla kuin testidatalla. Hyväksymistestaus saattaa paljastaa myös vaatimusmäärittelyongelmia, joissa järjestelmän laitteisto ei vastaa käyttäjän tarpeita tai järjestelmän suorituskykyä on mahdotonta hyväksyä.

Joskus hyväksymistestausta kutsutaan alfatestaukseksi (engl. *alpha testing*). Alfatestauksesta on kyse silloin, kun tilatut järjestelmät on kehitetty yksittäiselle asiakkaalle. Testausprosessia jatketaan niin kauan, kunnes järjestelmän kehittäjä ja asiakas ovat samaa mieltä siitä, että luovutettava järjestelmä vastaa hyväksyttävästi järjestelmän määrityksiä. Betatestauksesta (engl. *beta testing*) on kyse puolestaan silloin, kun järjestelmää markkinoidaan ohjelmistotuotteena. Betatestausta suorittavat luovutettavan järjestelmän potentiaaliset tilaajat, jotka tekevät sopimuksen järjestelmän käyttämisestä. Järjestelmää käyttäessään heidän tulee raportoida ongelmista järjestelmän kehittäjille. Betatestauksen tarkoituksena on siis paljastaa ja havaita tuotteen todellisesta käytöstä johtuvat virheet, joita järjestelmän rakentajat eivät ole saattaneet odottaa. Palautteen perusteella järjestelmään tehdään muutoksia, jonka jälkeen se vapautetaan täydentävään betatestaukseen tai yleiseen myyntiin.

3.7 Regressiotestaus

Regressiotestauksella (engl. *regression testing*) on tärkeä osa ohjelmistokehityksen elinkaaressa. Regressiotestauksen avulla on tarkoitus selvittää, että muunneltu ohjelma lisäyksen tai muutoksen jälkeen vastaa yhä määrityksiään ja ettei uusia virheitä ole päässyt ohjelmaan. Vaikka yleisimmin regressiotestausta käytetään ohjelmiston ylläpidon aikana, eivät muunneltavat ohjelmat kuitenkaan ole ainoa regressiotestauksen kohde. Regressiotestausta voidaan käyttää myös kehitystyössä varmistamaan, etteivät uusien alijärjestelmien integroimiset ole aiheuttaneet tahattomia sivuvaikutuksia [24]. Binderin [3] mukaan regressiotestausta puolestaan käytetään yleisimmin iteratiivisessa kehityksessä, debuggauksen jälkeen, uudelleenkäytettävien komponenttien uusien versioiden luomisessa, integraation ensimmäisenä vaiheena ja tukemassa sovelluksen ylläpitoa.

Ohjelmistojen ylläpito aiheuttaa jo tällä hetkellä varsin suuren osan ohjelmiston kustannuksista ja yhä kasvava ylläpidon osuus johtuu pitkälti testauksen aiheuttamista kustannuksista. Regressiotestauksen eräänä tarkoituksena onkin siis ohjelmistokustannuksien vähentäminen. Seuraavaksi käsiteltävä regressiotestauksen osuus pohjautuu pääosin Li´n ja Wahlin [21] artikkeliin.

Regressiotestausta ei kehitetä vain taloudellisista syistä, vaan sillä on tärkeä merkitys myös ohjelmiston kehittäjille ja käyttäjille ohjelmiston virheettömyyttä ja luotettavuutta varmistettaessa. Ohjelmiston virheettömyyden varmistamisen ja luotettavuuden kasvattamisen ansiosta regressiotestaus nähdäänkin tärkeänä testaustyyppinä. Koska regressiotestauksen lausekattavuus on noin puolet koko ohjelman kaikista lauseista, ei se yksistään riitä kattamaan koko testausta [3]. Binder muistuttaa myös, että myöhemmillä regressiotestauskerroilla ei voida saavuttaa suurempaa kattavuutta kuin aiemmilla kerroilla.

Regressiotestausta tarvitaan yleisimmin, kun ohjelmistoa on muunneltu ylläpidon aikana. Mukauttava ja täydentävä ylläpito saattavat aiheuttaa suorittamisen jälkeen ohjelmistomäärityksiin muutoksia, mutta korjaavan ylläpidon jälkeen niitä ei välttämättä ole tarvetta muuttaa. Näin ollen määritysten pysyessä ennallaan voidaan testitapauksia uudelleenkäyttää samanlaisina kuin aiemmin. Kun uusi inkrementti tuotetaan järjestelmään, on kuitenkin ensin ajettava kaikki ne aikaisemman inkrementin testisarjat,

jotka ovat yhä soveltuvia, ennen kuin ajetaan vastakehitetyt testit nykyiselle inkrementille. Tämän testauksen avulla paljastetaan huonot ja puutteelliset korjaukset aikaisemmista inkrementeistä, yhteensopimattomuudet ja ei-halutut sivuvaikutukset uudessa koodissa [3]. Asteittain kehittyvää regressiotestausta käytetään puolestaan muuttuneille määrityksille, jolloin myös testitapaukset vaativat uudelleensuunnittelua.

3.7.1 Regressiotestauksen strategiat

Kun olemassa olevan testisarjan uudelleenkäyttöä regressiotestauksessa mietitään, on puntaroitava, ajetaanko kaikki vai vain osa testitapauksista. Tämän takia regressiotestaus jaetaankin pääosiltaan kahteen eri strategiaan: koko ohjelmiston uudelleentestaukseen (engl. *retest-all*) ja valikoivaan (engl. *selective*) regressiotestaukseen. Koko ohjelmiston uudelleentestauksessa uudelleenkäytetään kaikki testit olettaen, että muutokset ovat vaikuttaneet ja alustaneet virheitä joka puolelle ohjelmistoa [35]. Ongelmaksi koko ohjelmiston uudelleentestauksessa saattaa muodostua tarpeettomien aikaa ja resursseja kuluttavien testien suorittaminen. Valikoivassa strategiassa käytetään vain osaa olemassa olevista testitapauksista, sillä muutoksien oletetaan vaikuttavan vain osaan ohjelmistoa. Näin ollen pystytään vähentämään uudelleentestauksesta aiheutuvia kustannuksia. Toisaalta on myös tärkeätä varmistaa, ettei regressiotestauksen testitapausten valinta tule kalliimmaksi kuin kokonaisen alkuperäisen testisarjan suorittaminen [36].

Regressiotestausprosessissa on tärkeätä huomioida tiettyjä ominaispiirteitä. On tunnistettava ne ohjelmistokomponentit, joita täytyy uudelleentestata, ja etsiä ne testit, joiden uudelleenajaminen on oleellista. Valikoitujen uudelleentestauskriteerit täyttävien testien jälkeen ei enää uusia testejä tarvita. Nopeampi virheiden havaitseminen regressiotestauksen aikana tarjoaa aikaisemman palautteen testattavasta järjestelmästä tukien myös aikaisempia strategisia päätöksiä julkaisuaikatauluista antaen näin myös suunnittelijoille mahdollisuuden aloittaa debuggaus aikaisemmin [26].

Regressiotestauskerran suorituksen jälkeen on myös hyödyllistä päivittää ja varastoida testaustieto myöhemmin tapahtuvaa uudelleenkäyttöä ajatellen. Sneed [29] huomauttaa, että nykyään testaajat eivät tunne koodia, vaan keskittyvät enemmän

mustalaatikkotestaukseen. Tämän vuoksi hän muistuttaa, että valikoiva testaaminen on vaikeata toteuttaa, jos ei ole otettu käyttöön linkkiä määrittelyiden, koodin ja testitapausten välillä. Muuten on vaikeata löytää yhteyksiä ja toteuttaa testausta. Orso ym [22] näkevät valikoivan regressiotestaamisen tekniikassa ongelmana myös, että tehokkaat valikoivat tekniikat ovat usein epätarkkoja ja saavuttavat vain vähän säästöjä testaustyössä. Kun taas tarkat tekniikat, jotka valitsevat jokaisen alkuperäisestä poikkeavasti käyttäytyvän ja muutetun testisarjan testitapauksen, ovat liian kalliita suurille järjestelmille. Monesti myös testaus, joka toimii kehitysvaiheessa, ei toimikaan enää ylläpitovaiheessa. Regressiotestauksella ei voida näin vähentää testien kehittämisen eikä suorittamisen tarvetta uusille ja muutetuille komponenteille tai järjestelmille [3]. Valikoiva testaaminen on monesti mahdollista lasilaatikkotestauksessa, jossa koodi tunnettaan. Dynaamisen linkittämisen ongelmana Sneed näkee sen kalliit kustannukset kuin myös suuret ajalliset tarpeet.

4 Testauksen automatisointi

Liike-elämän edustajat kohtaavat nykyaikana paineita kustannuksien pienentämisestä. Näin on asia myös ohjelmistotuotannossa, jossa ohjelmistojen elinkaaria pyritään supistamaan samanaikaisesti kun resursseja minimoidaan. Tämäntapaiset toimenpiteet eivät välttämättä kuitenkaan ole ratkaisu ongelmaan, sillä yli 90 %:a kehittäjistä ylittää toimituspäivämäärän [8]. Liian aikaisin tuotteen markkinoille saamisessa on myös omat riskinsä ajatellen yrityksen omaa kamppailua elämästä ja kuolemasta.

Apuna ongelmaan voidaan käyttää testauksen automatisointia (engl. *test automation*). Testauksen automatisoinnin avulla voidaan säästää aikaa, nopeuttaa kehitystä, laajentaa ulottuvuutta ja tehdä ohjelmointia tehokkaammaksi. Automatisointi ei kuitenkaan välttämättä aina ole vain positiivista, sillä se saattaa aiheuttaa turhaa resurssien tuhlausta. Mikään itsestään selvyys ei testauksen automatisointi siis ole, vaan sen toteuttamista kannattaa puntaroida ennakolta tarkasti ennen käyttöönottoa. Usein testauksen automatisointi ilman hyvää testaussuunnittelua saa aikaan paljon toimintaa, mutta vain vähän arvoa [20]. Kaner muistuttaa myös, että testauksen suunnittelu ilman hyvää ymmärtämystä testauksen automatisoinnista saattaa antaa automatisoimisesta liian positiivisen kuvan. Yleisimmin testaus kannattaa automatisoida, kun pitää suorittaa pitkä ja monimutkainen sadan, jopa tuhannen viestin pituinen testisarja [3]. Kaikkea ei välttämättä kuitenkaan kannata automatisoida, sillä yleisimmin parhaaseen tulokseen päästään yhdistelemällä eri testausmetodeja. Osa automatisoiduista testeistä ajetaan useammin ja osa testeistä suoritetaan vain kerran [20].

Jotta testausta ylipäätänsä päästään automatisoimaan, on testaustoimintojen eri vaiheet ymmärrettävä. Kaikkeahan testaamisessa ei voi automatisoida. Näin ollen on tärkeätä kyetä kartoittamaan eri testaustoimintoja ja niiden automatisoimismahdollisuuksia. Yleensä testauksen automatisoimisessa keskitytäänkin vain parin testaustoiminnon automatisoimiseen. Yksi hyvä mahdollisuus testauksen automatisoinnille on regressiotestaus, jossa samanlaisia testitapauksia suoritetaan uudelleen. Seuraava testauksen automatisoinnin luku perustuu pääosin Fewsterin ja Grahamin [11] teokseen.

4.1 Testaustoiminnot

Viiteen erilaiseen toimintoon jaetuista testaustoiminnoista kaksi ensimmäistä, testattavien asioiden tunnistaminen (engl. *identify test conditions*) ja testitapahtumien suunnittelu, ovat älyllisesti hallittavia ja usein vain kerran suoritettavia. Kolmas toiminto, testien rakentaminen, sijoittuu testausprosessissa älyllisesti hallittavien ja rutiinitoimintojen (engl. *clerical*) välimaastoon. Rutiinitoiminnot, testien suorittaminen ja saatujen tulosten vertailu odotettuihin tuloksiin suoritetaan testausprosessin aikana yleensä useammin kuin kerran. Näistä testaustoiminnoista älyllisesti hallittavat tuovat testaukseen laatua, kun taas rutiinitoimintojen avulla onnistutaan löytämään vain virheitä.

4.1.1 Testattavien asioiden tunnistaminen

Ensimmäisen testaustoiminnon, testattavien asioiden tunnistamisen, tarkoituksena on selvittää, mitä voidaan testata ja mitä testattavia asioita pidetään tärkeimpinä. Erilaisille järjestelmille ja testausluokille on erilaisia testattavia asioita, jotka voidaan verifioida testaamalla. Tällaisia testattavia asioita ovat esimerkiksi toiminnallisuustestaukset, suorituskykytestaukset ja turvallisuustestaukset. Koska kaikkea ei voida testata taloudellisista ja ajallisista rajoitteista johtuen, onkin tärkeätä keskittyä testaamaan niitä testaamattomia, kriittisiä osia, joihin rajoitetut resurssit riittävät [4].

4.1.2 Testitapausten suunnittelu

Testitapausten suunnittelussa määritetään, kuinka mitäkin tullaan testaamaan. Testisarja koostuu useista testitapauksista. Puolestaan testitapaus on sarja testejä, jotka sisältävät käsittelyjärjestyksen ja testauksen tavoitteet, esimerkiksi testien syyn ja tarkoituksen. Testitapauksen suunnittelun tuloksena syntyy runsaasti syötearvoja ja odotettuja vasteita, joita tarvitaan testien suorittamiseen ennaltasovitussa ympäristössä. Yamauran [37] mukaan testitapausten suunnittelussa on yksi tärkeä sääntö: käsitellään kaikkia ominaisuuksia, muttei tehdä liikaa testitapauksia.

Testitapausten suunnittelu on parasta toteuttaa rinnakkain yhdessä V-mallin vasemman puolen kehitystoimintojen kanssa ennen kuin ohjelmisto on rakennettu testattavaksi.

Tällöin tehokkaalla virheiden etsimisellä on todellista taloudellista hyötyä enemmän kuin tuhoavaa vaikutusta, sillä virheet voidaan korjata ennen niiden leviämistä. Erityisesti rinnakkaintoteuttaminen auttaa odotettavien vasteiden ennustamisessa.

Testitapausten suunnittelu on kaikista kriittisin tapahtuma luotettavalle ohjelmistotestaukselle. Formaalien menetelmien avulla pystytään määrittämään suurin osa tarvittavasta tiedosta ajatellen mustalaatikkotestausta. Muutenkin formaalien menetelmien käyttö testitapausten suunnittelussa vähentäisi huomattavasti virheitten määrää, koska testitapaukset olisi johdettu yksityiskohtaisesti antamatta minkäänlaista väärintulkinnan mahdollisuutta.

4.1.3 Testien rakentaminen

Testaustoimintojen kolmannessa vaiheessa, testien rakentamisessa, valmistellaan testitapausten toteuttamisessa tarvittavia testiskriptejä, -syötteitä, -dataa ja odotettuja vasteita. Testiskriptit ovat tyypillisesti tiedostoissa säilytettävää formaalilla syntaksilla kirjoitettua dataa ja/tai ohjeita, joita suoritetaan automaattisilla työkaluilla. Testiskripti voi toteuttaa yhden tai useamman testitapauksen, navigoinnin, alustus- tai alustuksen poistamisproseduurin tai verifioinnin. Testiskripti voidaan toteuttaa myös lomakkeella, jolloin sitä voidaan käyttää myös testien manuaalisessa suorittamisessa. Testisyötteet ja odotettavat vasteet saattavat sisältyä skriptiin tai olla ulkopuolella skriptiä erotettuna erillisiin tiedostoihin tai tietokantoihin.

Jotta testejä voidaan suorittaa, täytyy testitapausten edellytykset olla toteutettu. Esimerkiksi mikäli testi käyttää tiettyä dataa tiedostosta tai tietokannasta, tulee tiedoston tai tietokannan olla alustettu sisältämään kyseinen tieto, jonka testi odottaa löytävänsä. Testitapaus saattaa vaatia erityisen laitteiston tai ohjelmiston ollakseen saatavilla, esimerkiksi verkkoyhteyksille tai printterille. Näin ollen ympäristön täytyy olla valmis ennen kuin testitapaus voidaan ajaa.

Automatisoidun työkalun avulla voidaan odotettavat vasteet järjestää tiedostoihin, kun taas manuaalisessa testauksessa odotettavat vasteet voidaan merkitä manuaaliseen testiproseduuriin tai -skriptiin. Automatisoidussa työkalussa vertailun alustustyö saattaa

olla huomattavan paljon monimutkaisempia kuin mitä manuaalisessa, sillä automatisoidut työkalut vaativat kaiken merkitsemistä yksityiskohtaisesti, paljon tarkemmin kuin mitä manuaalinen testaus vaatii. Testin rakennustoiminnot olisi niin ikään hyvä valmistella ennakkoon V-mallin vasemmalla puolella, jolloin voidaan säästää aikaa myöhemmin.

4.1.4 Testien suorittaminen

Ohjelmiston testauksessa suoritetaan ennaltalaadittuja testitapauksia ennaltalaaditun testaussuunnitelman mukaisesti eri testaustasoilla. Manuaalinen testaus vaatii erillisen testaajan, jonka tehtävä on seurata kirjoitetun manuaalisen testauksen suorittamista. Testaajan tehtävänä on myös syöttää syötteet, tarkkailla vasteita ja tehdä huomioita ongelmista mikäli niitä esiintyy. Automatisoidussa testauksessa puolestaan riittää käynnistää testaustyökalu ja kertoa sille, mitkä testitapaukset sen tulee suorittaa. Testien suorittaminen voidaan aloittaa vasta, kun ohjelma on olemassa. Näin ollen tämä toiminto on V-mallin oikealla puolella.

4.1.5 Saatujen tulosten vertailu odotettuihin tuloksiin

Jokaisen testin varsinainen vaste täytyy tutkia, jotta voidaan varmistua testattavan ohjelmiston virheettömästä toiminnasta. Testien vertailua voidaan suorittaa sekä samanaikaisesti testauksen suorittamisen kanssa että vasta testitapausten suorittamisen jälkeen. Testauksen suorittamisen aikana tehty vertailu voidaan suorittaa näytölle lähetettyjen viestien avulla. Vasteita voidaan vertailla myös sen jälkeen, kun testitapaukset on suoritettu loppuun. Tällöin esimerkiksi tietokantaan tallennetuille tulostiedoille voidaan suorittaa tulosten vertailua. Automaattisessa testauksessa saatetaan tarvita näiden kahden lähestymistavan käyttämistä.

Yksinkertaisuudessaan saatujen tulosten vertailussa odotettuihin tuloksiin on kyse siitä, että kun varsinaiset ja odotetut vasteet ovat samoja, niin ohjelmisto hyväksytään. Jos ne puolestaan ovat erilaiset, niin ohjelmiston testi hylätään ja ohjelmistoa täytyy tutkia virheen aiheuttavan kohdan löytämiseksi. Saattaa olla, että ohjelmisto on virheellinen tai voi olla, että testit ajettiin väärässä järjestyksessä, odotetut vasteet olivat virheellisiä, testausympäristö ei ollut järjestelty oikein tai testi oli virheellisesti määritelty.

Vertailun ja verifioinnin välillä on eroja, sillä työkalut voivat vain vertailla, mutta ne eivät voi verifioida. Työkalut voivat vertailla testivasteiden sarjaa toiseen ja viestittää niiden erilaisuuksia. Mutta työkalu ei voi tietää ovatko vertailun kohteena käytetyt odotettavat vasteet oikeita. Tämän vuoksi tarvitaan verifiointia, joka on normaalisti testaajan tekemää työtä. Testaaja vahvistaa tai vakuuttaa, että tulokset on vertailtu perimmiltään oikein. Joissakin tilanteissa on mahdollista automatisoida odotettujen vasteiden generointi, mutta se ei ole kovinkaan yleistä teollisuustestauksessa kaupallisia testaustyökaluja käytettäessä.

4.2 Testaustoimintojen automatisointi

Kun mietitään testauksen automatisointia, on tärkeätä puntaroida yleensäkin automatisoimisen soveltuvuutta testauksessa. Testauksen automatisointi vaatii paljon lisäkustannuksia normaalin sovelluksen rakentamisen lisäksi. Näin ollen testauksen automatisointia ei kannata pitää itsestäänselvyytenä, vaan on tarkkaan mietittävä milloin automatisointi kannattaa. Jos testejä ei tarvitse toistaa, eivät automatisoinnin lisäkustannukset ole perusteltuja [3]. Toisaalta on myös sellaisia testauksia, joita ei voida tehdä manuaalisesti, jolloin automatisointi on ainoa keino toteuttaa testaus [8].

Käytännössä jokainen testaustoiminto voidaan automatisoida, mutta toisaalta mitään testaustoimintovaihettahan ei tarvittaisi, jos ei virheitä tehtäisi. Mutta jos testaustoimintoja ajatellaan automatisoida, on automatisointi parasta aloittaa rutiinitoiminnoista eli testauksen suorittamisesta ja saatujen tulosten vertailemisesta odotettuihin tuloksiin. Nämä ovat yleensä rutiininomaisempia ja useammin toistettavia vaiheita, ja siksi paremmin automatisoitavissa, myös taloudellisesta näkökulmasta ajateltuna. Automatisoidun ja manuaalisen testauksen yhteiskäytöllä saavutetaan myös hyvä tehokkuus, sillä osa tietyistä testitilanteista sopii paremmin manuaaliseen ja osa täysin automatisoituun [3].

Harju ja Koskela [15] näkevät automatisoidun testauksen korkean kattavuuden lisäksi tärkeänä myös testitapausten ylläpidettävyyden, jolloin testitapaukset ovat helpommin muokattavissa uusiin tarpeisiin. Testaustoimintojen automatisoinnilla ei välttämättä kannata hakea vain nopeampaa kehitysaikataulua, vaan säästetty aika kannattaa panostaa tärkeisiin ja aikaa vaativiin kohtiin. Testien tarkentamisen avulla päästään suurempaan

järjestelmän luotettavuuteen. Toisaalta automatisoidun testauksen valintaan saattaa vaikuttaa myös koulutuksen tarve, hinta sekä koulutukseen tarvittava aika, jonka vuoksi automatisoituun testaukseen vaihtaminen ei välttämättä aina kuitenkaan ole mahdollista.

Hutcheson [17] muistuttaa, että mikäli ohjelmistotestauksen automatisoinnin halutaan onnistuvan, on jatkuva tuki ja sitoutuminen järjestettävä. Hänen mielestään ei sovi myöskään unohtaa sitä, että testaus on tärkeämpää kuin automatisointi. Testauksen automatisointi vaatii onnistuakseen hyvät testausmetodit ja -metriikat.

4.2.1 Testitapausten suunnittelun automatisointi

On useita testaustyökaluja, joilla voidaan automatisoida testitapausten suunnittelu. Näitä työkaluja kutsutaan joskus myös testisyötteiden generointityökaluiksi ja niiden lähestymistapa on käytännöllinen joissakin tilanteissa. Täysin testisyötteiden generointityökalut eivät kuitenkaan voi korvata älyllisiä testaustoimintoja.

Testitapaussuunnittelun lähestymistavan yksi ongelmista on se, että työkalut saattavat generoida suuren määrän testejä. Joissakin työkaluissa testaajalla on mahdollisuus minimoida testien generoimista asettamalla ennalta kriteerejä. Tästä huolimatta työkalu saattaa yhä generoida liian monta testiä ajettavaksi kohtuullisessa ajassa. Työkalulla ei ole kykyä erotella tärkeitä testejä, jonka vuoksi tarvitaan ihmisen luovaa ja älyllistä tarkastelua avuksi. Työkalun avulla on siis mahdotonta valita vähässä ajassa suoritettavat tärkeät testitapaukset, ja vaikka testaus olisikin automatisoitu, ei ohjelmistoa voitaisi silti testata kokonaan puutteellisen ajan takia.

