Janne Rahkola

Ohjelmistojen validointi mittausjärjestelmissä

Insinööri (AMK)

Tieto- ja viestintätekniikka

Kevät 2022

KAMK • University of Applied Sciences

Tiivistelmä

Tekijä: Janne Rahkola

Työn nimi: Ohjelmistojen validointi mittausjärjestelmissä

Tutkintonimike: Insinööri (AMK), tieto- ja viestintätekniikka, älykkäät järjestelmät

Ohjaaja: Jaakko Vanhala

Asiasanat: automatisointi, Docker, ESP32, mittalaite, mittausjärjestelmä, ohjelmistotestaus, PlatformIO, validointi

Tutkimuksen lähtökohtana toimi Kajaanin Mittaustekniikan yksikössä kehitetty MITYSens-olosuhdejärjestelmä, jonka ohjelmiston laatu oli tarkoitus validoida. Aihepiiriä kartoittaessa kuitenkin selvisi, ettei validoinnille löydy yksiselitteistä ja tarkkaa määritystä. Täten validoinnin tarkastelua kohdennettiin uudelleen ja päähuomio asetettiin ESP32-kehitysalustaan ja lämpötila-anturiin DS18B20, joiden ohjelmistoa, toimintaa ja testausta tarkastellaan erilaisista näkökulmista.

Tutkimus alkaa ohjelmiston kehittämisen menetelmistä ja erilaisista ohjelmistokehityksen työkaluista, minkä jälkeen siirrytään tutkimuksen käytännön toteutukseen. Ohjelmistotestauksen pääasiallisina työkaluina käytetään Visual Studio Codea ja siihen asennettua liitännäistä PlatformIO, minkä lisäksi palvelimella tehtävät toimenpiteet suoritetaan GitLab-versionhallinnan Docker-konteissa. Kyseisten työkalujen avulla voidaan suorittaa niin yksikkötestejä kuin koko järjestelmää tarkastelevia toimenpiteitäkin.

Ylipäätään ohjelmistojen validoinnin keskiössä ovat erilaiset manuaaliset tai automatisoidut testit, joiden avulla on mahdollista varmentaa funktioiden, luokkien ja toiminnallisuuksien oikeellisuus. Toisaalta mittausjärjestelmien ohjelmistojen validoinnissa on laajemmassa perspektiivissä tarkasteltaessa huomioitava myös koko tuotantoketjun laatutekijät. Tällöin ohjelmiston elinkaaren aikana päädytään ensimmäisestä vaatimusmäärittelystä aina ohjelmiston ylläpitoon ja viimeisen vaiheen alasajoon saakka. Tosin on huomattava, että tämän tutkimuksen puitteissa laajemman näkökulman käsittelyä täytyi rajata voimakkaasti.

Mittausjärjestelmien validoinnin suhteen on myös huomattava, että ohjelmistojen ja mittalaitteen tulosten validointi eivät ole sama asia. Mittaustulosten validointi perustuu enimmäkseen referenssimittauksille ja niiden matemaattiselle tarkastelulle, mutta tässä tutkimuksessa aihetta oli vain mahdollista sivuta lyhyesti. Ehkä tulevissa tutkimuksissa validointia voidaan tarkastella perinpohjaisemmin, jolloin joitain tästä tutkimuksesta rajattuja aihepiirejä voidaan ottaa parempaan tarkasteluun.

KAMK • University of Applied Sciences

Abstract

Author: Janne Rahkola

Title of the Publication: Software Validation in Measurement Systems

Degree Title: Bachelor of Engineering, Information- and Communications Technology, Intelligent systems

Supervisor: Jaakko Vanhala

Keywords: automation, Docker, ESP32, measurement device, measurement system, software testing, PlatformIO, validation

This study is inspired by the MITYSens environmental sensing device, which is developed at the measurement technology research unit of MITY Kajaani. The mission was to validate the software quality of the produced device. However, when mapping the topic, it became clear that there was no unambiguous and precise definition for validation. Thus, the perspective of the validation was focused on the ESP32 development board and to the DS18B20 temperature sensor.

The review of this research begins from the methods of software development and goes on to the examination of various software development tools, followed by a practical implementation. In this study the main tools used for software testing are Visual Studio Code and the PlatformIO plugin. In addition, some operations are performed on the server using GitLab version control in which the Docker containers are operated. The tools mentioned can be used either to perform unit tests or system-wide procedures.

Typically, the focus of software validation is on various manual or automated tests which make it possible to verify the correctness of functions, classes or different kind of use cases. In broader perspective however, the validation of software for measurement systems should also take into account the quality factors of the entire production chain and life cycle of the software. Where the life cycle of the software begins from the first requirement definitions and goes all the way from the development and maintenance phases to the final scrapping of the software. It should however be noted, that within this study the handling of the broader perspective had to be restricted to a very limited view.

When validating measurement systems, it should also be noted that software validation and validation of measured results are two different things. The validation of measured results is mainly based on reference measurements with calibrated equipment in which validation procedure is done by mathematical analysis. But in this study however, it was only possible to briefly glance at that point of view. Perhaps in some other studies, different aspects of validation can be examined in a more diverse and in-depth way.

Sisällys

Lyhenteet ja termit

1-Wire	Yhdellä johtimella toteutettu sarjamuotoinen tiedonsiirtoratkaisu
Continuous delivery (CD)	Jatkuva toimitus. DevOps-menetelmä, jonka tarkoituksena on nopeuttaa ohjelmistotuotantoa automatisoimalla testausta ja ohjelmakoodin hyväksyntää
Continuous deployment (CD)	Jatkuva käyttöönotto. DevOps-menetelmä, jossa ohjelman julkaisu tuotantoympäristöön on automatisoitu
Continuous integration (CI)	Jatkuva integraatio. DevOps-menetelmä, jossa useiden tekijöiden päivitykset yhdistetään samaan ohjelmistoprojektiin
CI/CD-ohjelmistoputki	Automaattinen ohjelmakoodin käsittelyn prosessi, joka suoritetaan versionhallintapalvelussa tiedostoja päivitettäessä
DevOps	Monitulkintainen termi, joka toisaalta kuvaa modernia ohjelmistokehityksen työkulttuuria. Ja toisaalta tuotantotapaa, jossa CI/CD-ohjelmistoputki on käytössä
Git	Hajautettu versionhallintajärjestelmä, joka on alun perin Linus Torvaldsin kehittämä
Git Bash	Windows-sovellus, jonka avulla komentokehotteessa on mahdollista hyödyntää Git-käskyjä
GitLab	Eräs versionhallintapalvelu, joka tarjoaa tiedostojen ylläpidon, hajautetun ohjelmistokehityksen ja CI/CD-ohjelmistokehitysputken
IoT	Internet of Things, esineiden Internet
Iteraatio	Lyhyt, yleensä muutaman viikon, työskentelyjakso. Jakson aikana tehdään joko jonkin tuotteen tai ohjelmiston osa tai valmis tuote

JSON	JavaScript Object Notation. Tiedonvälittämiseen optimoitu tiedostomuoto, jonka dataformaatteja voivat olla: liukuluku (int tai float), merkkijono (string), totuusarvo (boolean), taulukko (array), objekti (object) ja tyhjä (null)
PlatformIO	Sulautettujen järjestelmien ohjelmointiin tarkoitettu ohjelmointiympäristö, jonka saa osaksi Visual Studio Codea
Repository	Versionhallinnan tiedostorakenne, jossa projektikohtaisia tiedostoja seurataan, päivitetään ja ylläpidetään.
Scrum	Ketterän ohjelmistokehityksen menetelmä, jossa pyritään asetettuihin päämääriin prosessin toisteisuuden ja oikean työnjaon avulla
Sprint	Ks. Iteraatio
Thingsboard	Avoimen lähdekoodin IoT-järjestelmä. Mahdollistaa datan käsittelyn ja visualisoinnin
Visual Studio Code tai VS Code	Visual Studio Code on Microsoftin kehittämä ohjelmointiympäristö, joka sisältää tekstieditorin ja ohjelmointikielen kääntäjän lisäksi toimintoja esimerkiksi debuggaukseen ja versionhallintaan

1 Johdanto

Tutkimuksessa käsitellään ohjelmistojen validointia mittausjärjestelmissä. Koen aiheen merkitykselliseksi, koska anturitekniikan parantuessa erilaisten järjestelmien hyödyntäminen mittauksissa yleistyy ja näillä näkymin sekä mittausten että mittausdatan määrä tulee jatkossa kasvamaan entisestään. Käytön lisääntyessä säilyy kuitenkin tarve mittausten tarkkuudelle, todentamiselle ja jäljitettävyysketjujen rakentamiselle. Edelleen mittausjärjestelmät ovat enenevässä määrin riippuvaisia ohjelmistoista ja niiden päivittämisestä. Täten on tarpeellista, että mittalaitteiden tulee olla valmistuessaan validoituja ja hyväksi havaittuja, mutta myös päivitysten jälkeen pysyttävä sellaisina. Tutkimuksen tarkastelussa Mittaustekniikan yksikössä rakennettu MITYSens-olosuhdejärjestelmä toimii inspiraationa ohjelmiston validoinnille.

Tutkimuksen aiheen määrittäminen ei kuitenkaan ollut yksinkertaista, sillä toistaiseksi mittalaitteen ohjelmiston validoinnille ei ole olemassa standardia. Tämän seurauksena mitään suoraa esimerkkitapausta tai lähdettä ei ole olemassa. Eri tulokulmista aihepiiriä sivuavia lähteitä ja aineistoja kuitenkin löytyi. Mainittakoon näistä esimerkiksi lukuisat Theseukseen ladatut opinnäytetyöt, jotka käsittelevät esimerkiksi ohjelmiston automatisoitua testausta ja PlatformIOta; sekä erinäiset kirjat, jotka käsittelevät esimerkiksi oikeaoppista ohjelmistojen kehitystä ja mittausten tekoa; sekä lukuisat internetlähteet, jotka tuovat lisävalaistusta erilaisten ohjelmien ja työkalujen käyttöön sekä monenlaisiin ohjelmointiratkaisuihin.

Sisältönsä puolesta tutkimus jakautuu kolmeen osaan. Ensin käsitellään lyhyesti MITYSens-olosuhdejärjestelmää, jotta tutkimuksen lähtökohdat tulevat esitellyksi. Seuraavaksi siirrytään teoreettiseen käsittelyyn, jossa käytetyt ohjelmat, menettelytavat ja tutkimuksen viitekehys pyritään esittämään ajantasaisesti ja asianmukaisesti. Kolmantena on käytännönläheisempi käsittely, jossa pyritään soveltamaan teoriaa käytännössä. Toki on huomattava, että vaikka tutkimuksen painopiste on ohjelmistoissa, niin luonnollisesti myös tekniset ratkaisut määrittävät mittalaitteen ominaisuuksia. Tutkimuksen sisällön hahmottamiseksi lukija voi myös tarkastaa Liitteen 1, jossa sisällysluettelo on eritelty vaihtoehtoisena visuaalisena esityksenä [Liite 1]. Tämä nähtiin hyödyllisenä, koska kuvan avulla eri lukujen välisten riippuvuussuhteiden havaitseminen on nopeaa ja yksinkertaista.

Tutkimusalueeseen liittyy lukuisia rajauksia. Toisaalta käytettyjen ohjelmien ja menetelmien käsittelyä on rajattu, jolloin insinööriopintojen puitteissa käytyjä yleisiä asioita ei pääsääntöisesti

eritellä. Mainittakoon näistä kuitenkin tässä yhteydessä esimerkiksi Git Bash, jonka avulla on suoritettu erinäiset versionhallinnan toimenpiteet. Ja toisaalta ESP32 sekä koekytkentäalusta ja niihin tehdyt yksinkertaiset kytkennät. Myös PlatformIOn käyttöön liittyvät toimenpiteet, kuten kirjastojen lisäys ja ohjelmien kasaaminen, ovat omaa aihepiiriään, mutta niiden erittely kasvattaisi opinnäytettä tarpeettomasti. Lisäksi edellä mainitut seikat ovat tästä tutkimuksesta kiinnostuneille joko entuudestaan tuttuja tai tarpeen vaatiessa kohtalaisen nopeasti erilaisten aineistojen avulla omaksuttavissa.

Toisaalta rajauksia on tehty aihepiiriin liittyen. Mainittakoon näistä esimerkiksi, että tutkimuksessa ohjelmointiin käytetään yhtä ohjelmaa, sitä varten sovelletaan yksittäistä ohjelmointialustaa ja versionhallintaan käytetään tiettyä palvelua, vaikka lukuisia muitakin vaihtoehtoja on olemassa. Lisäksi validoinnin osalta keskitytään valtaosin ohjelmiston sisäiseen tarkasteluun, vaikka myös mittaustulosten matemaattinen ja tilastollinen tarkastelu ovat keskeinen osa mittalaitteen toiminnan arviointia. Tutkimuksessa ei myöskään keskitytä mittalaitteen tuottamien tulosten varmentamiseen referenssimittausten avulla. Edelleen tutkimuksessa keskitytään vain mittalaitteeseen, eikä esimerkiksi tiedonsiirrossa tai tietokannassa tapahtuvaan datan käsittelyn tai korjaukseen. Ylipäätään tutkimuksessa on pyritty erinäisten esimerkkien avulla hakemaan ratkaisuja rajattuihin ja yksinkertaisiin tutkimusongelmiin.

Tutkimuskysymysten kannalta tämä tarkoittaa esimerkiksi seuraavaa: Millä menetelmillä ja työkaluilla modernia ohjelmistokehitystä tehdään? Mitä validointi on ja kuka sen suorittaa? Miten ohjelmiston validointi on mahdollista suorittaa? Kuinka moderni mittalaite rakennetaan modulaarisesti? Millaisia ohjelmistotestejä tarvitaan? Kuinka yhden mittalaitteen esimerkkitapausta voidaan soveltaa laajaan joukkoon mittalaitteita? Ja niin edelleen. Kuten edellä olevasta kysymyksien paljoudesta havaitaan, niin tutkimukseen liittyy monenlaisia osatavoitteita. Työn käsittelyn edetessä lukija myös toivottavasti saa vastauksen mahdollisimman moneen niistä.

Koko tutkimusta ajatellen keskeistä on termiin validointi liittyvä problematiikka. Tällöin tarkastellaan, että tehdäänkö oikeaa asiaa ja jos tehdään, niin tehdäänkö sitä oikealla tavalla. Määrityksen taustalla on lisäksi ajallinen ulottuvuus, joka voidaan tiivistää sanaan muutos. Mittalaitteiden osalta on nimittäin huomattava, että kaikkien järjestelmän osien ja ohjelmistojen jatkuvasta päivittymisestä, kehittymisestä ja vaihtumisesta seuraa jatkuvaa muutosta. Mutta vaikka sekä anturit, kehitysalustat, kehitysympäristöt ja kaikki muut osat muuttuvat ja päivittyvät ajan kuluessa, niin tarve toimiville ohjelmistoille ja niiden toiminnan varmentamisella ja todentamiselle ei poistu.

2 MITYSens-olosuhdejärjestelmästä

2.1 MITYSens-olosuhdejärjestelmän ominaisuuksista ja kehitysympäristöstä

MITYSens-olosuhdejärjestelmän tarkoituksena on seurata mittalaitteen ympäristöä. Ympäristön havainnointi tapahtuu mittalaitteen antureita hyödyntämällä, jolloin olosuhdeanturi tuottaa halutuin väliajoin lukuarvot havainnoistaan. Tämän tutkimuksen puitteissa mitattavaksi valittiin vain lämpötila (°C). Mittalaitteen rakenteesta johtuen myös muita asioita mitataan, mutta koska useiden suureiden todentaminen ja jäljitettävyys muodostuvat monimutkaiseksi sekä dataa tuottavien anturien että ohjelmistojen osalta, niin niiden käsittelyä ei tämän tutkimuksen puitteissa nähty järkeväksi.

IoT-kehitysalustoja on olemassa monenlaisiin tarpeisiin, mutta MITYSens-olosuhdejärjestelmän ytimenä on ESP32 [1]. Käytössä oleva ESP32 on Espressif Systemsin valmistama kehitysalusta, jossa yhdistyvät matala hinta ja monipuoliset ominaisuudet. Ominaisuuksia ovat esimerkiksi kaksi 240Mhz:n prosessoria, 3,3 voltin käyttöjännite sekä WLAN- ja Bluetooth-yhteydet [1]. ESP32-ohjelmoinnin mahdollistavana kehitysympäristönä toimii puolestaan Visual Studio Code ja siihen asennettu PlatformIO, joita tarkastellaan seikkaperäisemmin luvussa 4.3. Mainittakoon kuitenkin tiivistetysti ohjelmointiympäristön mahdollistavan lukuisia toimintoja, kuten esimerkiksi ohjelman kirjoittamisen ja kääntämisen sekä versionhallinnan.

2.2 Ohjelmistosta ja tiedonsiirrosta

Sulautetun laitteen ohjelmiston tulee mittalaitteessa olla monilta osin virheetön. Käytännössä tämä tarkoittaa esimerkiksi sitä, että sen tulee toimia pitkiä aikoja häiriöttä. Lisäksi mittalaitteen ohjelmiston arkkitehtuurin tulee mahdollistaa anturien lisäämiset ja poistot sekä päivitykset ja muokkaukset. Myös versionhallinta tulee hoitaa asianmukaisesti ja versionumerointi siten, että muutokset ovat jäljitettävissä ja tulokset todennettavissa.

Tiedonsiirron ja kommunikaation kannalta tämä tarkoittaa joko yhtä tai mahdollisesti useita tapoja kommunikoida mittalaitteen kanssa. MITYSens-olosuhdejärjestelmän tapauksessa tiedonsiirtotavat ovat WLAN ja BLE. Yleisemmin erilaisia taajuusalueita, kantamia ja tiedonsiirtonopeuksia erittelee esimerkiksi Juha Hauhia [2, s. 11].

