ERNO JOHANSSON

KÄYTTÖVARMUUDEN RAKENTAMINEN OSAKSI TEOLLISUUSROBOTIN SOLUOHJAINTA

Diplomityö

Tarkastajat: Prof. Hannu Koivisto ja
Assistant Prof. David Hästbacka
Tarkastajat ja aihe hyväksytty
27. syyskuuta 2017

TIIVISTELMÄ

ERNO JOHANSSON: Käyttövarmuuden rakentaminen osaksi teollisuusrobotin soluohjainta
Tampereen teknillinen yliopisto
Diplomityö, 81 sivua, 2 liitesivua
Toukokuu 2018
Automaatiotekniikan diplomi-insinöörin tutkinto-ohjelma
Pääaine: Automaation tietotekniikka
Tarkastajat: Prof. Hannu Koivisto ja Assistant Prof. David Hästbacka
Avainsanat: käyttövarmuus, luotettavuus, vikasietoisuus, soluohjain, DevOps

Ohjelma on käyttövarma kun se toimii määrittelynsä mukaisesti ja sen käyttö onnistuu sille asetettujen ehtojen vallitessa halutun ajan verran. Yhä monimutkaisempien ohjelmien osalta käyttövarmuuden varmistaminen on kuitenkin haastavaa. Työssä esiteltävä teollisuusrobottisolua ohjaava soluohjain on tästä esimerkkinä. Se sisältää useita virheitä, jotka heikentävät sen käyttövarmuutta. Työn tarkoituksena on löytää tapoja parantaa tämän ja uusien alkavien ohjelmistoprojektien käyttövarmuutta.

Tässä työssä tehtiin tutkimusta kahdessa vaiheessa, joista ensimmäisessä kartoitettiin kirjallisuustutkimuksena ohjelmistokehityksen keinoja ja toimintamalleja parantaa ohjelman käyttövarmuutta. Toisessa vaiheessa keinoja sovellettiin soluhjaimen toteuttamiseen ja siinä esiintyvien ongelmien korjaamiseen.

Ohjelman käyttövarmuutta parantavat keinot voidaan jakaa neljään ryhmään: virheiden esittelyä välttävät, ohjelman ajon aikaisista vioista toipuminen vikasietoisuudella, ohjelmaan jo esiteltyjä virheitä poistavat ja nykyisten sekä tulevien virheiden määrää ja vaikutusta arvioivat. Näihin ryhmiin kuuluvien keinojen avulla onnistuttiin työssä toteuttamaan käyttövarmuudeltaan hyväksyttävällä tasolla oleva soluohjain.

Keinojen lisäksi ohjelman käyttövarmuutta voidaan kasvattaa uusissa ohjelmistoprojekteissa myös ottamalla käyttöön toimintamalleja kuten jatkuva integrointi, jatkuva toimitus ja DevOps. Julkaisunopeuden kasvattamisen lisäksi niiden avulla voidaan vähentää virheiden esittelyä ohjelmaan ja tehostaa jo esiteltyjen virheiden nopeaa korjaamista automaattisilta testeiltä ja ohjelman käyttäjiltä saatavan palautteen avulla.

ABSTRACT

ERNO JOHANSSON: Development of Reliable Cell Controller
Tampere University of Technology
Diplomityö, 81 pages, 2 Appendix pages
May 2018
Master's Degree Programme in Automation Technology
Major: Information Systems in Automation
Examiner: Prof. Hannu Koivisto and Assistant Prof. David Hästbacka
Keywords: reliability, dependability, fault tolerance, cell controller, DevOps

Software is reliable when it performs its intended functions in a specified period of time under specified design limits in a system's environment, without experiencing failure. In complex software systems achieving reliability is demanding. The thesis introduces a cell controller software that is used to control industrial robot cells as an example. The software contains several faults that decrease its realiability. The goal of this thesis is to find ways to improve software reliability in this case and in future software projects.

The thesis work is split into two phases. First literature research is conducted to find ways to improve the reliability of software systems. Then learned methods are applied to fix faults in cell controller software.

Developing a reliable software can be divided into four groups: preventing faults from happening, recovering from run-time errors using fault tolerance, removing faults and evaluating faults that are left with respect to their occurrence and activation. The methods in these groups were used to build a cell controller that was accepted to be reliable by its users.

In addition to the methods introduced in this thesis reliability of software system in green field projects can be increased by using methodologies like continuous integration, continuous delivery and DevOps. The methodologies do not only decrease deployment times, but they can be used to reduce the amount of introduced faults and increase the efficiency of removing faults by using feedback from automatic tests and software's users.

ALKUSANAT

Diplomityö kirjoitettiin toimiessani projektipäällikkönä Eatech Oy:ssä. Idea työhön tuli käynnissä olevasta asiakasprojektista, jonka parissa työskentelin. Diplomityön kirjoitus alkoi huhtikuussa 2017 ja se saatettiin loppuun toukokuussa 2018.

Haluan kiittää Hannu Koivistoa ja David Hästbackaa työn ohjaamisesta. Haluan myös kiittää sekä työnantajaani Eatech Oy:tä että Pemamek Oy:tä, jotka yhdessä mahdollistivat tämän työn tekemisen. Erityiskiitokset Ville Laineelle hyvin sujuneesta yhteistyöstä ja Antti Kopposelle, joka mahdollisti ajan työn kirjoittamiselle.

Lopuksi haluan kiittää vaimoani Heini Johanssonia. Ilman hänen tukea ja kannustusta työ olisi vielä kirjoittamatta.

Tampere, 21.5.2018

Erno Johansson

SISÄLLYSLUETTELO

1. Johdanto 1
1.1 Motivaatio 1
1.2 Tavoite 1
1.3 Tutkimusmenetelmät 2
1.4 Työn rakenne 2
2. Lähtötilanne 4
2.1 Pemamek 4
2.2 Alkuasetelma 4
2.3 Rajoitteet 5
2.4 Järjestelmäkuvaus 5
2.5 Vanhan soluohjaimen ongelmia 7
3. Luotettavuutta lisäävät keinot 9
3.1 Virhe, vika ja häiriö 9
3.2 Keinojen ryhmittely 10
3.3 Virheiden välttäminen 11
3.3.1 Ohjelmiston vaatimusten määrittely 11
3.3.2 Tilakoneanalyysi 13
3.3.3 Muita analyysimenetelmiä 14
3.4 Vikasietoisuus 15
3.4.1 Poikkeustenkäsittely 15
3.4.2 Tilan tallennus 17
3.5 Virheiden poistaminen 17
3.5.1 Yksikkötestaus 18
3.5.2 Järjestelmätestaus tutkivana testauksena 19
3.5.3 Kuormitustestaus 20
3.6 Virheiden ennustaminen 21
3.6.1 Koodikattavuus 21