Testitapausten suunnittelun automatisointia voidaan suorittaa esimerkiksi koodin pohjalta generoimalla testisyötteitä, rajapintojen pohjalta voidaan generoida sekä syötteitä että vasteita, samoin kuin määritysten pohjalta generoitaessa. Koodipohjaiset metodit eivät voi generoida odotettuja vasteita, kun taas rajapinnan pohjalta generoitaessa on miinuksena se, että menetelmä voi vain osittain generoida odotettavia vasteita. Koodi- ja rajapintapohjaisten metodien avulla ei voida löytää määritysvirheitä, vaan määritysten laadun varmistamiseksi tarvitaan määritystenpohjaisia metodeja.

4.2.2 Testien suorittamisen automatisointi

Toisin kuin yleisesti ajatellaan, ei työkalulla testauksen suorittamisen nauhoittaminen (engl. *capture replay*) ja sitten nauhoituksen uudelleenajaminen olekaan testauksen automatisointia. Tällä tavalla nauhoittamalla toistettua testaamisen automatisointia ei mielletä kunnon testauksen automatisoinniksi, sillä testauksen suorittamisen automatisoinnissa on normaalisti mukana myös testauksen verifioinnin automatisointi.

Testitapausten yksityiskohtainen dokumentointi tekee automatisoinnista huomattavasti tehokkaampaa kuin jos niitä ei olisi lainkaan dokumentoitu. Yksityiskohtaiset manuaaliset skriptit antavat tarkan tiedon syötteestä ja siitä, mikä vaste on odotettavissa tälle syötteelle. Epämääräisissä manuaalisissa skripteissä testisyötteet eivät ole täsmällisesti määriteltyjä, jolloin eri testaajille jää liian monta tapaa tulkita samoja skriptejä. Tämän takia myös testaajat testaavat eri tavalla.

Testauksen automatisoinnissa työkalujen ymmärtämällä kielellä kirjoitetut skriptit generoidaan syötteiksi, joita käytetään testitapauksen syötteinä testattavalle järjestelmälle. Tämän jälkeen automaattisessa testien suorittamisessa työkalu jää odottamaan tietokoneelta vastetta. Työkalu kerää lokitiedostoon myös tiedot siitä, mitä tapahtui kun testitapaus suoritettiin. Koska kyseessä on vain automatisoitu testien suorittaminen, jää saadun vasteen vertaaminen odotettuun vasteeseen manuaalisen testaajan tehtäväksi. Näin ollen pelkkä testien suorittamisen automatisointi ei vielä riitä yksistään, jotta voitaisiin puhua kokonaisen testauksen automatisoinnista.

4.2.3 Testitulosten vertailun automatisointi

Testien verifiointi on tarkistusprosessi, jossa tutkitaan ohjelmiston antaman vasteen oikeellisuutta. Tätä tarkistusprosessin vertailua todellisten vasteiden ja odotettujen tulosten välillä voidaan suorittaa yhden tai useamman kerran. Jotkin testit vaativat vain yhden vertailun, kun toiset testit saattavat vaatia useita vertailuja.

Vertailu on luultavasti automatisoiduin tehtävä ohjelmistotestauksessa. Ihmiselle on vaikea suurien lukujen, näytön tulosteiden tai kaikenlaisen datan vertaileminen, minkä vuoksi virheiden tekeminen kasvaa niissä helposti. Toiseksi rutiininomaisessa työssä on helppo

tehdä virheitä ja ne tuntuvat tylsiltä toistettavuuden ja tarkan työn takia. Tämän vuoksi ne juuri ovatkin sopivia tehtäviä tietokoneen tekemiksi ja automatisoitaviksi.

Automatisoidun suorittamisen seurauksena syntyy paljon lopputuloksia, jotka tulisi verifioida. Suurin osa lopputuloksista pitäisi verifioida jollakin tavalla, vaikka kaikkia testejä ei tarvitsekaan yksityiskohtaisesti verrata niiden lopputuloksiin. Vaikka olisimmekin automatisoineet suorittamisen, emme ilman tulosten vertailun automatisointia voi kuitenkaan puhua automatisoidusta testauksesta.

Dynaamista vertailua (engl. *dynamic comparison*) suoritetaan samanaikaisesti testitapausten suorittamisen kanssa. Testien suoritustyökalut sisältävät normaalisti vertailemismahdollisuuksia vasteille, jotka on erityisesti suunniteltu dynaamiseen vertailuun. Dynaaminen vertailu onkin ehkä suosituin, koska sitä on paremmin tuettu kaupallisilla testauksen suorittamistyökaluilla.

Dynaamisessa vertailussa ohjelmiston vertailupisteeltä kaapattua varsinaista vastetta verrataan testien suorittamisen aikana odotettuihin vasteisiin. Joka kerta, kun vertailu on suoritettu, viesti kirjoitetaan lokitiedostoon näyttämään erilaisuuksien löytymisestä tai ei. Ongelmana dynaamisessa vertailussa on, että erilaisuuden löydyttyä työkalu kuitenkin jatkaa testitapauksen suorittamista. Tämä on sekä järkevää että hyödyllistä niin kauan kuin kaikki erilaisuudet eivät tarkoita katastrofaalista toimintahäiriötä. Ongelmana on, että työkalu jatkaa testien suorittamista kuitenkin tietämättömänä mahdollisesta sekasorrosta, joita se ehkä on aiheuttanut. Monimutkaisilla järjestelmillä testitapauksen keskeyttäminen saattaisi estää ei-toivotun lopputuloksen, mikäli testaussuunnitelma ei toteudu väärin korjatun tai poistetun datan takia. Tämän vuoksi voitaisiin laajemmissa järjestelmissä säästää kallista aikaa, mikäli tarpeeton testaus pystyttäisiin keskeyttämään jo hyvissä ajoin ennen loppuun suorittamista.

Erittäin harvoin kaikkia testitapausten vasteita verrataan sen epäkäytännöllisyyden ja tarpeettomuuden takia. Tämän vuoksi usein pyritäänkin vertaamaan vain tärkeimpiä ja merkittävimpiä piirteitä. Kun vasteiden vertailua voidaan suorittaa samanaikaisesti testin kanssa, puhutaan tällöin dynaamisesta vertaamisesta. Lopputuloksen vasteet voidaan myös tallentaa tiedostoihin tai tietokantaan, josta niitä voidaan vertailla vasta, kun testitapaus on

suoritettu. Tällöin vertailua kutsutaan suorituksen jälkeiseksi vertailuksi (engl. *post-execution comparison*). Automatisoitu testaus saattaa tarvita näiden molempien vertailujen yhdistettä.

4.3 Regressiotestauksen automatisointi

Vain komponenttitestauksessa tai järjestelmän aloitustestauksessa manuaalinen testaus on tehokas. Kun manuaalinen regressiotestauskaan ei tue riittävästi toistettavuutta eikä johdonmukaisuutta, on regressiotestaus hyvä automatisoida [3]. Regressiotestauksen tarkoituksena on löytää ja dokumentoida virheet sekä seurata niiden poistamista. Testaajan tarvitsee varmistua, ettei ohjelmiston toiminnallisuuden korjaaminen ole luonut uusia virheitä muualle ohjelmistoon. Regressiotestauksen avulla pyritään myös selvittämään, ettei uusia virheitä ole alustettu korjausprosessin aikana. Tämän regressiotestauksen alueen automatisointi tuottaa suurimman kustannuksellisen hyödyn. Kanerin ym. [20] mukaan yleensä testauksen automatisoinnilla löydetään kuitenkin hämmästyttävän vähän virheitä, mutta toisaalta automatisoidun regressiotestauksen avulla löydetään tyypillisesti enemmän virheitä kehitysvaiheen aikana kuin myöhemmin suoritettavilla testeillä. Tämä regressiotestauksen automatisoinnin alaluku perustuu pääosilta Dustinin, Rashkan ja Paulin [8] teokseen.

Monesti testaajilla on väärä kuva, että kerran manuaalisesti hyväksytysti testattua ohjelmistoa ei tarvitsisi enää testata. Testaajat eivät kuitenkaan ole huomioineet sitä, että moduulimuutokset saattavat aiheuttaa virheen, joka vaikuttaa myös muihin moduuleihin. Sovelluksen ollessa melko vakaa kannattaa testiryhmän keskittyä automatisoimaan osa tai kaikki regressiotestitapauksista. Erityisesti korkean toiminnallisen riskin, toistettavia tehtäviä ja uudelleenkäytettäviä moduuleita sisältävät regressiotestit kannattaa automatisoida. Regressiotestauksia voidaan uudelleenkäyttää monta kertaa kehityselinkaaren aikana, esimerkiksi testauksessa olevan sovelluksen erilaisiin uusiin julkaisuversioihin.

Regressiotesti on testi tai sarja testejä, joita suoritetaan vertaillessa järjestelmää tai tuotetta, kun osa järjestelmästä on muutettu. Tavoitteena on verifioida, ettei ei-toivottuja muutoksia

ole tapahtunut toiminnallisissa tehtävissä, kun järjestelmää tai tuotetta on muunnettu vastaamaan määrittelyjä. Automaattiset testaustyökalut tarjoavat yksinkertaistettua regressiotestausta.

Regressiotestaus tulee suorittaa jokaisen jo aiemmin testatun sovelluksen julkaisemisen jälkeen. Savutestaus (engl. *smoke testing*) on tavallista regressiotestausta pienempi ja nopeampi tärkeämpiin korkeamman tason toiminnallisuuksiin keskittyvä regressiotestaus. Tavallinen regressiotestaus laajentaa savutestausta käsittämään kaikki olemassa olevat toiminnallisuudet, jotka on jo osoitettu toimiviksi. Regressiotestauksen sarja edustaa osajoukkoa kaikista testausprosesseista, joita suoritetaan sovelluksen perustoimivuudelle. Se saattaa sisältää testimenetelmät, joilla on korkein todennäköisyys havaita suurin osa virheistä. Tämän tapainen testaus täytyy suorittaa automatisoidulla työkalulla, koska se on tavallisesti paljon ajallisia resursseja vievää ja ikävää sekä altista ihmisen tekemille virheille.

Kaner ym. [20] muistuttavat ylläpidon tärkeyttä regressiotestauksen automatisoinnissa. Mikäli ylläpito on laiminlyöty kokonaan, ei testitapauksia voi kauan käyttää, sillä testitapaukset itse alkavat kehittää virheitä testisarjoihin. Muita vaaratekijöitä, joita automatisoitu regressiotestaus saattaa kohdata, ovat muutokset käyttöliittymässä tai testauskielessä sekä työntekijöiden vaihtuminen, mikäli dokumentointi on ollut huonoa. Suurin vaaratekijä kuitenkin on hallitsemattomat ylläpitokustannukset, jotka saattavat aiheuttaa vaivaa automatisoidulle regressiotestaukselle. Toisaalta Pinkster [23] näkee merkittävimpänä etuna automatisoidulle regressiotestaukselle sen, että ensimmäistä kertaa voidaan kehittää sekä ylläpitää laajaa regressiotestausta suhteellisen pienellä ponnistelulla.

5 Sulautetut järjestelmät tietoliikenteessä

Tietotekniikan merkittävimpiä edistysaskelia viime vuosina on ollut tietoliikenteen kehittyminen. Aikaisemmin tietoliikenteen käyttö keskittyi pitkälti vain puheviestinnän puolelle, mutta nykyaikana datapohjainen tiedonsiirto valtaa alaa tietotekniikan yleisen kehityksen myötä, mikä on nähtävissä myös sulautettujen järjestelmien puolella. Tällä hetkellä tietoliikennetekniikassa langallinen ja erityisesti langaton datasiirto ovat nopeimmin kasvavia osa-alueita, joiden käyttö on merkittävässä osassa myös sulautetuissa järjestelmissä.

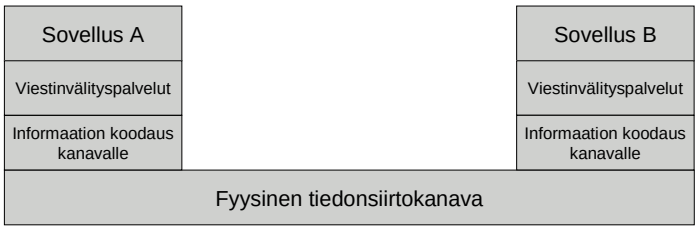
Sulautetut järjestelmät ovat järjestelmiä, joissa laitteisto ja ohjelmisto muodostavat erottamattoman kokonaisuuden. Monesti sulautetut järjestelmät muodostavat oman kokonaisuuden laajemmasta järjestelmästä. Sulautetut järjestelmät ovat useimmin reaaliaikaisia ja yleisimmin vuorovaikutuksessa ympäristön kanssa esimerkiksi seuraamalla ympäristön toimintaa, keräämällä tietoa ympäristöstä, ohjaamalla laitteita tai raportoimalla tietoja ylemmille tasoille, jolloin niiden yhtenä vaatimuksena voi olla tiedonsiirtäminen. Viimeisten tutkimusten mukaan jo arviolta 90 %:a kaikista prosessoreista on osana sulautettuja järjestelmiä [28].

Koska sulautettujen järjestelmien elinkaaret ovat erilaisia kuin tavallisilla ohjelmistoilla, on niille erikseen muokattu oma vaihemalli huomioimaan ohjelmiston ja laitteiston yhtäaikainen kehitys. Sulautetuille järjestelmille kehitetty oma yksinkertaistettu kehitysprosessimalli, usean V:n malli (engl. *multiple V-model*), helpottaa testaajan työtä. Myöhemmin käsiteltävä sulautettujen järjestelmien vaihemalli pohjautuu pääosiltaan Haikalan ja Märijärven [14] teokseen. Sulautettujen järjestelmien testauksen perusmalli ja regressiotestaus -luvut perustuvat puolestaan pääosin Broekmanin ja Notenboomin [6] teokseen.

5.1 Tietoverkkojen tietoliikenne

Tietoverkkojen pääperiaatteena on muiden tietotekniikan osa-alueiden tapaan eritasoisista palveluista koostuva järjestelmä. Lähtökohtana on, että kaksi ohjelmaa voi kommunikoida keskenään fyysisen tiedonsiirtokanavan välityksellä kuvan 9 mukaisella tavalla. Jotta

kommunikointi johtoa pitkin tai langattomasti voidaan toteuttaa, on vielä yhteisesti sovittava informaation siirtoon tapa.



Kuva 9: Perusajatus kahden sovelluksen kommunikoinnista fyysisen tiedonsiirtokanavan välityksellä.

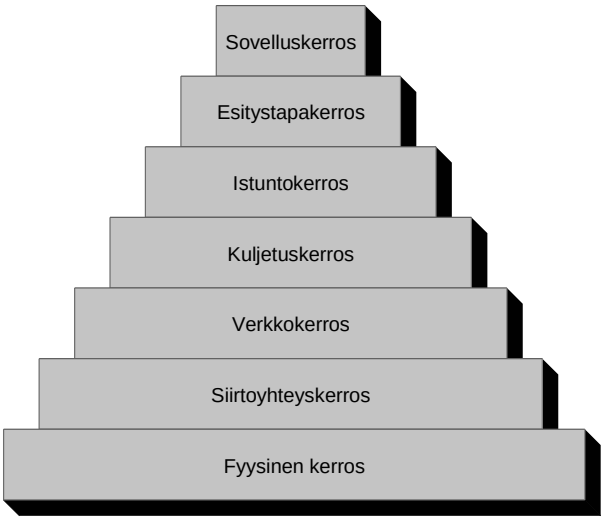
Jotta tietoliikenteessä tarvittavien yhteyksien luominen olisi helpompaa, on sitä varten kehitetty tietoliikenteen kommunikointiin oma standardi. OSI-malli on kansainvälisten standardointijärjestöjen standardi, joka on kehitetty kattamaan kaikenlaisen tiedonsiirron perusperiaatteet. Tiedonsiirrolla tarkoitetaan sellaista datansiirtoa, jossa päätelaitteina käytetään tietokonetta ja niiden oheislaitteita. Tällöin tietokoneen muistissa olevaa dataa siirretään toisen tietokoneen muistiin teksti-, ääni- tai kuvatiedosto muodossa.

5.1.1 OSI-malli

1980-luvun alkupuolella tietoliikennemarkkinoilla oli ongelmana, että monet verkkoratkaisut olivat laitevalmistajasidonnaisia yhteisten tietoliikennetoimintamallien puuttuessa. Tämän vuoksi ISO:n (International Organization for Standardization), jo 1970-luvun lopulla, asettama komitea julkaisi 1983 OSI-viitemallin (Open Systems Interconnection) yhtenäistämään tietoliikennettä. Viitemalli antoi perustan tietokoneiden toisiinsa liittämisessä hajautetuissa järjestelmissä. OSI-mallin päällimmäisenä tarkoituksena on jakaa tietoliikennejärjestelmän arkkitehtuuri kerroksiin. Kerroksien tehtävänä on tuottaa palveluja ylemmille kerroksille käyttäen hyödyksi alempien kerroksien antamia palveluja.

Täydellinen OSI-kerrosarkkitehtuuri on 7-tasoinen. Jokainen taso koostuu omasta protokollasta eli yhteyskäytännöstä, joka on tarkoitettu kahden samaan kerrokseen kuuluvan osapuolen väliseen keskustelukäytäntöön. OSI-viitemallin kerrokset ovat kuvan

10 mukaisesti alhaaltapäin lähtien: fyysinen kerros, siirtoyhteyskerros, verkkokerros, kuljetuskerros, istuntokerros, esitystapakerros ja sovelluskerros. Yleensä OSI-mallia sovelletaan ja siitä otetaan käyttöön vain tarvittavat kerrokset. Kerroksiin liittyviä tehtäviä on käyty seuraavassa läpi Granlundin [12] teoksen pohjalta.



Kuva 10: OSI-malli.

Fyysisen kerroksen tehtävänä on määritellä siirtoyhteyden mekaaniset, fyysiset ja toiminnalliset ominaisuudet, kuten bittien muunto signaaleiksi ja datan fyysinen siirto. Siirtoyhteyskerroksen tehtävänä on puolestaan kahden pisteen välisen virheettömän yhteyden huolehtiminen sekä pakettivirran järjestely ja kontrollointi. Toisen kerroksen tehtävänä on myös virheiden havaitseminen, niistä toipuminen ja tietovuon hallinta. Nykyisin linjayhteyksien luotettavuuden parannuttua siirtoyhteyskerroksen tehtävinä kuitenkin ovat vain virheellisten sanomien havaitseminen ja poistaminen siirtotieltä, sillä muut toipumiseen liittyvät tehtävät ovat tätä nykyään kuljetuskerroksen tehtäviä.

Verkkokerroksen tehtävänä on pakettien reititys ja tiedonvälitys isäntäkoneiden välillä. Verkkokerros erottaa ylemmät kerrokset riippumattomiksi verkon topologiasta. Näin verkkojen teknologioita ei ole sidottu vain johonkin tiettyyn. Tyypillisesti verkkokerroksen tehtäviin kuuluu myös reitittäminen.

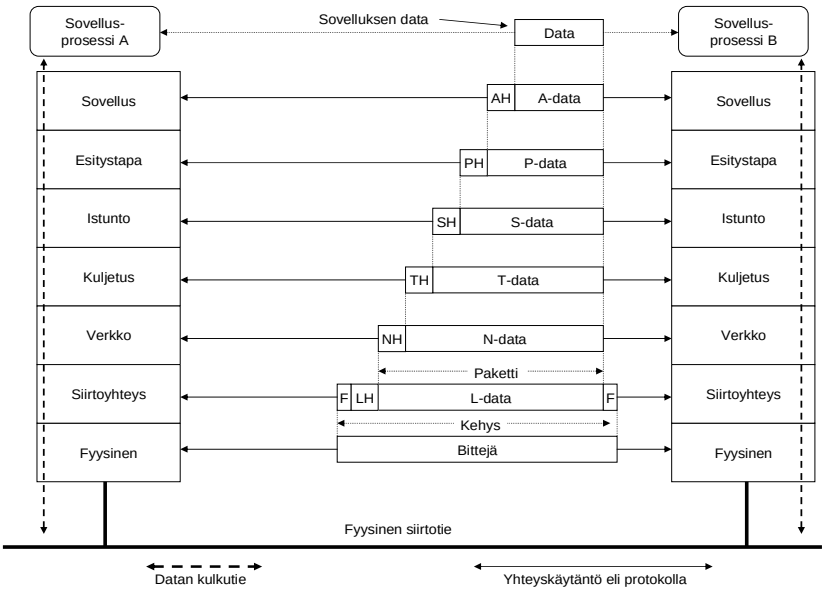
OSI-mallin neljännen kerroksen eli kuljetuskerroksen tehtävänä on kahden laitteen välillä olevan luotettavan vertaisyhteyden (engl. *peer-to-peer*) tarjoaminen. Kerroksen tehtäviin sisältyy myös vuon ohjausta, virheiden havaitsemista että virheistä toipuminen, joka on alun perin ollut siirtoyhteyskerroksen tehtävänä.

Istuntokerros puolestaan huolehtii sovellusten välisistä ohjaustoiminnoista. Istuntokerros muodostaa yhteyden ja varaa siihen liittyvän siirtoyhteyspalvelun, sopii yhteyteen liittyvistä ominaisuuksista osapuolten välillä, varmistaa yhteyden varmistuspisteillä sekä päättää yhteyden ja vapauttaa resurssit. Esitystapakerroksella sovitaan puolestaan päätelaitteiden välisen tiedon yhteisestä esitystavasta. Tehtäviin kuuluu datan esitysmuodosta, mahdollisesta pakkauksesta ja kryptauksesta sopiminen, jotta voidaan piilottaa eri laitteiden arkkitehtuureista johtuvia koodaustapoja. Sovelluskerroksen päätehtävänä on tarjota sovelluksille rajapinta OSI-järjestelmään ja sitä kautta mahdollistaa tiedonsiirto.

5.1.2 Viestien välitys tietoverkkoihin

OSI-mallin mukaisesti tapahtuvassa viestien välityksessä tietoverkkoihin menevä informaatio paketoidaan viestiksi eli paketiksi. Lähetettävästä informaatiosta riippuu käytettävän teknologian valinta. Perusperiaatteena viestien paketoimisessa on, että jokaisella kerroksella lisätään lähdetietoon omaa informaatiota kuvan 11 mukaisesti.

Yleensä lähetettävän informaation eteen lisätään tunnisteita viestin luonteesta, lähettäjästä ja vastaanottajasta. Kun viestiä kuljetetaan sovellukselta kohti fyysistä kerrosta, lisäävät erityyppiset välityspalvelut omat kehyksensä viestiin. Näin ollen erityyppisten tiedonvälityspalveluiden ei tarvitse tulkita koko viestiä, vaan ainoastaan päällimmäisen kehyksen osoite ja tunnustiedot. Muu tieto on palveluiden näkökulmasta lähetettävää viestiä eli dataa. Paketoidun viestin purkaminen tapahtuu sitten vastaanottavassa protokollapinossa, jossa tulkinta tapahtuu päinvastaisessa järjestyksessä kuin mitä paketoiminen. Paketissa olevien osoite ja tunnustietojen purkamisen avulla ohjataan paketin eteenpäinlaittamista. Erityisesti verkkokerroksen reitityksessä paketit voidaan ohjata nopeammin oikealle siirtotielle, kun koko pakattua pakettia ei tarvitse purkaa.



Kuva 11: OSI-viitekehys kuvaa protokollat kerroksittain [34].

5.1.3 Johdolliset ja langattomat siirtotiet

Jotta datasta muodostettua signaalia voitaisiin kuljettaa paikasta toiseen, tarvitaan siirtotie eli väylä. Aikaisemmin johdollisena siirtotienä on käytetty kuparijohdinta, mutta tällä hetkellä optiset kuidut ovat syrjäyttäneet kuparijohdot erityisesti televerkoissa. Toisaalta kuparin helppo liittäminen ja työstäminen ovat antaneet tekniikalle lisää elinikää.