2.3 Datan tallennus ja esittäminen

Jos ja kun anturin data halutaan tietokantaan, niin ensin anturikohtainen data luetaan ohjelmassa. Tämän jälkeen data asetetaan objekteina JSON-formaattiin, jossa kukin objekti koostuu avaimen ja arvon muodostamasta dataparista. Tarkemmin eriteltynä yksittäinen objekti koostuu yleensä merkkijonosta ja liukuluvusta, jolloin kokonainen JSON-viesti koostuu suuresta määrästä objekteja. Kun kaikki saatavilla oleva informaatio on luettu, niin JSON lähetetään säännöllisin väliajoin tietokantaan hyödyntäen esimerkiksi WLAN-yhteyttä.

Kun lähetetty data on vastaanotettu, niin se tallennetaan tietokantaan ja samalla data siirtyy Thingsboardin hyödynnettäväksi. Thingsboard sekä käsittelee vastaanotettua dataa että visualisoi tapahtuneet muutokset. Käytännössä käsittely voi tarkoittaa vaikkapa lämpötilojen keskiarvoistamista ja kuvaajaa, jossa lämpötilan arvo piirtyy taulukkoon kunkin ajanhetken tilanteen mukaisesti. Seikkaperäisemmin MITYSens-olosuhdejärjestelmän datan visualisointia ja muitakin piirteitä on tarkasteltu Juha Mustosen vuonna 2021 ilmestyneessä opinnäytteessä [3].

2.4 Järjestelmän vaatimuksia

Toisen pääluvun puitteissa on tähän saakka eritelty MITYSens-olosuhdejärjestelmän rakenteita, ominaispiirteitä ja käytettyjä ratkaisuja. Tosin niin, että tutkimuksen yksinkertaistamiseksi tarkastelu on rajattu koskemaan vain ESP32-kehitysalustaa ja lämpötilaa mittaavaa anturia. Valistunut lukija voi kuitenkin kysyä, mitä tekemistä tällä on ohjelmistojen validointi mittausjärjestelmissä -kysymyksen kannalta.

Ohessa muutamia näkökulmia ja vaatimuksia, joita modernin mittalaitteen suunnittelun, rakennuksen ja ylläpidon tulee huomioida. Yksi näkökulma liittyy esimerkiksi tuotantoprosessiin, jonka aikana validointiasiat huomioidaan, ohjelmaversiot rakennetaan ja työ dokumentoidaan. Toiseksi huomio kiinnittyy vaikkapa ohjelmistojen päivittämisen aiheuttamiin ongelmiin tulosten todentamisen ja jäljitettävyyden suhteen.

3 Nykyaikainen ohjelmistojen valmistaminen

3.1 Ohjelmistokehityksen historiasta

Monien on hankala hahmottaa, että ohjelmistoja on tehty erilaisilla laitteilla ja monilla ohjelmointikielillä jo vuosikymmenien ajan. Yhden varhaisimmista ja eniten yleisön tietoisuuteen levinneistä ohjelmistokehityksen malleista julkaisi vuonna 1970 Winston Royce [4]. Nykyisin vesiputousmallina tunnettua lineaarisen kehityksen mallia pidetään ohjelmistokehityksen arkkityyppinä. Mallissa edetään suunnitelmallisesti järjestelmän ja ohjelmiston vaatimuksista kohti ohjelmiston määrittelyä ja suunnittelua, josta edetään ohjelmoinnin ja testauksen kautta ohjelmiston varsinaiseen käyttöön. Alkuperäisessä mallissa eri vaiheiden välistä iteratiivista kehitystä korostetaan, mutta yleisesti vakiintuneen käsityksen mukaisesti vesiputousmallissa prosessi pitää etukäteen suunnitella, resursoida ja aikatauluttaa. Ohjelmistokehityksen luonteen vuoksi kaikkia vaatimuksia on kuitenkin lähes mahdotonta määrittää etukäteen, eikä tuotantoprosessin toteutus ole puristettavissa lineaariseksi tuotantoputkeksi. [5, s. 37; 6.]

Moderneista ohjelmistokehityksen menetelmistä tunnetuin lienee Scrum, joka on iteraatioihin perustuva ketterä ohjelmistokehitystapa [7]. Scrumissa työ jakautuu yhdestä neljän viikon työjaksoihin, joita Scrumissa kutsutaan sprinteiksi. Kussakin sprintissä tiimi pyrkii toteuttamaan vaatimusten mukaiset toiminnallisuudet, jotka sprintin alussa on määritelty. Scrum-menetelmässä keskeistä ovat palaverit, joita on ennen sprinttiä, jopa päivittäin sprintin aikana ja sprintin lopussa järjestetään katselmointipalaveri, jossa todetaan saavutetut asiat. Lisäksi sprintin jälkeen järjestetään kehityskohteita pohtiva retrospektiivi. [5, s. 49–51; 6; 8, s. 4.]

Scrumin työnjako jakautuu kolmen roolin kesken. Tärkein näistä on tuoteomistaja, joka muuntaa järjestelmän vaatimuksia työlistaksi ja priorisoi eri tehtävät. Lisäksi tuoteomistaja valitsee sprintin alussa kehitysjonon tehtävät ja lopussa katselmoi tehdyt tuotokset. Toinen rooli on Scrum master, joka vastaa työn tuotoksista ja jonka vastuulla on varmistaa tiimin työrauha, hankkia tarvittavat välineet ja organisoida lukuisat palaverit. Kolmas Scrum-menetelmän rooli on tiimi, joka koostuu minimissään kolmesta ja maksimissaan yhdeksästä henkilöstä. Tiimi on itseohjautuva ja pyrkii hoitamaan työlistasta valitut tehtävät (task). Tavoitteena Scrumissa on toiminnan läpinäkyvyys, mukautumiskyky ja toiminnan seuranta. Mutta oli kyseessä vesiputousmalli, Scrum tai joku muu, niin käytetyn ohjelmistokehitysmallin pyrkimyksenä on aina projektin eteneminen ja asetettujen tavoitteiden saavuttaminen [5, s. 47–49; 6; 8, s. 4–5.]

3.2 Sulautetun laitteen kehityksen ominaispiirteitä

Tavoitteiden ja projektinhallinnan kannalta ensimmäinen, ja kaikista tärkein, askel tuotteen aikaansaamiseksi on vaatimusmäärittely, jonka mukaisesti ohjelmistoprojektit joko onnistuvat tai ajautuvat vaikeuksiin. Vaatimusmäärittely jakautuu kahteen osaan: toiminnallisiin vaatimuksiin, kuten esimerkiksi, että mittalaite tuottaa arvoja, ja ei-toiminnallisiin vaatimuksiin, kuten esimerkiksi, että järjestelmä toipuu virhetilanteista. Määrittelyn ohella sulautetun laitteen kehitys jakautuu ohjelmiston suunnitteluun, toteutukseen ja testaukseen, joiden jälkeen siirrytään laitteen käyttöönottoon ja ylläpitoon. Jakoa on eritelty myös kuvassa 1. [5, s. 29–31 ja 61–68; 6.]

<table>
<tr><td rowspan="2">Määrittely</td><td>Ohjelmistosuunnittelu</td><td>Ohjelmointi</td><td>Ohjelmistotestaus</td><td rowspan="2">Käyttöönotto</td></tr>
<tr><td>Elektroniikkasuunnittelu</td><td>Elektroniikkatoteutus</td><td>Laitteen testaus</td></tr>
<tr><td>Dokumentaatio</td><td>Versionhallinta</td><td>Laadunvarmistus</td><td>Vaatimustenhallinta</td><td>EMC-testaus</td></tr>
</table>

Kuva 1. Sulautetun laitteen kehityksen osa-alueita. [5, s. 207; 9, s. 2.]

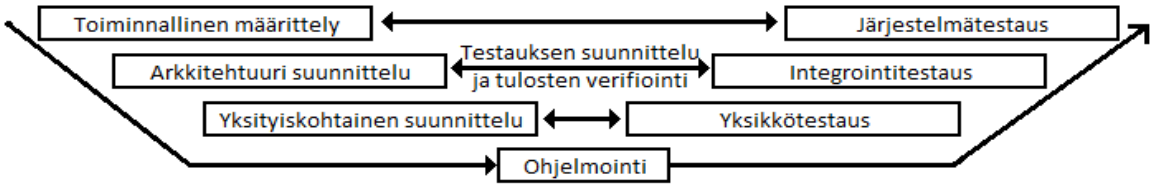
Kuten kuvasta 1 havaitaan, niin sulautetun laitteen kehityksessä tulee lisäksi huomioida kattavan ja ajantasaisen dokumentaation laatiminen, versionhallinta sekä laadunvarmistukseen ja vaatimustenhallintaan liittyvät seikat. Erityisesti muuttuvat vaatimukset ja niihin reagointi ovat olennainen osa ketterää sovelluskehitystä. Sulautetun laitteen onnistuminen vaatii myös elektroniikan suunnittelua, toteutusta ja testausta, mutta kyseisiä aiheita ja niiden sisältöä ei ole tämän tutkimuksen puitteissa mahdollista eritellä.

Sen sijaan tutkimuksen kannalta olennaisimmat kohdat löytyvät kuvan 1 lokeroista ohjelmistotestaus ja laadunvarmistus. Näiden avulla on nimittäin mahdollista päästä kiinni työn varsinaiseen aiheeseen. Laadunvarmistuksen osalta tämä tarkoittaa sitä, että projektien eteneminen on hallittua ja dokumentoitua, projektin laatuvaatimukset täyttyvät ja määritellyt toiminnallisuudet toteutuvat sekä sitä, että aikataulu ja budjetti eivät hajoa. Tarkan määrittelyn puuttuessa laadun mittaaminen on kuitenkin hankalaa. [5, s. 29–31 ja 61–68; 6.]

Yksi vaihtoehto laadun järjestelmälliseen kartoitukseen on auditointi- tai laadunhallintajärjestelmän standardin käyttöönotto. Ohjelmistostandardien osalta tunnetuimmat lienevät CMMI, IEEE 1012, ISO/IEC 25010 ja ISO/IEC 29119. Toistaiseksi kaikkien hyväksymää ja käyttämää ohjelmistoalan standardia ei kuitenkaan ole käyttöön vakiintunut, kuten 33 eri standardia Liitteessä 2 osoittavat. Laadunvarmistuksen ohella tärkeäksi todettiin ohjelmistotestaus, jota avataan seikkaperäisemmin seuraavassa luvussa. [5, s. 144–150; 10, s. 10–13; 11, s. 101 ja 139; 12; Liite 2.]

3.3 Ohjelmistotestaus

Ohjelmistotestauksella tarkoitetaan suunnitelmallista virheiden etsintää joko koko ohjelmaa tai sen osaa suorittamalla. Yksinkertaisin tapa manuaalisen tai automatisoidun testauksen havainnollistamiseen on V-mallin käyttö, jolloin suunnittelutaso ja testaustaso rinnastuvat. Tällöin testien tasoja ovat yksikkötestaus, integrointitestaus ja järjestelmätestaus, kuten kuvasta 2 havaitaan. Kaikkia ohjelmakoodin sisältämiä virheitä ei juuri koskaan voida poistaa, mutta tavoitteena on etsiä ja korjata järjestelmän toiminnan kannalta kriittiset virheet. Tällöin ohjelmisto todennäköisesti myös täyttää sille asetetut vaatimukset. Yleisesti pätevät testauksen toimintamallit voi myös halutessaan tarkastaa Liitteistä 3 ja 4. [5, s. 205–209; 6; 11, s. 103; Liite 3; Liite 4.]



Kuva 2. Ohjelmistotestauksen V-malli. [5, s. 207; 6.]

Testauksen perustaso on yksikkötestaus, jonka tavoitteena on todentaa esimerkiksi jonkin luokan tai moduulin toiminta. Integrointitestauksen tavoitteena on luokkien yhdistäminen, jolloin tarkastellaan jonkin osajärjestelmän toimintaa. Järjestelmätestauksen kohteena on koko järjestelmä, jota verrataan suunnitteludokumentteihin. Järjestelmätestiin voidaan myös sisällyttää hyväksymistestaus ja esimerkiksi luotettavuustestit. Yleisesti voi todeta, että mitä korkeammalla tasolla ohjelmisto-ongelma havaitaan, niin sitä työläämpää ja hankalampaa sen korjaaminen on.

Testitapausten valinta voidaan suorittaa esimerkiksi lasilaatikkotestauksena (white box), jossa ohjelmakoodi tunnetaan ja siten voidaan keskittyä tarkkoihin ohjelmiston rakenteisiin. Vastaavasti mustalaatikkotestauksessa keskitytään toiminnallisuuden tarkasteluun ohjelmakoodia tuntematta. Tällöin testitapaukset jaetaan ekvivalenssiluokkiin, jolloin testisyötteet ryhmitellään ohjelman toiminnan kannalta samalla tavalla toimiviin ryhmiin. [5, s. 209–210; 6; 11, s. 64–68.]

Käytännön ohjelmistotestaus on lisäksi jaettavissa staattiseen ja dynaamiseen testaukseen. Staattinen ohjelma-analyysi voi esimerkiksi tarkastella lähdekoodin rakennetta, syntaksia ja rajapintoja. Dynaamisessa testauksessa ohjelmakoodia puolestaan ajetaan, jolloin eri syötteet tuottavat erilaisia tuloksia. Tällöin on esimerkiksi syytä tarkastella koodikattavuutta, jota testit tarkastelevat. Lisäksi on huomattava, että ohjelmointikieli määrittää käytettävät työkalut. Tällä erää paras tuki löytyy käytetyimmille ohjelmointikielille, kuten JavaScript ja Python. [6; 11, s. 65; 12.]

3.4 DevOps

Nykyaikaisen lähestymiskulman ohjelmistokehityksen käsittelyyn tarjoaa myös termi DevOps, joka on yhdistelmä sanoista Dev (developers, eli ohjelmiston kehittäjät) ja Ops (operations, eli järjestelmän ylläpitäjät). Hankalahko termi on jaettavissa ainakin kahteen aihepiiriin, joita ovat yrityskulttuurin DevOps ja DevOps-tuotantotapa. Kulttuurillisessa mielessä DevOps on periaate, jonka mukaisesti kehitystiimit sisältävät kaiken tietotaidon. Usein DevOps kulttuurin vahvistaminen nähdään tarpeellisena, koska jos ohjelmistojen kehittäjät ja ylläpitäjät ovat toisistaan riippumattomia ja erillisiä yksikköjä, niin ryhmien osaaminen eriytyy ja tiedonkulku sekä kommunikaatio vaikeutuvat. Kulttuurillisesti tästä pyritään eroon, mikä tarkoittaa organisaation sisällä parempaa luottamusta ja työnjaollisesti tehokkaampaa vastuunkantoa. DevOps-kulttuurissa pyritään siis siihen, että tiimit pystyvät sisäisesti viemään ohjelmakoodiin toteuttamansa uudet toiminnallisuudet tuotantoympäristöön sekä testaamaan ja operoimaan niitä. [6; 13, s. 26; 14, s. 14.]

DevOps-tuotantotapa on työn tekninen toteutusmenetelmä, joka toimii yhdessä termin CI/CD kanssa. Jatkuvan integraation (CI) tarkoituksena on, että jokainen kehittäjä integroi tekemänsä työn versionhallinnassa ylläpidettävään koodiin mahdollisimman usein, jolloin työ jaetaan pieniin osiin ja ohjelmistoprojektin loppuun kasautuvalta integraatiohelvetiltä vältytään. Jos jokainen CI ohjelmistoputken läpäisevä ohjelmiston päivitys (commit) viedään automatisoidusti testien jälkeen tuotantoon, niin sitä nimitetään jatkuvaksi käyttöönotoksi (CD, Continous deployment). Jos vain ihmiset vievät ohjelmakoodin tuotantoon, puhutaan jatkuvasta toimitusvalmiudesta (myös CD, Continuous delivery). DevOpsille ei toistaiseksi ole yhtenäistä määritelmää, mutta tässä tutkimuksessa DevOps yhdistetään joko kulttuuriin tai tuotantotapaan liittyviin määrityksiin. [6.]

DevOpsin mukaiseen työskentelyyn voidaan liittää myös useita teknisiä käsitteitä, joita ovat esimerkiksi automatisoitu testaus, laskenta- ja tallennuskapasiteetin virtualisointi, palveluiden kontittaminen sekä pilvipalveluna toimivat palvelimet ja sovellusympäristöt. Edellä mainituista termeistä voi kiteyttää, että kasvanut laskentateho, kehittyneemmät ohjelmat ja tehokkaammat verkkopalvelut tarjoavat mahdollisuuksia, joita aiemmassa luvussa mainittu Winston Royce ei olisi voinut edes unelmoida. Toisaalta modernit välineet mahdollistavat DevOps-tyylisen ohjelmistojen valmistamisen, jossa hajautettua ohjelmistokehitystä on mahdollista tehdä onnistuneesti, mutta toisaalta ne eivät poista sovelluskehittäjien vastuuta luoda järkevää ohjelmistorakennetta, kirjoittaa mahdollisimman hyvää ohjelmakoodia ja testata ohjelman toiminnallisuutta erilaisin menetelmin. [6; 14, s. 19 ja 33–36.]