3.7 Toimintamallin muuttaminen 23
3.7.1 Ketterä ohjelmistokehitys 23
3.7.2 Jatkuva integrointi 24
3.7.3 Jatkuva toimitus ja -julkaisu 25
3.7.4 DevOps 26
4. Toteutusympäristö 28
4.1 PLC:n toimintaperiaate 28
4.1.1 Skannaus sykli 28
4.1.2 Ohjelmointikielet 30
4.2 Tiedonsiirron tekniikat 31
4.2.1 DeviceNet 31
4.2.2 TwinCAT ADS Communication Library 32
4.2.3 FTP 34
4.2.4 High Speed Ethernet Client (HSE) 35
4.3 Asynkroninen ohjelmointi 35
5. Työn kohteena oleva robottisolu 37
5.1 Rakenne 37
5.2 Soluohjaimen vaatimusten määrittely 38
5.3 Soluohjain 40
5.3.1 Hallittavan robottisolun työkierto 42
5.3.2 Työn suorittaminen 44
6. Luotettavuuden rakentaminen osaksi soluohjainta 47
6.1 Teollisuus-PC 47
6.1.1 Ohjelmistoarkkitehtuuri 47
6.1.2 Asynkroninen laiterajapinta 49
6.1.3 Töiden hallinnan yksikkötestaaminen 50
6.1.4 Töiden hallinnan koodikattavuus 51
6.1.5 Vikasietoisuuden parantaminen poikkeustenkäsittelyllä 53
6.1.6 Odottamattomasta viasta toipuminen 53

6.2 PLC 54
6.2.1 Tilakoneanalyysi robotin ohjaimelle 54
6.2.2 Käyttötapausten toteutuminen 58
6.3 Robotti 59
6.3.1 Liitynnät muuhun järjestelmään 59
6.3.2 Käyttötapausten toteutuminen 61
6.4 Testaus 63
6.4.1 Järjestelmätestaus 63
6.4.2 Kuormitustestaus 64
6.5 Vaihtoehtoinen toteutus - High Speed Ethernet Client 65
6.5.1 Muutoksen vaikutus 65
6.5.2 Käyttötapausten toteutuminen 66
7. Arviointi 68
8. Yhteenveto 71
Lähteet 73
A. Robotin pääohjelma 82
B. Apuohjelma robotin uuden työn luomiseen 83

KUVALUETTELO

1 Järjestelmän yleiskuvaus sekä sen muodostavien laitteiden väliset yhteydet ja tiedonsiirtoprotokollat. 6

2 Tilakone UML -syntaksilla. Muokattu lähteestä [21, s. 347]. 14

3 PLC:n yhden syklin aikaiset tehtävät. 29

4 IEC 61131-3 standardin mukaiset PLC ohjelmointikielet. Suomennettu lähteestä [59, s. 81]. 30

5 DeviceNet osana CIP OSI-mallia [62]. 32

6 ADS laitteiden väliset tiedonsiirtokanavat. Muokattu lähteestä [66]. . 33

7 Synkronisen- ja asynkronisen suorituksen vertailu. Muokattu lähteestä [79]. 36

8 Robottisolu, jossa on käytössä työn soluohjain. 37

9 Järjestelmän käyttötapaukset. 39

10 Soluohjaimen laitteiden sijoittuminen teollisuusverkon eri hierarkiatasoille. Suomennettu ja muokattu lähteestä [59, s. 318]. 41

11 Operaattorin työkierto robottiasemalla. 42

12 Soluohjainsovelluksen töiden hallintanäkymä. 43

13 Robotille käynnistettävän työn suorituksen vaiheet. 45

14 Soluohjainohjelmiston arkkitehtuuri. 48

15 dotCover -työkalun esittämä lausekattavuus robotin työn valinta moduulin lähdekoodeista. 52

16 PLC:n robottia ohjaava toteutus tilakaaviona. 57

17 DeviceNet-väylää pitkin PLC:ltä robotille menevä 100 BYTE:n taulukko. 59

18 DeviceNet-väylää pitkin robotilta PLC:lle saapuva 100 BYTE:n taulukko. 60

19 Robotille menevän taulukon 3:nnen BYTE:n binääriesitys. 62

TAULUKKOLUETTELO

1 Testitapaus, jolla saadaan 100% lausekattavuus ohjelmassa 2. 22

2 Testitapaukset, joilla saadaan 100% haarakattavuus ohjelmassa 2. . . 22

3 Vasemmalla esitetään työn kannalta merkittäviä käyttötapauksia, jotka sovelluksen täytyy toteuttaa. Oikealla puolella on esitetty, mitä rajapintametodeja kutsumalla High Speed Ethernet Client -kirjasto toteuttaa käyttötapauksen. 66

OHJELMALUETTELO

1 Virheen, vian ja häiriön esiintyminen koodissa. 9

2 Koodikattavuuden määrittäminen esimerkkikoodista. 22

3 Asynkroninen metodi, joka hakee robotilta uusimman hälytyksen. . . 49

4 Töiden prioriteetin kasvamista testaava yksikkötesti. 51

LYHENTEET JA TERMIT

ADS	Automation Device Specification. Sovelluskerroksen protokolla, joka mahdollistaa TwinCAT ekosysteemin ympärille rakennettujen laitteiden ja ohjelmien välisen tiedonsiirron
FTP	File Transfer Protocol. Internettiin liitettyjen koneiden väliseen tiedostonsiirtoon kehitetty sovelluskerroksen protokolla
HTTP	Hypertext Transfer Protocol. Tilaton sovelluskerroksen protokolla tiedonsiirtoon verkkoon liitettyjen laitteiden välillä
HSE	High Speed Ethernet Client. Protokolla, joka mahdollistaa Yaskawan robottiohjaimien ja kolmannen osapuolen ohjelmien välisen kommunikoinnin.
I/O	Input/Output. Kommunikointirajapinta, jonka avulla laitteet voivat lähettää tai vastaanottaa dataa toisiltaan.
IEC	International Electrotechnical Commission. Kansainvälinen sähköalan standardoimisjärjestö
IEEE	Institute of Electrical and Electronics Engineers. Maailman suurin kansainvälinen tekniikan alan järjestö, jonka tavoitteena on edistää teknologian kehitystä.
OPC	Open connectivity via open standards. Avoin standardi teollisuuden automaatiosovelluksien väliseen tiedonsiirtoon
PLC	Programmable logic controller. Ohjelmoitava logiikka
REST	REpresentational State Transfer. Arkkitehtuurityyli, jonka tarkoituksena on helpottaa hajautettujen hypermediajärjestelmien luontia ja organisointia
SFC	Sequental function chart. Vuokaavio. IEC 61131-3 mukainen PLC:n ohjelmointikieli.
SFTP	SSH File Transfer Protocol. Sovelluskerroksen protokolla tiedostojen salattuun siirtoon SSH-yhteyden avulla
SSH	Secure SHell. Istuntokerroksen protokolla salattujen yhteyksien luomista varten

ST Structured Text. Rakenteellinen teksti. IEC 61131-3 mukainen PLC:n ohjelmointikieli.