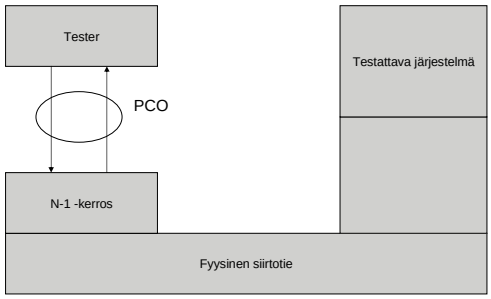
Johdollinen tiedonsiirto ei kuitenkaan ole ainoa mahdollisuus tietoliikenneratkaisuissa. Yhä enenemissä määrin johdollisen tiedonsiirron rinnalle on kehittymässä langattomia tiedonsiirtotekniikoita. Langattomassa tiedonsiirrossa tietoa siirretään sähkömagneettisten aaltojen avulla, joiden mahdollistamiseksi tarvitaan antenneja sekä lähettämisessä että vastaanottamisessa.

5.1.4 Protokollatestaus

Pystyäkseen toimimaan tietoliikenneohjelmistot vaativat yhtenäiset tavat toimia. Tämän vuoksi on kehitetty OSI-malli yhtenäistämään tiedonsiirtoa protokollien avulla. Jotta voitaisiin varmistua tuotteiden laadusta ja oikeellisuudesta, tarvitaan myös testausta.

Tietoliikenneprotokollien yhteydessä puhutaankin protokollatestauksesta, jossa vertaillaan ohjelmistojen oikeellisuutta viestien sisältöjä vertailemalla.

Protokollatestauksessa on kyse niin sanotusta mustalaatikkotestauksesta, jossa testattavaa järjestelmää testataan käyttäen hyväksi testausjärjestelmää. Mustalaatikkotestauksessa testausjärjestelmä seuraa ohjelmiston toimintoa PCO-kohdasta (engl. *Point of Control and Observation*). Testausjärjestelmä vaikuttaa testattavaan järjestelmään ulkopuolelta lähettämällä testattavalle järjestelmälle abstrakteja palveluprimitiivejä tai protokollaviestejä alempien kerroksien kautta [25]. Alempien kerroksien tehtävänä on toteuttaa yksi PCO ja sisältää koodausfunktiot sekä liitännät fyysiselle siirtotielle. Testattava järjestelmä puolestaan käsittelee viestit ja primitiivit tilanteen vaatimalla tavalla palauttaen vastauksena oman palveluprimitiivinsä tai protokollaviestinsä. Vastaanotettuaan vasteen on testausjärjestelmän tehtävänä vertailla PCO-kohdassa, onko vastaanotettu vaste odotetunlainen. Vertailun kohteena ei tarvitse olla vain osoite- ja tunnistetiedot, sillä myös viestien sisältämän datan oikeellisuutta voidaan tarkastella. Kuvassa 12 on havainnollistettu testattavan ohjelman toimintaa PCO:n kautta.



Kuva 12: Testeri seuraa ja vaikuttaa testattavaan ohjelman toimintaan PCO:n kautta [25].

Ohjelmistojen testaus voidaan jakaa myös erilaisiin pääalueisiin. Tietoliikenneohjelmistoissa pääpaino asettuu yleensä standardinmukaisuus- eli yhdenmukaisuustestaukseen (engl. *conformance testing*), jonka avulla pyritään varmistamaan protokollan tai ohjelmiston yhdenmukaisuus standardin kanssa siten, että toteutus vastaa yksikäsitteisesti standardissa annettuja vaatimuksia. Jotta tietyn järjestelmän yhdenmukaisuus voidaan testata, on järjestelmästä tiedettävä siihen toteutetut

ominaisuudet. Tätä varten on etukäteen laadittava toteutetuista ominaisuuksista lausunto, jonka pohjalta voidaan yhdenmukaisuustestausta suorittaa.

Tietoliikenneohjelmistojen yhteinen yhdenmukaisuustestaus perustuu ISO-9646 -standardisarjaan, joka määrittelee metodologian tietoliikenneprotokollien yhdenmukaisuustestaukselle [25]. Yhdenmukaisuustestauksen pohjalta tietoliikennelaitteille voidaan hakea tyyppihyväksyntää. Ilman tyyppihyväksyntää ei laitteelle voida antaa takuita toimia tietoliikennestandardien edellyttämällä tavalla. Yhdenmukaisuudesta hyötyvät erityisesti käyttäjät, sillä yhteinen standardi mahdollistaa eri laitteiden yhteenliittämisen.

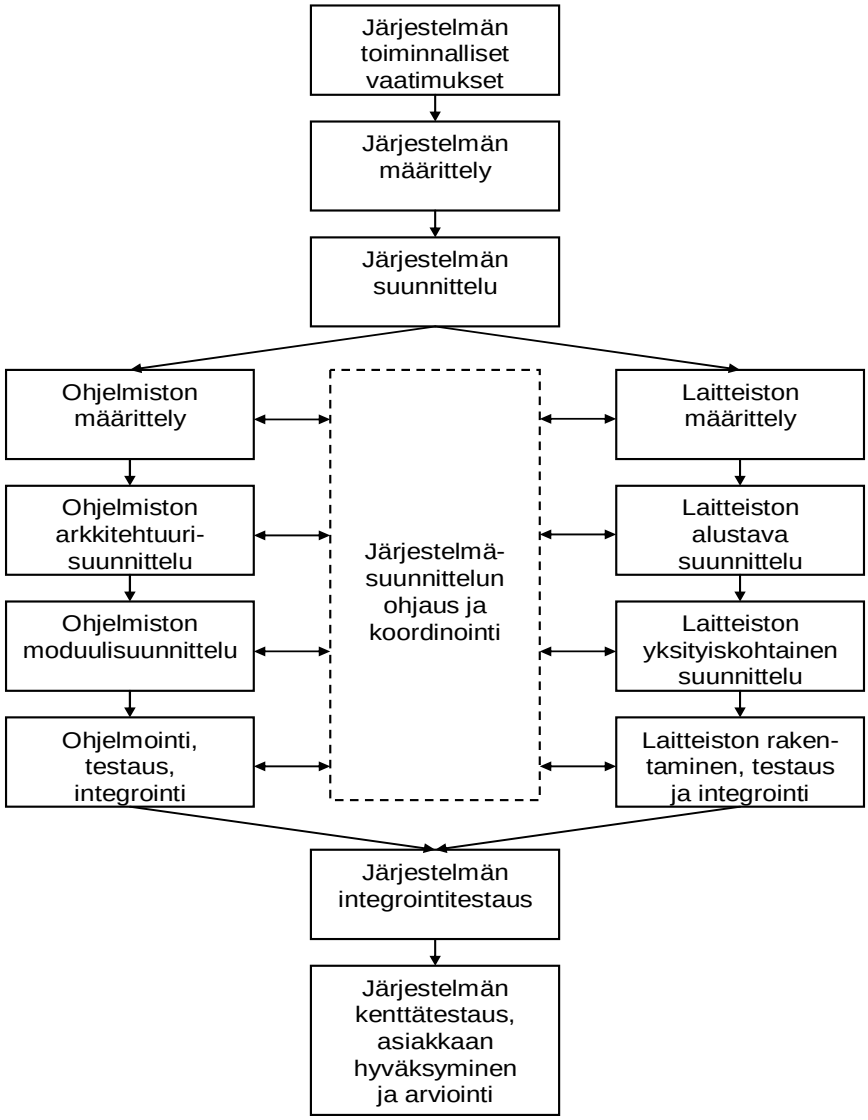
Perusperiaatteeltaan yhdenmukaisuustestaus on myös mustalaatikkotestauksen kaltainen. Yhdenmukaisuustestauksessa ei tunneta ohjelmiston sisältöä, vaan ainoastaan sen ulkoinen rajapinta ja odotettu toiminta. Valmiita testitapauksia on myös mahdollista saada protokollien testaukseen. Mikäli testitapauksia ei ole saatavilla tietyille protokollille, toteuttavat yritykset itse tällaisessa tapauksessa testitapaukset.

5.2 Sulautettujen järjestelmien vaihemalli

Melko tiiviisti tiedonsiirron kanssa tekemisissä olevat sulautetut järjestelmät poikkeavat hieman elinkaareltaan tavallisten ohjelmistojen elinkaarista. Sulautetuissa järjestelmissä eroavaisuus johtuu yhtä aikaa toteutettavista ohjelmistosta ja laitteistosta. Tavalliseen ohjelmistoprojektiin verrattuna sulautettujen järjestelmien projektissa ohjelmistoprojektin rinnalla samanaikaisesti etenee myös laitteistoprojekti. Järjestelmän kehitysprosessin eri aktiviteettien rinnakkainjakamisella voidaan nopeuttaa ja tehostaa prosessia huomattavasti [16]. Nykyaikana tuotteiden välinen kilpailu lyhentää niiden kehitysjaksoja teollisuudessa, jolloin uudet tuotesukupolvet ja -versiot vaativat yhä nopeampaa kehittämistä.

Sulautettujen järjestelmien esitutkimusvaiheen asiakasvaatimuksia seuraa järjestelmäsuunnitteluvaihe (engl. *system engineering, system design*), jossa tarkastellaan järjestelmää kokonaisuutena. Järjestelmäsuunnittelussa suunnitellaan kokonaisarkkitehtuuri ja eri osien väliset tehtävät sekä määritellään laitteiston ja ohjelmiston työnjako että rajapinnat. Järjestelmäsuunnittelun jälkeen työskentely jaetaan

ohjelmisto- ja laitteistoprojekteihin, joita suoritetaan erillään järjestelmäsuunnittelun ohjauksen ja koordinoinnin avulla.



Kuva 13: Sulautetun järjestelmän kehitysprojekti [14].

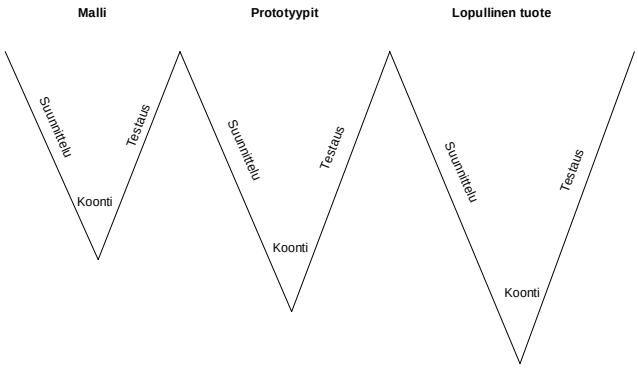
Ohjelmiston kehitystä joudutaan usein tekemään jo ennen kuin laitteistoa on edes olemassa, jolloin apuna käytetään niin sanottuja kehityslaitteistoja. Näin ollen protokollatestauskin voidaan aloittaa jo ennen kuin lopullinen laitteisto on saatu valmiiksi. Lopullinen testaaminen todellisessa ympäristössä voidaan kuitenkin aloittaa vasta, kun laitteisto on saatu täysin valmiiksi. Yleensä järjestelmän integrointi ei suju ongelmitta ohjelmisto- tai laitteistovirheiden takia. Näiden virheiden korjaukset pyritään yleensä

korjaamaan helpommin toteutettavilla ohjelmistomuutoksilla, joilla voidaan myös kiertää laitteistovikoja.

5.3 Sulautettujen järjestelmien testauksen perusmalli

Sulautetuissa järjestelmissä ei keskitytä ainoastaan suoritettavan koodin testaukseen, vaan huomioon on otettava myös laitteistolliset vaatimukset. Yleensä sulautettujen järjestelmien kehittämisessä todellisuutta on tuotteiden julkaisemisen sarja. Ensiksi rakennetaan tietokoneella järjestelmästä malli (engl. *model*), jolla simuloidaan järjestelmän käyttäytymistä. Mallista tuotetaan sen oikeaksi toteamisen jälkeen koodi, joka sulautetaan prototyyppiin (engl. *prototype*). Kokeiluluotoinen prototyyppilaitteisto muutetaan iteratiivisesti portaittain oikeaan laitteistoon, kunnes järjestelmä on rakennettu vastaamaan sen lopullista muotoa. Portaittaisessa rakentamisessa muutoksien tekeminen on halvempaa ja nopeampaa kuin samanlaisten toimenpiteiden tekeminen lopulliseen tuotteeseen (engl. *final product*).

Ohjelmistotestauksessa käytetyllä V-mallilla on myös sulautettujen järjestelmien testauksessa keskeinen osa, sillä sulautettujen järjestelmien usean V:n malli koostuu nimensä mukaisesti useasta peräkkäin suoritettavasta V-mallista. Yksinkertaistettua usean V:n mallia kuvataan yleensä yksinkertaistetusti kolmen peräkkäisen V-mallin avulla, jossa ensimmäisen vaiheen aikana luodaan malli, toisen aikana prototyyppi ja kolmannen aikana lopullinen tuote. Jokainen täydellinen V:n kehityselinkaaren vaihe sisältää suunnittelun, koonnin ja testauksen. Näin ollen jokaisen vaiheen jälkeen voidaan suorittaa täydellinen toiminnallinen testaus niin mallille, prototyypille kuin lopulliselle tuotteellekin, koska saman järjestelmän eri fyysisillä versioilla on samat toiminnalliset vaatimukset.



Kuva 14: Yksinkertaistettu usean V:n kehityselinkaari [6].

Koska usean V:n mallikin on yksinkertaistettu malli sulautetun järjestelmän kehitysprosessista, on tärkeätä ymmärtää, että kehitysvaiheessa prototyyppivaihe on usein iteratiivinen. Sulautettujen järjestelmien kehitysprosessi on usein monimutkainen johtuen ohjelmiston ja laitteiston rinnakkaisesta kehittämisestä. Tästä johtuen onkin tärkeätä, että projektiryhmien välillä on jatkuvaa kommunikointia, integrointia ja testausta. Tämä johtaa usein iteratiiviseen prosessiin, jossa järjestelmän toiminnallisuutta kasvatetaan pala palalta kohti lopullista tuotetta. Koska laajan ja monimutkaisen järjestelmän kehittäminen vaatii myös järjestelmän pilkkomista, johtaa se komponenttien rinnakkaiseen kehittämiseen ja portaittaiseen integrointiin. Eri komponenttien integrointi voi tapahtua useassa eri vaiheessa kehitysprosessia prototyyppivaiheelta aina lopullisen tuotteen täysin kehitettyyn komponenttiin.

Sulautetun järjestelmän kehittäminen on siis monimutkainen prosessi, jossa useat kehitys- ja testaustoiminnot tapahtuvat eri aikoina. Tämä vaatii testaus- ja kehittämisryhmiltä hyvää kommunikointia ja yhteistoimintaa, jonka vuoksi jokaisella projektissa mukana olevalla täytyy olla selkeä kuva kehitysprosessista. Erityisesti korkean suorituskyvyn vaatimuksia ja turvallisuusrajoitteita sisältävät sovellukset vaativat suuren määrän iteraatioita suunnitteluvaiheessa, jotta suunnitteluryhmä pystyy takaamaan hyvin testatun ja täysin debugatun lopullisen tuotteen [28].

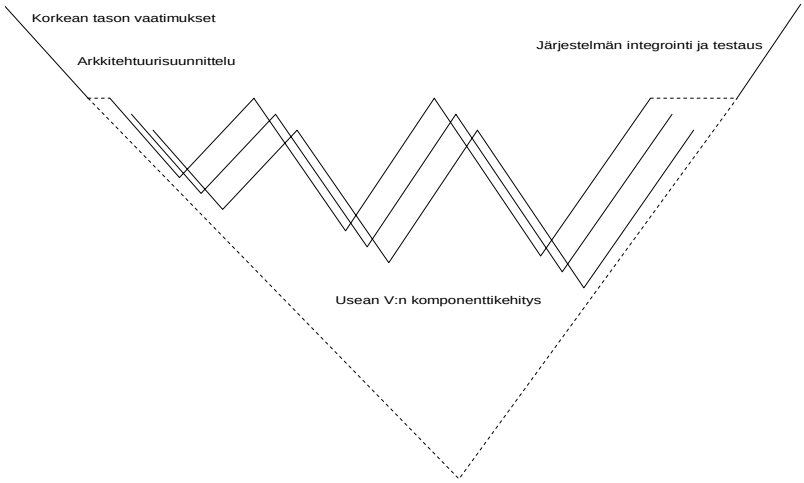
Teollisuudessa turvallisuuskriittisten sulautettujen järjestelmien testaamisessa on onnistuneesti käytetty verifiointimallia (engl. *verification pattern*, VP), joka on uusi testausta nopeuttava tekniikka. VP:n ensisijaisena tarkoituksena on luokitella järjestelmän

vaatimukset malleihin ja koodimallien (engl. *code templates*) käyttäminen puolestaan kehitettäessä testiskriptejä kaikkien samojen mallien järjestelmäskenaarioiden testaamista varten [33]. Tsai ym. [33] muistuttavat, että vaikka teollisuuden sulautettu järjestelmä tyypillisesti saattaa sisältää satojatuhansia skenaarioita, voidaan muutaman mallin avulla kattaa niistä suurin osa. Näin ollen VP:n avulla on mahdollista pienentää huomattavasti sulautettujen järjestelmien vaatimaa työmäärä että kustannuksia. Mallit tukevat myös testiskriptien johdonmukaisuutta, jolloin yksittäisten testaajien aiheuttamia vaihteluita voidaan eliminoida.

5.3.1 Sisäkkäinen usean V:n malli

Usean V:n mallin kolme peräkkäistä V-sykliä eivät ota huomioon sitä käytännön tosiasiaa, että hajotettu monimutkainen järjestelmä koostuu useammasta kuin kolmesta sisäkkäisestä V:stä. Yleisessä ja kaikille järjestelmille yhteisessä tavassa kehittää järjestelmää aloitetaan kehittäminen abstraktilla korkealla tasolla vaatimusten määrittämisellä. Tätä seuraavassa arkkitehtuurin suunnitteluvaiheessa määritetään, mitkä komponentit ovat tarvittavia toteutuksessa. Nämä kaksi vaihetta ovat yhteisiä kaikille järjestelmille, kun taas niiden jälkeen kehitettävät komponentit kehitetään erillään ja useammissa sisäkkäisissä V:ssä, joiden määrä ei ole ennalta määrätty. Lopuksi komponentit integroidaan kokonaiseksi järjestelmäksi, jolle suoritetaan testaus.

Todellisuudessa yksinkertaista V-mallia voidaan käyttää selventämään kehitysprosessia yleisellä tasolla. V-mallin vasemmalla puolella toteutetaan vaatimusmäärittelyt ja arkkitehtuurisuunnittelut. Keskimmäinen V-mallin osa sisältää rinnakkaisia kehityssyklejä kaikille komponenteille. V-mallin oikea puoli sitten käsittelee komponenttien integroinnin. Tämä voidaan käytännössä toistaa myös isojen ja monimutkaisten komponenttien kehittämisessä. Komponenttien koostuessa pienemmistä alikomponenteista ja järjestelmän koostuessa komponenteista kutsutaankin kyseistä kehityselinkaaren mallia sisäkkäiseksi (engl. *nested*).



Kuva 15: Sisäkkäinen usean V:n malli [6].

Käytännössä puhtaasti usean peräkkäisen V:n malli soveltuu pääasiassa parhaiten komponenttitasolle. Tuskin koskaan täydellistä järjestelmää mallinnetaan kokonaan ensin ja sitten tehdään prototyyppi kokonaan ja niin edelleen, vaan komponentit rakennetaan portaittain edeten. Tämän vuoksi kehitystoimintoja ja -kysymyksiä voi olla vaikea sijoittaa kolmeen V:hen usean V:n mallissa. Tällaisia ovat varsinkin kaikkialla kehitysprosessissa esiintyvät korkean ja matalan tason vaatimukset, varmuussuunnittelu, spesifien työkalujen suunnittelu ja rakentaminen.

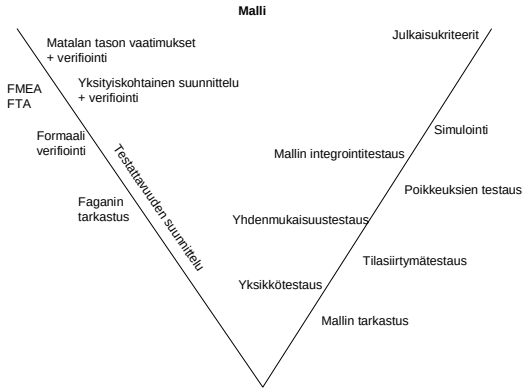
Kun järjestelmätason V-malli on yhdistetty komponenttitason usean V:n malliin, syntyy tuloksena niin sanottu sisäkkäinen usean V:n malli. Tässä mallissa voidaan sijoittaa kaikki testaukseen liittyvät toiminnot oikeille paikoille ja oikeille tasoille.

5.4 Regressiotestaus sulautettujen järjestelmien testauksessa

Koska sulautettujen järjestelmien testauksessa on vaikeata päästä käsiksi järjestelmän dataan, tarvitaan testaamista varmistamaan järjestelmän käyttäytyminen. Useasti tämän vuoksi järjestelmää tarvitsee testata myös pienempien lisäyksien jälkeen, jotta voidaan olla varmoja järjestelmän toimivuudesta. Näin ollen useasti toistettavassa sulautettujen järjestelmien testauksessa tarvitaan regressiotestausta. Usean V:n mallissa on käsitelty eri vaiheissa tapahtuvia testaustoimintoja.

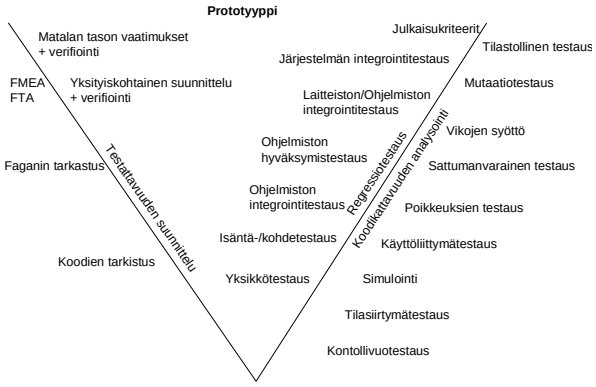
5.4.1 Testaustoiminnot usean V:n mallissa

Koska testausprosessi sisältää suuren määrän testaustoimintoja, vaatii monimutkainen järjestelmänhallitseminen apuvälineitä. Useiden erilaisten testauksen suunnitteluteknikoiden käyttö vaatii oman huomionsa ohjelmistoprosessin aikana. Jotta näiden testaustoimintojen jäsentämistä voitaisiin auttaa, on suunniteltu usean V:n malli. Yksistään monimutkainen usean V:n malli ei auta testauksen organisoinnissa, vaan myös testausta johtavalla testaussuunnittelijalla on oltava hyvä yleiskuva kaikista relevanteista testaustoiminnoista ja -kysymyksistä sekä niiden suhteista pystyäkseen hallitsemaan ja suunnittelemaan ne hyvin.



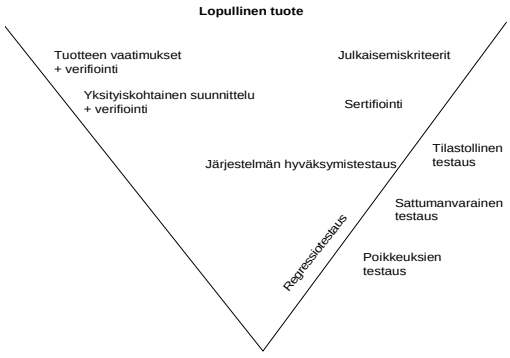
Kuva 16: Testaukseen liittyvien asioiden sijoittaminen mallin kehityssykliin [6].

Mallin kehityssyklissä keskitytään melko tasapuolisesti sekä mallin verifiointiin että validointiin. Alkuvaiheessa testattavuuden suunnittelun yhteydessä voidaan keskittyä mallin vika- ja vaikutusanalysointiin (engl. *failure mode and effect analysis*, FMEA) sekä vikapuuanalyysiin (engl. *fault tree analysis*, FTA) yhdessä formaalin verifioinnin ja Faganin tarkastusten kanssa. Mallin kehityssyklin aikana voidaan jo ensimmäistä kertaa suorittaa yhdenmukaisuustestausta tekeillä olevalle tuotteelle. Prototyyppivaiheessa testaaminen puolestaan keskittyy jo enemmän prototyypin validointiin unohtamatta kuitenkaan verifiointia. Lopullisen tuotteen testauksessa on enää lähinnä kyse tuotteen dynaamisesta validoinnista, jolloin järjestelmän testaus suoritetaan jo pääasiassa tuotteen omassa ympäristössä.