3.5 Ohjelmiston validointi ja verifiointi

Nykypäivänä erilaisten ohjelmistojen käyttö ja kehitys on johtanut pysyvään ohjelmistoriippuvuuteen. Vastaavasti kun ohjelmistojen merkitys on kasvanut, niin ne vaativat osakseen erilaisia suunnittelu- ja laatuprosesseja sekä erilaisia menetelmiä laadun varmistamiseksi. Tässä tutkimuksessa asiaa sivuavat laatustandardit myös mainittiin lyhyesti luvussa 3.2. Käytännön laatuun panostaminen on kuitenkin sitä, että ohjelmistoja pitää pystyä jatkuvasti päivittämään, muuttamaan ja korjaamaan toimintakyvyn ylläpitämiseksi ja haavoittuvuuksien korjaamiseksi. Kullakin organisaatiolla on omat tapansa laadunhallintaan, jolloin esimerkiksi automaatiolla voidaan helpottaa rutiininomaisten ja toisteisten tehtävien tekoa. Toisaalta pelkkien työkalujen lisäksi laadun parantaminen edellyttää myös asiaa hoitavien henkilöiden kiinnostusta ja ajantasaista osaamista. [12.]

Ohjelmiston laadunhallintaan liittyvät myös verifioinnin ja validoinnin (V&V) käsitteet. Tässä tutkimuksessa validoinnilla tarkoitetaan sitä, että ohjelmisto täyttää sille asetetut odotukset. Validointiin liittyy lisäksi kysymys asiakkaan tarpeista ja siitä, että tehdäänkö oikeaa tuotetta. Sen sijaan verifioinnissa, eli todentamisessa, pyritään siihen, että ohjelmisto toteuttaa vaatimusmäärittelyn aikana sille asetetut vaatimukset. Verifioinnissa vastataan ensisijaisesti kysymykseen, tehdäänkö tuote oikein. [6; 11, s. 135; 15, s. 26.]

Työekonomisesti verifioinnin kulmakivenä toimii ohjelmiston arviointi. Vastaavasti validoinnin työtapa on testaaminen. Ylipäätään ohjelmistokehityksessä validoinnin ja verifioinnin tavoitteena on, että ohjelmasta tulee “riittävän hyvä”. Tosin hyvyys on usein suhteellista ja riippuu paljon ohjelman käyttötarkoituksesta. Lisäksi on huomattava, että ohjelmiston ei välttämättä tarvitse olla täysin virheetön sopiakseen käyttötarkoitukseensa. Toisaalta vaatimukset voivat olla tiukemmatkin, sillä esimerkiksi lääkinnällisillä laitteilla on omat erikoisvaatimuksensa. [6; 12.]

On myös huomattava, että verifioinnin ja validoinnin menettelytapa vaihtelee ohjelmistokehitystavan mukaisesti. Esimerkiksi vesiputousmallissa vaatimusmäärittely tehdään lähtökohtaisesti heti alussa, jolloin asiakkaan tarpeiden pitäisi periaatteessa olla lyöty lukkoon. Vastaavasti ketterissä menetelmissä ohjelmistotuotanto tapahtuu vaiheittain, jolloin tarpeen vaatiessa vaatimuksia muutetaan asiakkaan toiveiden mukaisesti. Täten kehitystapa vaikuttaa myös siihen, miltä oikea tuote kullakin ajanhetkellä näyttää ja kuinka tuloksia varmennetaan. [6.]

4 Ohjelmistokehityksen työkalut

4.1 Docker

Moderni ohjelmistokehitys ei onnistu ilman oikeita työkaluja. Eräs tärkeimmistä on Docker Inc -emoyhtiön ylläpitämä Docker, joka mahdollistaa konttien ajamisen Docker Enginen avulla. Noin kymmenen vuoden olemassaolonsa aikana Dockerista on kehittynyt ohjelmistoalan standardi, sillä sitä voi hyödyntää lähes kaikilla ohjelmointikielillä ja käyttöjärjestelmillä. Käytännössä Docker mahdollistaa suhteellisen pienellä laskentateholla yksittäisten, jotakin ohjelmaa tai prosessia ajavien Docker imagejen luomisen. Imageja puolestaan ajetaan konteissa, jolloin Docker mahdollistaa itsenäisten konttien pystyttämisen ja purkamisen. Tällöin niiden avulla voidaan joko suorittaa yksittäinen palvelu tai tarvittaessa pystyttää samassa tarkoituksessa kymmeniä tai satoja uusia kontteja. Lukuisia kontteja tarvitaan esimerkiksi, jos niiden avulla halutaan skaalata palveluntarjoajan kovaa kysyntää vastaava määrä palvelua tuottavia elementtejä. [16; 17.]

Tämän tutkimuksen puitteissa olennaista ovat yksittäiset kontit ja se, että ne voivat toimia itsenäisesti. Tämä tarkoittaa sitä, että kutakin konttia kohti luodaan omat asetukset, kirjastot, ohjelmakoodit ja muut toiminnot. Tästä seuraa esimerkiksi, että Dockeria on mahdollista hyödyntää ohjelmien asennukseen ja ohjelmistotestien suorittamiseen. Konttien käyttöön liittyy toki myös omat toimintonsa ja komentonsa, kuten pystyttäminen, käynnistys, pysäytys ja alasajo. Tarvittaessa kontit voivat myös kommunikoida keskenään ja yhdistää itsensä internetiin. Dockerissa on myös lukuisia muita teknisiä ominaisuuksia ja käyttötapoja, mutta niiden käsittely ei tässä yhteydessä ole mahdollista eikä järkevää. Tiivistettynä voi kuitenkin todeta, että Docker on asioita mahdollistava yleisohjelmisto, eikä niinkään yksittäisen työvaiheen tai ongelman ratkaisu. [16; 18.]

4.2 GitLab ja Gitlab Runner

Versionhallinta on ohjelmistokehityksen keskeinen tekijä ja vaikka versionhallintajärjestelmiä on muitakin, niin tämän tutkimuksen puitteissa keskitytään GitLabiin. Sitä hallinnoi GitLab Inc, joka on listattu Yhdysvaltain teknologiapörssi Nasdaqiin. Ohjelmistona GitLab on monipuolinen ja sen kautta on mahdollista tehdä erilaisia versionhallinnan toimenpiteitä. Näitä ovat esimerkiksi projektien luominen, Git -komentojen käyttäminen, CI/CD-ohjelmistoputken luonti ja käyttäminen, muistiinpanojen teko ja erilaisten ohjelmistoprojektinhallinnan työkalujen käyttö. [17; 19.]

Ohjelmistokehityksen tarpeita ajatellen keskeisin GitLabin toiminto on projektikohtaisten repositoryjen käsittely. Tällöin tiedostoja voidaan siirtää päätelaitteen ja serverin välillä, minkä lisäksi repositoryn tiedostojen päivityksen yhteydessä voidaan ajaa CI/CD-ohjelmistoputken mukaiset toimenpiteet. Varsinaisesti toimenpiteet määrittää repositoryyn asetettu .gitlab-ci.yml -tiedosto, jonka asetusten perusteella saapunutta ohjelmistokoodia vaiheittain käsitellään. Tärkeä GitLabin ominaisuus on myös projektien jakaminen erilaisiin kehityshaaroihin, jolloin esimerkiksi uusia ominaisuuksia on mahdollista testata ilman pelkoa pääohjelman toiminnan häiriöistä. [19.]

Jatkuvaa integraatiota ja CI/CD-ohjelmistoputken käyttöä varten tarvitaan lisäksi GitLab Runner. Se on varsinaisen versionhallinnan rinnalla toimiva aputyökalu, jonka tehtävä on suorittaa versionhallintaan lisättyä ohjelmakoodia. On huomattava, että automaattinen versionhallinta vaatii toimiakseen tiettyjen ehtojen täyttymistä. Ensinnäkin projektille pitää olla määritetty käytettävissä oleva GitLab Runner, toiseksi ohjelmistoputken ja GitLab Runnerin käyttämä tunniste tulee olla identtinen ja kolmanneksi käyttäjän määrittämä .gitlab-ci.yml -tiedosto tulee olla kunnollinen. Ehtojen täytyttyä CI/CD-ohjelmistoputki toimii ja GitLab Runner tekee tehtävänsä. [19; 20.]

4.3 Visual Studio Code ja PlatformIO

Visual Studio Code (VS Code) on Microsoftin kehittämä ja ylläpitämä ilmainen ohjelmistonkehitystyökalu, joka mahdollistaa ohjelmoinnin useilla eri ohjelmointikielillä. Ilmaisen jakelun ohella VS Coden suosion syitä lienevät esimerkiksi monipuoliset toiminnot, asetusten muokattavuus, suuri määrä tuettuja lisäosia ja liitännäisiä sekä yhteensopivuus versionhallinnan kanssa. Täten lähdekoodin kirjoitus, suorittaminen ja versionhallinta yhdistyvät samaan ohjelmaan. [21.]

Runsaiden muokkausmahdollisuuksien seurauksena VS Coden käyttöönotto vaatii hieman ohjelmakohtaista asetusten säätöä. Tämän tutkimuksen puitteissa olennaisimmat lienevät liitännäinen C/C++-kielituelle, järjestelmän yhdistäminen GitLab-versionhallintaan ja PlatformIO-liitännäisen asennus, josta laajemmin seuraavassa kappaleessa.

PlatformIO on VS Coden liitännäinen, jonka avulla sulautettujen laitteiden kehittäminen on mahdollista. Myös PlatformIO tukee eri käyttöjärjestelmiä ja erilaisia toimintoja. Toiminnoista mainittakoon esimerkiksi, että asetusten määritys tapahtuu tiedostossa platformio.ini, jossa ohjelmointialustan asetukset ja käytetyt kirjastot määritellään. PlatformIO tukee myös ESP32-alustaa, minkä seurauksena ohjelmakoodi on mahdollista kasata, rakentaa ja ladata ohjelmointialustalle. [21; 22.]

4.4 Ohjelmistotestauksen työkaluja

Kuten tutkimuksessa aiemmin mainittiin, on ohjelmistolle mahdollista tehdä staattista ja dynaamista ohjelma-analyysiä. Käytännössä erilaisten ohjelmiston osien testaus vaatii oikeat työkalut, jotka valitaan ohjelmointikielen ja ohjelmointiympäristön perusteella. Työkalut voivat kohdistua esimerkiksi yksikkötesteihin, ylläpitoon, käytettävyyteen tai turvallisuuteen. Toimintoja on myös mahdollista liittää CI/CD-ohjelmistoputkeen, jolloin versionhallinnan tiedostojen muuttuessa työkalut käyvät läpi saapuneet tiedostot. Kyseisessä luvussa käydään esimerkinomaisesti läpi joitain C++-ohjelmointikieltä tukevia ei-kaupallisia ohjelmia, joita voidaan soveltaa myös MITYSens-olosuhdejärjestelmän toimintojen tarkkailuun. On tosin huomattava, että internetin myötä työkalujen ja ohjelmien suosio voi muuttua nopeasti, jolloin uudet ja kehittyneemmät työkalut voivat syrjäyttää vanhentuneet. [12.]

Yksikkötesteihin on mahdollista soveltaa esimerkiksi CppUnit-ohjelmaa, jonka avulla on mahdollista saada automatisoiduista testeistä tietoja tekstitiedostoina tai XML-muodossa. CppUnit on xUnit-yksikkötesteihin käytetyn JUnit-ohjelman versio, joka tukee C++-ohjelmointikieltä. On tosin huomattava, että CppUnit on päivittynyt viimeksi vuosia sitten, joten se on jokseenkin vanhentunut. Yleisemmin ohjelman toimintaa on mahdollista tutkia esimerkiksi Cppcheck-työkalua hyödyntäen. Käytännössä vikojen havaitsemiseen käytetty koodianalyysi etsii ohjelmistosta poikkeavaa toimintaa ja vaarallisia koodirakenteita. Lisäksi työkalun tavoitteena on havaita koodista vain todelliset virheet, eikä tuottaa turhia varoituksia. On kuitenkin huomattava, että myös Cppcheck on parhaimmillaankin vain täydentävä osa laadunvarmistusta. [12; 23; 24.]

Turvallisuusanalyysiin on puolestaan käytettävissä vaikkapa Flawfinder-työkalu. Käytännössä se tarkastelee koodia ja raportoi riskitason mukaan lajiteltuina mahdollisista tietoturvan heikkouksista. Täten Flawfinderistä voi olla hyötyä joidenkin haavoittuvuuksien ja tietoturvaongelmien löytämiseen ja poistamiseen ennen varsinaista ohjelman julkaisua. [12; 25.]

Edellä mainittujen lisäksi sekä kaupallisia että ei-kaupallisia ohjelmia on muitakin. Eräs näistä on jo aiemmin mainittu VS Code erilaisine liitännäisineen, josta esimerkiksi PlatformIOon on olemassa työkalu yksikkötestien tekoa varten. On kuitenkin muistettava, että automatisoitu koodianalyysi ei ainakaan vielä ole hopealuoti ohjelmiston validointiin. Eikä automaatio korvaa esimerkiksi huolellisen suunnittelun ja muun testauksen puutteita. Täten ohjelmiston toiminnan todentaminen on koneellisen käsittelyn sijaan edelleen lähes sataprosenttisesti seurausta ohjelmiston tekijöiden ja testaajien osaamisesta ja ammattitaidosta. [26.]

5 Mittaukset, jäljitettävyys ja validointi

5.1 Periaatteita ja huomioitavia seikkoja

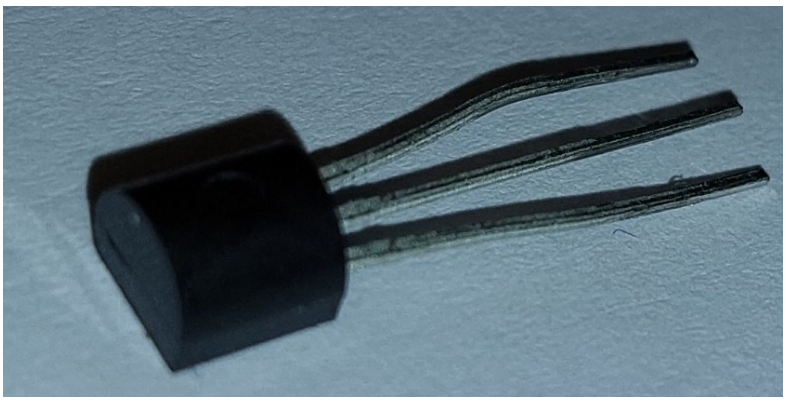
Kun mittalaitteita tarkastellaan, niin viime kädessä tullaan mittalaitteen tuottamiin arvoihin. Arkielämässä mittalaitteen tuottamiin arvoihin myös luotetaan varsin kritiikittömästi, sillä esimerkiksi kuumemittarin oletetaan automaattisesti näyttävän oikein. Jos oman mittalaiteen tuottamat arvot kuitenkin halutaan mahdollisimman varmennetuiksi, niin päädytään esimerkiksi todenmukaisuuden ja täsmällisyyden käsitteisiin. Tässä tapauksessa todenmukaisuudella tarkoitetaan sitä, että mittausten antama keskiarvo ja tulosten reaalinen arvo pitävät yhtä. Vastaavasti täsmällisyydellä tarkoitetaan sitä, että mittalaitteesta saatavat yksittäiset tulokset ovat lähellä toisiaan, eikä data-arvojen välillä mittaussarjan sisällä esiinny suurta heittoa. [27, s. 19–20.]

Modernin anturitekniikan kehittyessä ja ohjelmistojen jatkuvasti päivittyessä havaitaan kuitenkin, että koska antureilla voidaan mitata valtava määrä eri asioita, niin vastaavasti niiden antamien tulosten varmentaminen muodostuu ongelmalliseksi. Esimerkiksi Juha Hauhian luokkatiloja monitoroivan laitteen yhteydessä anturit mittaavat seuraavia arvoja: lämpötila, ilmankosteus, liike, hiilidioksidi ja valoisuus [2, s. 27–30]. Vastaavasti jos mittaaja haluaa todentaa, että esimerkiksi hiilidioksidin määrä on reaalisesti se, mitä anturi ilmoittaa, niin hänen pitäisi rakentaa monimutkainen mittausjärjestely. Mittausjärjestelyn kaksi keskeisintä seikkaa olisivat ympäristö, jossa hiilidioksidin määrää olisi mahdollista skaalata minimaalisesta määrästä maksimaalliseen sekä kalibroitu referenssimittari, jonka perusteella olisi mahdollista arvioida omaa mittalaitetta. Lisäksi jos kaikkia viittä mainittua arvoa haluaisi tarkastella, niin kukin tarvitsisi oman järjestelynsä.

Periaatteellisella tasolla kaikki tulokset ovat kuitenkin jäljitettävissä, kunhan hyödynnetään kalibroituja mittalaitteita. Tällöin SI-järjestelmässä määritellyt suureet ovat johdettavissa eritasoisiin kansainvälisiin ja kansallisiin mittanormaaleihin. Kansallisella tasolla jako tehdään lisäksi yleensä joko primäärinormaaleihin, sekundaarinormaaleihin tai käyttönormaaleihin. Tämän tutkimuksen puitteissa kiinnostus kohdistuu lähinnä loppukäyttäjän suorittamien anturilla tehtävien mittauksen vertailuun yritysten käyttönormaalin välillä, mutta periaatteessa kalibroitujen mittalaitteiden avulla jäljitettävyysketju on rakennettavissa aina kansainväliselle tasolle saakka. [28, s. 15–17.]

5.2 Lämpötilan mittaaminen

Tutkimuksen rajauksen ja relevantin esimerkin havainnollistamiseksi luvussa käsitellään lämpötilan mittausta ja Dallasin DS18B20-lämpötila-anturia. Kyseisen anturin ominaisuuksiin kuluu esimerkiksi parhaimmillaan 12 bitin tarkkuudella tuotettujen lämpötila-arvojen mittaaminen -55°C – +125°C vaihteluväliltä, jolloin tutkimuksen todennäköiset mittausolosuhteen arvot rajautuvat vaihteluvälille -30°C – +30°C astetta. Datalehden arvojen mukaisesti mittaustarkkuus on ±1°C astetta vaihteluvälillä -30°C – -11°C astetta ja ±0,5°C astetta välillä -10°C – +30°C astetta. Täten datalehdessä määritellyt arvot asettavat maksimivirheen, jonka rajoissa ehjä ja toimiva anturi toimii. Lisäksi on huomattava, että anturin hyödyntäminen yli tuhannen käyttötunnin ajan voi aiheuttaa tuloksissa ±0,2°C siirtymän. Kuvassa 3 näkyvään ja 1-Wire-väylään asennettavan laitteen fyysiset mitat ovat 4×3.5×19 millimetriä. [29.]