TLS Transport Layer Security. Salausprotokolla, jonka avulla kaksi ohjelmaa voivat kommunikoida suojatun yhteyden yli säilyttäen datan eheyden

TwinCAT Total Windows Control and Automation Technology. Ohjelmisto, jonka tarkoituksena on muuntaa Windows -pohjainen PC reaaliaikaiseksi PLC:ksi ja mahdollistaa sen ohjelmointi

UML Unified Modeling Language. Ohjelmistotekniikassa käytetty standardoitu graafinen mallinnuskieli

1. JOHDANTO

1.1 Motivaatio

Alati monimutkaisemmissa ohjelmistototeutuksissa ei ole itsestään selvää, että ohjelma toimii kaikissa tilanteissa määrittelynsä mukaisesti tai että sen määrittelyssä itsessään ei esiinny virheitä. Tärkeimpänä ohjelman laatuun vaikuttavista tekijöistä pidetään luotettavuutta. Sitä tutkivalle tieteenalalle on vakiintunut nimitys *software reliability engineering*. Eräs sen keskeisimmistä tutkimuskohteista on ohjelmiston *käyttövarmuus*, joka on arvio järjestelmän häiriövapaasta toiminnasta määrätyn aikaikkunan sisällä ennalta määrätyssä ympäristössä [1, s. 170].

Luotettavuuden laiminlyönti ohjelmistokehityksessä on isompien ohjelmistoprojektien osalta johtanut uutisotsikoihin, kuten esimerkiksi Suomen junaliikenteestä vastaavan VR:n uuden myyntijärjestelmän käyttöönoton yhteydessä tapahtui vuonna 2011. VR:n tietohallintojohtaja Jukka-Pekka Suonikon mukaan järjestelmän käyttöönoton yhteydessä kävi ilmi, että järjestelmän arvioitu 0,4 miljoonan asiakasmäärä oli ensimmäisenä päivänä kolminkertainen ja käytetystä sovellusalustasta (*application server*) löytyi virhe, joka lisäsi järjestelmän kuormitusta. Järjestelmän häiriö ilmeni lopulta siten, että lipunmyyntijärjestelmä jouduttiin sulkemaan useiden ensimmäisten päivien ajaksi, jolloin junalippuja sai ostettua vain konduktööreiltä. [2] [3]

VR:n esimerkki on yksi monesta ohjelmistoprojektista, jossa luotettavan ohjelmistokehityksen keinojen soveltaminen olisi voinut olla edistämässä parempaa lopputulosta. Luotettavuus ei ilmesty itsestään osaksi ohjelmistototeutusta, vaan se rakennetaan siihen.

1.2 Tavoite

Tämän diplomityön tarkoituksena on selvittää kuinka ohjelmistokehityksen keinoin voidaan rakentaa ohjelma, jonka käyttövarmuusaste on korkea. Keinoja sovelletaan teollisuusrobotin ohjainsovelluksen toteuttamisessa. Taustalla on hitsaus- ja tuotan-

toautomaatio toimittaja Pemamekin aiemmin aloittama keskeneräinen ohjelmistoprojekti, jonka käyttövarmuudessa on ilmennyt ongelmia. Tavoitteen toteutumista arvioidaan ohjelmistokehittäjän, järjestelmätestauksen ja asiakkailta saatavan palautteen avulla. Työn toisena tavoitteena on, että korkeaan käyttövarmuuteen tähtäävän ohjelmistokehityksen keinoja sekä niiden soveltamisesta hankittua kokemusta voidaan hyödyntää uusissa alkavissa ohjelmistoprojekteissa.

1.3 Tutkimusmenetelmät

Tutkimus koostuu kahdesta vaiheesta. Ensimmäisessä vaiheessa kartoitetaan luotettavan ohjelmistokehityksen keinoja kirjallisuustutkimuksena. Tavoitteena on löytää keinoja, joiden avulla ohjelmistotuotteen käyttövarmuutta voidaan kasvattaa.

Toisessa vaiheessa toteutetaan tapaustutkimus, jossa sovelletaan kirjallisuudesta löytyneitä keinoja teollisuuden käyttöön tarkoitetun robotin ohjainsovelluksen toteutuksessa. Samalla arvioidaan, kuinka keinojen soveltaminen vaikutti ohjelman käyttövarmuuteen.

1.4 Työn rakenne

Luvussa 1 kerrotaan, miksi ohjelmiston käyttövarmuus on tärkeää ja minkä takia sitä kannattaa tavoitella ohjelmistokehityksessä. Luvussa esitetään työn tavoite ja millä tutkimusmenetelmillä tavoite pyritään saavuttamaan, sekä kuvataan työn rakenne.

Luvussa 2 kuvataan työn taustalla oleva projekti ja sen asettamat reunaehdot työn toteutukselle. Luvussa pohditaan syitä, jotka johtivat juuri tämän diplomityön kirjoittamiseen käyttövarmuuden näkökulmasta.

Luku 3 sisältää työn teoreettisen tarkastelun ja alkaa esittelemällä luotettavuuden käsitteitä. Tämän jälkeen esitetään kirjallisuudesta löytyviä keinoja, joiden avulla voidaan lisätä ohjelmistokehityksen luotettavuutta, johon myös käyttövarmuus kuuluu.

Luvussa 4 kuvataan työn toteutusympäristö. Tämä sisältää kuvauksen työssä käytettävästä PLC:stä, tiedonsiirtoprotokollista ja tekniikoista.

Luvussa 5 kerrotaan työn kohteena olevasta robottisolusta, jota soluohjaimen on tarkoitus hallita. Luvussa esitellään robottisolu, soluohjain, ja kerrotaan mitä ohjelmistovaatimuksia soluohjaimelta vaaditan. Luvussa kuvataan robottisolun työkierto eli

minkälaisen työprosessin soluohjaimen käyttö mahdollistaa. Laitteiden välisen viestiliikenteen yleiskuvan luomiseksi luvussa kuvataan soluohjaimen tärkein tehtävä, joka on mahdollistaa robotin töiden käynnistäminen.

Luvussa 6 on kuvattu ja arvioitu kuinka järjestelmän luotettavuutta on pyritty lisäämään, miten toteutus ratkaisee luvussa 2.5 kuvatut ongelmat ja miten luvussa 5.2 kuvatut ohjelmistovaatimukset toteutuvat. Luku on jaettu osiin siten, että tarkasteltavana on kerrallaan yksi järjestelmän kolmesta toteuttavasta laitteesta. Luvussa kerrotaan järjestelmälle suoritetusta testaamisesta, jonka avulla pyritään osoittamaan, että soluohjaimelle asetetut ohjelmistovaatimukset on saavutettu. Luvun lopussa käydään läpi vaihtoehtoinen toteutus, jonka sopivuutta korvaavana tekniikkana arvioidaan.

Luvussa 7 arvioidaan kuinka luvun 3 kirjallisuudesta löytyneiden keinojen käyttö lisäsi ohjelman käyttövarmuutta ja kuinka tämä on verrattavissa kirjallisuudessa esitettyyn.

Luku 8 päättää työn. Siinä kootaan tiivistetysti yhteen, kuinka soluohjainta ja kehitysprosessia voidaan tulevaisuudessa uudistaa.