Kuva 17: Testaukseen liittyvien asioiden sijoittaminen prototyypin kehityssykliin [6].

Regressiotestaus tulee mukaan vasta prototyypin ja lopullisen tuotteen vaiheessa. Prototyyppivaiheessa regressiotestausta voidaan hyödyntää ohjelmiston integrointitestauksessa, ohjelmiston hyväksymistestauksessa sekä laitteiston ja ohjelmiston integrointitestauksessa. Lopullisen tuotteen testauksessa regressiotestausta voidaan käyttää ennen järjestelmän hyväksymistestausta. Unohtaa ei kuitenkaan sovi, että regressiotestauksella on myös keskeinen osa ylläpitovaiheessakin. Tällöin regressiotestauksen avulla järjestelmä voidaan validoida siihen tehtyjen muutosten jälkeen.



Kuva 18: Testaukseen liittyvien asioiden sijoittaminen lopullisen tuotteen kehityssykliin [6].

6 Case-tapauksen teknologiat

Tässä luvussa käsitellään case-tapaukseen tarvittavia teknologioita. Ensimmäisessä alaluvussa on käyty läpi LOPO-projektissa valmistettua LR-WPAN -laitetta (engl. *Low-rate wireless personal area network*) ja laitteista muodostettua verkkoa. LR-WPAN -standardin alaluku pohjautuu IEEE:n (The Institute of Electrical and Electronics Engineers) 802.15.4 -standardia käsittelevään Gutiérrezin, Callawayn ja Barrettin [13] teokseen. Kolmannessa alaluvussa käsitelty TTCN-3 (The Testing and Test Control Notation version 3) on pääosiltaan ETSI ES 201 873 -sarjan (The European Telecommunications Standard Institute) standardiosiin 1 [9] ja 5 [10] pohjautuva alaluku.

6.1 LOPO-projekti

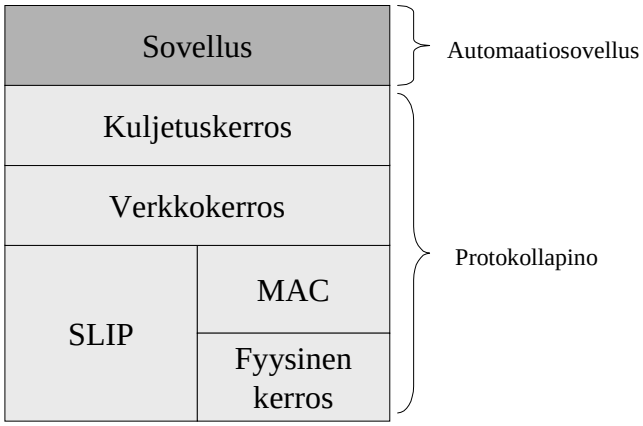
Vuonna 2002 alkaneessa Chydenius-Instituutin Tietoliikennelaboratorion LOPO-projektissa (LOw POwer) on keskitytty lyhyen kantaman langattoman tiedonsiirron tutkimiseen. Projektissa on ollut tarkoituksena suunnitella ja toteuttaa hyvin vähällä virralla toimiva laite, joka olisi kuitenkin riittävän älykäs yksinkertaisen reitittävän verkon toteuttamiseen. LOPO-projektin konkreettisena tavoitteena oli rakentaa kotiautomaatiosovellus, jossa muutamilla laitteilla voitaisiin käsitellä mittaus- ja ohjaustietoja. Projektin tarkoituksena oli myös toteuttaa langaton verkko, jota voitaisiin käyttää alustana verkon jatkotutkimukselle.

6.1.1 LR-WPAN -laite

LOPO-projektissa modulaariseksi suunniteltu LR-WPAN -laite koostuu radio-osasta, I/O- ja CPU-kortista. Testausta varten on laitteeseen lisätty mahdollisuudet lisäominaisuuksille, kuten näppäimistölle ja näytölle. Kokonaisuudessaan ohjelmistokin on suunniteltu modulaariseksi, jotta uusien ominaisuuksien lisääminen olisi helpompaa. Ohjelmisto on toteutettu protokollapinon ja sovellusohjelman avulla kuvan 19 mukaisesti.

Kuva 19: LOPO-projektin ohjelmistoarkkitehtuuri [32].

LR-WPAN -standardi tarjosi valmiina langattoman tiedonsiirron fyysisen kerroksen ja osan siirtoyhteyskerroksesta (MAC), jotka on toteutettu LR-WPAN -piirillä. Muut verkon kerroksiin ja sovellukseen tarvittavat ominaisuudet toteutettiin LOPO-projektin toimesta kontrolleriohjelmistoon. 802.15.4 -standardi pohjautuu OSI-mallin protokollapinoon, josta on toteutettu kaksi alimmaista kerrosta. Näin ollen OSI-malli toimi jatkossakin hyvänä lähtökohtana, koska siitä pystyi valitsemaan vain välttämättömät kerrokset toteutukseen. LOPO-projektissa toteutetun ohjelmiston rakenne ilmenee kuvasta 20. Ohjelmistoon on toteutettu lisäksi SLIP-protokolla laitteen mahdollista tietokoneeseen liittämistä varten.

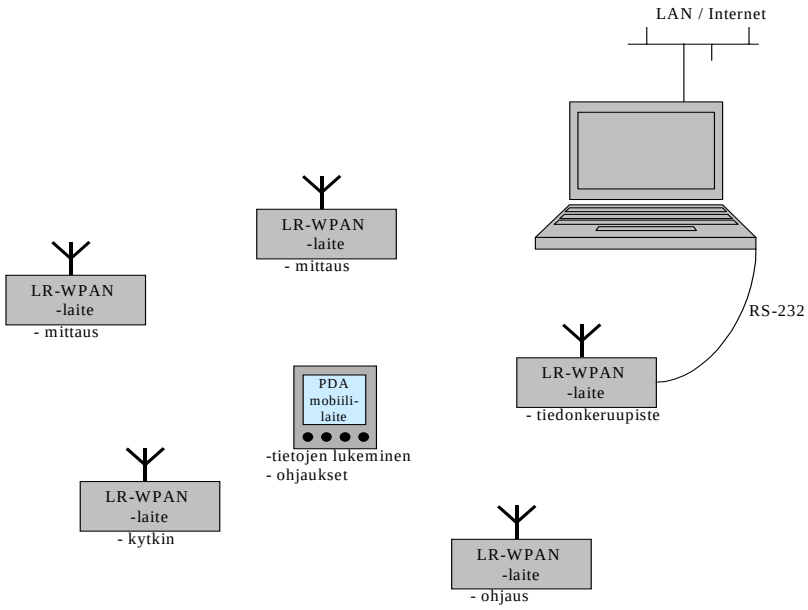


Kuva 20: Ohjelmiston rakenne langattomassa laitteessa [32].

6.1.2 LR-WPAN -sensoriverkko

LR-WPAN -verkko voidaan rakentaa kahdentyyppisistä eri käyttötarkoitukseen sopivista laitteista. Täyden toiminnon laitteet (engl. *full function* device, FFD) voivat toimia verkkoa

kontrolloivina laitteina, kun taas alemman toiminnon laitteet (engl. r*educed function device*, RFD) ovat yksinkertaisempia tukiaseman palveluita tiedonsiirtämisessä tarvitsevia laitteita. Kuvasta 21 ilmenee eräs tapa muodostaa LR-WPAN -verkko.



Kuva 21: LOPO-projektin verkon rakenne [32].

Ainakin yksi verkon laitteista on yhteydessä tietokoneeseen RS-232 -sarjaliikenneprotokollan välityksellä toimien näin linkkinä verkon ja muiden tietojärjestelmien välillä. Tiedonkeruulaite toimii siten apuna verkon mittaus- ja ohjaustietojen siirtämisessä sekä konfigurointiasioissa. Verkon laitteilla on muutenkin erilaisia tehtäviä. Osa laitteista voi toimia ulkoisen ympäristön mittauksessa, toinen puolestaan voi käsitellä mitattua tietoa ja kolmas hoitaa kommunikoinnin viereisten laitteiden kanssa.

6.2 LR-WPAN

Sulautettujen järjestelmien mahdollistaman kontrolloinnin ja valvonnan voimakas kasvu lähes kaikissa elektronisissa laitteissa on lisännyt ongelmia sovellusten liitettävyyksissä. Valmistajat käyttävät erilaisia sekä standardoituja että patentoituja kommunikointirajapintoja erilaisten laitteiden välillä. Yleensä kommunikointi on hoidettu kaapelin välityksellä, mutta aina tämä ei ole mahdollista kustannuksista tai muista syistä

johtuen. Kun oheislaite ei ole fyysisesti yhdistettynä toiseen laitteeseen, voidaan turvautua langattomaan yhteyteen.

Yleensä erilaiset sovellukset voivat tarjota lisäkustannuksin tehokkaampia langattomia kommunikointijärjestelmiä, kuten esimerkiksi matkapuhelimen, WLAN:n (Wireless Local Area Network) tai muiden vastaavien tapojen avulla. Toisaalta sovellukset voisivat lisätä suosiotaan ja käytettävyyttä, jos halvat langattomat kommunikointiratkaisut olisivat mahdollisia. Yhteisen standardin avulla eri valmistajien laitteiden yhteensopivuus antaisi loppukäyttäjille suoraa etua.

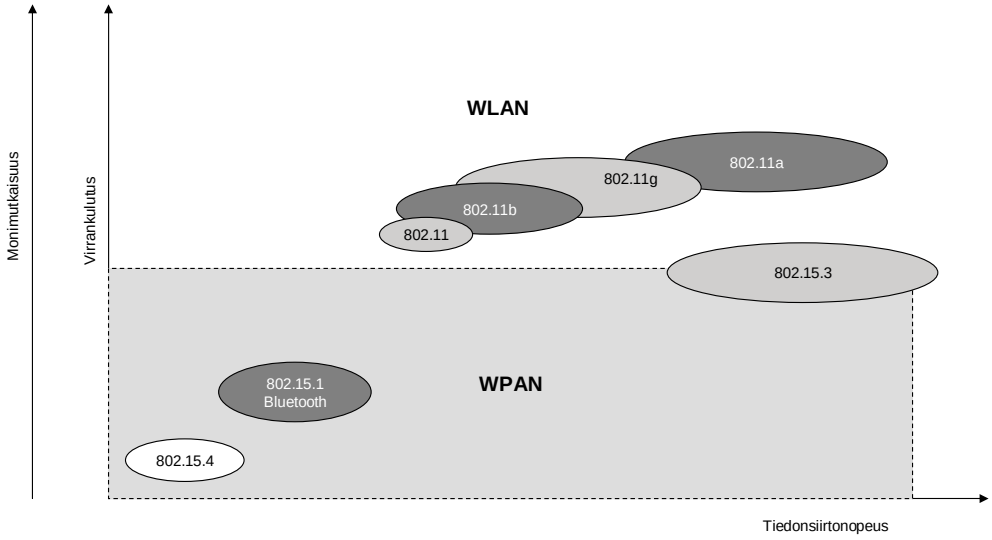
LR-WPAN on lyhyen kantaman hidas langaton verkko, joka on suunniteltu halvaksi ja vähän virtaa kuluttavaksi. Määritelmä saattaa kummastuttaa ajatellessa nykyajan langatonta teknologiaa, sillä yleensä kommunikoinnissa pyritään nopeaan tiedon siirtämiseen ja laadukkaaseen palveluun.

IEEE:n 802 -standardisarjan eräs langaton laajennus on Wireless Local Area Networks (WLAN) eli langaton lähiverkko, joka on suunniteltu tehokkaaseen tiedonsiirtoon verkossa. WLAN-järjestelmän vaatimuksina ovat saumaton verkossa eteneminen (engl. *roaming*), sijainninseuranta, viestin edelleenvälittäminen, mahdollisimman laaja vaikutusalue ja kapasiteetti suurelle laitteistojen määrälle. WPAN puolestaan on suunniteltu ulottumaan kaikkiin suuntiin 10 metriin asti ja peittämään liikkeessä tai paikalla olevaa laitetta ympäröivän alueen.

WPAN:a käytetään siirtämään tietoa suhteellisen lyhyen lähetin-vastaanotin -välimatkan välillä. WPAN-yhteys sisältää vain vähän tai ei ollenkaan infrastruktuuria, toisin kuin WLAN-ympäristö. Vähäinen infrastruktuuri sallii pienen virrankulutuksen sekä halvan ratkaisun toteuttaa kyseisillä laitteilla verkko.

IEEE 802.15 -työryhmä on määrittänyt WPAN:n kolmeen eri luokkaan, tiedon siirtonopeuden, pariston virrankulutuksen ja palvelun laadun (QoS, engl. *quality of service*) mukaan. IEEE:n 802.15.3 -standardi on High Rate Personal Area Network (HR-WPAN) eli lyhyen kantaman nopea langaton verkko, joka on tarkoitettu hyvää palvelun laatua vaativille multimediasovelluksille. Medium Rate WPAN 802.15.1, kuten esimerkiksi

Bluetooth, on suunniteltu korvaamaan kaapeli kuluttajien elektronisissa laitteissa keskittyen kännyköihin ja kämmentietokoneisiin. MR-WPAN mahdollistaa myös äänisovelluksiin sopivan hyvän palvelun laadun. Viimeinen luokista on LR-WPAN (IEEE 802.15.4), joka keskittyy toisenlaisiin tavoitteisiin kuin ylläolevat. LR-WPAN:n päällimmäisenä tavoitteena ei ole ollut nopea tiedonsiirto eikä hyvä palvelun laatu. Kuvasta 22 käy ilmi 802-standardin WPAN- ja WLAN-operointialueet.



Kuva 22: Erilaisten WLAN- ja WPAN-standardien operointialueet [13].

IEEE:n 802 -kommunikointistandardi määrittää vain kaksi alimman tason kerrosta OSI-protokollamallista. Standardissa on määritelty fyysinen (engl. *physical*, PHY) sekä siirtoyhteyskerroksen alikerros MAC (engl. *Data Link*). Muita kerroksia ei ole standardissa määritelty, vaan ne ovat yritysten tai vastaavien oman kiinnostuksen mukaan määriteltävissä.

	802.11b WLAN	Bluetooth WPAN	Low Rate WPAN
Kantama	~100 m	~10-100 m	10 m
Tiedonsiirtonopeus	~2-11 Mb/s	1 Mb/s	< 0.25 Mb/s
Virrankulutus	Keskikokoinen	Alhainen	Erittäin alhainen
Koko	Suurin	Pienempi	Pienin
Kustannus/Monimutkaisuus	Korkea	Keskikokoinen	Erittäin matala

Taulukko 2: LR-WPAN:n vertailu muihin langattomiin teknologioihin. [13]

Kuten taulukossa 2 on esitetty, keskittyy LR-WPAN -verkko lyhyeen kantamaan, matalaan tiedonsiirtonopeuteen, pieneen virrankulutukseen ja kokoon sekä erittäin mataliin kustannuksiin ja matalaan monimutkaisuuteen. Näiden ominaisuuksien vuoksi LR-WPAN -tekniikka eroaakin varsin paljon muista langattomista tekniikoista. Vaikka LR-WPAN:n kantamaksi on standardissa annettu 10 metriä, on LOPO-projektissa tehdyissä mittauksissa laitteella päästy ulkoilmamittauksissa 100 metrin kantamaan.

6.2.1 Langaton sensoriverkko

Langaton sensoriverkko (engl. *wireless sensor network*, WSN) on langattomien verkkojen sovelluksiin keskittyvä alijoukko, joka mahdollistaa yleensä sensorien ja releiden liitettävyyden ilman kaapelia. Sensoriverkot voidaan luokitella esimerkiksi sensorityyppien, sovellusten, toimintaympäristön ja verkkoparametrien perusteella.

Langattomalla sensoriverkolla on kolme kiinnostuksen kohdetta. Ensiksikin on tarvetta halpaan sensorien asentamiseen, sillä halvan sensorin hinta saattaa nousta kalliiksi, mikäli asennuskustannukset maksavat liikaa. Lisäksi kaapelointisäännöt teollisuudessa ja asumisympäristössä saattavat vaatia lisämateriaalia tai -toimintoja kaapelin asentamisessa. Toiseksi kaapelit vaativat liittimiä, jotka voivat irrota, kadota, olla väärin yhdistettyjä tai rikkoontuneita. Kolmantena näkökohtana on langattoman sensoriverkon muodon järkevä alemman tason ylläpidettävyysjärjestelmä, joka mahdollistaa sensoririkkaan ympäristön,

josta voidaan generoida runsaasti tietoa teollisen toiminnan kehittämistä varten. Langaton sensoriverkko sallii yleensä myös suurelta laitemäärältä ja teollisuusjärjestelmiltä kerätyn usein toistuvan tiedon tiivistämisen. Suuri määrä johdollisesti keskitettyjä järjestelmiä tuo kompleksisuutta järjestelmään, eikä ole näin mahdollinen useissakaan tapauksissa.

Langaton sensoriverkkoratkaisu tarjoaa tiedonsiirtoyhteyden ilman liittimiä. Toinen etu langattomassa ratkaisussa on liikkuvuus, vaikka langattoman sensoriverkon yhteydessä tämä kyky onkin vaihdettu helppoon asennettavuuteen. Liikkuvuus ei ole normaalisti vaatimus langattomille sensoriverkkojärjestelmille. Liikkuvat laitteet kuitenkin asettavat vaatimuksia joustavasta verkonhallinnasta, joka tuo verkkoon ad-hoc -piirteen.

6.3 TTCN-3

LR-WPAN -laite koostuu monista erilaisista testattavista osista, kuten tässä työssä testattavasta järjestelmästä ja lisäksi virransäästö- ja reititysmoduuleista. Testattava järjestelmä muodostaa yhden osan suuremmasta kokonaisuudesta, kun ollaan kehittämässä ja rakentamassa toimivaa laitetta. Testausjärjestelmän tehtävänä on mahdollistaa erilainen tietoliikennetestaaminen, jotta kehitettävää järjestelmää voidaan testata sen sitä vaatiessa.

Tietoliikenneohjelmistojen testaaminen on monesti hankalampaa ja vaikeampaa kuin tavallisten ohjelmistojen, sillä tietoliikenneohjelmistot ovat monesti monimutkaisempia ja niiden tietoihin käsiksi pääseminen on vaikeaa. Mustalaatikkotestaukseen perustuva TTCN-3 -testauskieli, jossa seurataan testattavan ohjelmiston toimintaa tai siihen vaikutetaan ulkopuolelta tietämättä mitään ohjelmiston yksityiskohtaisesta toteutuksesta, soveltuu näin hyvin LR-WPAN -laitteen testauskieleksi.

TTCN-3 on joustava ja tehokas kieli, joka on suunniteltu erityyppisille reaktiivisille järjestelmille määritysten testaamiseen yli monimuotoisten kommunikointiporttien. TTCN-3 vaatii toimivan MAC-tason, jotta sen päällä olevaa protokollapinoa päästään testaamaan, siis pelkkä fyysinen kerros ei vielä riitä testausta varten. TTCN-3 -kielellä on merkittäviä etuja verrattuna myös muihin perinteisiin testausmetodeihin ja -kieliin. TTCN-3:n etuina voidaan pitää esimerkiksi sen yleiskäyttöisyyttä, uudelleenkäytettävyyttä ja työkaluriippumattomuutta, jonka vuoksi se erottuu edukseen monista muista perinteisistä

testausmetodeista ja skriptauskielistä. Tavallisimmat TTCN-3:n sovellusalueet ovat protokollatestaus, palvelutestaus (engl. *service testing*), moduulitestaus, CORBA-pohjaisten alustojen testaus ja niin edelleen.

ISO 9646 -standardisarjaan perustuva tietoliikenneohjelmistojen testaus määrittelee metodologian tietoliikenneprotokollien yhdenmukaisuustestaukselle, joka luo pohjan myös tietoliikennelaitteiden tyyppihyväksynnälle [25]. Yhdenmukaisuus- eli konformanssitestauksen avulla pyritään selvittämään, että järjestelmä toimii sille asetettujen toiminnallisten vaatimusten mukaisesti. TTCN on alun perin suunniteltu juuri tietoliikenneprotokollien yhdenmukaisuustestaukseen, jonka vuoksi sitä on ajateltu tyyppihyväksyntätestauksessa käytettäväksi. TTCN-3 ei kuitenkaan ole edeltäjiensä tapaan rajoittunut ainoastaan yhdenmukaisuustestaukseen, vaan sitä voidaan käyttää myös monenlaisessa muussa testauksessa esimerkiksi yhteentoimivuus-, kestävyys-, regressio-, järjestelmä- ja integrointitestauksessa. Yleisesti TTCN kuitenkin nähdään hyväksi apuvälineeksi protokollatestauksessa, sillä sen avulla voidaan melko yksinkertaisesti todentaa rajapintojen toiminnallisuus standardin mukaisesti. On myös syytä huomata, että testausta ei tarvitse rajoittaa ainoastaan yhdelle protokollalle, vaan se soveltuu hyvin eri protokollatasojen sekä datan testaukseen. Toisaalta TTCN-3:n avulla voidaan myös testata verkkoa ja sen toimintaa olosuhteita muuttamalla.

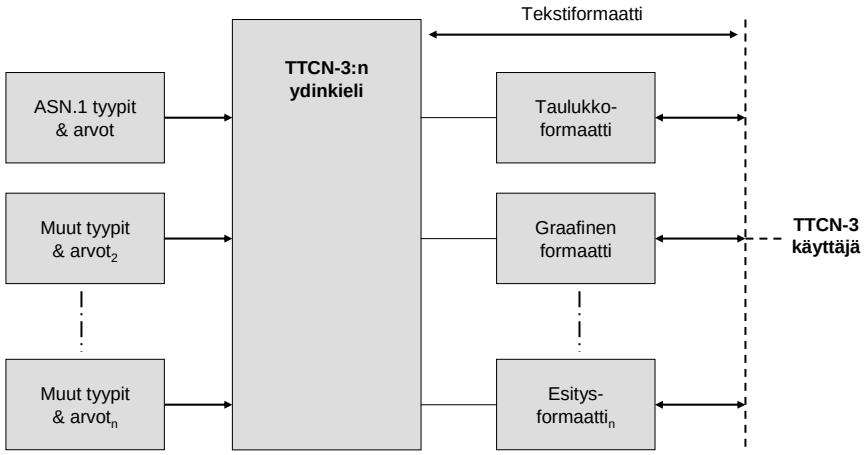
ETSI:n stardardoimaan abstraktiin testien kuvauskieleen perustuvan TTCN-3:n etuna muihin testauskieliin on se, että testauksen jatkokehittely on helpompaa kuin itse tehdyillä ja räätälöidyillä testausjärjestelmillä. Itse tehtyihin räätälöityihin, monesti C- tai C++-kieleen pohjautuviin, testeihin verrattuna TTCN-3 tarjoaa oman testausta varten suunnitellun testikielen. TTCN:n monipuolisuuden ansiosta sitä voidaan käyttää myös ohjelmiston kehitysajan eri vaiheissa.

Muihin itse räätälöityihin testausjärjestelmiin nähden eduksi voidaan katsoa se, että testausta voidaan jatkokehittää standardin mukaisesti. TTCN-3:lla on standardi testauskieli sekä -arkkitehtuuri, jonka vuoksi se on myös paljon yleiskäyttöisempi kuin monet muut testausjärjestelmät. TTCN-3:n standardi arkkitehtuuri on suunniteltu siten, että testausjärjestelmä on siirrettävissä ympäristöstä toiseen vain adapteriin tehtävien

muutosten jälkeen. Lopullisen tuotteen testaustakin ajatellen TTCN-3 antaa etuja, sillä sen avulla voidaan suorittaa lopullinen testaus standardinmukaisesti ISO-9646 -sarjan tietoliikennelaitteiden tyyppihyväksyntään perustuen.