Kuva 3. Dallasin DS18B20-lämpötila-anturi.

Mittalaitteen ja siihen kytketyn anturin tuottamia arvoja on mahdollista kokeilla eri tavoin. Joko virtuaalisesti, jolloin ohjelmisto tuottaa anturille simuloidun arvon, tai reaalisesti, jolloin anturilla mitataan esimerkiksi olosuhdekaapin tuottamia lämpötiloja. Olosuhdekaapilla voidaan esimerkiksi luoda mittausjärjestely, jossa ilmankosteus on vakioitu ja testin aloituslämpötila on -30°C astetta ja lämpötila nousee tunnin välein +10°C astetta. Täten seitsemän tunnin mittaus kartoittaa koko vaihteluvälin -30°C – +30°C astetta. Lisäksi mittausten ja todentamisen kannalta on olennaista, että anturin tuottamia arvoja voidaan yhtäaikaisesti verrata kalibroidun referenssimittarin tuottamiin arvoihin. [30, s. 12–16.]

5.3 Mittalaitteen validointi ja mittausten kaupalliset validointipalvelut

Vaikka tässä tutkimuksessa keskitytään lähes täysin ohjelmiston toiminnan tarkasteluun, niin on silti havaittava ero mittalaitteen ohjelmakoodissa ja sen tuottamissa tuloksissa. Toisin sanoen voidaan kysyä, toimiiko täydellinen mittalaite ohjelmistonsa puolesta oikealla tavalla vai tuottaako mittari aina mahdollisimman oikeita arvoja. Voi nimittäin olla tilanne, jossa ohjelmisto toimii halutusti, mutta mittalaitteen vääränlainen kalibrointi romuttaa saadut tulokset. Tämän lisäksi on huomioitava, että onnistuneen mittausjärjestelmän kannalta pelkkä mittalaite ja anturi eivät muodosta koko järjestelmää. Koska järjestelmään liittyy myös tiedonsiirtoa, tallennusta ja käsittelyä. Jossain tapauksessa esimerkiksi verkkoyhteyksien katkeaminen saattaa katkaista anturidatan saapumisen, vaikka ohjelmisto ja mittalaite toimisivat muuten suunnitellusti.

Mittalaitteen luotettavuuden ja validoinnin osalta päädytään lisäksi matemaattisiin käsitteisiin, kuten mittausepävarmuuteen. Esimerkiksi kun mittauksia suoritetaan ja validoidaan, niin tarvitaan kalibraattoria, joka lähettää tietyn määrän pulsseja ajan funktiona. Vastaavasti kun mittalaitteen ja kalibraattorin tuloksia verrataan keskenään, niin molemmissa laitteissa tulisi olla sama määrä lähetettyjä ja mitattuja pulsseja. Tämän lisäksi mittausaika pitää olla riittävä ja mittaus tulee toistaa useita kertoja. Kun saadut arvot taulukoidaan, niin niiden perusteella voidaan laskea muuttujia, kuten kattavuuskertoimella kaksi oleva standardiepävarmuus. Sen avulla voidaan puolestaan tarkastella mittarin totuudenmukaisuutta ja luotettavuutta. Varsinaisesti matemaattinen tarkastelu vaatisi kuitenkin osakseen oman tutkimuksensa, joten tässä yhteydessä riittänee toteaminen näkökulman olemassaolosta. [27, s. 36; 31, s. 11–14; 32.]

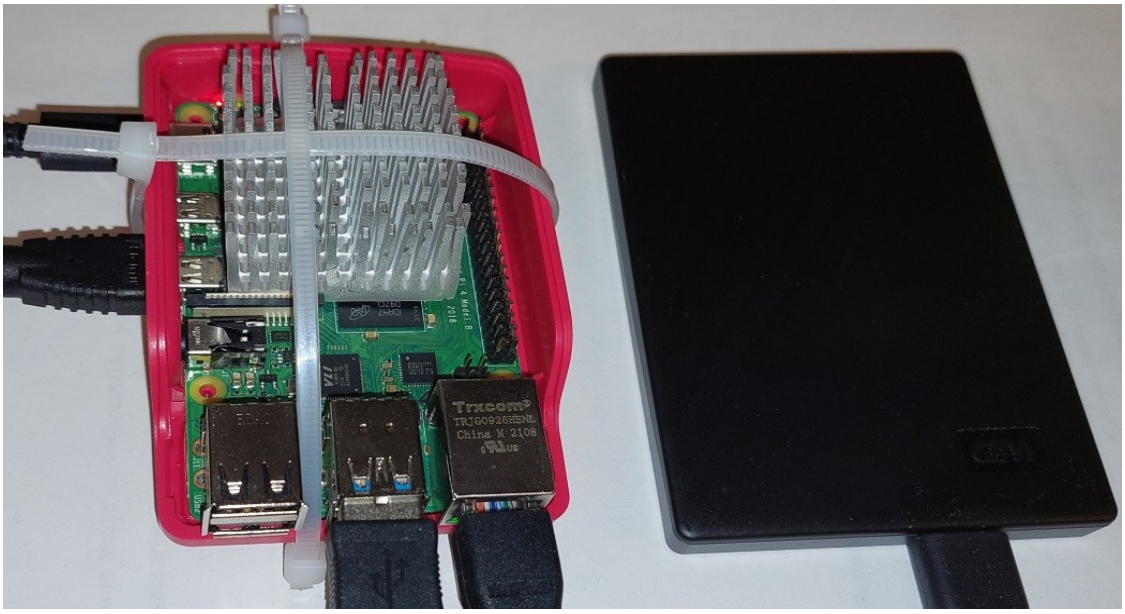
Luotettavien mittalaitteiden, työkalujen ja olosuhteiden vaatimukset ovat myös luoneet markkinat erilaisille palveluille. Suomessa tunnetuin kalibrointilaitteita valmistava yritys lienee Beamex, jonka tuotteita käytetään globaalisti tuhansissa yrityksissä. Sen sijaan validointipalvelua tarjoavat esimerkiksi Kiwa, Vaisala ja VTT MIKES, joista kukin tarjoaa muiden palveluidensa ohella ratkaisunsa myös lämpötila-arvojen tarkistamiseen ja varmentamiseen. Usein arvojen oikeellisuus ei myöskään ole vapaaehtoista, vaan erilaisilla mittareilla, pumpuilla, vaaoilla ja muilla laitteilla on lakisääteinen velvoite tuottaa paikkansa pitäviä arvoja. Erään esimerkin tarkasta aihepiirikohtaista säädöksestä ja palveluratkaisusta tarjoaa Kärsämäellä toimiva Haka-Kone Oy, joka tarjoaa palvelua standardin IEC EN 60974-14 mukaiseen kaarihitsauslaitteiden kalibrointiin ja validointiin. [33; 34; 35; 36; 37.]

6 Mittalaitteen ohjelmistosta ja testausympäristöstä

6.1 Testausympäristön pystyttäminen

Mittalaitteen ohjelmiston varmennus vaatii toimiakseen CI/CD-ohjelmistoputken, jonka toteutus valittiin tehtäväksi GitLab-versionhallintapalvelussa [19]. Tämän jälkeen selvitettiin vaihtoehdot, joiden mukaisesti järjestelmä voitiin joko rakentaa itse tai hankkia ostopalveluna. Parempien säätömahdollisuuksien vuoksi palvelinympäristö päädyttiin rakentamaan ja asentamaan itse. Alla olevissa kappaleissa on hyvin typistetysti eritelty vaiheet, joiden mukaisesti asennustoimenpiteet tehtiin.

Asentamisen ensimmäinen vaihe oli Raspberry Pi 4B -kämmentietokoneen (jatkossa raspi), USB-kiintolevyn ja muiden osien tilaaminen ja kasaaminen. Toisessa vaiheessa raspille asennettiin käyttöjärjestelmäksi Ubuntu Server 20.04 LTS, jolloin raspin käynnistyminen USB-kiintolevyltä ja nettiyhteyksien asetus vaativat toimiakseen pienet säädöt. Fyysisten osien kokoonpano havaitaan kuvasta 4.



Kuva 4. Raspberry Pi 4B, jossa siili viilennystä varten. Kuvassa myös ulkoinen kiintolevy.

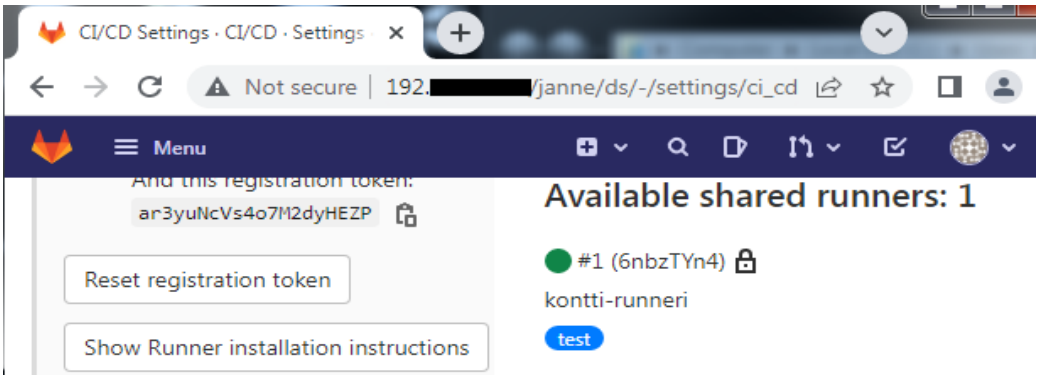
Kolmannessa vaiheessa käyttökuntoiselle kämmentietokoneelle asennettiin konttiohjelma Docker, joka nähtiin tarpeellisena, jotta toiminnalliset vaatimukset on mahdollista saavuttaa [16]. Palveluiden asennus kontteihin mahdollistaa myös palvelimen hallitun käynnistämisen ja tarvittaessa alasajon, jolloin palvelimen toiminta on ennustettavaa ja vakaata.

Neljännessä vaiheessa Docker-konttiin asennettiin GitLab-versionhallintapalvelu, jossa olennaisin havainto oli käyttää raspille yhteensopivaa versiota arm64v8 [38; 39]. GitLabin käynnistyminen palvelimella mahdollisti myös samassa lähiverkossa olevan tietokoneen yhdistämisen palveluun, jolloin selainriville syötetty IP-osoite ja käyttäjätunnuksen sekä salasanan syöttäminen mahdollistivat pääsyn palveluun. Tällöin myös normaalit versionhallintatoimenpiteet, kuten repositoryjen kopiointi ja siirto, joko palvelimelta etäpäätteelle tai päinvastoin, tulivat HTTP-yhteyden avulla mahdolliseksi. Viidennessä vaiheessa toiseen konttiin asennettiin GitLab Runner, jonka tehtävä on käydä läpi GitLabiin lisättyjä tiedostoja [40]. Serverillä ajossa olevat kontit havaitaan kuvasta 5.

```
ubuntu@ubuntu:~$ docker ps
CONTAINER ID   IMAGE                              COMMAND                  NAMES
8e0dd81a75c5   gitlab/gitlab-runner:latest        "/usr/bin/dumb-init ."   gitlab-runner
09ff0b50db5b   yrzr/gitlab-ce-arm64v8:latest      "/assets/wrapper"        gitlab-ce      (healthy)
ubuntu@ubuntu:~$
```

Kuva 5. Palvelimella ajettavat Docker-kontit. (Muuttujat CREATED, STATUS ja PORTS on editoitu selkeyttämisen vuoksi kuvasta pois.)

Kuudennessa vaiheessa GitLab Runner rekisteröitiin GitLabiin, jolloin CI/CD-ohjelmistoputken käyttö tuli mahdolliseksi [40]. Edellä mainittujen kuuden työvaiheen jälkeen ympäristö on pystytetty ja käyttäjän on mahdollista lähettää ohjelmakoodia palvelimelle testattavaksi. GitLabin toimivuus ja GitLab Runnerin käyttökelpoisuus havaitaan kuvasta 6.



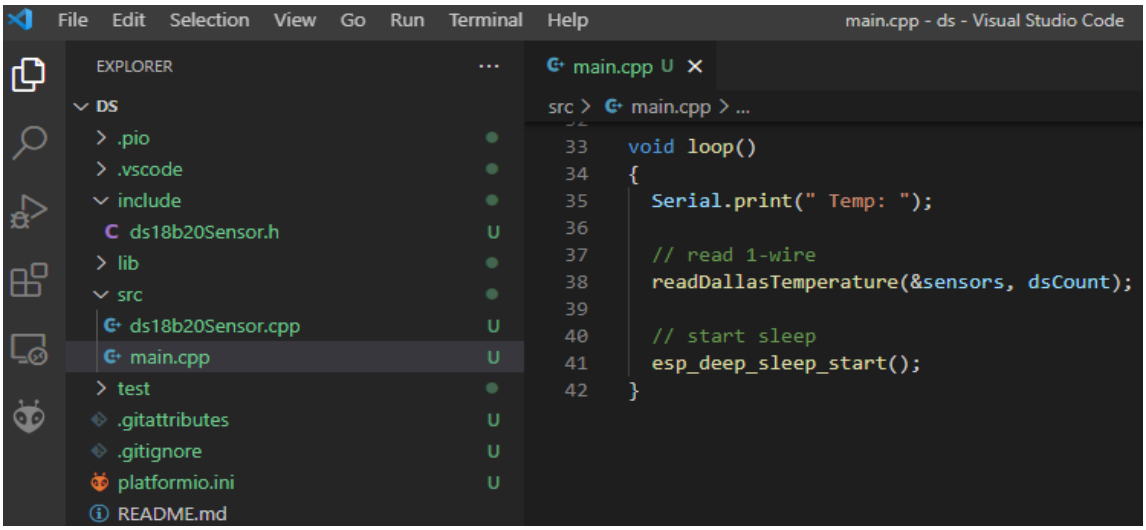
Kuva 6. Raspberry Pi 4B:lle asennettu GitLab ja GitLab Runner.

Yllä olevasta kuvasta on olennaista huomata, että IP-osoitteen ja käyttäjätunnuksen asettamisen sekä repositoryn luonnin jälkeen päästään käsiksi kansioon ds. Lisäksi GitLab Runner, joka on tässä tapauksessa nimeltään kontti-runneri, on käytettävissä kaikkiin projekteihin, jotka käyttävät tunnisteenaan sanaa test. Varsinaisesti CI/CD-ohjelmistoputken määrittämiseen käytetään kuitenkin kunkin repositoryn juuressa olevaa tiedostoa .gitlab-ci.yml, jonka määrittelemien asetusten mukaisesti GitLab Runner aktivoituu ja ohjelmistoputki toimii.

6.2 Mittalaitteen ohjelmistosta

Mittalaite koostuu käytetyistä osista, niiden vaatimista kytkennöistä ja mittalaitetta varten räätälöidystä ohjelmistosta, jolloin periaatteena on yhdistää pääohjelmaan erilliset anturien vaatimat aliohjelmat. Tässä tutkimuksessa käytössä ovat ESP32 ja DS18B20, joiden käyttämä ohjelmisto koostuu pääohjelmasta main.cpp sekä anturin vaatimien tiedostojen ds18b20Sensor.h ja ds18b20Sensor.cpp sisällöistä [Liite 5]. Lukijan on myös huomattava, että esimerkin kohteena oleva ohjelma on hyvin yksinkertainen ja pelkistetty. Kuitenkin ESP32 muistiin ladattuna se tuottaa DS18B20-anturin mittaamia lämpötiloja.

Pääohjelman main.cpp tarkoituksena on ottaa käyttöön vaadittavat määritykset, asetukset ja ajaa ohjelmakoodia, kunnes tulee käsky ohjelman keskeytyksestä tai virta katkaistaan. Määrityksiin kuuluu esimerkiksi sen pinnin asettaminen, jolla DS18B20 dataliikenne hoidetaan. Asetukset puolestaan määrittävät sen, että järjestelmä tunnistaa kunkin väylässä olevista lämpötila-antureista. Tässä tapauksessa varsinaisessa ohjelmakierrossa printataan teksti, luetaan anturin tuottamia arvoja ja asetetaan ESP32 virtaa säästävään nukkumistilaan, josta laite herää, kun haluttu ajanjakso on kulunut. Kyseinen toiminto havaitaan myös kuvasta 7.