2. LÄHTÖTILANNE

2.1 Pemamek

Työ toteutettiin yhdessä suomalaisen hitsaus- ja tuotantoautomaatioon erikoistuneen yrityksen Pemamekin kanssa. Se valmistaa sekä suunnittelee automatisoituja hitsaus- ja tuotantojärjestelmiä, sekä työkappaleiden käsittelylaitteita. [4]

Pemamek toimittaa automaatioratkaisuja muun muassa konepajateollisuuteen, liikkuvien työkoneiden hitsaukseen, rakennusteollisuuden teräsrakenteiden hitsaukseen, laivanrakennukseen, kattilateollisuuteen, tuulivoimateollisuuteen sekä prosessi- ja ydinvoimateollisuuteen. Toimitukset ovat maailmanlaajuisia ja niistä yli 90% menee vientiin. [4]

Useat Pemamekin hitsausautomaatioon liittyvät toimitukset ovat tarkoitettu raskaiden kappaleiden hitsaukseen, jossa materiaalipaksuudet ja kappaleiden koko ovat merkittäviä. Tällä on merkitystä yhden kappaleen käsittelyyn kuluvaan aikaan, joka on suurikokoisilla kappaleilla kuten tuulivoimaloiden osilla, useita tunteja. [5]

2.2 Alkuasetelma

Tullessani ensimmäisen kerran mukaan projektiin, jonka tavoitteena oli tuotteistaa soluohjainsovellus, oli sen toteutus vielä hyvin keskeneräinen. Merkittävä osuus sen tärkeimmistä toiminnoista oli toteuttamatta tai toimi epäluotettavasti, minkä vuoksi se ei ollut vielä missään tuotantokäytössä. Tässä vaiheessa soluohjaimen vanha projektiryhmä siirtyi tekemään muita projekteja, ja soluohjaimen kehittämisestä tuli minun tehtäväni.

Projektiin perehtyminen oli aluksi aikaa vievää, sillä siitä uupui ohjelmiston vaatimukset ja muut kirjalliset dokumentaatiot. Lähtötilanteena oli määrittelemätön järjestelmä, jonka toiminta ei vastannut haluttua.

Koska kirjalliset dokumentaatiot puuttuivat, pidettiin työn alussa useita palavereja niin vanhan projektiryhmän kuin toimiala-asiantuntijoiden kanssa, joissa kartoi-

tettiin soluohjaimelta vaadittavia toiminnallisuuksia ja reunaehtoja. Aikaisessa vaiheessa kävi ilmi, että työssä käytettävät teknologiavalinnat olivat rajoitettu kuten luvussa 2.3 kerrotaan. Haastattelujen myötä selvisi, että soluohjaimen tulee kyetä suorittamaan sille asetettuja tehtäviä keskeytyksettä, sillä sen elinkaari osana tuotantoprosessia on pitkä ja sen käyttöaste tulee olemaan korkea.

2.3 Rajoitteet

Työn teknologiavalintoihin liittyy rajoitteita, jotka ovat peräisin työn tilaajalta – Pemamekilta. Pemamek on solminut kumppanuusohjelmia muun muassa japanilaisen Yaskawan kanssa, joka toimittaa heidän robotiikkaan liittyvät laitteet, joihin kuuluu myös tässä työssä käytetty Motoman DX200-robottiohjain. Automaatiojärjestelmän alustana Pemamekilla on käytössä saksalaisen Beckhoffin toimittamat järjestelmät.

Toimijoiden kanssa pidempään kestänyt yhteistyö on kerryttänyt yrityksen työntekijöille kokemusta ja osaamista, jonka vuoksi teknologioiden vaihtaminen toiseen tarkoittaisi vanhasta kerätystä tiedosta luopumista ja henkilökunnan mittavaa uudelleen kouluttamista. Tämä ei ole järkevää, minkä vuoksi nämä teknologiat sanelevat reunaehtoja työn teknologioiden valinnassa.

Beckhoffin käyttö ohjaa myös ohjelmointikielen valintaa, sillä valmistajan tarjoamat kirjastot on tarkoitettu käytettäväksi Windows ympäristössä. Myös useat soluohjaimen toimitusprojektiin liittyvät oheisohjelmat toimivat ainoastaan Windows -ympäristössä. Koska työn kirjoittaja omaa useamman vuoden kokemuksen .NET -ympäristössä työskentelystä, toteutuskieleksi työhön valikoitui luontevasti $C\#$, josta oli työn kirjoitushetkellä käytössä versio 6.0.

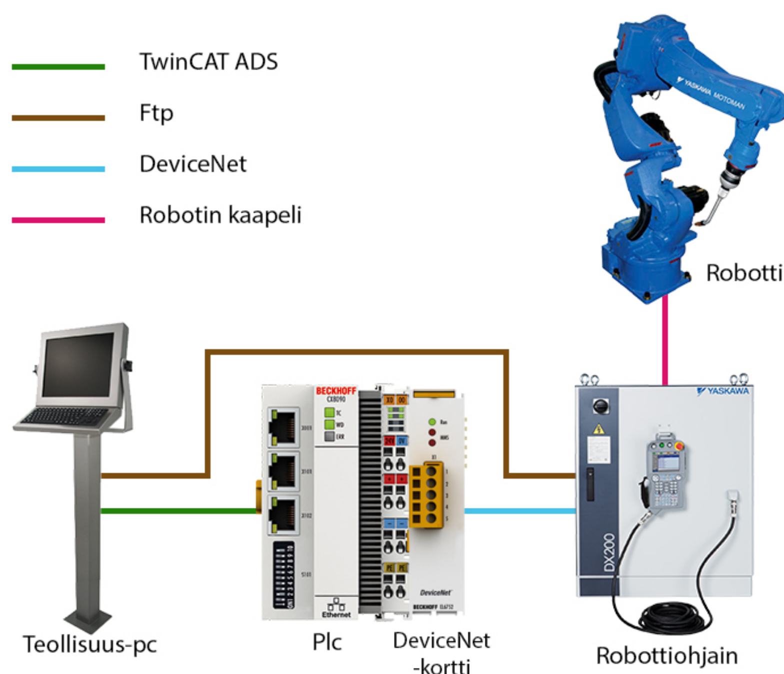
Koska soluohjaimen toimitukselle oli asetettu toimitusaikataulu, ei ohjelmistoa voinut lähteä toteuttamaan täysin uudelleen, vaan toimivaksi todettuja tekniikoita tuli hyödyntää. Tämän vuoksi toteutetulle soluohjaimelle esitetään myös vaihtoehtoinen toteutus luvussa 6.5, jonka sopivuutta arvioidaan siitä löytyvän teknisen dokumentaation pohjalta.

2.4 Järjestelmäkuvaus

Järjestelmää on tarkoitus käyttää kappaletavarateollisuudelle tarkoitettujen robottisolujen ohjainohjelmistona – soluohjaimena. Robottisolulla tarkoitetaan kokonaisuutta, joka sisältää muun muassa robotin, robottiohjaimen, kappaleenkäsittelypöytiä ja turvallisuuteen liittyviä laitteita. Robotin tarkoituksena on osallistua jonkin

kappaleen jatkokäsittelyyn. Pemamekin tapauksessa tähän liittyy useimmiten kappaleen työstämistä robottihitsauksen avulla.