6.3.1 TTCN-3 -kielen arkkitehtuuri

Kuten kuvassa 23 käy ilmi, on TTCN-3 rakennettu tekstikielestä, joka mahdollistaa rajapinnat erilaisille tiedon kuvauskielille ja esittämisformaateille. Syntaktisesti TTCN-3 on melko erilainen kuin aikaisemmat TTCN-versiot. Kuitenkin useat hyviksi havaitut TTCN-perustoiminnot on säilytetty ja joitakin tapauksia on parannettu.



Kuva 23: TTCN-3 -kielen arkkitehtuuri [9].

TTCN-3:n ydinkieli on tekstipohjainen ohjelmointikieli, joka tarjoaa työkalujen välisen formaattien vaihdon. Ydinkieltä voidaan käyttää itsenäisesti ilman esitysformaatteja. TTCN-3:n graafinen esitysformaatti puolestaan visualisoi testitapaukset käyttäen sekvenssikaaviota, kun taas taulukkoesitysformaatti on suunniteltu yhdenmukaisuustestausta varten. Taulukko- ja graafinen formaatti eivät ole käytettävissä ilman ydinkieltä, sillä esitysformaattien käyttö ja toteutus täytyy tehdä ydinkieleen pohjautuen.

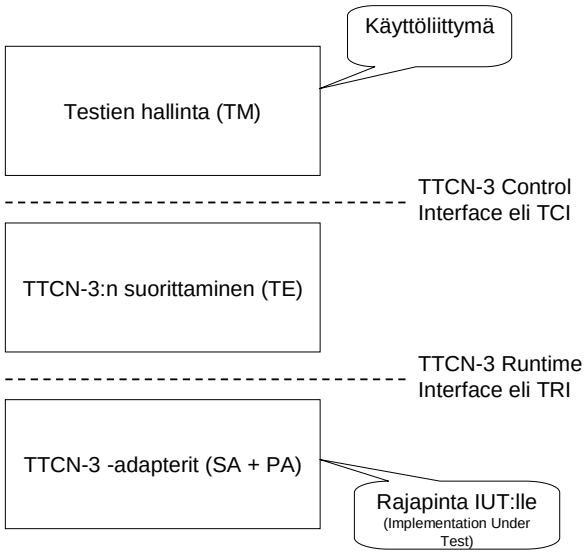
Taulukko- ja graafinen formaatti ovat ensimmäisiä esitysformaatteja, joita TTCN-3 -käyttäjä voi käyttää. Myös muiden formaattien standardisointi tai patentointi esitysformaateiksi on mahdollista TTCN-3 -käyttäjien toimesta. TTCN-3 on täysin

yhteensovitettu ASN.1:n (Abstract Syntax Notation One) kanssa. Tämän vuoksi sitä voidaan vapaasti käyttää yhdessä TTCN-3 -moduuleiden kanssa vaihtoehtoisena tietotyypin ja arvon syntaksina.

6.3.2 TTCN-3:n testausjärjestelmän arkkitehtuuri

TTCN-3 -testausjärjestelmä voidaan ajatella käsitteelliseksi sarjaksi vuorovaikutusmoduuleita, joista jokainen moduuli (engl. *entity*) vastaa yksityiskohtaisista toiminnallisuuden näkökohdista testausjärjestelmän toteuttamisessa. Nämä moduulit hallitsevat testien suorittamisen, tulkkauksen tai TTCN-3 -koodin kääntämisen, toteuttavat asianmukaisen kommunikoinnin testattavan järjestelmän (engl. *system under testing*, SUT) kanssa, toteuttavat ulkoiset funktiot ja käsittelevät ajastinoperaatioita.

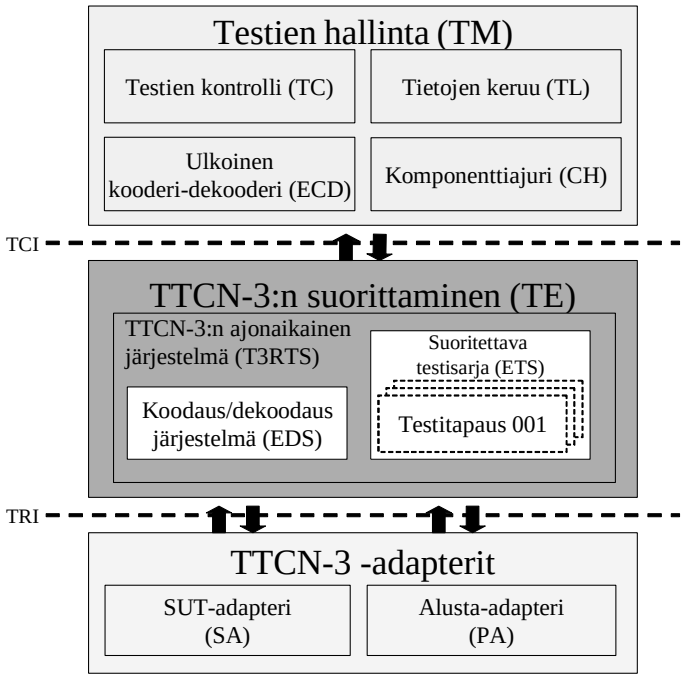
TTCN-3 -testausjärjestelmän moduulit jaetaan kuvan 24 osoittamalla tavalla kolmeen yleiseen osaan: testien hallintaan (engl. *test management*, TM), TTCN-3:n suorittamiseen (engl. *TTCN-3 executable*, TE) ja TTCN-3 -sovittimeen eli adapteriin, joka koostuu SUT-adapterista (SA) ja alusta-adapterista (engl. *platform adapter*, PA).



Kuva 24: TTCN-3 -testausjärjestelmän yleinen arkkitehtuuri.

Testien hallinta

Testien hallinta voidaan jakaa vielä testin suorittamisen kontrolloimiseen ja testitapahtumien tiedonkeruuseen. Testien kontrolloimisen (engl. *test control*, TC) vastuulla on testausjärjestelmän kokonaisvaltainen hallinta. Kun testausjärjestelmä on alustettu, käynnistetään testauksen suorittaminen TC-moduulisssa. Moduuli vastaa varsinaisten TTCN-moduulien, moduuliparametrien ja/tai IXIT-informaation (Implementation Extra Information for Testing) levittämisestä TE:n niitä tarvitessa. IXIT-informaatio on asiakkaan toimesta testauslaboratoriolle tuotettua lisätietoa testattavasta protokollasta tai järjestelmästä sekä testausympäristöstä. Tavallisesti TC-moduuli toteuttaa myös testausjärjestelmän käyttöliittymän.



Kuva 25: TTCN-3 -testausjärjestelmän rakenne [10].

Testitapahtumien ja muun vastaavan tiedon tallentamista hoitaa tietojen keruu -moduuli (engl. *test logging*, TL), jonka vastuulla on testilokien ylläpito. TL-moduulin rajapintaa ei ole suunnattu vain johonkin tiettyyn TM:n moduuliin, minkä ansiosta myös jokainen TE-moduulin osa saattaa lähettää tiedonkeruupyynnön TL-moduulille. Näin ollen sekä TE-

moduulien että esimerkiksi TC:n tallentamia testinhallintatietoja voidaan tallentaa TL-moduulin toimesta.

Testinhallinta sisältää myös ulkoisen kooderi-dekooderimoduulin (engl. *external CoDecs*, ECD) ja komponenttiajurimoduulin (engl. *component handler*, CH). ECD:tä voidaan käyttää TE:n sisäänrakennetun kooderi-dekooderin kanssa joko rinnakkain tai sen sijasta. Niiden molempien tehtävänä on tiedon koodauksen ja dekoodauksen yhdistäminen TE:n sisällä tapahtuvaan viesti- tai proseduuripohjaiseen kommunikointiin. CH on puolestaan vastuussa jaetuista rinnakkaisista testauskomponenteista.

TTCN-3:n suorittaminen

TTCN-3:n suorittamismoduulin vastuulla on abstraktien TTCN-3 -testisarjojen (ATS) suorittaminen ja tulkkaus. Suorittaminen voidaan jakaa kolmeen vuorovaikutukselliseen moduuliin: suoritettavaan testisarjaan (engl. *executable test suite*, ETS), TTCN-3:n ajonaikaiseen järjestelmään (engl. *TTCN-3 runtime system*, T3RTS) ja koodaus/dekoodaus -järjestelmään (engl. *encoding/decoding system*, EDS).

ETS-moduulin vastuulla on testitapausten suorittaminen ja tulkkaus sekä TTCN-3 -moduuleissa määriteltyjen testitapahtumien järjestyksen ja yhteensopivuuden käsittely. ETS toimii vuorovaikutuksessa T3RTS-moduulin kanssa lähettäen, vastaanottaen ja kirjaten testaustapahtumia testitapausten suorittamisen aikana. Se myös luo ja poistaa TTCN-3 -testikomponentteja sekä käsittelee ulkoisia funktiokutsuja, toimintaoperaatioita ja ajastimia. ETS-moduuli ei ole itse suorassa vuorovaikutuksessa SA:han TRI:n (TTCN-3 Runtime Interface) kautta.

T3RTS-moduuli on vuorovaikutuksessa TM-, SA- ja PA-moduulien kanssa TCI- (TTCN-3 Control Interface) ja TRI-rajapintojen kautta halliten samalla TE:n sisäisiä ETS- ja EDS-moduuleita. T3RTS alustaa adapterit sekä ETS- että EDS-moduulit. Tämä moduuli suorittaa kaikki välttämättömät toiminnot testitapausten tai funktioiden suorittamisen käynnistyksen yhteydessä yhdessä ETS-moduulin parametrien kanssa. Se kyselee TM-moduulilta ETS-moduulin vaatimia moduuliparametrien arvoja lähettäen lokitietoa takaisinpäin. Se myös kerää ja tekee päätöksen ETS-moduulin palauttamien yhdistettyjen testitapausten perusteella, siitä onko testi hyväksytty vai hylätty.

T3RTS-moduulin vastuulla on myös TTCN-3 -komponenttien luonti ja poisto, viestien TTCN-3 -semantiikka ja proseduuripohjainen kommunikointi, ulkoisen funktion kutsuminen, toimintaoperaatiot ja ajastimet. T3RTS siis ilmoittaa SA:lle siitä, mikä viesti- tai proseduurikutsu on lähetettävä testattavalle järjestelmälle (SUT) tai alusta-adapterille (PA) siitä, mikä ulkoinen funktio on suoritettava tai mitkä ajastimet on käynnistettävä, pysäytettävä, kysyttävä tai luettava. Samoin T3RTS ilmoittaa ETS-moduulille testattavalta järjestelmältä saapuvista viesteistä tai proseduurikutsuista sekä aikalisätapahtumista. Ennen kuin lähetettäviä tai vastaanotettavia viestejä ja proseduurikutsuja voidaan tehdä, on T3RTS:n herätettävä EDS-moduuli niiden koodausta tai dekoodausta varten. T3RTS-moduulin tehtävänä on siis toteuttaa kaikki viesti- ja proseduuripohjainen kommunikointi testikomponenttien välillä.

EDS-moduulin vastuulla on testidatan koodaus ja dekoodaus. TTCN-3 -moduulin suorituksessa määriteltyä testidataa käytetään kommunikointioperaatioissa testattavan järjestelmän kanssa. Jos testidatan koodausta ei ole määritelty TTCN-3 -moduulissa, ei data-arvoja voida koodata ilman erityisiä toimenpiteitä. EDS-moduuli ei ole suorassa vuorovaikutuksessa SA:han TRI-rajapinnan kautta, vaan vaatii aina T3RTS:n avukseen yhteyden muodostamisessa.

TTCN-3:n abstrakteissa testisarjoissa selvitetyt ja nimetyt ajastimet voidaan käsitteellisesti luokitella selvästi TE:n sisälle. Vaikka ajastimet luodaan TE:ssä, ovat ne kuitenkin PA:n toteuttamia.

TTCN-3 -adapterit

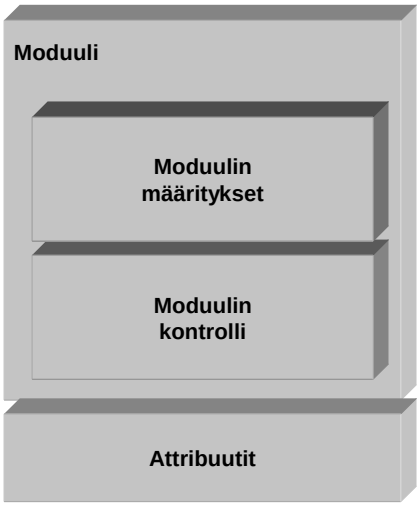
TTCN-3 -adaptereista SA muokkaa TTCN-3 -testausjärjestelmän ja testattavan järjestelmän välisen viesti- ja proseduuripohjaisen kommunikoinnin testausjärjestelmässä yksiselitteiseksi. SA vastaa testausjärjestelmän todellisen rajapinnan toteuttamisesta TTCN-3 -testikomponenttien kommunikointiporttien ja testausjärjestelmän rajapinnan porttien välillä. SA hoitaa lähetettyjen kyselyjen ja testattavan järjestelmän toimintaoperaatioden levittämisen TE:lta testattavalle järjestelmälle sekä ilmoittaa TE:lle kaikista saapuvista testitapahtumista liittämällä ne myös TE:n porttijonoon. Testattavan järjestelmän ja TE:n yhteiset proseduuripohjaiset kommunikointioperaatiot on toteutettu

SA:ssa. SA on vastuussa erilaisten viestien erottamisesta proseduuripohjaisen kommunikoinnin välillä ja niiden levittämisestä joko testattavalle järjestelmälle tai TE:lle. TTCN-3:n proseduuripohjainen kommunikointisemantiikka käsitellään TE:ssä. SA:lla on rajapinta TE:n kanssa, jota käytetään testattavan järjestelmän viestien lähettämisessä SA:lle ja vaihdettaessa koodattua testidataa kahden moduulin välillä yhdessä testattavan järjestelmän kommunikointioperaatioiden kanssa.

Toinen TTCN-3 -adaptereista, PA, toteuttaa puolestaan TTCN-3:n ulkoiset funktiot ja määrittää TTCN-3 -testausjärjestelmälle yksikäsitteisen ajan. Tässä moduulissa siis toteutetaan niin ulkoiset funktiot kuin kaikki ajastimetkin. Ajastimen tapaukset on luotu TE:ssä. PA:n vastuulla on ilmoittaa TE:lle ajastimen määräajan umpeutumisesta.

6.3.3 TTCN-3 -testisarjan rakenne

TTCN-3 -testimääritykset rakennetaan moduuleista. Kuvassa 26 on kuvattu moduulin yleinen rakenne. TTCN-3 -moduuli koostuu määrittely-, kontrolli- ja attribuuttiosasta. Moduulia, joka on määritelty täydellisin testimäärittelyin, kutsutaan usein testisarjaksi.

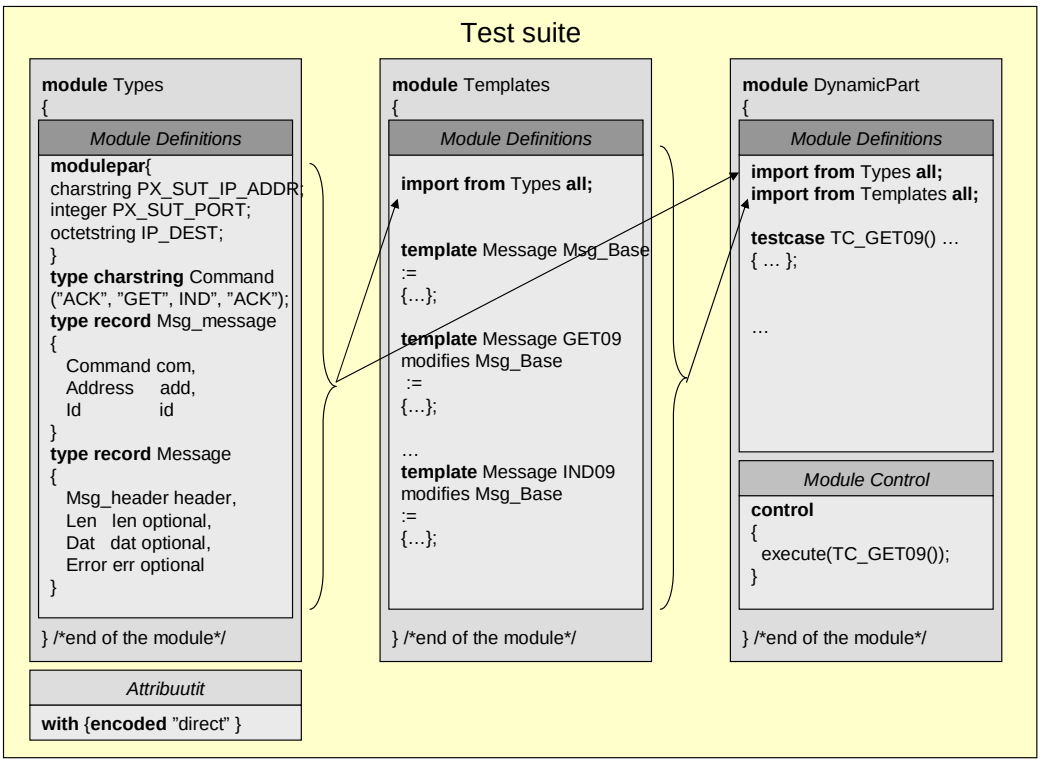


Kuva 26: TTCN-3 -moduulin yleinen rakenne.

Määrittelyosassa määritellään sovelluksen spesifiset tietotyypit, testidata, testausjärjestelmän arkkitehtuuriset elementit ja toiminnot, jotka kuvaavat testausjärjestelmän dynaamista käyttäytymistä. Moduulin kontrolliosassa on varsinaisten

testien toteuttaminen tai testitapausten hallinta. Kontrolliosassa on mahdollista esimerkiksi kontrolloida testien suorittamisen järjestystä, testien suorittamismääriä, testitapausehtojen suorittamisen muodostaminen riippumatta muiden testitapausten tuloksista. On myös mahdollista valita tai hylätä testitapaukset riippuen tietyistä kriteereistä. Attribuuttien määritysosa sallii erilaisten tarkenteiden yhdistämisen moduulissa oleviin määrityksiin tai määritysryhmiin. Attribuuttien määrittelyosassa määritellään erilaiset määritteet ja määrittelyryhmät, joita moduulissa käytetään. Määrittelyosa sallii nähtävillä olevien attribuuttien määrittämisen suhteessa yksityiskohtaisiin esitysformaatteihin, koodaukseen ja koodauksen muutosohjeisiin tai jonkun käyttäjän määrittelemiin määritteisiin.

Täydellinen testisarja saattaa koostua useammasta kuin yhdestä moduulista. Käyttämällä modularisoituja testisarjoja edistetään testisarjojen osien käytettävyyttä. Kuvassa 27 on yksi esimerkkitapaus, jossa testisarja on jaettu pienempiin moduuleihin.



Kuva 27: TTCN-3 -moduulien suhteet.

Moduulin määritysosassa esitetyt määrittelyt ovat globaalisti näkyviä juuri tässä kyseissä moduulissa. Näin ollen moduulissa tehtyjä globaaleja määrittelyjä voidaan käyttää ilman

erilisiä toimenpiteitä kaikkialla moduulissa aina kontrolliosaa myöten. Jos puolestaan määrittely on suoritettu funktion tai testitapauksen sisällä, on muuttuja ainoastaan paikallisesti näkyvä.

Kun testisarja sisältää useita moduuleja, ei yhdessä moduulissa tehtyihin määrityksiin voida viitata ennen kuin, ne on tehty näkyviksi tuomalla määritykset moduuliin import-komennolla. Yleensä tyyppimäärittelyt tehdään omassa Types-moduulissa, erityiset testin mallit (templates) ilmoitetaan Template-moduulissa ja testauksen dynaaminen kuvaaminen on määritetty DynamicPart-moduulissa.

7 Case-tapaus: LR-WPAN -laitteen testaaminen

Case-tapauksessa käsittelen tutkimani sulautettuihin järjestelmiin kuuluvan LR-WPAN -laitteen testaamista. Pääasiassa testaamista tarkastellaan LR-WPAN -laitteen kehittämisen näkökulmasta, toisaalta tulevaisuudessa eteentulevaan sensoriverkon testaamiseen on tarkoitus vähän syventyä. Koska tehtävänäni oli vain testausjärjestelmän rakentaminen, käyn tässä luvussa läpi testauksen etenemistä sekä testitapausten sisältöä yksinkertaisen esimerkin avulla. Esimerkki on vain yksi pieni osa suuremmasta testauskokonaisuudesta käsittäen tällä hetkellä vain yhden laitteen yhden toiminnallisuuden tarkastelun. Varsinainen testaus koostuu useammista testitapauksista, joiden yhdistämisen pohjalta rakentuu koko testitapahtuma. Lopullista testitapahtumaa voidaan sitten toistaa jokaisen muutoksen, inkrementaalisen lisäyksen tai uusien osien integroimisen jälkeen, kun halutaan varmistua laitteen vastaavan määrityksiä myös siihen tehtyjen muutoksien jälkeen.

7.1 Case-tapauksen rajaus

Testaamisen tarkastelu on rajattu sulautettujen järjestelmien kehitystyössä käytettävän inkrementaalisen mallin mukaan, jota myös LOPO-projektissa on käytetty ohjelmistotuotannon vaihemallina. Testaus aloitetaan ensimmäisen inkrementin jälkeen, jonka seurauksena mallista valmistuu prototyyppi. Testausta ajatellen LR-WPAN -laitteen ensimmäisen prototyypin katsotaan valmistuneen, kun laitteen protokollapino on saatu valmiiksi. Tämän jälkeen jokaisen uuden ominaisuuden lisäämisen tai vanhojen asioiden muokkaamisen jälkeen katsotaan uuden inkrementin valmistuneen, jolloin laite on testattava. Näin ollen ensimmäisen prototyypin valmistuttua on tarkoituksena regressiotestata järjestelmä jokaisen siihen tehdyn lisäyksen tai muutoksen jälkeen.

Case-tapauksessa on tarkoituksena keskittyä ohjelmistotuotannon testausvaiheen tarkasteluun. Testausvaihe on jaettu V-mallin mukaisesti helpommin jäsennettäviksi osiksi. Case-tapauksessa keskitytään regressiotestaukseen, jota suoritetaan järjestelmäntestausvaiheessa dynaamisen mustalaatikkotestauksen avulla. Koska sulautettujen järjestelmien laitteiden tietoihin pääseminen on vaikeata, oli

testaustekniikaksi valittu TTCN-3 hyvä ratkaisu sitäkin ajatellen. TTCN-3 mahdollisti laitteen mustalaatikkotestauksen selvittäen näin laitteen toimintaa käyttötilanteessa.

Järjestelmätasolla suoritettavan testauksen tarkastelutavaksi valittiin regressiotestaus, koska järjestelmätasolla suoritettava testaaminen vaatii paljon toistettavuutta. TTCN-3 mahdollisti myös automatisoinnin, mikä oli yksi syy kyseisen teknologian valintaan. Automatisointi mahdollisti regressiotestauksen monipuolisemman käytön tarjoamalla mahdollisuuden myös kuormitustestaukseen, jossa suoritetaan, erityisesti tietoliikenteessä vaatimia, toistoja samoille laitteille peräkkäin. Käsin tehtynä kuormitustestauksen vaatimien toistojen määrät olisi ollut mahdotonta toteuttaa kuin myös testitulosten vertailu, joka meidän käyttöön saamassamme TTCN-3 -testaustyökalussa oli automatisoitu yhdessä testien suorittamisen kanssa. Nämä antoivat muutenkin hyvän mahdollisuuden tarkastella automatisoitua regressiotestausta.