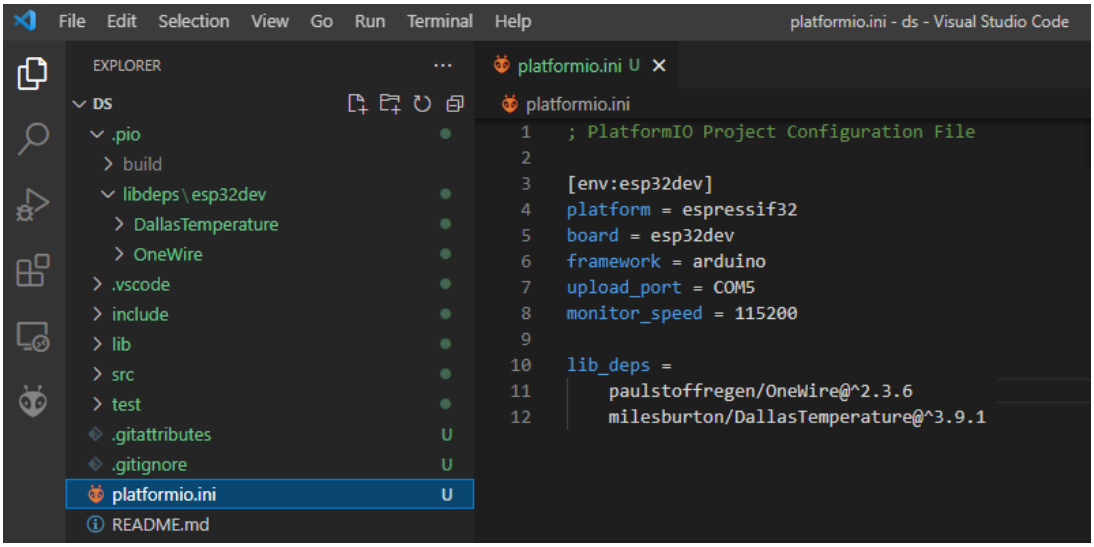
Kuva 7. Visual Studio Code -näkymä, kansion ds-rakenne ja osa tiedostosta main.cpp.

Kuvasta 7 havaitaan myös ohjelmistorakenteen jakautuminen eri kansioihin, joista tässä tapauksessa ovat käytössä kansiot include ja src. Kansioista löytyvät lisäksi tiedostot ds18b20Sensor.h ja ds18b20Sensor.cpp, joiden tehtävänä on tuottaa toiminnot DS18B20-anturien tunnistamiseksi ja yhden anturin lukemiseen. Rakenteellisesti erillisiä header -tiedostoja tarvitaan esimerkiksi funktioiden alustamista ja käyttöä varten, mikä parantaa ohjelman rakennetta ja helpottaa käsittelyä.

6.3 Mittalaitteen määritykset ja kirjastot

Mittalaitteen ohjelmisto vaatii toimiakseen myös muita määritteitä, kuin edellisessä luvussa mainitut kolme tiedostoa. Tässä tapauksessa määritykset löytyvät ensisijaisesti tiedostoista platformio.ini sekä OneWire.h ja DallasTemperature.h. On myös huomattava, että luvussa käsitellyissä asioissa on ensisijaisesti kyse periaatteesta. Täten halutut toiminnot ja käytetyt osat vaativat aina toimiakseen sekä oikeanlaisen kansiorakenteen että tarpeelliset tiedostot, jolloin erilaisten luokkien ja funktioiden käyttö tulee mahdolliseksi.

Tutkimuksessa tarkasteltavan mittalaitteen toiminnan kannalta olennaisin tiedosto lienee platformio.ini, joka mahdollistaa ohjelman rakentamisen ja lataamisen ESP32:lle. Kyseinen tiedosto määrittää siten halutun käyttöalustan, hyödynnettävät kirjastot sekä tarvittaessa tietokoneen ja ESP32 välisen USB-yhteyden asetukset. Kuvasta 8. havaitaan esimerkiksi, että käytetty alusta on espressif32, kortin nimi on esp32dev ja porttina toimii tässä tapauksessa COM5.



Kuva 8. Visual Studio Code -näkymä muista määrityksistä, jossa avoinna tiedosto platformio.ini.

Kuten kuvan 8 platformio.ini kohdasta lib_deps havaitaan, niin PlatformIOon on lisätty kaksi kirjastoa. Tämä havaitaan tiedostopolusta .pio\libdeps\esp32dev, josta löytyvät tiedostot DallasTemperature\DallasTemperature.h ja OneWire\OneWire.h. Kyseiset kirjastot ja niiden sisältö ovat tarpeellisia DS18B20-anturin vaatiman 1-Wire-väylän käyttämiseksi ja ESP32 vastaanottaman signaalin muokkaamiseksi ymmärrettäviksi toiminnoiksi ja lämpötila-arvoiksi. Toki on muistettava, että pätevä sulautettujen laitteiden ohjelmoija ei käytä valmiita kirjastoja, vaan tekee haluamansa toiminnot itse. Aina se ei kuitenkaan ole osaamisen tai työekonomisten syiden takia mahdollista. [41; 42.]

7 Yksikkötestit PlatformIOn avulla

7.1 Testien lähtökohdista ja toteutuksesta

Yksikkötesteillä pyritään laadunvarmennukseen ja siihen, että jokin ohjelmiston osa todennetusti toimii. Tässä luvussa PlatformIOta käytetään kahden esimerkkitapauksen suorittamiseen, joiden avulla havainnollistetaan yksikkötestien käyttömahdollisuuksia ja mallia testien toteuttamiseksi. Mainittakoon myös, että yksikkötestien soveltaminen ei onnistu pelkkien Liitteessä 5 eriteltyjen ohjelmakoodien avulla, vaan testit vaativat omanlaisensa kansiorakenteen ja suoritettavat tiedostot [Liite 5]. Käytännössä tiedostojen repositoryyn tarvitaan kansio test, jossa suoritettavien testien kansiot ja itse testitiedostot ovat. Muutama esimerkki kansiorakenteen havainnollistamiksi havaitaan myös Liitteiden 6 ja 7 vasemmasta laidasta [Liite 6; Liite 7].

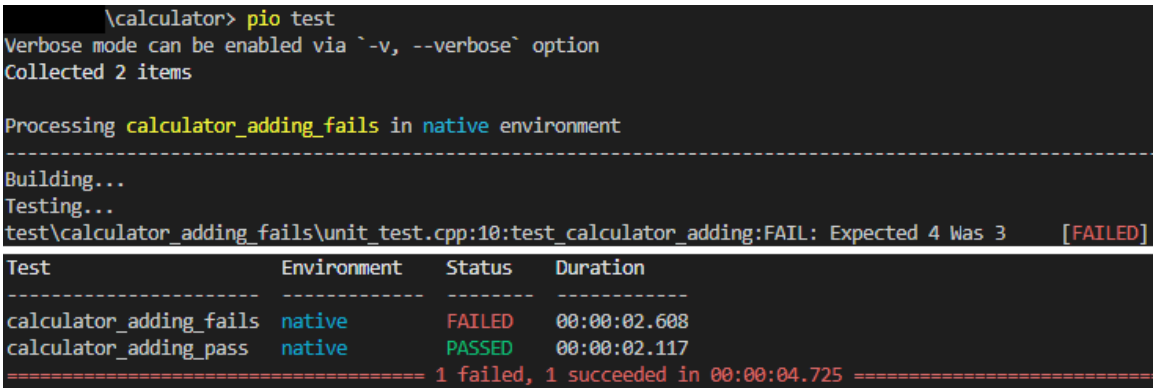
Toteutuksen suhteen PlatformIOn yksikkötestien kulmakivenä toimii kirjasto unity.h, joka mahdollistaa erilaiset funktiot. Funktioista tärkeimmät lienevät UNITY_BEGIN() ja UNITY_END(), joiden väliin varsinaiset yksikkötestit asetetaan. Yksikkötestien funktiot puolestaan ajetaan komennolla RUN_TEST(), jonka sisällä määrättyjä ehtoja testataan. Ehdoille asetettavat funktiot voivat puolestaan olla esimerkiksi muotoa TEST_ASSERT(condition), joka asettaa varsinaisen suoritusrajan funktion hyväksymiselle tai hylkäämiselle. Esimerkit toimintatavasta käydään tarkemmin läpi kahden seuraavan alaluvun puitteissa, mutta Liitteitä 6 ja 7 on mahdollista tarkastella myös periaatteen ja käytännön erojen tarkasteluun. Tällöin suurin poikkeavuus mallina käytettyyn kirjastoon lienee funktion TEST_ASSERT ehtojen asettaminen erillisiksi aliohjelmiksi, jolloin ohjelmakoodin käsittely ja muokkaus ovat hieman yksinkertaisempia. [26; 43; Liite 6; Liite 7.]

7.2 Yksikkötesti calculator

Luvussa tarkastellaan hyvin pelkistettyä yksikkötestiä calculator, jonka tavoitteena on havainnollistaa yksikkötestien rakennetta, toteutusta sekä testiympäristön määritystä. Mainittakoon heti alussa, että yksikkötestit voidaan toteuttaa erilaisissa testiympäristöissä. Tässä yhteydessä käytetään esimerkin vuoksi esp32dev-ympäristön sijaan ympäristöä native, joka hyödyntää alustariippumatonta ohjelmointikielen kääntäjää [26]. Native-ympäristön käyttö on mahdollista, koska testi ei tässä tapauksessa vaadi käyttöönsä ulkoisia ohjelmointirajapintoja. Tiedoston platformio.ini sisältö havaitaan kuvasta 9.

Kuva 9. Muista tutkimuksen sisällöistä poiketen platformio.ini-ympäristö ei ole esp32dev.

Varsinaisessa yksikkötestissä verrataan, vastaako aliohjelman tuottama arvo etukäteen määrättyä referenssiarvoa. Liitteestä 6 havaitaan, että adding() aliohjelmaan sijoitetaan esimerkiksi luvut kaksi ja kaksi [Liite 6]. Niitä puolestaan verrataan haluttuun arvoon neljä, joka on asetettu aliohjelman test_calculator_adding() sisällä funktiossa TEST_ASSERT_EQUAL(). Yksikkötestin toiminnan tarkastelua varten on myös hyödyllistä luoda sekä testin läpäisevä että virheen tuottava ratkaisu. PlatformIOn komentokehotteesta tulostettu raportti havaitaan kuvasta 10.



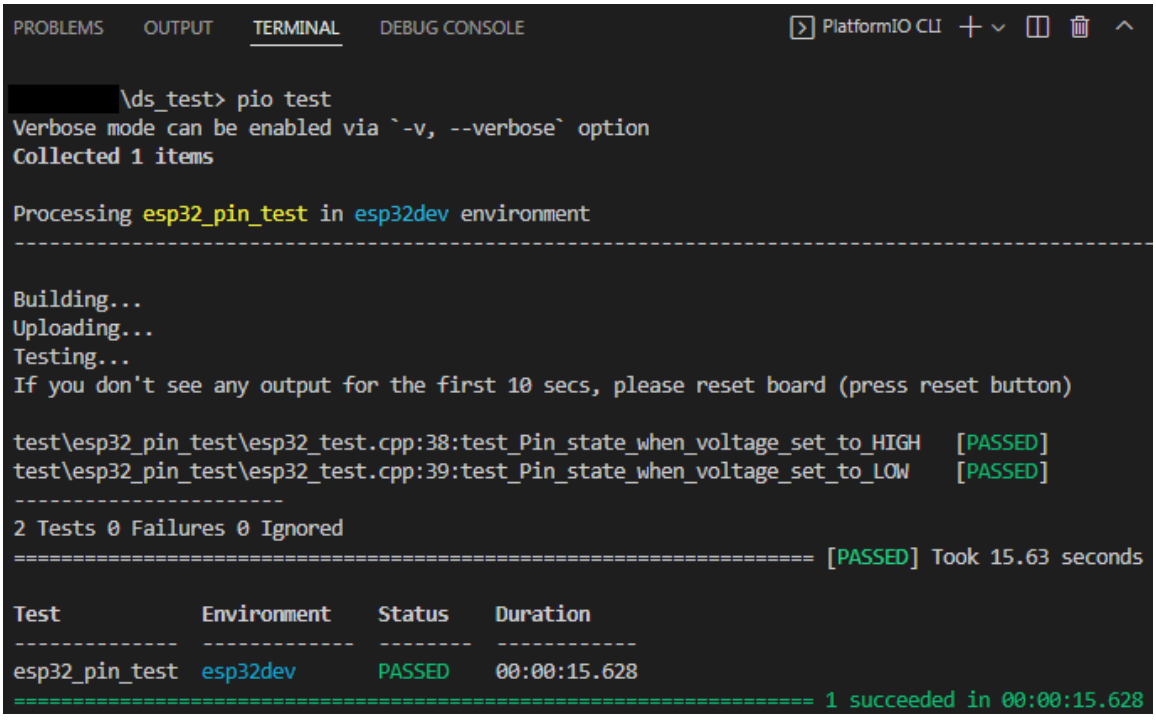
Kuva 10. PlatformIOn terminaalista kopioidusta raportista muokattu kuva calculator_adding_fails ja calculator_adding_pass -yksikkötesteistä.

Kuvasta 10 voidaan päätellä, että kahdesta yksikkötestistä toinen on onnistunut ja toinen on epäonnistunut. Epäonnistuneesta testistä voidaan myös havaita, että testissä odotettu arvo on neljä ja aliohjelman tuottama arvo on kolme. Lisäksi testin suorittamiseksi käytetään ympäristöä native, jolloin toiminnallisuuden varmistamiseksi olennaisinta on GCC-ohjelmointikielen kääntäjän olemassaolo [44]. Ajallisesti kahden yksikkötestin suorittaminen vei noin viisi sekuntia.

Yhteenvetona luvun sisällöstä voidaan todeta, että unity.h-kirjastoa hyödyntäen VS Codella ja PlatformIOlla on mahdollista suorittaa yksikkötestejä. Kaikki testattavat asiat täytyy kuitenkin suunnitella, toteuttaa ja suorittaa tapauskohtaisesti, jolloin testiraportit paljastavat ohjelmiston toimintatavat. Ajoittain myös epäonnistuneista testeistä voi saada hyödyllistä tietoa, esimerkiksi vertailemalla raporteista, kuinka ohjelmisto reaalisesti toimii ja miten sen pitäisi toimia. On myös huomattava, että muista tutkimuksen esimerkeistä poiketen ohjelmiston rakenne ei muodostu setup()- ja loop() -osioista, vaan ohjelmakoodi suoritetaan kokonaisuudessaan main()-funktiossa.

7.3 Yksikkötesti ESP32-pinnien toiminnasta

Oheisessa luvussa tarkoituksena on dynaamisen testin avulla tutkia, toimiiko ESP32 halutulla tavalla vai ei. Käytetty ohjelmakoodi on Liitteeseen 7 kopioitu esp32_test.cpp, jonka tarkoituksena on yhden lähtöpinnin tilan seuraaminen [Liite 7]. Käytännössä tämä tapahtuu siten, että lähtöpinni asetetaan joko tilaan HIGH (1) tai LOW (0) ja vastaavasti tulopinni seuraa tilojen muutoksia. Jos tulopinni rekisteröi lähdön muutoksen yksikkötestin edellyttämällä tavalla, eli lähtöpinnin tila HIGH rekisteröi tulopinnissä tilan HIGH. Tai jos lähtöpinnin tila LOW rekisteröi tulopinnissä tilan LOW, niin testi menee läpi ja tulostuu teksti PASSED. Vastaavasti jos syystä tai toisesta tilat eivät ole yhtenevät, niin testi epäonnistuu ja tulostuu teksti FAILED. PlatformIOn komentokehotteen kautta tuotettu ja pio test -komennolla koostettu raportti havaitaan kuvasta 11.



Kuva 11. PlatformIOn terminaalista kopioitu esp32_pin_test yksikkötestin raportti.

Kuvasta 11 voidaan päätellä muutamia seikkoja. Ensinnäkin testi on onnistunut, koska lähtöpinnin ja tulopinnin arvot ovat identtiset molemmissa testitapauksissa. Lisäksi voidaan havaita, että testin suorittamiseksi tarvitaan ympäristöä esp32dev, jonka määrityksiä avattiin aiemmin kuvassa 8. Edelleen havaitaan, että testin koostaminen ja läpäisy veivät jonkin verran aikaa, sillä suoritukseen kului yli 15 sekuntia.

Yhteenvetona voi todeta, että yksikkötestin perusteella ESP32 toimii ohjelman kasaamisessa ja kyseisten pinnien tarkastelussa, kuten kuuluukin. Täten sitä on mahdollista hyödyntää vaikkapa jonkin tietystä pinnistä anturille syötettävän käyttöjännitteen tarkkailuun. Toisaalta jos vastaavaa järjestelyä on tarkoitus hyödyntää jonkin mittalaitteen tarkoitusperiä ajatellen, niin asia pitää huomioida jo piirilevyä suunniteltaessa. Tämä siitä syystä, että ohjelman toiminnan onnistumiseksi piirilevyssä tulee olla joko mahdollisuus tilapäiseen kytkentään tai kiinteä johdin testauksen suorittamista ajatellen. Lisäksi ESP32 täytyy olla käytettävissä testijärjestelyssä, joka sekä hankaloittaa testien tekoa että hidastaa niiden läpimenoaikaa.

7.4 Pohdintaa yksikkötesteistä

Yksikkötestien suunnittelussa ja toteutuksessa on olennaista tuntea ohjelmiston toimintaperiaate sekä panostaa testaukseen järjestelmän kriittisimmissä osissa. Tämä vaatii kuitenkin Liitteessä 4 eriteltyä testauksen lähestymistapaa sekä ohjelmiston arviointia [Liite 4]. On olennaista ymmärtää, mitä eri ohjelmiston aliohjelmat ja funktiot tekevät, kuinka ne toimivat ja miten niiden toimintaa tulisi testata. Aiemmissa luvuissa esiteltiin kaksi PlatformIOlla suoritettua testitapausta, mutta ne toimivat lähinnä havainnollistavina esimerkkinä käytännön toteutuksesta. Ohjelmistokehityksen iteratiivisen luonteen ja mittausjärjestelmän modulaarisen rakenteen vuoksi useiden tarkasti räätälöityjen yksikkötestien avaaminen ei myöskään palvele tutkimuksen tarkoitusta.