Kuvassa 1 esitetään laitteet ja niiden väliset kommunikointiprotokollat. Teollisuus-PC:llä ajettava ohjainsovellus, jota nimitetään työssä *soluohjainsovellukseksi*, on yhteydessä Beckhoffin CX8090 ohjelmoitavaan logiikkaan *TwinCAT ADS* -protokollan avulla. Työssä käytetään termin *ohjelmoitava logiikka* sijasta johdonmukaisesti kirjainlyhennettä PLC, joka tulee sanoista *Programmable Logic Controller*. ADS -yhteyttä hyödynnetään, kun robotille annetaan ohjauskomentoja. Teollisuus-PC:n ja robottiohjaimen välisen FTP-yhteyden avulla sovellus puolestaan siirtää robotille työtiedostoja, jotka voivat esimerkiksi sisältää tiedon siitä, kuinka robotin tulee hitsata tietynlainen työkappale. PLC:hen on kiinnitetty EL6752 DeviceNet-kortti, jonka avulla se kommunikoi DeviceNet-väylää pitkin DX200-robottiohjaimelle, joka puolestaan välittää ohjauskomennot servomoottorien liikkeeksi varsinaiselle Motoman robotille.



Kuva 1 *Järjestelmän yleiskuvaus sekä sen muodostavien laitteiden väliset yhteydet ja tiedonsiirtoprotokollat.*

Järjestelmän tämän hetkisessä kehitysvaiheessa tavoitellaan aluksi yhden robotin ohjausta, mutta jatkokehitysvaiheessa ohjattavien robottien määrää on tarkoitus kasvattaa. Termejä robottiohjain ja robotti käytetään työssä useimmiten tarkoittamaan samaa asiaa eli robottiohjainta. Tärkeintä on ymmärtää, että robotti ei vastaanota signaaleja muilta laitteilta kuin robottiohjaimelta, jonka takia kaikki viestiliikenne

kulkee sen kautta.

2.5 Vanhan soluohjaimen ongelmia

Seuraavaksi esitetään keskeisiä ongelmia, joihin tulee työn edetessä löytää ratkaisu, jotta soluohjaimen tuotantokäyttö olisi mahdollista. Useimmat ongelmista liittyvät soluohjaimen käyttövarmuuteen. Suurin osa niistä tuli ensimmäisen kerran esille, kun soluohjaimen muodostavat ohjelmamoduulit saatiin otettua testikäyttöön tuotantoasemaa vastaavalle asemalle, jossa niille suoritettiin *integraatiotestausta*. Integaatiotestauksessa ohjelman muodostavat komponentit liitetään yhteen ja testataan niiden yhteistoimintaa [1, s. 112] eli tässä tapauksessa koko soluohjaimen toimintaa tuotantokäyttöä vastaavalla testiasemalla.

Työjonoon peräkkäin ajettavista robotin töistä vain osa käynnistyi, ja robotti saattoi pysähtyä kesken työn suorituksen ilman syytä. Pysähtymisen jälkeen oli mahdollista, että robotti lukkiutui, jolloin se ei vastannut soluohjaimelta tuleviin pyyntöihin.

Sovelluksen kautta käynnistettävä työ ei ajoittain käynnistynyt. Sovellus ei indikoinut ongelmasta, vaan jätti käskyn suorittamatta. Tässä tilanteessa robotin työ täytyi käynnistää *käsiohjaimesta*. Käsiohjain on robotin ohjaamiseen tarkoitettu laite, jonka päätarkoituksena on mahdollistaa robotin liikeratojen ohjelmointi ja tarjota käyttöliittymä näiden suorittamiseksi.

Soluohjaimen vanha arkkitehtuuri oli vahvasti riippuvainen graafisesta käyttöliittymästä, joka aiheutti useita ohjelmiston todentamiseen liittyviä ongelmia. Ohjelman testaaminen oli hankalaa, ja automaattisten testien luominen järkevällä työmäärällä mahdotonta, sillä kirjoitetut testit riippuivat ohjelman logiikan lisäksi graafisesta käyttöliittymästä. Näin ollen teollisuus-PC:n ja robotin välisen kommunikoinnin todentaminen määrittelynsä mukaiseksi oli osoittautunut käytännössä erittäin haastavaksi.

Uusien asiakkuuksien myötä soluohjaimelta vaadittavat toiminnalliset ominaisuudet lisääntyivät ja tuli tarpeelliseksi, että myös muut ohjelmat pystyisivät hallitsemaan robottisolua. Näissä tilanteissa olisi tarvittu käyttöön vain robottisolun kanssa kommunikointiin kykenevä osuus ilman graafista käyttöliittymää. Vanha arkkitehtuuri ei kuitenkaan mahdollista käyttöliittymän ja logiikan irrottamista toisistaan. Jos vanhasta arkkitehtuurista ei luovuta, tulee soluohjaimen olla taustalla käynnissä, kun muut ohjelmat lähettävät viestinsä sen kautta robotille, jonka seurauksena järjestelmän monimutkaisuus lisääntyy ja heikentää sen käyttövarmuutta.

Vanha toteutus oli rakennettu synkroniseksi. Ohjelma jäi odottamaan kaikkia ro-

botille lähteviä viestejä. Tämä aikaansai sen, että käyttöliittymä saattoi pysähtyä pahimmillaan useiksi sekunneiksi toimintojen välissä. Tänä aikana ohjelmaa ei voinut käyttää. Mikä tahansa pitkäkestoisempi toiminto pysäytti käyttöliittymäsäikeen suorituksen. Tämä aikaansai sen, että käyttäjälle välittyi kuva sovelluksen kaatumisesta.

3. LUOTETTAVUUTTA LISÄÄVÄT KEINOT

3.1 Virhe, vika ja häiriö

Ennen kuin voidaan kuvata keinoja lisätä ohjelman luotettavuutta, tulee ymmärtää mitkä tekijät heikentävät sitä. Laprie määrittelee ohjelman luotettavuutta heikentäviksi käsitteiksi: virheen, vian ja häiriön [6]. Puhekielessä termejä käytetään usein ristiin ja niiden käyttö ei ole vakiintunutta edes kirjallisuudessa, kuten standardi [1, s. 86] toteaa asian olevan. Tässä työssä termejä käytetään seuraavasti.

Virhe (*fault*) on syy, joka johtaa vikaan [6, s. 4]. Lähdekoodin 1 rivillä 1 esiintyy ohjelmavirhe, josta käytetään myös nimitystä bugi [1, s. 86]. Siinä muuttujaan suoritetaan vertailun sijasta sijoitus. Tässä työssä ohjelmistossa esiintyvistä virheistä käytetään bugi termin sijasta johdonmukaisesti termiä virhe.

```
1 if(IsNormalDrivingMode = true)
2 {
3     return;
4 }
5 else
6 {
7     EnableCruiseControlMode();
8 }
```

Ohjelma 1 *Virheen, vian ja häiriön esiintyminen koodissa.*

Vika (*error*) on järjestelmän tila, jonka on mahdollista johtaa häiriöön [6, s. 4]. Esimerkki tällaisesta tilasta on, kun virheen sisältämä ohjelmakoodi suoritetaan. Lähdekoodissa 1 se tarkoittaa rivien 1-4 suoritusta. Muita esimerkkejä viasta ovat asiakasvaatimusten väärinymmärtäminen tai niissä esiintyvä puute [7, s. 2].