Case-tapauksen päällimmäisenä tarkoituksena on keskittyä testaamaan LR-WPAN -laitteen käyttäytymistä. Valittujen teknologioiden pohjalta laitteen käyttäytymistä tarkastellaan annettujen syötteiden pohjalta saatujen vasteiden vertaamisella odotettuihin vasteisiin. Vertailemisen avulla on tarkoitus varmistua laitteen oikeanlaisesta käyttäytymisestä kehitysvaiheen järjestelmätestauksessa. TTCN-3 mahdollistaa myös laitteen testaamisen inkrementaalisessa kehitysvaiheessa jokaisen siihen tehdyn muutoksen jälkeen, näin voidaan varmistua laitteen toimimisesta ennen seuraavaan vaiheeseen siirtymistä.

TTCN-3 tarjosi myös protokollatestaukselle mahdollisuuden. Jokaisen protokollakerroksen testaaminen oli mahdollista yhdessä verkon testaamisen kanssa. Näin ollen ensin testattuja ja toimiviksi havaittuja laitteita voidaan myöhemmin testata myös osana suurempaa verkkoa. Tällöin on mahdollista laitteen oikeanlaisella käyttäytymisellä varmistua myös verkon oikeanlaisesta käyttäytymisestä. Toisaalta yksistään verkkoakin voidaan testata, jolloin ei sovi unohtaa olosuhteiden muutoksesta aiheutuvaa vaikutusta verkolle.

7.2 Testausjärjestelmä

Testausjärjestelmä on monesti yksilöllinen, minkä vuoksi sen siirtäminen on vaikeaa. LOPO-projektissa valitut testaustekniikat ja standardit mahdollistavat testausjärjestelmän

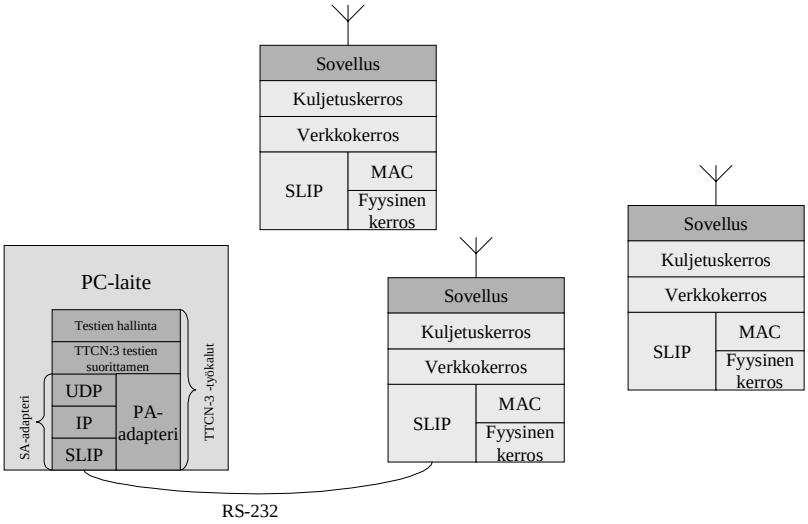
muuttamisen toisenlaiseksi melko vähillä toimenpiteillä. TTCN-3 -standardin tarkoituksena onkin ollut, että se on kieli- ja alustariippumaton, jolloin se mahdollistaa laajan käyttömahdollisuuden. Monet muut testaustyökalut tai itse räätälöidyt työkalut ovat usein tiettyihin ohjelmointikieliin sidottuja, kuten esimerkiksi C:hen tai C++:aan. Samoin on monesti myös käyttöjärjestelmien osalta, että testaustyökalu ei vaadi testattavalta järjestelmältä jotain tiettyä käyttöjärjestelmää toimiakseen. TTCN-3 tarjoaa siis vapauksia tässä asiassa, sillä se teknologiana antaa mahdollisuuden eri ohjelmointikielillä ja käyttöjärjestelmillä toteutettujen testattavien järjestelmien testaamiselle.

7.2.1 Testausjärjestelmän konfiguraatio

LOPO-projektissa toteutetun LR-WPAN -laitteen testaustekniikaksi valittiin standardoituun kieleen ja arkkitehtuuriin perustuva TTCN-3. Testausalustaksi saimme käyttöön OpenTTCN Oy:ltä OpenTTCN Tester for TTCN-3 -testaustyökalun. Helppokäyttöisen OpenTTCN:n protokollatesterin avulla testaus voidaan automatisoida systemaattiseksi sekä ISO/IEC 9646 ja TTCN-3 -standardeihin perustuvaksi.

OpenTTCN:n tarjoama testaustyökalu on käyttövalmis ohjelmistonkehittäjille ja testaajille. Se koostuu testien ajoympäristöstä eli testien suorittamisosasta ja testien hallinnasta, joka sisältää käyttöliittymän testaajalle. Näin testausjärjestelmän rakentamisessa tehtäväksi jäi testattavan järjestelmän ja TTCN-protokollatesterin yhteenliittäminen.

TTCN-3 tarjoaa tekniikkana standardin arkkitehtuurin avulla helpon tavan toteuttaa testausjärjestelmän yhteenliittäminen SA-adapteriin valmiin TRI-rajapinnan kautta. SA-adapteriin tarvittiin pieniä lisäyksiä, jotta yhteys testattavaan järjestelmään onnistui RS-232 -sarjaliikenneprotokollan avulla. Toteutettavana oli sarjaportista lukemisen ja sarjaporttiin kirjoittamisen lisäksi UDP-, IP- ja SLIP-protokollien toteuttaminen SA-adapteriin, minkä jälkeen testausjärjestelmä oli valmis käytettäväksi. Myös tarvittavat muutokset ulkoiseen kooderi/dekooderiin teimme itse, sitä mukaan kun tarvetta ilmeni uusien tietotyyppien ilmaannuttua.



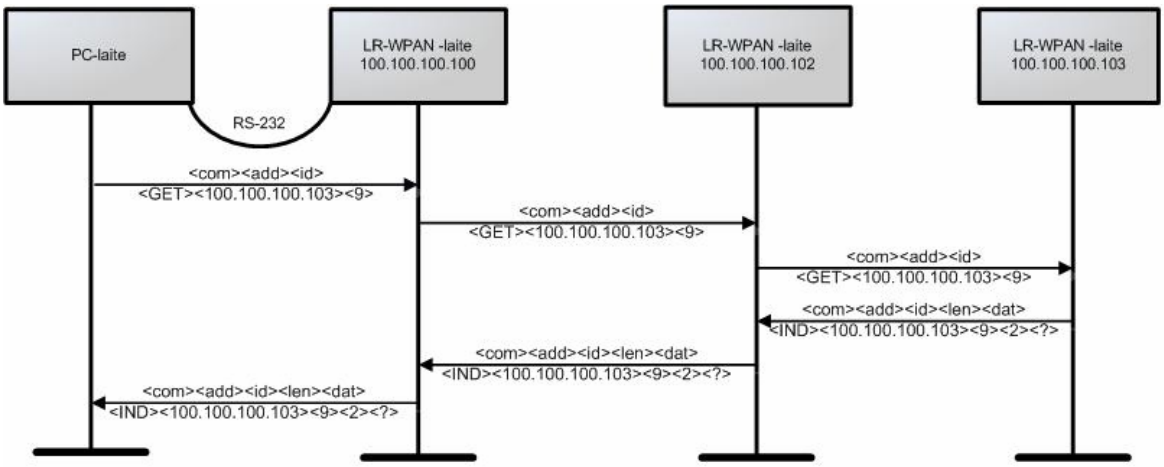
Kuva 28: LR-WPAN -laitteen testausjärjestelmä.

OpenTTCN-3 -testeri toimii tietokoneella, joka on yhteydessä LR-WPAN -sensoriverkon yhteen tiedonkeruulaitteeseen RS-232 -sarjaliikenneprotokollan välityksellä. Näin ollen testausympäristö ei aiheuta minkäänlaisia muutoksia normaaliin LR-WPAN -verkkoon. Armour [1] muistuttaa, että jos emme voi todistaa riittävän yksityiskohtaisesti ja kontrolloidusti asiakkaan ympäristössä tapahtuvaa tilannetta, emme voi myöskään paljastaa julkaistavan ohjelmiston virheitä. Näin ollen on erityisen tärkeätä voida testata laite ilman, että siihen joudutaan tekemään muutoksia testausta varten. Kuvasta 28 käy ilmi sekä tietokoneen että LR-WPAN -laitteiden arkkitehtuuri ja niiden välinen yhteys.

7.2.2 Datapaketin eteneminen testausjärjestelmässä

Testausjärjestelmän konfiguraatio muodostuu yhdestä PC-laitteesta, yhdestä tiedonkeruulaiteesta ja kahdesta tavallisesta mittaavasta LR-WPAN -laitteesta. Tiedonkeruulaitteena toimivan laitteen IP-osoite on 100.100.100.100 ja mittaavien 100.100.100.102 sekä 100.100.100.103. Tavallisten mittaavien LR-WPAN -laitteiden määrä on rajoitettu kahteen vain malliesimerkissä, mutta normaalisti kyseisten laitteiden määrällä ei ole testausjärjestelmään merkitystä. Testauksessa tarkastellaan mustalaatikkotestaustekniikan omaisesti lähetettävän syötteen aikaansaaman vasteen oikeellisuutta odotetunlaiseen vasteeseen PCO-kohdassa.

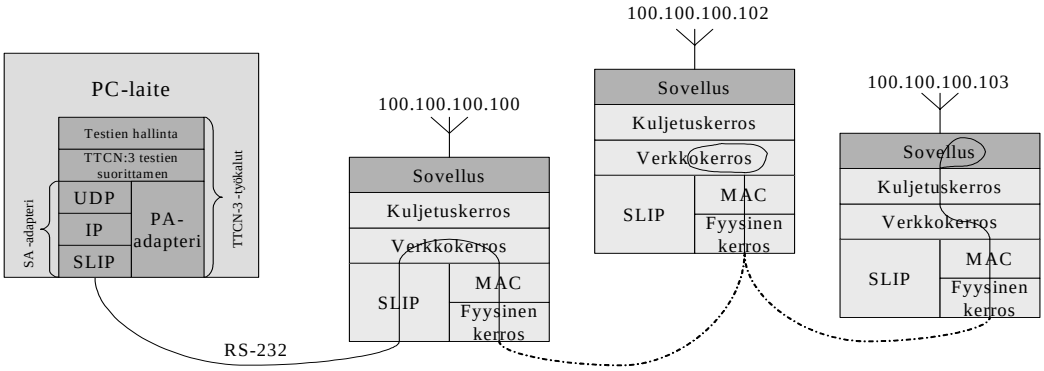
Yksinkertaisuudessaan testitapaus etenee seuraavasti. Moduuleissa määritelty viesti lähetetään testeriltä koodattavaksi. Koodattu data lähetetään edelleen testattavalle järjestelmälle SA:n välittämänä. Datapaketti, joka lähtee RS-232 -sarjaliikennekaapelia pitkin, on paketoitu UDP-, IP- ja SLIP-protokollilla. Paketoitu datapaketti itse sisältää komennon GET, kyselyssä kohteena olevan laitteen IP-numeron 100.100.100.103, sekä kyselyn kohteena olevan laitteen muokattavan arvon id-numeron 9. Kuvassa 29 on esitetty sekvenssikaaviona datapaketin eteneminen testausympäristön laitteissa. Kaaviosta käy myös ilmi datapaketin sisältö ilman siihen liitettäviä UDP-, IP- ja SLIP-protokollia. Kyseisessä testitapauksessa keskitytäänkin ainoastaan sovelluksen datan sisällön testaamiseen, jolloin PCO on sijoitettu UDP-, IP- ja SLIP-protokollien purkamisen jälkeen. Mutta yhtä hyvin testauksen kohteena voisivat olla UDP-, IP- tai SLIP-protokollakenttien tiedotkin. Nyt datan testaamisessa keskitytään enemmänkin datan oikeellisuuden testaamiseen. Oikeellisuutta testattaessa keskitytään siihen, että data tulee perille oikeassa järjestyksessä ja on odotetunlaista.



Kuva 29: Datapaketin eteneminen testausympäristön laitteissa.

Kuten kuvasta 30 ilmenee, puretaan ensin SLIP-protokolla RS-232-sarjaliikennekaapelia pitkin 100.100.100.100-laitteelle lähetetystä viestistä. Tämän jälkeen reititysominaisuuden omaavan verkon 100.100.100.100-laitteen verkkokerroksella tarkistetaan, oliko datapaketti tarkoitettu kyseiselle laitteelle. Koska datapaketti ei kuulunut kyseiselle laitteelle, laitetaan se MAC- ja fyysisen kerroksen kautta eteenpäin radiotielle. Tämän jälkeen datapaketin vastaanottaa 100.100.100.102-laite, jonka protokollapinossa se nousee fyysisen ja MAC-

kerroksen kautta verkkokerrokselle tarkistettavaksi. Koska pakettia ei ollut tällekään laitteelle tarkoitettu, laitetaan se takaisin samaa reittiä radiotielle. Kun oikea 100.100.100.103-laite löytyy, päästää se datapaketin verkkokerroksen ja kuljetuskerroksen kautta sovellustasolle.



Kuva 30: Testausjärjestelmän konfiguraatio.

Vastaavasti takaisin vastauspaketti kulkee samaa reittiä kuin kyselypaketti, mutta vastakkaiseen suuntaan. Vastauspaketin sisältö on hieman muuttunut, sillä komento on vaihtunut IND:ksi. Parametrina saatu IP-osoite ja id-arvo ovat säilyneet puolestaan muuttumattomina. Kyselypakettiin verrattuna kuuluu vastauspakettiin lisäksi kentät len ja dat, jotka ovat vastauspaketin lopussa. Dat-kenttä kertoo kyselyn seurauksena saadun tiedon. Tässä kyseisessä tapauksessa ?-merkillä hyväksytään kaikki arvot. Len-kenttä kertoo puolestaan dat-kentän pituuden, joka on kyseisessä tapauksessa 2 tavua. Kun vastauspaketti on saapunut testattavalta järjestelmältä verkkoteitse SA:lle, lähetetään se dekoodauksen kautta TTCN-3:n arvona testerille. Testerillä toteutetaan sitten ajonaikana suoritettava vertailu.

7.3 Testisarjan TTCN-3 -kieliset moduulit

Case-tapauksen testisarja koostuu neljästä moduulista, joista yksi on parametreja sisältävä moduuli. Kolme muuta moduulia on pyritty jakamaan tietotyyppien ja mallien määrittämisen sekä suorituksen yksityiskohtaisemman sisällön mukaan. Moduuleihin jaon avulla on pyritty testitapausten ylläpitämisen helpottamiseen. Seuraavassa alaluvussa

käydään yksinkertaisen esimerkin avulla läpi osa moduulien sisällöstä, jotka kokonaisuudessaan ovat nähtävissä liitteinä (Liiteet 1-4).

7.3.1 UDPModule1

Esimerkissä 1 on UDPModule1-moduulissa määritettävät sovelluksen tietotyypit, joista ensimmäisenä on määritelty testikomponentissa parametrina käytettävä address-tietotyyppinen tietue (engl. *record*). Toisena moduulissa on määritelty käytettävät parametrit. Tämän jälkeen on määritelty datapaketeissa esiintyvät perustietotyypit Command, Address, Id, Len, Dat ja Error. Raw on puolestaan virhetilanteissa käytettävä tietotyyppi.

```
module UDPModule1
{
        // Address type used for UDP packets.
        type record address
        {
                charstring host,
                integer portField
        }

        modulepar
        {
                // IP address and port number of the SUT.
                charstring PX_SUT_IP_ADDR;
                integer PX_SUT_PORT;

                // destination IP
                octetstring IP_DEST;
        }

        //type record Message
        type charstring Command ("ACK", "GET", "IND", "SET")
        type octetstring Address length(4)
        type integer Id (0..12)
        type integer Len (0..32)
        type octetstring Dat
        type integer Error (0..110)

        // For sending invalid packets and for receiving packets of invalid
        // format that cannot be properly decoded by the SUT adapter.
        type octetstring Raw;

        type record Msg_header
        {
                Command         com,
                Address         add,
                Id              id
        }
```

```
        type record Message
        {
                Msg_header      header,
                Len             len             optional,
                Dat             dat             optional,
                Error           err             optional
        }
}
```

Esimerkki 1: UDPModule1-moduuli.

Koska jokainen lähetettävä ja vastaanotettava datapaketti sisältää Command-, Address-, ja Id-kentät, on niistä koottu yhteinen tietuetyyppi Msg_header. Valinnaisina (engl. *optional*) kenttinä puolestaan datapaketeissa ovat Len-, Dat- ja Error-kentät. Näin ollen jokainen datapaketti, koostuu aina Msg_header:sta sekä valinnaisesti käytettävistä Len-, Dat- ja Error-kentistä, joiden pohjalta muodostetaan Message-tietue. Myöhemmin message-tietuetta tullaan käyttämään UDPModule2:ssa, kun perustietotyyppejä sisältävistä tietueista rakennetaan malli (engl. *template*).

7.3.2 UDPModule2

Esimerkissä 2 olevassa UDPModule2-moduulissa määritellään perustietotyyppejä sisältävistä tietueista kootut mallit. Ennen kuin UDPModule1-moduulissa määritettyjä tietotyyppejä voidaan käyttää UDPModule2-moduulissa, on ne tuotava näkyviksi import-komennolla. Tämän jälkeen UDPModule1:n tietotyypit on käytettävissä tämän moduulin sisällä ilman määrittämisiä.

Moduulissa on import-komennon jälkeen alustettu Message-tyyppinen Msg_Base -malli, joka toimii Message-tietuetta käyttävien mallien pohjana. Msg_Base -mallissa header-tietue on alustettu tyhjäksi, jolloin sen kaikkiin kenttiin odotetaan jotain tietoa. Toisin on len-, dat-, err-kenttillä, jotka on alustettu omit:lla. Tällöin alustus antaa mahdollisuuden jättää kyseiset kentät pois, mikäli niitä ei tarvita datapaketin rakentamisessa.

Msg_Base -mallia modifioimalla voidaan rakentaa uusia tarvittavan muotoisia malleja. Kyseisessä esimerkissä sekä GET09- että GET09_a -mallit ovat modifioitu Msg_Base:sta. Modifioitujen mallien GET-kyselyissä tarvitaan vain header-osaa, jolloin alustamisen yhteydessä len-, dat- ja err-kentät jätetään huomioimatta. Modifioiduissa malleissa

käytetään IP_DEST -parametria, jonka arvo on määritelty UDPModuleParamers-moduulissa 100.100.100.103:ksi.

GET-kyselyissä ensimmäisessä GET09:ssä menee syötteenä järjestelmälle alun perin oikeaksi ymmärrettävä datapaketti. Toisessa GET09_a:ssa puolestaan järjestelmältä pyydetään lämpötilaa sellaiselta id:ltä, jota ei ole olemassa. Näin ollen ensimmäisen vastauksen tulee olla odotetunlainen. Toisessakin vastaukseksi odotetaan oikeanlaista vastetta, kun on huomattu, että kyseessä on virheilmoitus.

```
module UDPModule2
{
        import from UDPModule1 all;

        template Message Msg_Base :=
        {
                header :=
                {
                        com := "",
                        add := ' 'O,
                        id := 0
                },
                len := omit,
                dat := omit,
                err := omit
        }

        template Message GET09 modifies Msg_Base :=
        {
                header :=
                {
                        com := "GET",
                        add := IP_DEST,
                        id  := 9
                }
        }

        template Message IND09 modifies Msg_Base :=
        {
                header :=
                {
                        com := "IND",
                        add := IP_DEST,
                        id := 9
                },
                len := 2,
                dat := ?
        }

        // Wrong id, ind->ack and error-message (unknown id := 101)
        //*********************************************
        template Message GET09_a modifies Msg_Base :=
        {
                header :=
                {
                        com := "GET",
                        add := IP_DEST,
                        id  := 16
                }
        }
```

```
template Message ACK09_a modifies Msg_Base :=
{
                header :=
                {
                                com := "ACK",
                                add := IP_DEST,
                                id := 16
                },
                err := 101
}

//*********************************************

// Too short packet.
template Raw Codec_Test_INV_1 := '0102030405'O;

// Unrecognized frame type.
template Raw Codec_Test_INV_2 := 'FF0203040506'O;

// GET frame too long.
template Raw Codec_Test_INV_3 := '01020304050607'O;

// SET frame too short.
template Raw Codec_Test_INV_4 := '020203040506'O;

// IND frame too short.
template Raw Codec_Test_INV_5 := '030203040506'O;

// ACK frame too short.
template Raw Codec_Test_INV_6 := '040203040506'O;

// ACK frame too long.
template Raw Codec_Test_INV_7 := '0402030405060708'O;

}
```

Esimerkki 2: UDPModule2-moduuli.

UDPModule2-moduulissa on Msg_Base:sta modifioidut myös vastauksissa käytettävät mallit. GET09:n kysymykseen odotetaan IND-tyyppistä vastausta, jonka dat-kentän pituus (len) on kaksi tavua. Itse dat-kentän tiedon sisältö ei ole oleellinen testattaessa, jonka vuoksi vastaukseksi voidaan hyväksyä mikä vaan vastaus ?-merkillä. Jos emme olisi edes varmoja tuleeko vastausta vai ei, voisimme käyttää *-merkkiä, jolloin alustettuun kenttään ei välttämättä tarvita lainkaan vastausta.

GET09_a:n kyselyssä vasteeksi odotetaan ACK-tyyppistä vastausta, sillä lähetetty kysely kohdistuu virheelliselle id-numerolle. Tämän vuoksi vastauksena saadaan virheellisestä id-numerosta (16) johtuva ACK-vaste, jossa väärästä virhetyypistä aiheutuvassa vasteessa err-kenttä saa arvon 101.

Kyseisessä esimerkissä molempiin odotettaviin vasteisiin verrattavat saadut vasteet ovat samanlaisia, jolloin testitapaukset hyväksytään oikeanlaisina. Mikäli emme olisi huomanneet, että teemme GET-kyselyn virheelliselle id-numerolle, niin olisimme

odottaneet vastausta IND-tyyppisenä. Tällöin odotettava vaste olisi eronnut saadusta vasteesta ja testitapaus olisi hylätty.

UDPModule2-moduulin lopussa on kelpaamattomille lähetys- ja vastaanottopakettien formaateille omat virhemallit. Virhemalleja käytetään, kun SUT-adapteri (SA) ei voi asianmukaisesti koodata saamiansa paketteja.

7.3.3 UDPModule3

UDPModule3-moduulissa on kuvattu testauksen dynaaminen osa. Moduuli koostuu kahdesta osasta, ensin on määrittelyosa ja lopussa kontrolliosa, jossa selvitellään testitapausten suorittamisjärjestystä. Samoin kuin UDPModule2-moduulissa, täytyy muiden aikaisempien moduulien määritykset ja alustukset tuoda näkyviksi import-komennolla.