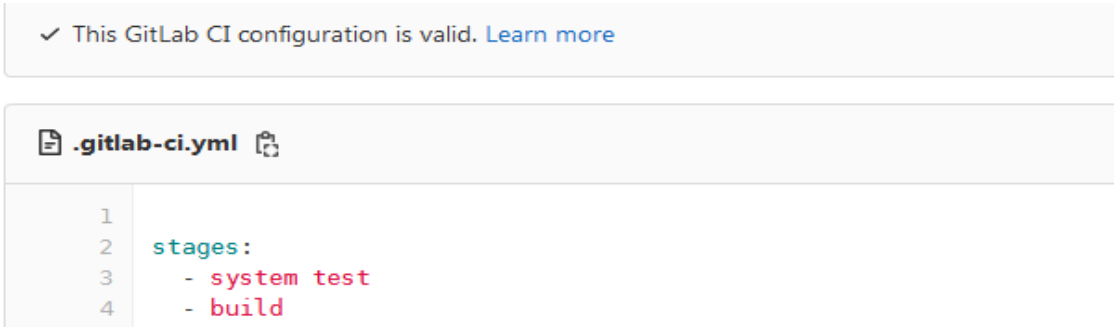
Lisäksi on todettava, että C++ yksikkötestejä on mahdollista toteuttaa useilla työkaluilla. Unityn ohella tunnetuimmat lienevät CppUTest ja Google Test, joista jälkimmäistä Samuli Mononen tarkastelee vuonna 2020 ilmestyneessä opinnäytteessään. Tämän lisäksi saadakseen yksikkötesteissä koodikattavuuden sataan prosenttiin, tarvitaan erilaisille ohjelman osille simuloituja arvoja tuottavia koodirakenteita. Englanniksi näistä käytetään termejä fake, stub ja mock, joiden käyttämiseksi puolestaan tarvitaan omat työkalunsa kuten CppUMock tai gMock. [45; 46.]

Luvussa on käsitelty päätelaitteella tehtyjä ja PlatformIOn avulla suoritettuja yksikkötestejä. Testejä on mahdollista hyödyntää esimerkiksi tarkasteltaessa, tarjoaako rajapinta tietyssä formaatissa halutun tiedon tai onko luettu arvo oikeaa suuruusluokkaa. Yksikkötestit on mahdollista suorittaa myös alustariippumattomasti joko päätelaitteella tai Docker-kontissa, mutta toisaalta tämä edellyttää erilaisia arvoja simuloivien rakenteiden käyttöä. Varsinaisen testin suorittamisen lisäksi saadut raportit tulee tallentaa, arkistoida ja hyödyntää ohjelmistokehityksen tarpeita ajatellen. Raportit ovat myös hyödynnettävissä osana ohjelmistojen validoinnin perusaineistoa.

8 Palvelimella tehtävät toimenpiteet

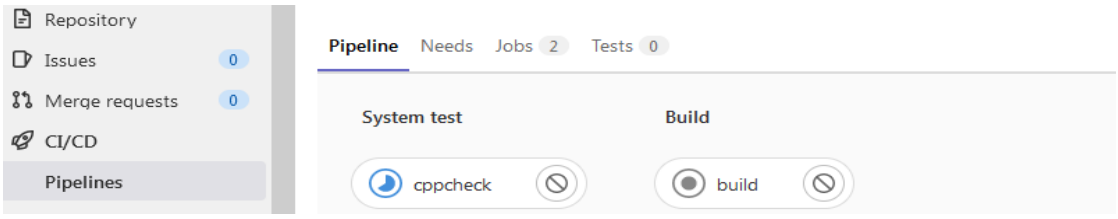
8.1 Lähtötilanne ja työvaiheiden määrittäminen

Edellisessä luvussa eriteltyjen yksikkötestien suunnittelu ja toteutus eivät testaa koko järjestelmään. Täten ne eivät yksistään riitä yksiselitteisen ja tyhjentävän ratkaisun tuottamiseksi, kun haetaan vastausta tutkimuksen aiheeseen, joka on edelleen ohjelmistojen validointi mittausjärjestelmissä. Lisäksi koko järjestelmää ja sen logiikkaa tarkastelevia testejä suoritetaan tässä luvussa päätelaitteen sijaan palvelimella. Tarkastelussa eritellään kahta esimerkkiä, joiden tarkoituksena on havainnollistaa GitLabissa suoritettavaa toimintatapaa. Molemmissa esimerkkitapauksissa työvaiheiden kulmakivenä on määritystiedosto .gitlab-ci.yml. Kuvasta 12 havaitaan tiedoston sisältöä sekä siinä asetetut ohjelmistoputken työvaiheet ja työjonon kulku.



Kuva 12. CI/CD-ohjelmistoputken toiminta määritetään tiedostossa .gitlab-ci.yml.

Kuvan mukaisesti työvaiheet ovat system test ja build. On myös huomattava, että GitLabin automatiikka ilmoittaa tiedoston rakenteen oikeellisuuden. Jos tiedosto on määritelty väärin, ilmestyy näytölle raksi ja teksti invalid. Vastaavasti jos tiedosto on määritelty oikein, ilmestyy ruudulle kuvassa 12 nähtävä teksti valid. Repositoryn tiedostojen päivittyessä versionhallinnan näkymä ilmenee kuvasta 13. Kuvia vertailemalla havaitaan myös, että ohjelmistoputken automatiikka suorittaa järjestyksessä määrittämämme työvaiheet, joista tarkemmin seuraavissa luvuissa.



Kuva 13. GitLab-ohjelmistoputken näkymä toiminnan ollessa kesken.

8.2 Testaus Cppcheckin avulla

Käyttäjän lisätessä tai muokatessa versionhallinnan tiedostoja suoritetaan määritetyt toimenpiteet. Kuten aiemmin todettiin, ohjelmistoputken määritysten mukaisesti CI/CD-työjonon ensimmäinen vaihe on Cppcheck. Käytännössä tämä tarkoittaa, että pystytetään Docker-kontti, joka voidaan toteuttaa esimerkiksi kuvan 14 mukaisesti.

```
5  cppcheck:
6    stage: system test
7    tags:
8      - test
9    image: ubuntu:latest
10   script:
11     - apt-get update
12     - apt-get install -y cppcheck
13     - chmod +x ci/cppcheck.sh
14     - ci/cppcheck.sh
15   artifacts:
16     paths:
17       - cppcheck
```

Kuva 14. Cppcheckin määrittäminen osana .gitlab-ci.yml -tiedostoa.

Kuvasta havaitaan kontin pystyttämisen rakenne. Olennaista on esimerkiksi käyttää haluttua GitLab Runneria, jonka asetukset havaittiin luvun 6.1 kuvasta 6 ja jota voidaan nyt hyödyntää käyttämällä tunnistetta test. Lisäksi konttiin on olennaista pystyttää Linux-ympäristö muuttujien image ja script avulla sekä tallentaa Cppcheckin raportit muuttujan artifacts avulla. Varsinainen kontissa suoritettava ohjelmakulku havaitaan kuvasta 15.

```
1   Running with gitlab-runner 14.4.0 (4b9e985a)
2     on kontti-runneri 6nbzTYn4
214 Checking include/ds18b20Sensor.h ...
215 1/3 files checked 20% done
216 Checking src/ds18b20Sensor.cpp ...
217 2/3 files checked 48% done
218 Checking src/main.cpp ...
219 3/3 files checked 100% done
248 Job succeeded
```

Kuva 15. Cppcheckin suorittamisen vaiheita palvelimen Docker-kontissa.

Kuvan 15 mukaisesti kontti-runneri hoitaa Cppcheckin pystyttämisen ja repositoryn tiedostojen läpikäynnin. Käytännössä tämä tarkoittaa kolmea, Liitteessä 5, eriteltyä tiedostoa [Liite 5]. On tosin huomattava, että Cppcheck ei tuota hyödyllistä informaatiota, jos ohjelmakoodi on rakenteellisesti kunnossa. Vastaavasti jos Liitteen 8 mukaisesti ohjelmakoodiin luodaan rakenteellinen

virhe, eli anturien määrää tutkiessaan aliohjelma palauttaa aina vakioarvon nolla [Liite 8]. Päädytään tulokseen, että vaikka ohjelmakoodi on syntaksin osalta kunnossa, niin Cppcheck havaitsee virheen ja tuottaa raportin käyttämättömästä muuttujasta.

Luvussa havaitaan käyttötapa eräästä staattisen analyysin työkalusta. Edelleen havaitaan, että tiedoston .gitlab-ci.yml määritysten ollessa kunnossa ohjelma menee aina läpi. Lopputulos ei tällöin ole niinkään ohjelmakoodin välitön hyväksyminen tai hylkääminen, vaan staattista analyysia käsittelevä laadullinen raportti. Raportista voi puolestaan joissain tapauksissa olla hyötyä ohjelmiston rakenteen ongelmien havaitsemisessa. Ensisijaisesti kyseessä on kuitenkin tiettyjä ongelmia kartoittava apuväline, eikä pätevä yleisratkaisu.

8.3 Ohjelmakoodin rakentaminen ja versiointi

Cppcheckin ohella toinen esimerkki työvaiheen määrittämisestä, työjonon rakentamisesta ja suoritettavasta tehtävästä on muuttuja build. Tässä tapauksessa tarkoituksena on koostaa ja rakentaa versionhallintaan saapuva lähdekoodi ESP32:lla ajettavaksi binääriksi. Työvaiheen suorittamisen apuna käytetään repositorystä löytyvää tiedostoa platformio.ini, jota käsiteltiin aiemmin luvun 6.3 kuvassa 8. Palvelimella tehtävä PlatformIOn määritys havaitaan puolestaan kuvasta 16.

```
20  build:
21    stage: build
22    tags:
23      - test
24    image: python:latest
25    script:
26      - apt-get update
27      - apt-get install -y python3-pip
28      - pip install -U platformio
29      - platformio run -e esp32dev
30      - mv .pio/build/esp32dev/firmware.bin esp32_v001.bin
31    variables: {PLATFORMIO_CI_SRC: "src/main.cpp"}
32    artifacts:
33      paths:
34        - esp32_v001.bin
```

Kuva 16. Ohjelmakoodin rakentamisen määrittäminen osana .gitlab-ci.yml -tiedostoa.

Kuvan 16 määritykset poikkeavat kuvan 14 vastaavista, mutta ohjelman logiikka ja kontin pystyttämisen rakenne ovat identtisiä. Kuvan 16 tapauksessa kontissa ajettavana ohjelmana toimii Cppcheckin sijaan PlatformIO ja pääohjelman asetukset määrittävä tiedosto main.cpp. Tavoitteena on saada ohjelmakoodi suoritettua, jolloin onnistuneen suoritteen tuottama binääri nimetään

tässä tapauksessa manuaalisesti tiedostoksi esp32_v001.bin. Vastaavasti työvaiheen epäonnistumisesta seuraa ohjelmistoputken virhe ja työjonon keskeytyminen.

Toimivista ohjelmistoversioista on havaittava, että jos tarkastellaan vain ohjelmakoodin vähittäisiä muutoksia, niin päivityshistorian jäljittämisestä ei ole ohjelmiston validoinnin ja mittalaitteen elinkaaren seurannan kannalta valtavaa hyötyä. Vastaavasti ammattimaisessa ohjelmistotuotannossa semanttisen versionumeroinnin käyttöönotto on tarpeellista, koska selkeiden kehitysvaiheiden erottelusta ja niiden asianmukaisesta nimeämisestä saadaan konkreettista hyötyä.

Käytännössä tämä tarkoittaa versionumeroinnin jakautumisen kolmeen pisteellä eroteltavaan numeroon, kuten 3.0.0. Tällöin ensimmäistä numeroa vasemmalta kutsutaan englanninkielisellä termillä major, joka erottelee keskenään sopimattomat ohjelmistoversiot toisistaan. Keskimmäistä termiä kutsutaan puolestaan termillä minor, jossa ohjelman toiminnallisuutta laajennetaan ilman sopivuusongelmia. Viimeisin numero on patch, jolla korjataan yksittäisiä virheitä. Järjestelmällinen ja oikeaoppisesti toteutettu versionumerointi selkeyttää ohjelmistokehitystä ja mahdollistaa tarvittaessa paluun edellisten, ja toimivaksi koeteltujen, versioiden käyttöön. [47.]

8.4 Pohdintaa palvelimella tehtävistä toimenpiteistä

Kyseisessä luvussa käsitellyt kaksi esimerkkiä tarjoavat pienen tirkistyksen erilaisista versionhallinnan yhteyteen rakennettavista testauksen toteutustavoista ja käyttömahdollisuuksista. Toisaalta suunnittelua ohjaa esimerkiksi käyttäjän osaaminen, laitteen vaatimukset ja käytetty ohjelmointikieli. Vaihtoehtoisen lähestymiskulman erilaisista toteutustavoista tarjoaa vaikkapa Esa Hannila testauksen automatisointia käsittelevässä opinnäytteessään tai Joel Känsälä opinnäytteessään, jonka aihe on testausautomaation kehittäminen Robot Frameworkillä [48; 49].

Aihepiirin lopuksi on todettava, että jaottelu päätelaitteella ja palvelimella tehtävistä toimenpiteistä on jossain määrin keinotekoinen. Tähän on lähinnä syynä Docker, joka mahdollistaa riippumattomuuden erilaisista ympäristöistä. Konttien avulla on myös mahdollista rakentaa hyvin monipuolisia ja räätälöityjä ohjelmistoratkaisuja, mutta toisaalta niiden rakentaminen edellyttää vankkaa asiantuntemusta käytetyn laitteen ominaisuuksista ja niiden vaatimista testeistä. Yleisemmin tarkasteltuna palvelimelle toteutettu ratkaisu aiheuttaa kehittämiseen, käyttöön ja ylläpitoon omat vahvuutensa ja heikkoutensa. Esimerkiksi palvelimella määritetty ohjelmistoputki palvelee kaikkia kehittäjiä tasapuolisesti, eikä vain yksittäistä päätelaitteen käyttäjää.

9 Yhteenveto

Tutkimuksen lähtökohtana oli MITYSens-olosuhdejärjestelmä, jonka laatua oli tarkoitus validoida. Ajankäytöltään ohjelmistojen validointi mittausjärjestelmissä -tutkimusta tehtiin vaihtelevalla työtahdilla noin yhdeksän kuukauden ajan syksystä 2021 kevääseen 2022. Tutkimuksen kuluessa validoinnin määritykseksi vakiintui ohjelmistolle asetettujen odotusten täyttyminen. Rajaukseltaan tutkimuksen aihe muodostui kuitenkin ongelmalliseksi, sillä tutkimuksen aikana selvisi, ettei ohjelmistojen validointiin ole yhtä ja yksiselitteistä keinoa. Sen sijaan ohjelmistojen validointia ja laatutekijöitä määrittäviä standardeja on kymmeniä, eivätkä ne sovellu byrokraattisen luonteensa vuoksi erityisen hyvin pienen kehitysyksikön toimintaan.

Tutkimuksen tarkastelu tiivistyi vähitellen. Ensin pyrittiin kattamaan olosuhdejärjestelmän kaikki osat ja täydellinen ohjelmisto, mutta lopullisessa versiossa käsitellään vain kehitysalustaa ja lämpötilaa mittaavaa anturia. Tutkimuksen sisältöä ohjaavan päätöksen taustalla on, että opinnäytteen sisältöä kohdentamalla työn kokonaisuus pysyi hallittavana ja laajuus kohtuullisena. Sisällöllisesti käsittelyssä on toisaalta ESP32-kehitysalusta ja pääohjelman erittely. Toisaalta käsittelyssä on lämpötila-anturi DS18B20, joka muodostaa esimerkin mittalaitteen modulaarisuudesta ja anturin vaatimien aliohjelmien integroinnista osaksi mittalaitteen pääohjelmaa. Sisältöä ohjaa myös se, että lopputuotteena syntyvä opinnäyte on julkinen dokumentti. Tällöin esitystavan on syytä olla mahdollisimman yleispätevä, eikä kaikkiin yksityiskohtiin pureutuva datalehti.

Rakenteeltaan tutkimus jakautuu teoreettisen viitekehyksen erittelyyn sekä käytännön toimia havainnollistaviin käsittelylukuihin. Teoreettisessa osuudessa tarkoituksena on esitellä, kuinka modernia ohjelmistokehitystä tehdään, millä työkaluilla se tapahtuu ja miten validointi määritetään. Käytännön osuudet puolestaan jakautuvat lukuihin, joissa tarkastellaan palvelinta ja testausympäristöä, mittalaitteen komponentteja ja ohjelmistoa sekä erilaisia ohjelmistotestejä. Käsittelyn puitteissa ohjelmistotestejä tosin tarkastellaan ennemminkin yleisellä tasolla testausmenetelmien, työkalujen ja periaatteiden muodossa kuin yksiselitteisenä keinona validointiin. Tutkimuksen lopussa olevat liitteet jakautuvat pääosin teoriaosuutta tukeviin validointia ja testausta erittteleviin aineistoihin sekä käytännön osuudessa hyödynnettyjen ohjelmakoodien esimerkkeihin.

Tutkimuksen käytännön toteutukseen vaikutti myös työekonomiset syyt. Esimerkiksi opinnäytteessä tarkastellun palvelimen pystyttäminen aloitettiin osien tilaamisesta, erilaisten ohjelmien asennuksesta ja toimintakuntoon saattamisesta, mikä söi osansa opinnäytteen tekoon tarkoitetusta työajasta. Myös mittalaitteen kokoonpanon määritys ja käytettyjen ohjelmistokoodin osien

eristäminen veivät hieman työaikaa. Tämän lisäksi aikaa kului varsinaisten päätelaitteella ja palvelimella suoritettujen testien rakentamiseen ja todentamiseen. Näiden lisäksi aikaa kului teorian rakentamiseen ja varsinaiseen kirjoitustyöhön. On todettava, ettei minkään yksittäisen työvaiheen suorittaminen aiheuttanut huomattavaa kuormitusta tai ajankäyttöä. Mutta kokonaisuutena ne aiheuttivat vaikutuksen, ettei mihinkään aihepiiriin ollut opinnäytteen laajuuden puitteissa mahdollista paneutua kovinkaan syvällisesti.