Häiriö (*failure*) on viasta aiheutuva poikkeama ohjelman suorituksessa, joka ei vastaa ohjelman määrittelyä, josta käy ilmi, kuinka ohjelman halutaan toimivan [1, s. 86]. Lähdekoodista 1 nähdään, että ohjelma toimii oikein, niin kauan kuin ajotilaksi

ei yritetä asettaa vakionopeudensäätimellä tapahtuvaa ajoa, sillä kyseinen osa koodia on virheen takia saavuttamattomissa. Yksinkertainen koodiesimerkki osoittaa, että kaikki virheet ja niistä seuraavat viat eivät johda häiriöön. Tämän seurauksena ohjelmaan voi jäädä piileviä virheitä, joiden esiintymistä ei huomata, mutta jotka voivat myöhemmässä kehitysvaiheessa tulla esille, kun ohjelmaa muutetaan.

3.2 Keinojen ryhmittely

Ohjelman käyttövarmuutta voidaan parantaa lisäämällä sen luotettavuutta. Luotettavuutta lisääviä keinoja on tutkittu vähintään 50 vuotta [8]. Tänä aikana on kertynyt lukuisia keinoja, minkä vuoksi niiden ryhmittelyn tarve on ilmeinen. Ryhmät selventävät missä ohjelman elinkaaren vaiheessa ja millä tavalla keinoja voi soveltaa. Tässä työssä luotettavuutta lisäävät keinot ryhmitellään neljään ryhmään:

- virheiden välttämiseen
- vikasietoisuuteen
- virheiden poistamiseen ja
- virheiden ennustamiseen.

Tässä muodossa keinot on esittänyt ensimmäisen kerran Laprie [6]. Samankaltaiseen jakoon ovat päätynyt myös Heimerdinger ja Weinstock [9] sekä Anderson [10, s. 4]. Anderson tosin jättää virheitä ennustavan ryhmän kokonaan pois.

Virheiden välttämis- ja vikasietoisuustekniikat pyrkivät mahdollistamaan sellaisten ohjelmien toteuttamisen, joihin voidaan luottaa. Virheiden poistaminen ja -ennustaminen puolestaan tarjoavat työkaluja, joiden avulla voidaan varmistua siitä, että tähän tavoitteeseen on päästy. [8] On ehdotettu, että näiden tekniikoiden käytön tulisi kuulua jokaiseen ohjelmistokehitysprosessiin omana vaiheenaan erityisesti ohjelmistokehityksessä, joka tähtää luotettavan ohjelmistotuotteen toteuttamiseen [11]. Lyu toteaa, että paras tapa saavuttaa luotettavasti toimiva järjestelmä, on hyödyntää tekniikoista jokaista kehityksen aikana [12, s.33]

Tässä työssä ei pyritä esittelemään kaikkia luotettavuutta lisääviä keinoja, vaan keskitytään aluksi esittelemään ne keinot, jotka ovat työn soveltavassa osuudessa käytössä. Valintakriteereinä keinoille ovat toimineet mm. niiden soveltuvuus ketterään ohjelmistokehitykseen (*agile software development*) ja se että niitä on voinut soveltaa jo käynnissä olevalle projektille. Näiden lisäksi käydään läpi kuinka käyttövarmuuden kasvattaminen ohjelmaan voisi toimia, jos projekti aloitettaisiin puhtaalta

pöydältä. Näiden periaatteiden käyttöönottaminen jo käynnissä olevalle projektille on sanottu olevan erittäin tuskallista [13, s. 59], mikä johtuu osittain siitä, että ohjelman arkkitehtuurin täytyy mm. tukea kattavaa automaatista testaamista.

3.3 Virheiden välttäminen

Virheiden välttämisessä on tarkoituksena parantaa ohjelman luotettavuutta estämällä virheiden syntyminen ohjelmistoon jo suunnitteluvaiheessa tai minimoimalla virheiden syntymisen todennäköisyys kehityksen aikana. Se on ensimmäinen keino luotettavamman ohjelman aikaansaamiseksi. Sen käyttöä on helppo perustella kuten Lyu toteaa asian olevan:

> A fault which is never created costs nothing to fix.[14]

Koska virheiden välttämistekniikat ovat ohjelmiston elinkaaren ensimmäinen keino luotettavamman ohjelman aikaansaamiseksi, sen aikana tehdyillä ratkaisuilla on muita keinoja suurempi vaikutus projektin lopulliseen onnistumiseen tai epäonnistumiseen [8].

3.3.1 Ohjelmiston vaatimusten määrittely

Ohjelmiston vaatimusten määrittelyä pidetään yleisesti ohjelmistokehityksen yhtenä tärkeimmistä osuuksista. Riittämätön informaatio ohjelmiston käyttäjältä, puutteelliset vaatimukset, muuttuvat vaatimukset tai väärinymmärretyt tavoitteet ovat suurimpia syitä miksi ohjelmistoprojektit epäonnistuvat [15, s. xxv].

Alun perin ohjelmiston vaatimusten määrittely on tapahtunut projektin elinkaaren alussa laatimalla kirjallinen dokumentti, jota on nimitetty *ohjelmiston vaatimusmäärittelyksi*. Dokumentista on käynyt ilmi mitä toiminnallisuuksia, tehokkuusvaatimuksia, rajoitteiden noudattamista tai muita ominaisuuksia ohjelmalta vaaditaan [1, s. 187]. Onnistunut vaatimusten määrittely sisältää edelleen samojen asioiden löytämistä, mutta niiden kaikkien tunnistaminen heti projektin alussa ei ole tarpeellista, vaan uusia vaatimuksia syntyy kehitysprosessin edetessä, kun tietämys kehitettävästä asiasta lisääntyy. Näin toimivat esimerkiksi ketterät ohjelmiston projektinhallintamenetelmät kuten *scrum*.

Ohjelmiston vaatimusten määrittelyn laiminlyönnin sanotaan johtavan useimmissa tapauksissa kehnoon ohjelmistoon, eikä sen luotettavuutta voida varmistaa pelkästään vikasietoisuustekniikoita hyödyntämällä [16, s. 22]. Ohjelmistovaatimusten

tulisi olla tarkkoja, eivätkä ne saisi jättää tilaa tulkitsemiselle. Esimerkiksi ilmailualan kaupallisten lentokonejärjestelmä ohjelmistojen hyväksymiseen keskittyvän dokumentin *DO-178C, Software Considerations in Airborne Systems and Equipment Certification* perustana pidetään juuri laadukkaita ohjelmistovaatimuksia [17, s. 97]. Autojen turvallisuuteen keskittyvä standardin *ISO 26262-1:2011* yhtenä osana on kuvata kuinka toiminnallinen turvallisuus huomioidaan vaatimusten määrittelyssä. *ISO/IEC 25030:2007 Software product Quality Requirements and Evaluation (SQuaRE)* puolestaan pyrkii tarjoamaan keinoja, joiden avulla voidaan tunnistaa ohjelmiston luotettavuusvaatimuksia.