UDPModule3-moduulissa ensimmäisenä olevassa switch case -tyyppisessä altstep DefaultAltstep -funktiossa annetaan lopulliset päätökset testitapausten oikeellisuudesta. Tämän jälkeen kaikki porttityypit määritetään inout-tyyppisiksi, jolloin portit voivat sekä lähettää että vastaanottaa viestejä. Sitten moduulissa nimetään ja alustetaan testauskomponentit, jotka myöhemmin preamble-funktiossa yhdistetään toisiin map-komennolla. Testauskomponenttien käyttöönotosta enemmän alaluvussa 8.4.2, jossa käsitellään testien suorittamista tarkemmin.

```
module UDPModule3
{

        import from UDPModule1 all;
        import from UDPModule2 all;

        altstep DefaultAltstep() runs on TestComponent
        {
                [] p.receive
                {
                        setverdict(fail);
                        stop;
                }
                [] p.getcall
                {
                        setverdict(fail);
                        stop;
                }
                [] p.getreply
                {
                        setverdict(fail);
                        stop;
```

```
                        }
                        [] p.catch
                        {
                                setverdict(fail);
                                stop;
                        }
                        [] T_GUARD.timeout
                        {
                                setverdict(inconc);
                                stop;
                        }
}

type port PortType mixed
{
                        inout all;
}

type component TestComponent
{
                        port PortType p;

                        const float T_GUARD_DEFAULT := 5.0;

                        timer T_GUARD := T_GUARD_DEFAULT;

                        var address sut_addr :=
                        {
                                host := PX_SUT_IP_ADDR,
                                portField := PX_SUT_PORT
                        };

                        var default compDefaultRef;
}

type component TestSystemInterface
{
                        port PortType tsiPort;
}


function preambleSetup() runs on TestComponent
{
                        map(mtc:p, system:tsiPort);
                        T_GUARD.start;
                        compDefaultRef := activate(DefaultAltstep());
}

function postambleRelease() runs on TestComponent
{
                        deactivate;
                        T_GUARD.stop;
                        unmap(mtc:p, system:tsiPort);
}

//Testi-caset

testcase TC_GET09() runs on TestComponent system TestSystemInterface
{
                        preambleSetup();

                        p.send(GET09) to sut_addr;
                        p.receive(IND09) from sut_addr;

                        setverdict(pass);

                        postambleRelease();
}
```

```
        testcase TC_GET09_a() runs on TestComponent system TestSystemInterface
        {
                preambleSetup();

                p.send(GET09_a) to sut_addr;
                p.receive(ACK09_a) from sut_addr;

                setverdict(pass);

                postambleRelease();
        }

        control
        {
                execute(TC_GET09());
                execute(TC_GET09_a());
        }
}
```

Esimerkki 3: UDPModule3-moduuli.

UDPModule3-moduulin lopussa on määritetty testitapaukset ja niiden sisäinen toiminta. Moduulin lopussa on kontrolliosa, jossa on määritelty testitapausten käsittelyjärjestys.

7.3.4 UDPModuleParameters

UDPModuleParameters -moduulissa on puolestaan alustettu moduuleissa käytettävät parametrit ja annettu niille arvot. Parametrimoduulin ansiosta syötettävien parametrien muutokset tarvitsee tehdä ainoastaan yhteen kohtaan. Näin voidaan vähentää aikaa parametrien ylläpidosta, toisin kuin jos jokainen kohta jouduttaisiin käymään erikseen käsin korjaamassa.

```
modulepar
{
        // IP address and port number of the SUT.
        charstring PX_SUT_IP_ADDR := "127.0.0.1";
        integer PX_SUT_PORT := 6510;

        // destination IP
        octetstring IP_DEST := '64646463'O;
}
```

Esimerkki 4: UDPModuleParameters-moduulissa tehtävät sijoitukset.

Moduulissa suoritetaan UDPModule1:ssä esiteltyjen parametrien arvojen sijoitus. PX_SUT_IP_ADDR- ja PX_SUT_PORT-parametreja käytetään TestComponent:ssa address-tyyppisen muuttujan arvoina. IP_DEST-parametrin avulla määrätään lähettävän kyselyn kohdeosoite. IP_DEST:ä käytetään myös odotetun vastauksen rakentamisessa tarvittavassa osoitteessa.

7.4 Testaustoimenpiteet

Käyttöliittymä säätelee testerin toimintaa, sillä se tarjoaa käyttäjälleen työkalut testaamista varten. Käyttöliittymän avulla testaajan on mahdollista valita testejä, hallita testien ajamista ja standardoitujen testilokien tuottamista. Mutta ennen kuin päästään suorittamaan testausta, on testausjärjestelmä alustettava.

7.4.1 Testauksen alustaminen

Jotta testaus voitaisiin aloittaa, on aluksi testitiedostoissa kuvatut abstraktit testiaineistot (engl. *abstract test suite*, ATS) ladattava testerin tietokantaan. Näin testiaineistot ovat tietokannassa suoritusvalmiina ajonaikaista suorittamista varten, sillä lataamisen aikana oikeanlaiset TTCN-3 -moduulit esikäännetään ja tallennetaan järjestelmän tietokantaan. Samalla skriptit myös antavat tarkat arvot moduuliparametreille, jotka on määritelty parametritiedostossa (UDPModuleParameters).

Testiaineiston tietokantaan lataamisen jälkeen on varmistettava, että testattava toteutus (engl. *implementation under testing*, IUT) on käynnistetty. Tämän jälkeen on käynnistettävä SA-adapteri, jonka välityksellä testauskomponentti (TestComponent) on yhteydessä testattavaan järjestelmään. Jokaiselle testattavalle järjestelmälle on rakennettava oma SA-adapteri, jonka ansiosta testeri voidaan pitää samanlaisena testattavasta järjestelmästä riippumatta.

Kun SA-adapteri on käynnistetty, on sen paikkatiedot vielä tallennettava järjestelmän tietokantaan. T3RTS liitetään SA-adapteriin TRI-rajapinnan avustuksella. Tämän jälkeen, kun tarvittavat komponentit on määritetty, voidaan aloittaa testitapausten suorittaminen.

7.4.2 Testin suorittaminen

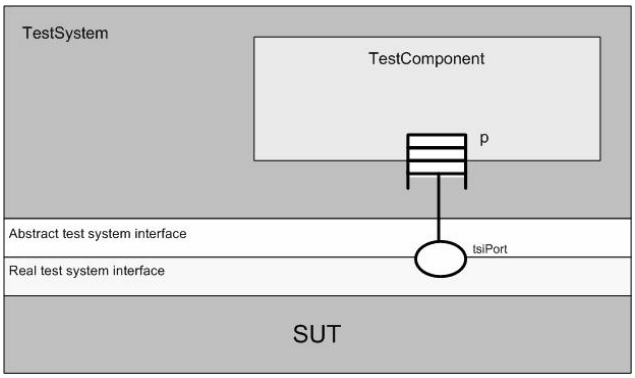
Testin suorittaminen alkaa UDPModule3:n kontrolliosasta, josta käy ilmi testitapausten suorittamisjärjestys. Kontrolliosassa kutsutaan testitapauksia halutussa järjestyksessä esimerkin 5 mukaisesti. Kyseisessä esimerkkitapauksessa on kaksi suoritettavaa testitapausta TC_GET09 sekä TC_GET09_a.

```
control
{
                    execute(TC_GET09());
                    execute(TC_GET09_a());
}
```

Esimerkki 5: UDPModule3:n kontrolliosa.

Kontrolliosasta kutsutussa testitapauksessa määritetään, missä testikomponentissa ja -järjestelmässä testitapaus suoritetaan. Itse testitapauksessa ensimmäisenä käydään preambleSetup-funktiossa yhdistämässä päätestikomponentin (engl. *main test component*, mtc) portti (p) ja testausjärjestelmän portti (tsiPort) map-komennolla. Yhdistämisessä portit yhdistetään kuvan 31 osoittamalla tavalla. Alustusasetuksissa käynnistetään myös testitapauksen ajastin mahdollista ajanottoa varten.

Testijärjestelmän esimerkkitapauksessa on vain yksi testikomponentti TestComponent, joka toimii näin päätestikomponenttina. Jokaisessa testijärjestelmässä on vain yksi päätestikomponentti, jolla voi olla rinnakkaiskomponentteja (engl. *parallel test component*, ptc). Päätestikomponentti vastaa testitapauksen ajon lopettamisesta, viimeistään silloin kuin se tuhotaan. Päätestikomponentti on myös vastuussa rinnakkaiskomponenttien luomisesta ja valvomisesta sekä testitulosten laskennasta ja määräämisestä. Testikomponentti ja rajapinta testattavaan järjestelmään kuuluvat testijärjestelmään (TestSystem). Tämän testijärjestelmän tehtävänä on testitapausten ajaminen, jolloin kommunikointi testattavaan järjestelmään tapahtuu yhdistettyjen porttien välityksellä.



Kuva 31: Testikomponentin ja testattavan järjestelmän yhdistäminen.

Testattava järjestelmä ja testijärjestelmä muodostuvat kahdesta eri tasosta muodostuvasta kommunikointirajapinnasta, abstraktista ja varsinaisesta rajapinnasta. Näin ollen

testijärjestelmässä käytettäviä komponenttien portteja voidaan kuvata abstraktia rajapintaa käyttäen, vaikka kommunikointi tapahtuukin varsinaisella rajapinnalla, joka on TTCN-3:n vaikutusalueen ulkopuolella.

```
testcase TC_GET09() runs on TestComponent system TestSystemInterface
{
                    preambleSetup();

                    p.send(GET09) to sut_addr;
                    p.receive(IND09) from sut_addr;

                    setverdict(pass);

                    postambleRelease();
}
```

Esimerkki 6: Testitapauksen TC_GET09:n sisältö.

Kun testikomponentit on yhdistetty toisiinsa ja muut ennakkovalmistelut tehty, voidaan p.send(GET09)-funktion avulla lähettää kysely sut_addr:lle, joka on määritetty TestComponent:ssa. Lähetettävä kysely GET09 rakentuu UDPModule2:ssa ja UDPModule1:ssä tehtyjen määritysten mukaan. GET09 on modifioitu Msg_Base-templaatista ottamalla käyttöön vain header:n com-, add- sekä id-kentät, joihin on sijoitettu komento (GET), parametrina tuleva IP_DEST (100.100.100.103) sekä id-numero (9). Msg_Base-templaatin valinnaisia kenttiä ei tarvita GET09-templaatissa ja ne jätetään näin huomioimatta.

Kun viesti on lähetetty, jäädään odottamaan testattavalta järjestelmältä tulevaa vastetta. Odotettavan vasteen funktiokutsu on toteutettu p.receive(IND09):n avulla. Odotettavan vasteen arvot on rakennettu IND09-templaatissa, joka on modifioitu myös Msg_Base:sta. Header:n odotettavina arvoina ovat IND, parametrina tuleva IP_DEST (100.100.100.103) ja id-numero (9). Näiden lisäksi vasteena tulevan datapaketin oletetaan sisältävän header-otsikon lisäksi kahden tavun mittaisen datan. Len kertoo odotettavan datan pituuden, kun taas dat on odotettavan datan sisältö. Kun IND09:n dat-kentässä käytetään ?-merkkiä, hyväksyy se datavasteena kaiken saapuvan tiedon.

Kun vaste on saatu, verrataan sitä odotettuun vasteeseen. Mikäli vaste oli odotetunlainen, annetaan testitapaukselle tuomio hyväksytty (pass). Jos taas vaste ei täytä odotetun vasteen mukaisia arvoja, tuomioksi tulee hylätty (fail). Mikäli taas ajastin ylittää sille varatun ajan, on tuomiona tulokseton (inconc). Ei-hyväksyttyjen vasteiden käsittely on toteutettu

UDPModule3:n DefaultAltstep-funktiossa, jossa tuomiot määrätään. Jokaiselle testitapaukselle annetaan erikseen tuomio, joista sitten kootaan yhteinen tuomio. Yksikin ei-hyväksytty tuomio aiheuttaa ei-hyväksytyn kokonaistuomion.

7.5 Case-tapauksen pohtiminen

LR-WPAN -laitteen tietoihin käsiksi pääseminen on erityisen vaikeata. Tämä tekee myös testauksesta hankalan, koska laitteen käyttäytymistä on vaikeata tietää. Case-tapaukseen valittu testauskieli, TTCN-3, ja OpenTTCN Oy:n OpenTTCN Tester for TTCN-3 -testaustyökalu, oli tarkoitettu juuri tämänlaisen tietoliikennetestaukseen mahdollistaen järjestelmän testaamisen yli monimuotoisten kommunikointiporttien.

TTCN-3 -standardiin perustuvan testaustyökalun soveltuvuus testaukseen oli mielestäni hyvä, sillä se mahdollisti yhdenmukaisuustestauksen. Standardinmukaisuus mahdollisti kehitysvaihetestauksen lisäksi sen, että kyseisellä testaustyökalulla on mahdollisuus suorittaa myös lopullinen yhdenmukaisuustestaus tuotteelle tietoliikenneprotokollien standardiin perustuen. Näin ollen se mahdollistaa lopulliselle tuotteelle myös tyyppihyväksynnän suorittamisen. Omalla kohdalla tehtävänäni ei ollut suorittaa yhdenmukaisuustestausta, sillä laite oli vasta kehitysvaiheessa. Enemmänkin tehtävänäni oli testausjärjestelmän rakentaminen ja malliesimerkkien tekeminen myöhemmin testausasioista vastaavalle henkilölle.

Testaustyökalu automatisoi testien suorittamisen ennaltalaadittujen testitapausten pohjalta varmistaen näin, että uudelleentestattaessa testitapaukset ja syötteet olivat samoja. Testaustyökalu mahdollisti myös odotettujen vasteiden automaattisen vertailun saatuihin vasteisiin, joka työnä on erityisen rutiininomaista ja virhealtista ihmisen tekemänä. Testaustyökalun tekemän automaattisen testaamisen ansiosta regressiotestatessa säästyi resursseja muihin tärkeämpiin tehtäviin kehitystyössä. Mutta ennen kuin ajallista hyötyä regressiotestauksesta saatiin, vaativat ennakkovalmistelut toisaalta huomattavan paljon enemmän valmisteluja ja aikaa kuin manuaalisessa testauksessa. Case-tapauksessamme saimme osan adapterista jo valmiiksi tehtynä, mutta silti ennakkovalmistelut veivät aikaa suhteellisen paljon.

Yksi suuri ja tärkeä etu kyseissä testausjärjestelmässä ja -teknologian valinnassa oli myös se, että järjestelmään ei tarvinnut tehdä muutoksia testausta ajatellen. Näin ollen järjestelmän laitteet pystyttiin testaamaan normaalissa ympäristössä täysin ilman erillisen kalliin testerin hankintaa tai muutoksien tekemistä järjestelmään. Manuaalisesti ilman testaustyökalua joudutaan laitteissa mahdollisesti käyttämään näyttöä, johon tehdään välitulostuksia laitteen tilan ja elämän selvittämiseksi. Välitulostusten lisäksi ylimääräisten viiveiden lisääminen järjestelmään aiheuttaa myös muutoksia varsinaiseen järjestelmään. Testauksen jälkeen nämä laitteeseen normaalisti kuulumattomat muutokset joudutaan poistamaan, eikä näin ollen voida täysin olla varmoja sen toiminnasta muutosten jälkeen.

Vaikka tällä hetkellä keskitytään testauksessa pelkästään laitteiden testaamiseen, mahdollistaa testausympäristö myös kokonaisen verkon testaamisen. Tulevaisuudessa on näin ollen mahdollista samalla selvittää myös verkon laitteiden toimivuus osana verkkoa. Myös pienet testaukset muutaman laitteen verkkoa koskien vaikuttivat positiivisilta, sillä näin voitiin keskittyä testaamaan verkkoa erilaisilla laitesijoitteluilla.

Automatisoidussa testauksessa testitapaukset voidaan toistaa samanlaisina myöhemminkin. Manuaalisessa testauksessa käsin syötettävien syötteiden yhdenmukaisuudesta aikaisemmin suoritettujen testien kanssa ei voida olla täysin varmoja. Koska automatisoitu testaaminen on helppo ja nopeampi tapa testata kuin manuaalinen, suoritetaan sitä useammin kuin manuaalista. Testausmäärän kasvamisen avulla voidaan olla varmempia järjestelmän toimivuudesta ja laadusta. Manuaalisen testauksen etuna Fewster [11] näkee sen, että ihminen osaa testauksen aikana miettiä, mitä virheitä koodiin tehty muutos voisi aiheuttaa, pystyen muokkaamaan testausta tämän pohjalta. Automatisoitu testaus keskittyy vain sille määrättyjen testien orjalliseen suorittamiseen. Manuaalisen testauksen avulla päästää nopeammin testaamaan joitain tiettyjä yksittäisiä kohtia.

Muutenkin testattaessa usein toistettavia kohteita havaitsin automatisoidun regressiotestauksen edut verrattuna manuaaliseen vastaavaan. Automatisoidun jo kertaalleen suoritetun regressiotestauksen toistaminen oli huomattavasti manuaalista nopeampi. Erityisesti kuormitustestaamisessa automatisointi toi huomattavia etuja, sillä työkalun avulla pystyttiin kuormittamaan sekä laitteita että verkkoa. Etua lisäsi myös se,

että testitapauksia voitiin toistaa samanlaisina kerta toisensa perään sisältäen näin myös automaattisen tulosten vertailun.

Standardi testausarkkitehtuuri ja -kieli yhdessä neutraalin teknologian kanssa eivät sido testausta liiaksi tiettyyn ympäristöön tai ohjelmointikieleen. Kun vielä TTCN-3 tukee modulaarisuutta testimoduulien rakentamisessa, on testisarjojen ylläpito myös hallittavissa. Case-tapauksen esimerkeissä eteen tuli kuitenkin ylläpidollisiakin ongelmia. Ensinnäkin esimerkeissä oleva testitapausten rakentaminen ei välttämättä ole oikeanlainen, sillä laajentamalla testitapauksia testimallien määrä kasvaa helposti liian suureksi, jonka vuoksi saattaisi olla parempi tuoda parametrina enemmän tietoa kuin tällä hetkellä tapahtuu. Miettimistä vaatii myös testitapausten nimeäminen sekä testitulosten tallentaminen.

Valitun TTCN-3 -testauskielen haasteena voidaan pitää melko vaikeasti hahmoteltavaa arkkitehtuuria ja erityisesti sen yksityiskohtia. Alussa ongelmia tuotti standardiin perustuvan testerin toiminnan selvittäminen. Yhdessä testerin toiminnan selvittämisen kanssa oli arkkitehtuurin hahmottaminen, joka vaati myös oman aikansa. Ainahan uuden kielen opiskelu vaatii oman työnsä, mutta TTCN-3:n kohdalla se ei kuitenkaan tuottanut liiallisia ponnisteluita, sillä TTCN-3 -kieli on kuitenkin melko lähellä perinteisiä ohjelmointikieliä ja muutenkin melko helposti ymmärrettävää.

Kokonaisuudessaan case-tapauksessa esiinnousseet asiat kuitenkin herättivät positiivisia mielikuvia, vaikka ongelmitta ei selvitty. Erityisesti olen huomannut, kuinka paljon testerin käytöllä voidaan helpottaa laitteen testausta. Yhdistelemällä muihin apuvälineisiin testerin käyttö, päästään käsiksi helposti myös sellaisiin tietoihin mihin muuten ei päästäisi. Tietenkin testerin ja testauskielen arvioiminen olisi helpompaa pidemmän aikavälin päästä, kun kokemuksia niistä olisi saatu enemmän. Näin ollen testauksen kehittymisen jatkotutkiminen olisi mielenkiintoista jonkin ajan kuluttua, kun enemmän tietoa testausjärjestelmästä, -kielestä ja -työkalusta on saatu. Tällä hetkellä työkalun ja kielen valinta kuitenkin vaikuttavat hyviltä, sillä testauksen jatkokehittäminen näyttää hyvältä ajatellen valittuja teknologioita. Myös testitapausten ylläpidettävyys herättää kiinnostusta jatkoa ajatellen. Erityisen mielenkiinnon kohteena on, kuinka hyvin omat

testitapaussuunnittelut toimivat testauksen edetessä. Tai kuinka testitapauksia tulisi kehittää, jotta niistä tulisi entistä paremmin ylläpidettäviä.

8 Yhteenveto

Kokonaisvaltaisella testaamisella päästään parhaaseen lopputulokseen järjestelmän oikeellisuutta ja laatua tutkittaessa. Tämän vuoksi testausta ei saisi kohdistaa vain tiettyyn vaiheeseen, vaan sitä tulisi tehdä läpi koko järjestelmän elinkaaren aina järjestelmän alkuvaiheista käyttöönottoon ja ylläpitoon saakka. Jokaisella eri vaiheella on oma tehtävänsä testauksessa. Parhaaseen lopputulokseen pääsemisessä on hyvä käyttää virheiden estämistä ja olemassa olevien virheiden löytämistä eri testausvaiheissa tukemaan toisiaan kokonaisvaltaisessa testaamisessa.

Tutkielmassani olen keskittynyt automatisoidun regressiotestauksen merkitykseen sulautetun järjestelmän kehitystyössä. Aluksi olen käynyt yleisemmällä tasolla läpi testausta ja ohjelmistotuotantoa, jotka olen sitten liittänyt sulautetun järjestelmän vastaaviin vaiheisiin. Case-tapauksessani olen tutkinut tarkemmin automatisoidun regressiotestauksen soveltuvuutta sulautettuun järjestelmään lyhyen kantaman langattoman LR-WPAN -laitteen ja TTCN-3 -testerin avulla.

Regressiotestaus on yleensä integrointi- ja järjestelmävaiheessa suoritettavaa testausta, jossa löydettävien virheiden korjaamishinnat nousevat jo melko korkeiksi. Tästä huolimatta regressiotestauksella on tärkeä osa sulautetuille järjestelmille ominaisessa inkrementaalisessa kehitystyössä ja ennen kaikkea ylläpidossa. Inkrementaalisessa kehitystyössä pienien lisäyksien jälkeen voidaan näin regressiotestata tarvittavat kohteet. Yleensä paras hyöty regressiotestauksesta saadaan, kun automatisoidaan usein toistettavat testausaktiviteetit, testitapausten suorittaminen ja saatujen vasteiden vertailu odotettuihin vasteisiin. Automatisoimalla erityisesti loppuvaiheessa suoritettavat regressiotestaukset voidaan varmistaa myös laitteen kattava testaaminen, sillä usein loppuvaiheessa suoritettava testaus jää pois ajanpuutteen takia.

Sulautetun järjestelmän voi testata automatisoidun regressiotestauksen ansiosta jokaisen pienen muutoksen tai lisäyksen jälkeen. Testauksen tarkoituksena on varmistaa, että järjestelmä vastaa vaatimuksia jokaisen muutoksia tai lisäyksiä sisältävän inkrementin jälkeen. Automatisoitu regressiotestaus mahdollistaa myös, että testaus voidaan suorittaa samanlaisena, samoilla syötteillä kuin aiemmilla kerroilla, kun taas manuaalisesti

testattaessa voi olla vaikeata todistaa testikertojen yhdenmukaisuus. Monesti juuri loppuvaiheessa testauksia saatetaan supistaa tai jättää kokonaan pois, mikäli aikataulun kanssa on ongelmia. Automatisoitu regressiotestaus nopeuttaa testaamista vapauttaen työntekijöitä myös muihin tehtäviin. Automatisoinnin avulla voidaan näin järjestelmää testata useammin, jolloin testauksesta saadaan myös kattavampi ja se voidaan kohdistaa myös tarkemmin testausta enemmän vaativiin kohtiin. Huomattavana säästönä automatisoinnissa on myös se, että testien suorittamisen voi keskittää myös työajan ulkopuolelle. Automatisoidun regressiotestauksen edut tulevat esille vasta, kun testausta toistetaan useamman kerran. Tämän vuoksi on tärkeätä pohtia eri testauksen sopivuutta automatisoitavuuden näkökulmasta.

Vaikka regressiotestaus tuleekin järjestelmän kehittämisessä melko loppuvaiheessa, ei testauksen suunnittelua, alustamista ja toteuttamista kannata kuitenkaan jättää viime hetkeen. Testauksen suunnittelu ja suunnittelun toteutus on hyvä aloittaa yhdessä järjestelmän kehittämisen rinnalla V-mallin mukaisesti, jolloin se myös tukee järjestelmän kehittäjien työtä. Tämän vuoksi kehittäjien ja testaajien odotetaankin toimivan tiiviissä yhteistyössä läpi koko järjestelmän kehittämisen ajan.

Testauksessa, kuten muissakin asioissa, eteen tulevana ongelmana on koulutuksen järjestäminen henkilökunnalle. On tarkkaan harkittava testauksen automatisoinnista saatava hyöty ja siihen menevät kustannukset, jotta voidaan olla varmoja saatavista eduista. Tärkeätä on myös muistaa, että järjestelmämuutokset vaikuttavat testitapauksiin, joiden ylläpidon suunnittelulle kannattaa varata riittävästi aikaa. Ilman hyvin suunniteltuja testitapauksia ja testausjärjestelmää menetetään automatisoinnista saatava etu nopeasti, mikäli testejä ei voida uudelleenkäyttää.