Tutkimuksen sisällöstä on myös olennaista kysyä validointiin liittyvän ohjelmistotestauksen periaatteista, eli mitä testataan, miksi testataan ja miten testataan. Lisäksi tähän vaikuttaa ohjelmiston kehitystapa, sillä periaatteessa vanhassa vesiputousmallissa ohjelmiston vaatimukset ja niiden määritykset asetetaan jo ohjelmistokehityksen alussa. Nykyaikainen ohjelmistokehitys kuitenkin tapahtuu iteratiivisesti ja vähittäin erilaisia toiminnallisuuksia lisäten, jolloin myös validointi pirstaloituu pienemmiksi osakokonaisuuksiksi. Sulautetuissa järjestelmissä asiaa toki monimutkaistaa edelleen käytetyn mikrokontrollerin ominaisuuksien huomiointi sekä laitteen kehitys modulaarisesti, jolloin erilaiset komponentit ja niiden vaatimat ohjelmistot testeineen yhdistetään osaksi kehitystyötä ja validointia.

Yhtä kaikki tutkimuksen edetessä ESP32 ja Raspberry Pi 4 ovat tulleet jossain määrin tutuiksi. Tämän lisäksi Visual Studio Code, PlatformIO ja GitLab havaittiin päteviksi työkaluiksi ohjelmistokehityksen, erilaisten testien ja versionhallinnan toimenpiteitä ajatellen. Tutkimusta tehtäessä kyseisistä työkaluista ja esimerkiksi Dockerin toiminnasta selvisi allekirjoittaneelle huomattava määrä uusia käyttötapoja ja ominaisuuksia. Ylipäätään tutkimuksen koostaminen vaati perinpohjaista aihepiiriin perehtymistä ja riittävää ajankäyttöä kirjoitustyön edistämiseksi sekä tutkimuksen viimeistelemiseksi.

Lopuksi on todettava, että aihepiirin tarkastelussa yllätti tähän saakka tehdyn tutkimuksen vähäisyys. Suomessa ja maailmalla on kuitenkin huomattava määrä eri tavoin mittaamiseen liittyviä tai mittalaitteiden ohjelmistoja valmistavia yrityksiä. Ilmeisesti valtaosa yritysten käyttämistä toimintamalleista on kuitenkin kunkin talon sisäisiä, eikä niitä ole mahdollista tai järkevää jakaa yritysten ulkopuolelle. Täten julkista aineistoa on niukasti tarjolla, eikä myöskään mitään yleispätevää periaatetta validoinnille ole muodostunut. Mahdollisia syitä tälle voivat toki olla myös riski raskaalle byrokraattiselle painolastille tai ohjelmoinnin välineiden ja työkalujen jakautuminen äärettömäksi määräksi erilaisia yrityskohtaisia sovelluksia, mutta yhtä kaikki validoinnista on kirjoitettu yllättävän vähän. Tosin nyt julkaisuja on yksi enemmän.

Lähteet

1 Espressif, ESP32 Series. [viitattu 22.4.2022]. Saatavilla: https://www.espressif.com/sites/default/files/documentation/esp32_datasheet_en.pdf

2 Hauhia Juha, KAMK-anturiverkko – sensoriverkko luokkatilojen monitorointiin. KAMK, 2020. Saatavilla: https://urn.fi/URN:NBN:fi:amk-2020060316554

3 Mustonen Juha, Puhdastilojen Olosuhdeseurantajärjestelmän Käyttöliittymä. KAMK, 2021. Saatavilla: https://urn.fi/URN:NBN:fi:amk-202102021830

4 Royce W. Winston, Management of the development of Large Software Systems. [viitattu 22.4.2022]. Saatavilla: http://www-scf.usc.edu/~csci201/lectures/Lecture11/royce1970.pdf

5 Haikala Ilkka ja Mikkonen Tommi. Ohjelmistotuotannon käytännöt, Talentum Media Oy. Hämeenlinna 2011.

6 Luukkainen Matti ja Ilves Kalle, Ohjelmistotuotanto 2021. Helsingin yliopisto. [viitattu 22.4.2022]. Saatavilla: https://ohjelmistotuotanto-hy.github.io/

7 Beck Kent et. al, Manifesto for Agile Software Development. [viitattu 22.4.2022]. Saatavilla: http://agilemanifesto.org/

8 Lehtonen Teijo et. al, Sulautettujen järjestelmien ketterä käsikirja, Painosalama Oy, Turku 2014. Saatavilla: https://urn.fi/URN:ISBN:978-951-29-5838-2

9 Karppinen Markku, Kehitysalustan suunnittelu ja toteutus sulautettujen järjestelmien opetuskäyttöön. KAMK, 2020. Saatavilla: https://urn.fi/URN:NBN:fi:amk-2020052212914

10 Sassi Saara, Laadunhallintajärjestelmän käyttöönoton perusteet ohjelmistoalan yrityksessä. Jyväskylän yliopisto, 2020. Saatavilla: http://urn.fi/URN:NBN:fi:jyu-202006124150

11 Kasurinen Jussi Pekka, Ohjelmistotestauksen käsikirja. Docendo, 2015. Saatavilla rajoitetusti: https://www.ellibslibrary.com/book/9789522911025

12 Software Verification and Validation Technologies and Tools. [viitattu 22.4.2022]. Saatavilla rajoitetusti: https://ieeexplore.ieee.org/abstract/document/8648264

13 Komulainen Jari, DevOps:n käyttöönotto pienessä yksikössä. OAMK, 2020. Saatavilla: https://urn.fi/URN:NBN:fi:amk-202202192674

14 Laihonen Paul, Adoption of DevOps Practices in the Finnish Software Industry: an Empirical Study. Aalto-yliopisto, 2018. Saatavilla: http://urn.fi/URN:NBN:fi:aalto-201810175475

15 1012-2016 - IEEE Standard for System, Software, and Hardware Verification and Validation. [viitattu 22.4.2022]. Saatavilla rajoitetusti: https://ieeexplore.ieee.org/document/8055462

16 Docker overview. [viitattu 22.4.2022]. Saatavilla: https://docs.docker.com/get-started/overview/

17 GitLab Inc. [viitattu 22.4.2022]. Saatavilla: https://www.crunchbase.com/organization/gitlab-com

18 Use containers to Build, Share and Run your applications. [viitattu 22.4.2022]. Saatavilla: https://www.docker.com/resources/what-container

19 GitLab Docs. [viitattu 22.4.2022]. Saatavilla: https://docs.gitlab.com/ee/

20 GitLab Runner. [viitattu 22.4.2022]. Saatavilla: https://docs.gitlab.com/runner/

21 Visual Studio Code. [viitattu 22.4.2022]. Saatavilla: https://code.visualstudio.com/docs

22 PlatformIO Home. [viitattu 22.4.2022]. Saatavilla: https://docs.platformio.org/en/latest/home/index.html

23 CppUnit - C++ port of JUnit. [viitattu 22.4.2022]. Saatavilla: https://sourceforge.net/projects/cppunit/

24 Cppcheck Github repository. [viitattu 22.4.2022]. Saatavilla: https://github.com/danmar/cppcheck

25 Flawfinder. [viitattu 22.4.2022]. Saatavilla: https://sourceforge.net/projects/flawfinder/

26 PlatformIO. Unit Testing. [viitattu 22.4.2022]. Saatavilla: https://docs.platformio.org/en/latest/plus/unit-testing.html

27 Hiltunen Erkki et. al (toim.), Laadukkaan mittaamisen perusteet. Mittaustekniikan keskus, 2011. [viitattu 22.4.2022]. Saatavilla: https://www.vttresearch.com/sites/default/files/pdf/MIKES/2011-J4.pdf

28 Järvinen Jaana, S. Eerola ja M. Kaukonen (toim.), Metrologiasta lyhyesti. Mittaustekniikan keskus, 2008. [viitattu 22.4.2022]. Saatavilla: https://www.vttresearch.com/sites/default/files/pdf/MIKES/metrologiasta_lyhyesti_nettiin.pdf

29 DS18B20. Programmable Resolution 1-Wire Digital Thermometer. [viitattu 22.4.2022]. Saatavilla: https://datasheets.maximintegrated.com/en/ds/DS18B20.pdf

30 Pyykkönen Teemu, Olosuhdekaapin käyttöönotto ja testausjärjestelmän toteutus. KAMK, 2015. Saatavilla: https://urn.fi/URN:NBN:fi:amk-201505198764

31 Weckström Thua (toim.), Lämpötilan mittaus. Mittaustekniikan keskus, 2005. [viitattu 22.4.2022]. Saatavilla: https://www.vttresearch.com/sites/default/files/pdf/MIKES/2005-J4.pdf

32 Petri Koposen sähköposti Janne Rahkolalle 1.2.2022

33 Meidän tarinamme. Beamex Oy Ab. [viitattu 22.4.2022]. Saatavilla: https://www.beamex.com/fi/tietoa-meista/meidan-tarinamme/

34 Mittauslaitteiden kalibrointi. Kiwa Oy. [viitattu 22.4.2022]. Saatavilla: https://www.kiwa.com/fi/fi/palvelutyyppi/mittauslaitteiden-kalibrointi/

35 Lämpötilan ja kosteuden kartoitus- ja validointipalvelu. VAISALA. [viitattu 22.4.2022]. Saatavilla: https://www.vaisala.com/fi/products/maintenance-and-support-services/continuous-monitoring-system/mapping-service

36 Kalibrointi- ja mittauspalvelut. VTT Oy. [viitattu 22.4.2022]. Saatavilla: https://www.vttresearch.com/fi/palvelut/kalibrointi-ja-mittauspalvelut

37 Hitsauskoneiden kalibrointi, validointi ja huolto. Haka-Kone. [viitattu 22.4.2022]. Saatavilla: http://www.haka-kone.fi/hitsauskoneiden-kalibrointi-validointi-ja-huolto/

38 GitLab Docker images. [viitattu 22.4.2022]. Saatavilla: https://docs.gitlab.com/ee/install/docker.html

39 GitLab Community Edition docker image for arm64v8. [viitattu 22.4.2022]. Saatavilla: https://hub.docker.com/r/yrzr/gitlab-ce-arm64v8

40 Deploy Gitlab Runner With Docker. [viitattu 22.4.2022]. Saatavilla: https://blog.programster.org/deploy-gitlab-runner-with-docker

41 Library for Dallas/Maxim 1-Wire Chips. [viitattu 22.4.2022]. Saatavilla: https://github.com/PaulStoffregen/OneWire

42 Arduino plug and go library for the Maxim (previously Dallas) DS18B20 (and similar) temperature ICs. [viitattu 22.4.2022]. Saatavilla: https://github.com/milesburton/Arduino-Temperature-Control-Library

43 UNITY - Unit Testing for C (especially Embedded Software). [viitattu 22.4.2022]. Saatavilla: https://www.throwtheswitch.org/unity

44 Using GCC with MinGW. [viitattu 22.4.2022]. Saatavilla: https://code.visualstudio.com/docs/cpp/config-mingw

45 Embedded C/C++ Unit Testing Basics. [viitattu 22.4.2022]. Saatavilla: https://interrupt.memfault.com/blog/unit-testing-basics

46 Mononen Samuli, Evaluation of Test-Driven Approaches for Embedded Software Development. Aalto-yliopisto, 2020. Saatavilla: http://urn.fi/URN:NBN:fi:aalto-202008235020

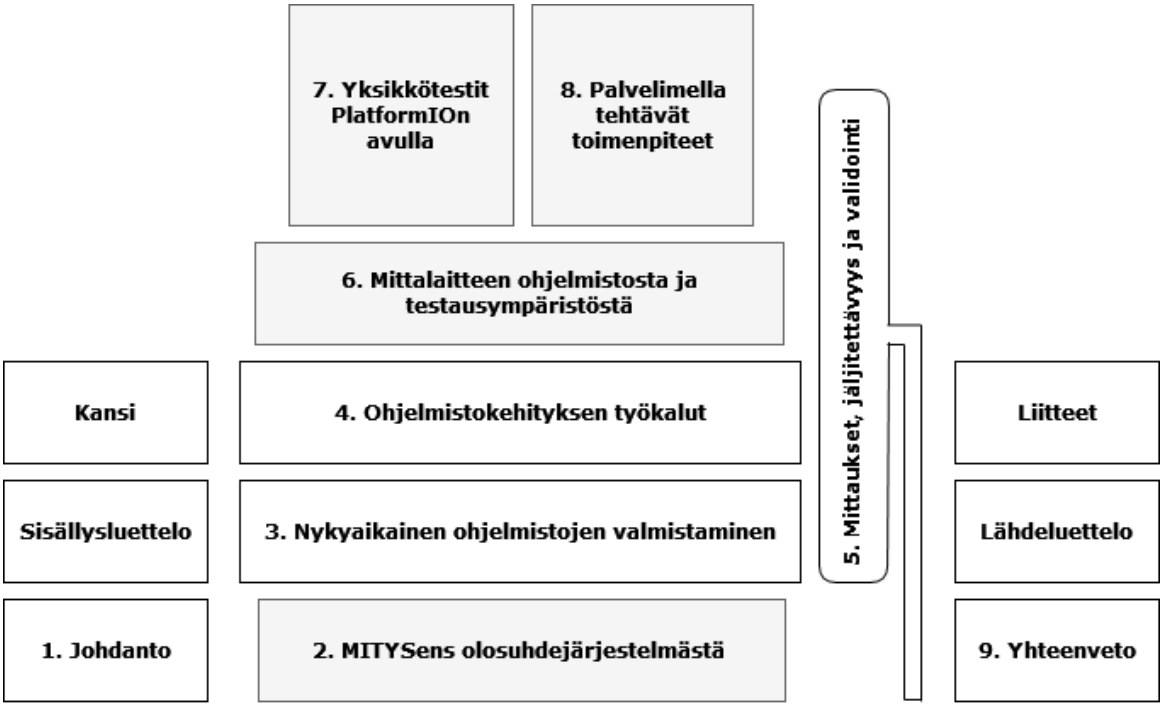
47 Semantic Versioning 2.0.0. [viitattu 22.4.2022]. Saatavilla: https://semver.org/

48 Hannila Esa, Ohjelmistojen automaattinen testausjärjestelmä. OAMK, 2018. Saatavilla: https://urn.fi/URN:NBN:fi:amk-201802202635

49 Känsälä Joel, Testausautomaation kehittäminen Robot Frameworkillä. KAMK, 2021. Saatavilla: https://urn.fi/URN:NBN:fi:amk-2021120223329

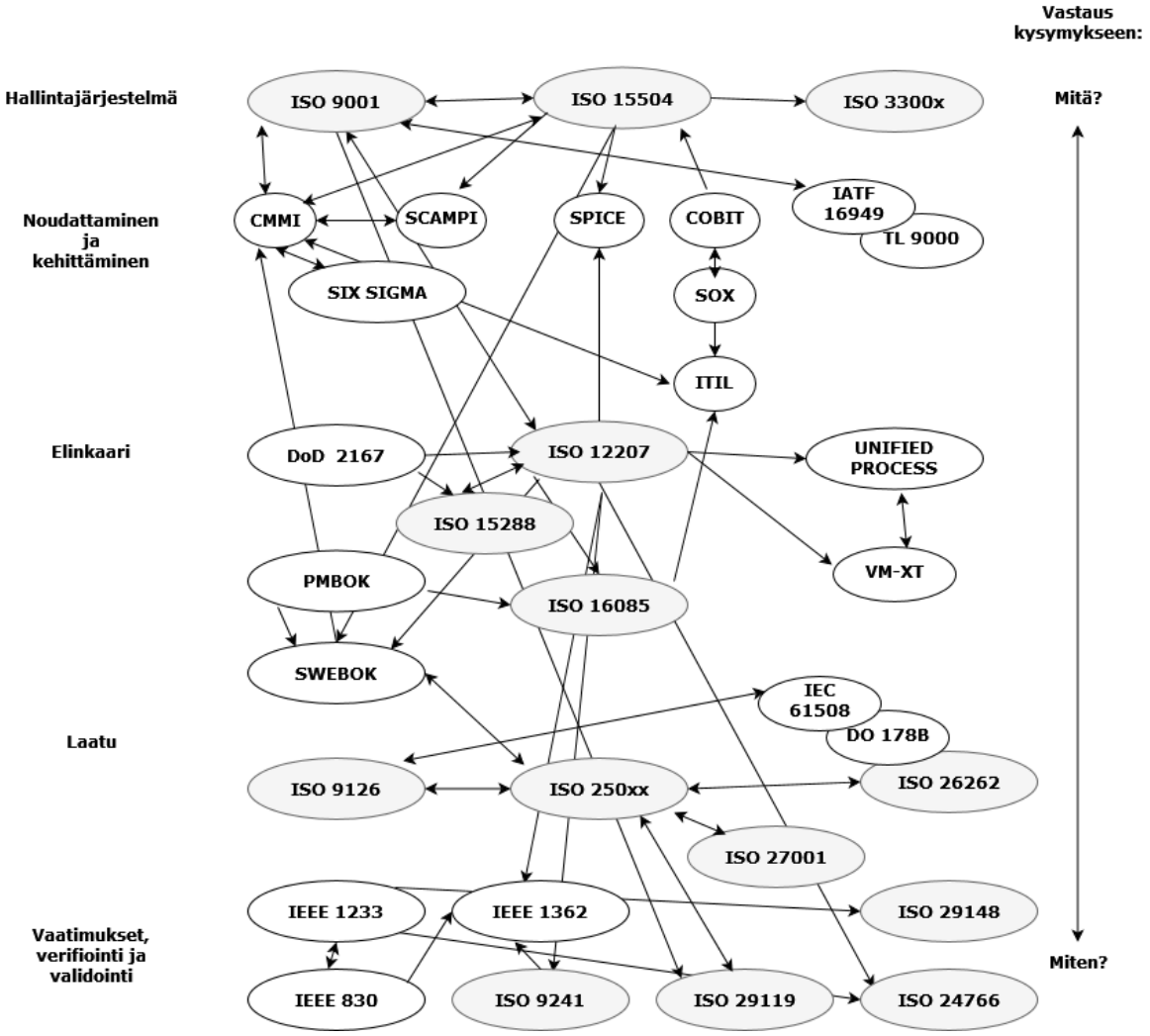
(Lukijalle huomio! Jos ja kun elektroniset viitteet muuttuvat tai vanhenevat, niin ne voivat silti olla käytettävissä Internet Archive -palvelun avulla. Hae siis kutakin osoitetta mahdollisimman läheisellä päivämäärällä viitepäivämäärään verraten. Osoitteesta: https://archive.org/web/)