Ohjelmiston vaatimusten määrittely alkaa ohjelmistovaatimusten selvittämisellä ja analysoinnilla. Tähän kuuluvat myös luotettavuudelle asetettavien vaatimusten tunnistaminen. Tarkoituksena on ymmärtää määriteltävä tuote. Vaatimusten tarkempi analysointi, esimerkiksi useammasta näkökulmasta tapahtuva tarkastelu käyttötapauskuvausten avulla, vähentää tarvetta vaatimusten uudelleen kirjoittamiselle. Käyttötapauskuvausten tarkoitus on esitellä määrämuotoisena dokumenttina minkälaisilla vaiheilla ohjelma ja sen käyttäjä pääsevät haluttuun tavoitteeseen. [17, s. 107-109]. Scrumissa tätä tarkoitusta varten käytetään käyttäjätarinoita (*user story*), jotka ovat lyhyitä kuvauksia ohjelman käyttäjän näkökulmasta kirjoitettuna.

Vaatimusten kirjoittamisessa tulisi välttää toteutusyksityiskohtia ja keskittyä kuvaamaan vaatimukset korkeammalla tasolla, mutta esittämään asiat kuitenkin riittävän yksiselitteisesti. Tässä apuna voivat toimia kuvat ja kaaviot. Jokainen vaatimus tulisi priorisoida, jotta kehitysprosessi keskittyisi toteuttamaan tärkeimpiä tai kiireellisimpiä asioita ensin. Prioriteetin merkitystä voi korostaa perustelemalla osana vaatimusta, miksi sen merkitys koko järjestelmän kannalta on tärkeää. [17, s. 109-119].

Vaatimuksia dokumentoidaan useammasta näkökulmasta. Eniten on toiminnallisia vaatimuksia. Ne määrittelevät toiminnon, jonka järjestelmän tai järjestelmän komponentin täytyy toteuttaa [1, s. 96]. Ei-toiminnallisiin vaatimuksiin kuuluvat toiminnallisten vaatimusten ulkopuolelle jäävät vaatimukset, jotka voivat liittyä esimerkiksi ohjelmalle asetettaviin luotettavuus-, turvallisuus- tai käytettävyysvaatimuksiin. Lisäksi tulisi dokumentoida järjestelmän käyttämät rajapinnat, joihin kuuluvat: käyttöliittymä, laitteisto, ohjelmakirjastot ja kommunikointi. Yksinkertaisimmillaan näiden dokumentointi voi tapahtua viittaamalla rajapinnan omaan dokumentaatioon. [17, s. 115-119].

Lopuksi vaatimukset tulisi validoida, jotta niissä esiintyvät virheet saataisiin kiinni ennen kuin vaatimuksia aloitetaan toteuttamaan. Todentamisessa voidaan hyö-

dyntää vertaisarviointia, joka voi kevyimmillään tapahtua kehitysryhmän sisällä. Kehitysryhmä määrittelee uusia vaatimuksia ja samaan aikaan tarkastaa toistensa määrittelemiä vaatimuksia. Virallisemmassa validoinnissa on useampia osapuolia mukana ja siinä edetään ennalta laaditun dokumentin mukaan, josta käy ilmi mitä asioita validoinnissa tulee huomioida. [17, s. 123-125]

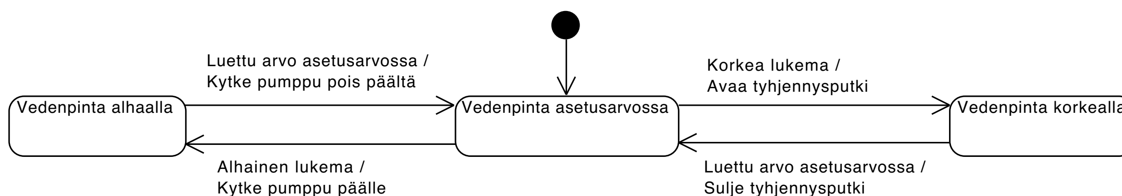
Vaatimusten määrittely on iteroiva prosessi, kun puutteita huomataan, missä tahansa vaiheessa, palataan takaisin alkuun — vaatimusten selvitys ja analysointi -vaiheeseen. Vaatimusten jälkikäteen tapahtuvan muuttamisen on toimittava tehokkaasti, jotta vaihe ei jäisi suorittamatta. Sähköpostin sijaan voidaan hyödyntää tähän tarkoitukseen laadittuja vaatimustenhallinta -ohjelmia. [17, s. 128-130] Ohjelmiston vaatimusten määrittelystä löytyy useita kirjoja, kuten [17] ja [15], joista prosessista voi lukea tarkemmin. Laadukkaiden vaatimusten laatimisesta ohjeistetaan myös standardissa *ISO/IEC/IEEE 29148:2011*.

3.3.2 Tilakoneanalyysi

Tilakoneanalyysi perustuu nimensä mukaisesti tilakone -mallin analysointiin. Tilakone on ohjelmistojen suunnitteluvaiheessa yleisesti käytössä oleva mallinnusmenetelmä, jonka hyödyntäminen erityisesti ohjelman todentamisvaiheessa on tyypillistä [18][19]. Tilakoneanalyysiä voidaan käyttää muun muassa turvallisuus- ja vikasietoisuussuunnitelmien analysointiin, ohjelman turvallisuusvaatimusten määrittämiseen, turvallisuuskeskeisten toimintojen havaitsemiseen ohjelmasta ja vikojen havaitsemiseen ja niistä toipumiseen. Sen käyttöä ennen ohjelman yksityiskohtaisten suunnitelmien tekemistä pidetään tehokkaimpana, mutta se soveltuu käytettäväksi missä tahansa projektin vaiheessa esimerkiksi vaihtoehtoisten toteutusten etsimisessä. [20, s. 92]

Tilakone on järjestelmän tiloja ja niiden välistä vaihtumista kuvaava malli. Tiloja merkitään UML:n mukaisessa notaatiossa kuvan 2 mukaisesti pyöristettyinä suorakaiteina ja niiden välisiä siirtymiä nuolilla. Kun siirtymän ehto toteutuu järjestelmän ollessa siirtymää edeltävässä tilassa, järjestelmä siirtyy uuteen tilaan ja toteuttaa tilan mukaisen toiminnon, joka voi olla esimerkiksi ulostulon päälle kytkeminen. Vaihtoehtoisesti toiminto voidaan merkitä tapahtuvaksi myös siirtymäehtoon kuten kuvassa 2 on tehty. Täytetty musta ympyrä kuvaa aloitustilaa, josta mallin kuvaama järjestelmä käynnistyy.

Tilakoneessa 2 tarkkaillaan vedenpinnan korkeutta säätävää järjestelmää. Riippuen vedenpinnan korkeutta mittaavan sensorin mittaustuloksesta, järjestelmä kytkee pumpun päälle ja pois päältä, sekä avaa ja sulkee tyhjennysputken.