Case-tapaukseni loi hyvän kuvan automatisoidun regressiotestauksen hyödyistä ja haitoista sulautettujen järjestelmien kehitystyössä. Suurimmaksi hyödyksi nousi omassa tapauksessani automatisoinnin mahdollistama kuormitustestaus, jonka käyttämisen mahdollisuus avasi uusia mahdollisuuksia testausnäkökulmassa. Myös muita kysymyksiä ja mahdollisuuksia heräsi tätä tutkimusta tehdessäni. Ensinnäkin mielenkiintoa herätti regressiotestauksen koko elinkaaren aikaiset erilaiset käyttömahdollisuudet, ja kuinka

hyvin niiden avulla voidaan tukea järjestelmän kehittämistä läpi elinkaaren. Toinen kiinnostava aihe olisi regressiotestauksen ylläpitoon liittyvien ongelmien miettiminen. Kuinka testitapaukset tulisi suunnitella, jotta niiden ylläpito olisi helppoa ja hallittua koko elinkaaren ajan. Myös yleisen testausnäkökulman selvittäminen sulautettujen järjestelmien kehitystyössä herättää kiinnostusta tulevaisuutta ajatellen. Siitä kuinka testaaminen on mukana tukemassa järjestelmän kehittämistä aina määrittelystä ja suunnittelusta käyttöönottoon ja ylläpitoon.

Lähteet

[1] Armour Phillip G., *The Business of Software: The Unconscious Art of Software Testing*, Communications of the ACM, No.1, Vol. 48 (January 2005), s. 15-18.

[2] Beizer Boris, ”Software testing techniques”, International Computer Press, Boston, 1990.

[3] Binder Robert V., ”Testing object-oriented systems: models, patterns and tools”, Addison-Wesley, Reading, Massachusetts, 1999.

[4] Black Rex, ”Managing the Testing Process: Practical Tools and Techniques for Managing Hardware and Software Testing”, Wiley Publishing, Inc., New York, 2002.

[5] Boehm Barry W., *Guidelines for Verifying and Validating Software Requirements and Design Specifications*, Euro IFIP 79, P. A. Samet (editor), North-Holland Publishing Company, IFIP, 1979.

[6] Broekman Bart ja Notenboom Edwin, ”Testing Embedded Software”, Addison-Wesley, London, 2003.

[7] Burnstein Ilene, ”Practical Software Testing”, Springer-Verlag, New York, 2003.

[8] Dustin Elfriede, Rashka Jeff ja Paul John, ”Automated Software Testing: introduction, management and performance”, Addison-Wesley, Boston, 1999.

[9] ETSI ES 201 873-1 v.2.1.1, Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; Part 1: TTCN-3 Core Notation, February 2003.

[10] ETSI ES 201 873-5 v.1.1.1, Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; Part 5: TTCN-3 Runtime Interface (TRI), February 2003.

[11] Fewster Mark ja Graham Dorothy, ”Software Test Automation: Effective use of test execution tools”, Addison-Wesley, Harlow, 1999.

[12] Granlund Kaj, ”Tietoliikenne”, Teknolit, Jyväskylä, 1999.

[13] Gutiérrez José A., Callaway Jr. Edgar H. ja Barrett Jr. Raymond L., ”Low-Rate Wireless Personal Area Networks: Enabling Wireless Sensors with IEEE 802.15.4”, IEEE, New York, 2003.

[14] Haikala Ilkka ja Märijärvi Jukka, ”Ohjelmistotuotanto”, 8. uudistettu painos, Satku, Helsinki, 2002.

[15] Harju Hannu ja Koskela Mika, ”Kustannustehokas ohjelmiston luotettavuuden suunnittelu ja arviointi”, VTT Tiedotteita – Research Notes 2193, VTT, Espoo, 2003.

[16] Honka Hannu, ”A simulation-based approach to testing embedded software”, Technical Research Centre of Finland, VTT Publications 124, Espoo, 1992.

[17] Hutcheson Marnie, *The evolution of an automated software test system*, kirjassa ”Software Test Automation: Effective use of test execution tools”, Addison-Wesley, Harlow, s. 326-341, 1999.

[18] Myers Glenford, ”The Art of Software Testing”, John Wiley & Sonc, Inc., New York, 1979.

[19] Mäkelä Marko, *Rinnakkaisuus hallintaan – Formaalit menetelmät mutkikkaiden järjestelmien suunnittelun apuna*, Prosessori, ES marraskuu 2001, s. 68-71.

[20] Kaner Cem, Bach James ja Pettichord Bret, ”Lessons learned in software testing: a context-driven approach”, John Wiley & Sons, Inc., New York, 2002.

[21] Li Yuejian ja Wahl Nancy J., *An Overview of Regression Testing*, ACM SIGSOFT Software Engineering Notes, No.1, Vol. 24 (January 1999), s. 69-73.

[22] Orso Alessandro, Shi Nanjuan ja Harrold Mary Jean, *Scaling Regression Testing to Large Software Systems*, Proceedings of the 12th ACM SIGSOFT twelfth international symposium on Foundations of software engineering, October 2004. Saatavilla PDF-muodossa <URL: http://delivery.acm.org/10.1145/1030000/1029928/p241-orso.pdf?key1=1029928&key2=8903566011&coll=GUIDE&dl=GUIDE&CFID=37177467&CFTOKEN=77531189 >.

[23] Pinkster Iris, *Regression testing at ABN AMRO Bank Development International*, kirjassa ”Software Test Automation: Effective use of test execution tools”, Addison-Wesley, Harlow, s. 465-473, 1999.

[24] Pressman Roger S., ”Software Engineering: A Practitioner’s Approach”, 5. painos, European Adaptation, Adapted by Darrel Ince, McGraw-Hill, London, 2000.

[25] Puro Panu ja Puro Vesa-Matti, *Protokollatestaus uudelle vuosituhannelle*, Prosessori, ES marraskuu 1999, s. 81-82.

[26] Rothermel Gregg ja Elbaum Sebastian, *Putting Your Best Tests Forward*, IEEE Software, September/October 2003, s. 74-77.

[27] Royce Winston W., *Managing the Development of Large Software Systems*, Proc. IEEE WESCON, August 1970, s. 1-9.

[28] Schulz Stephan, Buchenrieder Klaus J. ja Rozenblit Jerzy W., *Multilevel Testing of Desing Verification of Embedded Systems*, Design & Test of Computers, IEEE, vol. 19 (2002), s. 60-69.

[29] Sneed Harry M., *Reverse Engineering of Test Cases for Selective Regression Testing*, Proceedings of the Eight European Conference on Software Maintenance and Reengineering, 2004. Saatavilla PDF-muodossa <URL: http://ieeexplore.ieee.org/iel5/9013/28613/01281407.pdf?tp=&arnumber=1281407&isnumber=28613&arSt=69&ared=74&arAuthor=Sneed%2C+H.M.%3B >.

[30] Sommerville Ian, ”Software Engineering”, 5. painos, Addison-Wesley, Harlow, 1998.

[31] Spillner Andreas, *The W-Model – Strengthening the Bond Between Development and Test*, STAReast 2002, Orlando, Florida , USA, 15.-17. May 2002. Saatavilla PDF-muodossa <URL: http://www.stickyminds.com/sitewide.asp?ObjectId=3572&Function=DETAILBROWSE&ObjectType=ART. >.

[32] Tikkakoski Merja, ”LR-WPAN sensoriverkkotekniikkana”, Pro gradu, Jyväskylän yliopisto, Tietotekniikan laitos, 2003.

[33] Tsai W.T., Yu L., Zhu F. ja Paul R., *Rapid Verification of Embedded Systems Using Patterns*, Computer Software and Applications Conference, 2003. COMPSAC 2003. Proceedings. 27th Annual International, 3-6 Nov. 2003, s. 466-471.

[34] Uotila Pekka, ”Tietoliikenteen tekniikka: verkot ja protokollat”, Satku, Espoo, 1997.

[35] Von Mayrhauser Anneliese, *Maintenance and Evolution of Software Products*, Advances in Computers, vol. 39 (1994), s. 1-49.

[36] Xu Lihua, Dias Marcio ja Richardson Debra, *Generating Regression Tests via Model Checking*, Computer Software and Applications Conference, 2004. COMPSAC 2004. Proceedings of the 28th Annual International, 28-30 Sept. 2004, s 336 – 341.

[37] Yamaura Tsuneo, *How to Design Practical Test Cases*, IEEE Software, vol. 15 (1998), s. 30-36.

[38] Younessi Houman, ”Object-Oriented Defect Management of Software”, Prentice Hall PTR, Upper Saddle River, N. J., 2002.

Liitteet

Liite 1. UDPModule1

```
module UDPModule1
{
		// Address type used for UDP packets.
		type record address
		{
				charstring host,
				integer portField
		}

		modulepar
		{
				// IP address and port number of the SUT.
				charstring PX_SUT_IP_ADDR;
				integer PX_SUT_PORT;

				// destination IP
				octetstring IP_DEST;
		}

		//type record Message
		type charstring Command ("ACK", "GET", "IND", "SET")
		type octetstring Address length(4)
		type integer Id (0..12)
		type integer Len (0..32)
		type octetstring Dat
		type integer Error (0..110)

		// For sending invalid packets and for receiving packets of invalid
		// format that cannot be properly decoded by the SUT adapter.
		type octetstring Raw;

		//*************************************************

		type record Msg_header
		{
				Command		com,
				Address		add,
				Id		id
		}

		type record Message
		{
				Msg_header	header,
				Len		len		optional,
				Dat		dat		optional,
				Error		err		optional
		}

		//*************************************************
		// ID_OWN_IP
		//*************************************************
		type octetstring octet1 length(1)
		type octetstring octet2 length(1)
		type octetstring octet3 length(1)
		type octetstring octet4 length(1)

		type record IP
		{
				octet1		oct1,
				octet2		oct2,
				octet3		oct3,
				octet4		oct4
```

```
}

type record Message_OwnIP
{
        Msg_header      header,
        Len             len,
        IP              ownip
}

//*************************************************
// ID_DEVICE_NAME
//*************************************************
type octetstring Device_Name length(1..21)

type record Message_Dev_Name
{
        Msg_header      header,
        Len             len,
        Device_Name     dev_name
}

//*************************************************
// ID_ROLE
//*************************************************
type octetstring Role length(1)

type record Message_Role
{
        Msg_header      header,
        Len             len,
        Role            role
}

//*************************************************
// ID_IP_DATA
//*************************************************

type octetstring versio_ihl length(1)
type octetstring type_of_service length(1)
type octetstring time_to_live length(1)
type octetstring protocol length(1)

type record IP_Data
{
        versio_ihl      vrs_ihl,
        type_of_service tos,
        time_to_live    ttl,
        protocol        protoc,
        IP              source_address,         //look id_own_ip
        IP              subnet_mask,            //look id_own_ip
        IP              default_gateway         //look id_own_ip
}

type record Message_IP_Data
{
        Msg_header      header,
        Len             len,
        IP_Data         ipdata
}
}
```

Liite 2. UDPModule2

```
module UDPModule2
{
		import from UDPModule2 all;

		template Message Msg_Base :=
		{
				header :=
				{
						com := "",
						add := ' 'O,
						id := 0
				},
				len := omit,
				dat := omit,
				err := omit
		}

		//*************************************************
		//		GET GET GET
		//*************************************************

		//		GET 09
		//*************************************************
		template Message GET09 modifies Msg_Base :=
		{
				header :=
				{
						com := "GET",
						add := IP_DEST,
						id  := 9
				}
		}

		template Message IND09 modifies Msg_Base :=
		{
				header :=
				{
						com := "IND",
						add := IP_DEST,
						id := 9
				},
				len := 2,
				dat := ?
		}

		// Wrong id, ind->ack and error-message (unknown id := 101)
		//*************************************************
		template Message GET09_a modifies Msg_Base :=
		{
				header :=
				{
						com := "GET",
						add := IP_DEST,
						id  := 16
				}
		}

		template Message ACK09_a modifies Msg_Base :=
		{
				header :=
				{
						com := "ACK",
						add := IP_DEST,
						id := 16
```

```
                    },
                    err := 101
}

//*********************************************
template Message GET10 modifies Msg_Base :=
{
                    header :=
                    {
                                        com := "GET",
                                        add := '64646464'O,
                                        id  := 10
                    }
}

template Message IND10 modifies Msg_Base :=
{
                    header :=
                    {
                                        com := "IND",
                                        add := '64646464'O,
                                        id := 10
                    },
                    len := 2,
                    dat := ?
}

//*********************************************
//                                  SET SET SET
//*********************************************

template Message_OwnIP Msg_Ownip :=
{
                    header :=
                    {
                                        com := "",
                                        add := ' 'O,
                                        id := 0
                    },
                    len := 0,
                    ownip :=
                    {
                                        oct1 := ' 'O,
                                        oct2 := ' 'O,
                                        oct3 := ' 'O,
                                        oct4 := ' 'O
                    }
}

//*********************************************
template Message_OwnIP SET00 modifies Msg_Ownip :=
{
                    header :=
                    {
                                        com := "SET",
                                        add := IP_DEST,
                                        id := 0
                    },
                    len := 4,
                    ownip :=
                    {
                                        oct1 := '64'O,
                                        oct2 := '64'O,
                                        oct3 := '64'O,
                                        oct4 := '67'O
                    }
}

template Message ACK00 modifies Msg_Base :=
```

```
{
		header :=
		{
			com := "ACK",
			add := IP_DEST,
			id := 0
		},
		err := 0
}

//***********************************************
template Message_Dev_Name Msg_DevName :=
{
		header :=
		{
			com := "",
			add := ' 'O,
			id := 0
		},
		len := 0,
		dev_name := ' 'O
}

template Message_Dev_Name SET01 modifies Msg_DevName :=
{
		header :=
		{
			com := "SET",
			add := IP_DEST,
			id := 1
		},
		len := 10,
		dev_name := '4C616974654C61697465'O		//laitelaite

}

template Message ACK01 modifies Msg_Base :=
{
		header :=
		{
			com := "ACK",
			add := IP_DEST,
			id := 1
		},
		err := 0
}

//***********************************************
template Message_Role Msg_Role :=
{
		header :=
		{
			com := "",
			add := ' 'O,
			id := 0
		},
		len := 0,
		role := ' 'O
}

template Message_Role SET02 modifies Msg_Role :=
{
		header :=
		{
			com := "SET",
			add := IP_DEST,
			id := 2
		},
		len := 1,
```

```
                role := '00'O
}

template Message ACK02 modifies Msg_Base :=
{
                header :=
                {
                                com := "ACK",
                                add := IP_DEST,
                                id := 2
                },
                err := 0
}

//*********************************************
template Message_IP_Data IP_Data_Base :=
{
                header :=
                {
                                com := "",
                                add := ' 'O,
                                id := 0
                },
                len := 0,
                ipdata :=
                {
                                vrs_ihl := ' 'O,
                                tos := '00'O,
                                ttl := ' 'O,
                                protoc := ' 'O,
                                source_address :=
                                {
                                                oct1 := ' 'O,
                                                oct2 := ' 'O,
                                                oct3 := ' 'O,
                                                oct4 := ' 'O
                                },
                                subnet_mask :=
                                {
                                                oct1 := ' 'O,
                                                oct2 := ' 'O,
                                                oct3 := ' 'O,
                                                oct4 := ' 'O
                                },
                                default_gateway :=
                                {
                                                oct1 := ' 'O,
                                                oct2 := ' 'O,
                                                oct3 := ' 'O,
                                                oct4 := ' 'O
                                }
                }
}

//*********************************************
template Message_IP_Data SET03 modifies IP_Data_Base :=
{
                header :=
                {
                                com := "SET",
                                add := IP_DEST,
                                id := 3
                },
                len := 16,
                ipdata :=
                {
                                vrs_ihl := '44'O,
                                tos := '00'O,
                                ttl := '0A'O,           //10
```

```
                                        protoc := '11'O,                        //17
                                        source_address :=
                                        {
                                                oct1 := '64'O,
                                                oct2 := '64'O,
                                                oct3 := '64'O,
                                                oct4 := '67'O
                                        },
                                        subnet_mask :=
                                        {
                                                oct1 := 'FF'O,
                                                oct2 := 'FF'O,
                                                oct3 := 'FF'O,
                                                oct4 := 'E0'O
                                        },
                                        default_gateway :=
                                        {
                                                oct1 := '64'O,
                                                oct2 := '64'O,
                                                oct3 := '64'O,
                                                oct4 := '64'O
                                        }
                        }
}

template Message ACK03 modifies Msg_Base :=
{
                        header :=
                        {
                                        com := "ACK",
                                        add := IP_DEST,
                                        id := 3
                        },
                        err := 110
}

//***********************************************

// Too short packet.
template Raw Codec_Test_INV_1 := '0102030405'O;

// Unrecognized frame type.
template Raw Codec_Test_INV_2 := 'FF0203040506'O;

// GET frame too long.
template Raw Codec_Test_INV_3 := '01020304050607'O;

// SET frame too short.
template Raw Codec_Test_INV_4 := '020203040506'O;

// IND frame too short.
template Raw Codec_Test_INV_5 := '030203040506'O;

// ACK frame too short.
template Raw Codec_Test_INV_6 := '040203040506'O;

// ACK frame too long.
template Raw Codec_Test_INV_7 := '0402030405060708'O;

}
```

Liite 3. UDPModule3

```
module UDPModule3
{

	import from UDPModule1 all;
	import from UDPModule2 all;

	altstep DefaultAltstep() runs on TestComponent
	{
		[] p.receive
		{
			setverdict(fail);
			stop;
		}
		[] p.getcall
		{
			setverdict(fail);
			stop;
		}
		[] p.getreply
		{
			setverdict(fail);
			stop;
		}
		[] p.catch
		{
			setverdict(fail);
			stop;
		}
		[] T_GUARD.timeout
		{
			setverdict(inconc);
			stop;
		}
	}

	type port PortType mixed
	{
		inout all;
	}

	type component TestComponent
	{
		port PortType p;

		const float T_GUARD_DEFAULT := 5.0;

		timer T_GUARD := T_GUARD_DEFAULT;

		var address sut_addr :=
		{
			host := PX_SUT_IP_ADDR,
			portField := PX_SUT_PORT
		};

		var default compDefaultRef;
	}

	type component TestSystemInterface
	{
		port PortType tsiPort;
	}


	function preambleSetup() runs on TestComponent
```

```
{
		map(mtc:p, system:tsiPort);
		T_GUARD.start;
		compDefaultRef := activate(DefaultAltstep());
}

function postambleRelease() runs on TestComponent
{
		deactivate;
		T_GUARD.stop;
		unmap(mtc:p, system:tsiPort);
}

//Testi-caset

testcase TC_GET09() runs on TestComponent system TestSystemInterface
{
		preambleSetup();

		p.send(GET09) to sut_addr;
		p.receive(IND09) from sut_addr;

		setverdict(pass);

		postambleRelease();
}

testcase TC_GET09_a() runs on TestComponent system TestSystemInterface
{
		preambleSetup();

		p.send(GET09_a) to sut_addr;
		p.receive(ACK09_a) from sut_addr;

		setverdict(pass);

		postambleRelease();
}

testcase TC_GET10() runs on TestComponent system TestSystemInterface
{
		preambleSetup();

		p.send(GET10) to sut_addr;
		p.receive(IND10) from sut_addr;

		setverdict(pass);

		postambleRelease();
}

testcase TC_SET00() runs on TestComponent system TestSystemInterface
{
		preambleSetup();

		p.send(SET00) to sut_addr;
		p.receive(ACK00) from sut_addr;

		setverdict(pass);

		postambleRelease();
}

testcase TC_SET01() runs on TestComponent system TestSystemInterface
{
		preambleSetup();

		p.send(SET01) to sut_addr;
		p.receive(ACK01) from sut_addr;
```

```
                setverdict(pass);

                postambleRelease();
        }

        testcase TC_SET02() runs on TestComponent system TestSystemInterface
        {
                preambleSetup();

                p.send(SET02) to sut_addr;
                p.receive(ACK02) from sut_addr;

                setverdict(pass);

                postambleRelease();
        }

        testcase TC_SET03() runs on TestComponent system TestSystemInterface
        {
                preambleSetup();

                p.send(SET03) to sut_addr;
                p.receive(ACK03) from sut_addr;

                setverdict(pass);

                postambleRelease();
        }

        control
        {
                execute(TC_GET09());
                execute(TC_GET10());
                execute(TC_SET00());
                execute(TC_SET01());
                execute(TC_SET02());
                execute(TC_SET03());
        }
}
```

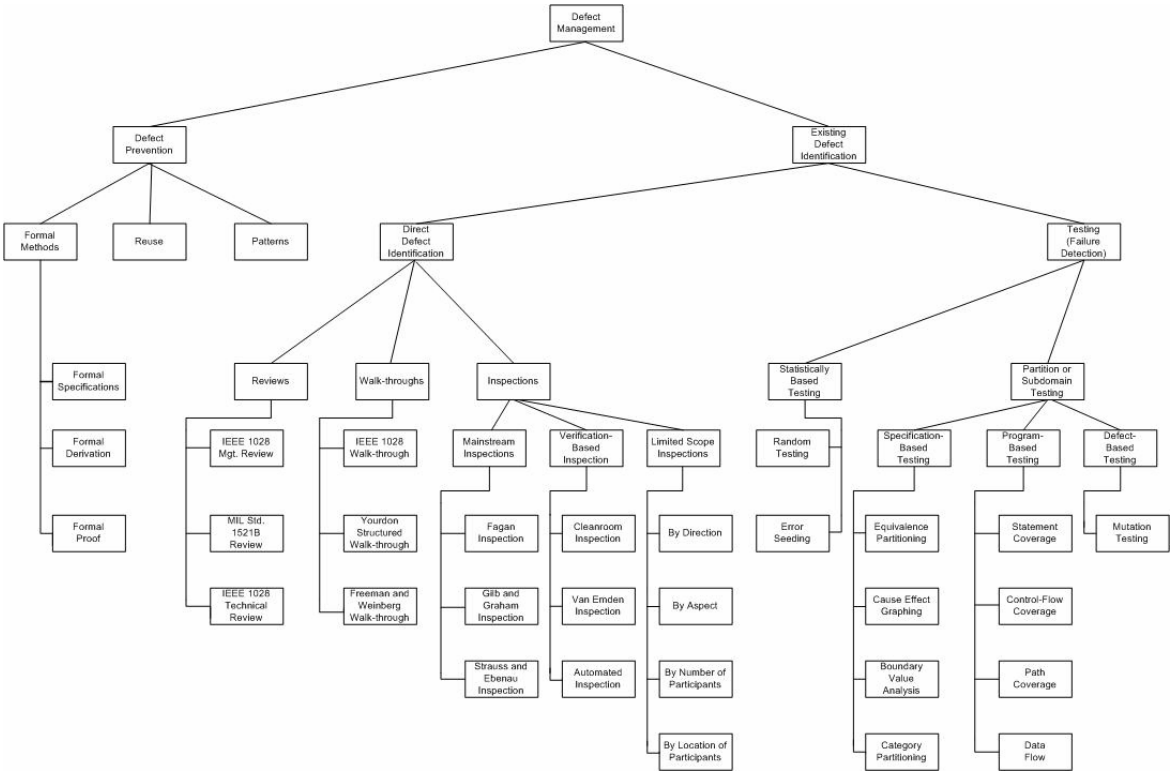
Liite 4. UDPModuleParameters

```
module UDPModuleParameters
{

		modulepar
		{
			// IP address and port number of the SUT.
			charstring PX_SUT_IP_ADDR := "127.0.0.1";
			integer PX_SUT_PORT := 6510;

			// destination IP
			octetstring IP_DEST := '64646467'O;
		}
}
```

Liite 5. Virheiden hallinta Younessin mukaan



Kuva 32: Younessin virheiden hallinnan jakamistapa [38].