Tutkimuksen rakenteen kuvaus visuaalisena esityksenä



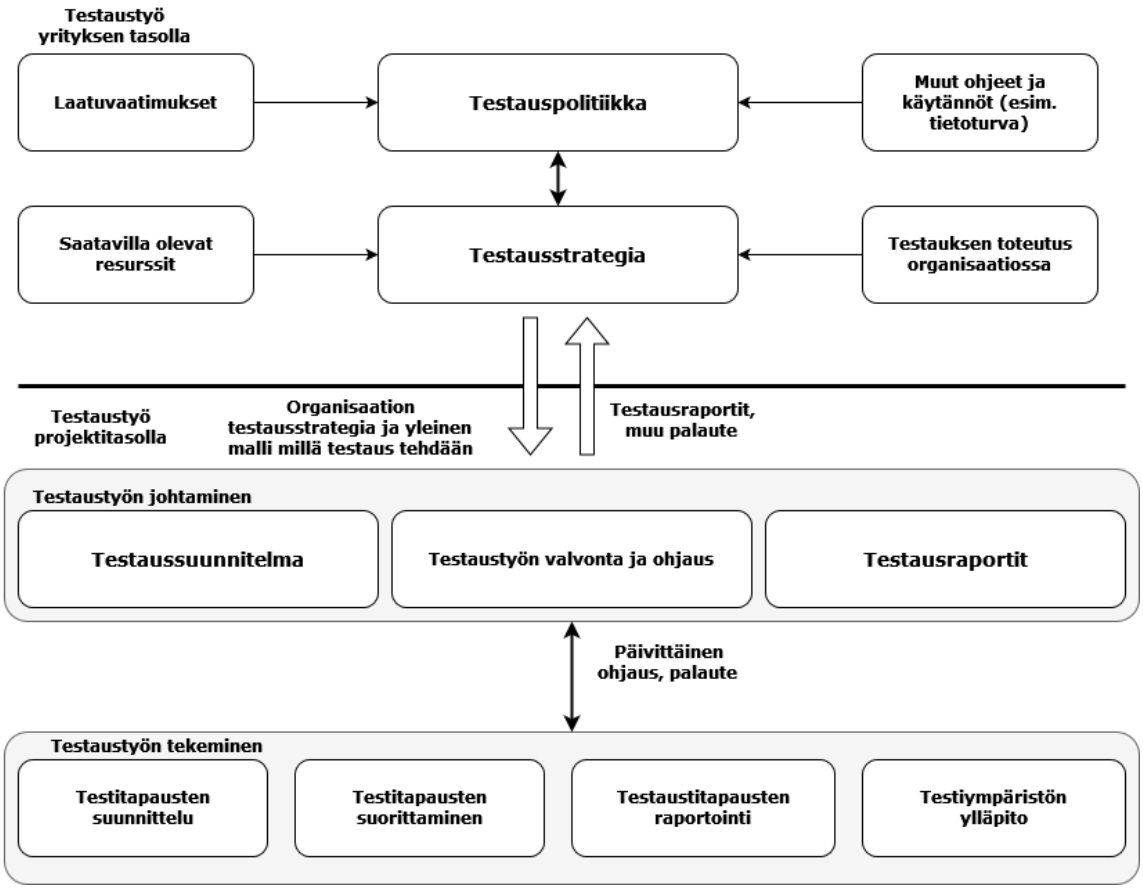
Lähde: Janne Rahkola.

Ohjelmistoalan standardeja, jotka liittyvät välittömästi tai välillisesti validointiin ja verifiointiin.



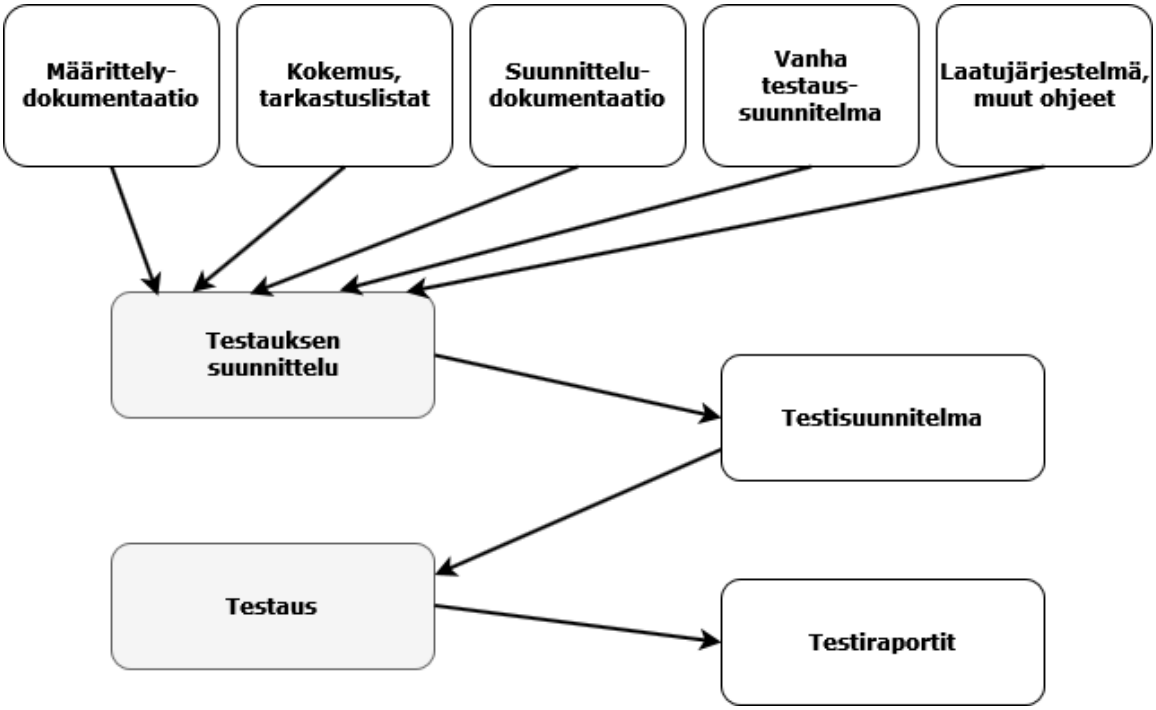
Lähdeviite: 12. (Software Verification and Validation Technologies and Tools.)

Ohjelmistotestauksen yleinen toimintamalli ISO/IEC 29119 periaatteilla.



Lähdeviite: 11. (Kasurinen Jussi Pekka, Ohjelmistotestauksen käsikirja. Sivu 103)

Testauksen suunnitteluun vaikuttavista tekijöistä.



Lähdeviite: 5. (Haikala Ilkka ja Mikkonen Tommi, Ohjelmistotuotannon käytännöt. Sivu 216)

Pääohjelma: main.cpp

src > main.cpp > ...

```cpp
// Basic DS18S20 sensor platformio software with sleep

// def bus
#define ONE_WIRE_BUS      15

// include libs
#include <Arduino.h>
#include <OneWire.h>
#include <ds18b20Sensor.h>

// ds18 settings
OneWire oneWire(ONE_WIRE_BUS);
DallasTemperature sensors(&oneWire);

// DS18B20 count
uint8_t dsCount              = 0;

// deep sleep time. * 1000000ULL modifier for second convert
uint64_t const sleepTimeSeconds    = 10 * 1000000ULL;

void setup()
{
  // init serial
  Serial.begin(115200);

  // enable sleep
  esp_sleep_enable_timer_wakeup(sleepTimeSeconds);

  // init bus
  dsCount = initDallasBus(&sensors);
}

void loop()
{
  Serial.print(" Temp: ");

  // read 1-wire
  readDallasTemperature(&sensors, dsCount);

  // start sleep
  esp_deep_sleep_start();
}
```

Lähde: Janne Rahkola

Aliohjelmat:

ds18b20Sensor.h

```
#ifndef DS18B20SENSOR_H
#define DS18B20SENSOR_H

#include <OneWire.h>
#include <DallasTemperature.h>

int initDallasBus(DallasTemperature *dallasSensors);

void readDallasTemperature(DallasTemperature *dallasSensors,
                           uint8_t dsSensorsCount);

#endif
```

ds18b20Sensor.cpp

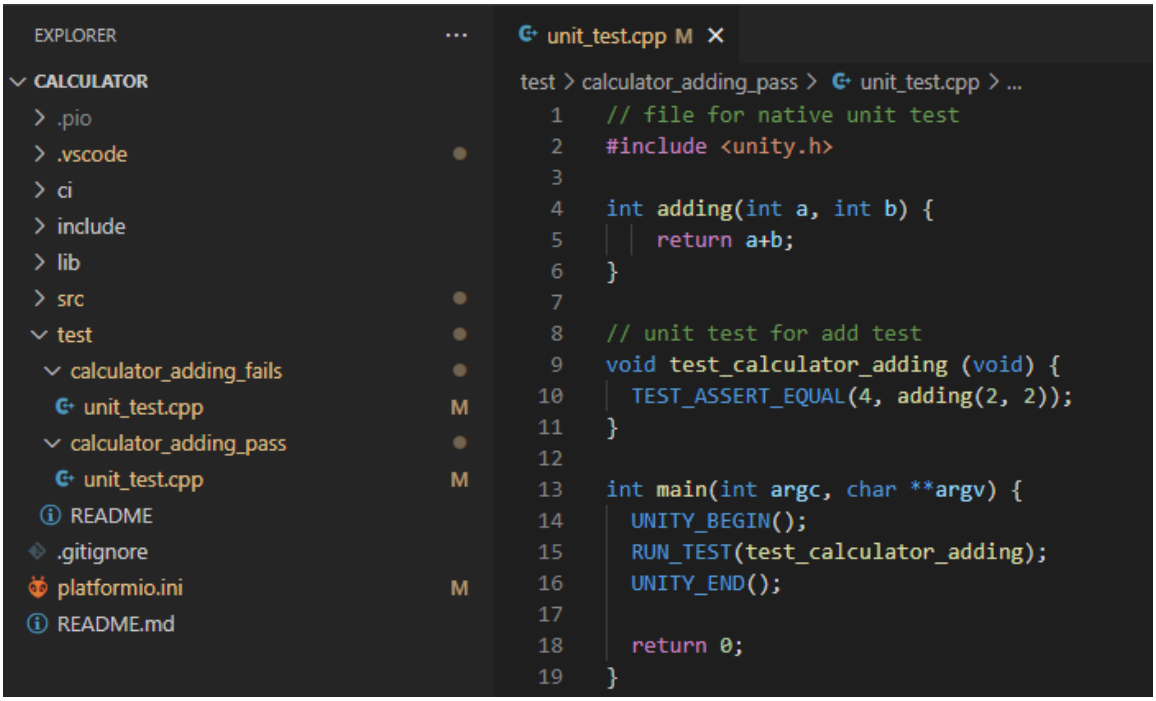
```
#include "ds18b20Sensor.h"

int initDallasBus(DallasTemperature *dallasSensors)
{
    dallasSensors->begin();
    uint8_t temperatureDevicesCount = dallasSensors->getDS18Count();
    return temperatureDevicesCount;
}

void readDallasTemperature(DallasTemperature *dallasSensors, uint8_t dsSensorsCount)
{
    float temp = 0;
    dallasSensors->requestTemperatures();
    temp = dallasSensors->getTempCByIndex(0);
    Serial.println(temp);
}
```

Lähde: Janne Rahkola

Unity.h yksikkötestejä havainnollistava unit_test.cpp



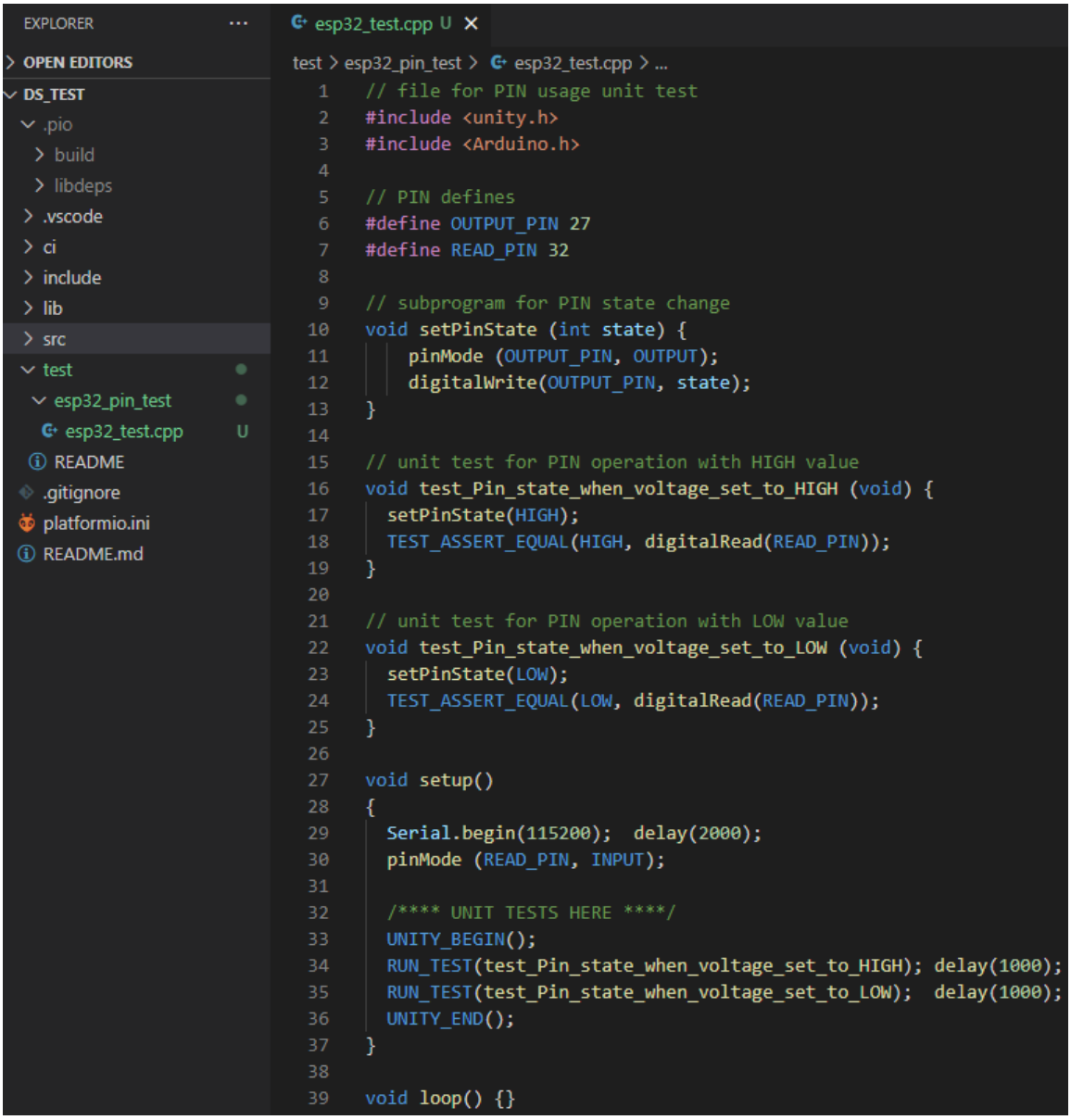
Lähde: Janne Rahkola

Huomio! Calculator_adding_fails -testiin verrattuna ainoa muutos on funktiossa:

TEST_ASSERT_EQUAL(4, adding(2, 1));

Tarkoituksena on, että saattamalla funktio tarkoituksellisesti epätodeksi voidaan tarkastella myös ohjelman tuottamaa virheraporttia.

ESP32 kytkentää tarkasteleva yksikkötesti esp32_test.cpp



Lähde: Janne Rahkola

Aliohjelmaan tahallaan aiheutettu rakenteellinen virhe ja sen seuraus.

ds18b20Sensor.cpp

```cpp
#include "ds18b20Sensor.h"

int initDallasBus(DallasTemperature *dallasSensors)
{
    dallasSensors->begin();
    uint8_t temperatureDevicesCount = dallasSensors->getDS18Count();
    return 0;
}

void readDallasTemperature(DallasTemperature *dallasSensors, uint8_t dsSensorsCount)
{
    float temp = 0;
    dallasSensors->requestTemperatures();
    temp = dallasSensors->getTempCByIndex(0);
    Serial.println(temp);
}
```

Cppcheckin raportti cppcheck_all.log

cppcheck_all.log

```
src/ds18b20Sensor.cpp:6:37: style: Variable 'temperatureDevicesCount' is assigned a value that is never used. [unreadVariable]
    uint8_t temperatureDevicesCount = dallasSensors->getDS18Count();
                                    ^
src/main.cpp:32:0: style: The function 'loop' is never used. [unusedFunction]

^
src/main.cpp:20:0: style: The function 'setup' is never used. [unusedFunction]

^
nofile:0:0: information: Cppcheck cannot find all the include files (use --check-config for details) [missingInclude]
```

Lähde: Janne Rahkola