***Kuva** 2 Tilakone UML -syntaksilla. Muokattu lähteestä [21, s. 347].*

Järjestelmän tilakoneen valmistuttua suoritetaan vaaratilanteiden tunnistaminen, jossa voidaan käyttää apuna toimiala-asiantuntijoita. Vaaratilanteella tarkoitetaan järjestelmän toiminnanaikaista tilaa, joka voi aiheuttaa onnettomuuden [22, s. 184]. Suositeltu tapa toteuttaa tämä on aloittaa etsimällä mallista vaaratilanne, ja pyrkiä korjaamaan mallia siten, että vaaratilannetta ei pääse syntymään. Menetelmän ongelmana pidetään sitä, että korjattuihin vaaratilanteisiin ei välttämättä ole mahdollista edes päästä, joten vaaratilanteita saatetaan poistaa enemmän kuin niitä oli alun perin olemassa. [20, s. 92]

3.3.3 Muita analyysimenetelmiä

Tilakoneanalyysin lisäksi on olemassa myös muita analyysejä, joiden avulla voidaan pyrkiä estämään virheiden syntymistä ohjelmaan. Lyhyesti esiteltynä näitä ovat muun muassa: HAZOP, FMEA, FTA, ETA ja SSA.

HAZOP (Hazard and Operability Study) on poikkeamatarkastelu, jonka käyttöä suositellaan ryhmätyöskentelyn muodossa toimintaparametrien muutoksista tai puutteellisesta suunnittelusta johtuvien häiriöiden ja niiden aiheuttamien onnettomuusriskien kartoittamiseen. [23, s. 66-67]

FMEA (Failure Mode and Effect Analysis) eli vika-, ja vaikutusanalyysi. Sen avulla selvitetään ohjelman kaikkien moduulien vioittumistavat. Sitten vian vaikutusta moduulin käyttövarmuuteen arvioidaan, jonka jälkeen siirrytään arvioimaan vikojen vaikutusta koko järjestelmän näkökulmasta. [24, s. 179]

FTA (Fault Tree Analysis) vikapuuanalyysi eroaa muista analyysimenetelmistä sillä, että sen avulla pyritään tarkastelemaan ohjelmaa eri näkökulmasta ylhäältä alaspäin. Siinä lähdetään liikkeelle määrittelemällä vaaratapahtumia, joihin johtavia syitä lähdetään tämän jälkeen etsimään loogisella kaaviolla. Vaaratapahtumaan johtavat syyt ja näiden syyt alkavat pian muistuttaa puumaista rakennetta. Sen käyttöä suositellaan, kun on tarve selvittää ja poistaa vaaratapahtuman aiheuttaja tai katkaista syyketju, joka siihen johtaa. [23, s. 64-66]

ETA (Event Tree Analysis) on tapahtumapuuanalyysi, joka muistuttaa vikapuuanalyysiä, mutta siinä aloitussolmuna on vioittumisen syy, jota lähdetään pilkkomaan tunnistamalla siitä aiheutuvia seurauksia. Se soveltuu syy-seuraussuhteeltaan yhdensuuntaisten tai toisistaan riippumattomien tapahtumien mallintamiseen ajallisesti etenevissä tapahtumaketjuissa. [23, s. 68]

SSA (Software Sneak Analysis) on ohjelmiston oikopolkuanalyysi, jossa pyritään paikantamaan odottamaton ohjelman suoritus, joka voi johtaa vaaratilanteeseen tai olla esteenä tietyn halutun tapahtuman suorittamiselle. Analyysin käyttö edellyttää ohjelmistodokumentaation lisäksi kokenutta tarkastelijaa. [23, s. 69-71]

3.4 Vikasietoisuus

Vikasietoinen järjestelmä pyrkii toimimaan vikojen ilmaantuessa virheettömästi. Vikasietoisuus saavutetaan erilaisilla ohjelman ajon aikana suoritettavilla tekniikoilla, joiden avulla pyritään poistamaan vikoja ennen kuin ne aiheuttavat ohjelmaan häiriöitä. Toisin sanoen järjestelmä pyrkii toimimaan oikein, vaikka sen lähdekoodissa esiintyisi virheitä. Vikasietoisuuden tekniikoilla ohjelman on mahdollista havaita ja eristää vikatilanteita sekä palautua niistä. [6, s. 23]

3.4.1 Poikkeustenkäsittely

Poikkeustenhallinnassa on vikasietoisuuden kannalta kyse keinosta palautua vikatilanteesta. Poikkeuskäsittelyä pidetään yhtenä tehokkaimmista tavoista käsitellä ohjelman ajonaikaisia vikatilanteita [25]. Suurin osa eniten käytetyistä ohjelmointikielistä sisältää tuen poikkeustenhallinnalle [26, s. 3].

Poikkeusten avulla tavallinen ohjelman suoritus erotetaan poikkeuksellisesta suorituksesta. Poikkeava suoritus voi olla esimerkiksi tilanne, jossa lukua yritetään jakaa nollalla. Poikkeustenkäsittelystä vastaava koodi eristetään omaksi koodilohkokseen, jota nimitetään poikkeuksenkäsittelijäksi. Muu osuus koodista sisältää tavallisen ohjelman suorituksen. Tämä tekee koodista luettavampaa. [26, s. 3]

Kun ohjelman suoritus havaitsee poikkeuksellisen tilanteen, se heittää poikkeuksen. Poikkeus on virhetilannetta varten luotu poikkeuksen sisältämä luokka, joka sisältää oleellista tietoa tapahtuneesta virheestä. Ohjelman suoritus siirtyy tämän jälkeen automaattisesti oikeaan poikkeuksenkäsittelijään, jonka tulisi tietää kuinka poikkeavasta tilanteesta toivutaan.

Poikkeus voi lentää syvällä funktioiden kutsuhierarkiassa, mutta poikkeuksen poikkeuskäsittelijä voi sijaita samassa hierarkiassa useita funktiokutsuja aikaisemmassa vaiheessa. Tämä mahdollistaa poikkeuksen käsittelyn siinä kohdassa ohjelmaa, jossa sen käsittely on mahdollista. Poikkeustenkäsittelijä valitaan sen mukaan, minkä tyyppinen poikkeus on heitetty. Jokaisella poikkeuksenkäsittelijällä on oma lohkonsa, jonka sisään kirjoitetusta koodista heitetyt poikkeukset voivat aktivoida poikkeuksenkäsittelijän. [26, s. 3]

C++ -kielen kehittäjä Stroustrupin mukaan poikkeustenkäsittelyn käytölle voidaan nähdä edellä listattujen lisäksi seuraavia etuja:

- Poikkeustenkäsittelyn avulla funktioiden paluuarvot voidaan varata oikein toimivan koodin käyttöön. Tällä vältytään ongelmalta, jossa funktio ei voi palauttaa sopivaa virhekoodia tilanteessa, jossa kaikki numeromuotoiset paluuarvot ovat sopivia arvoja
- Poikkeustenkäsittely yhdistää eri ohjelmamoduuleista koostuvan ohjelman virheenhallintamekanismin yhtenäiseksi [27, s. 355-388]

Poikkeuskäsittelymekanismin päälle on rakennettu useita poikkeustenkäsittely tekniikoita (*exception handling techniques*), joiden avulla ohjelman käyttövarmuutta voidaan kasvattaa. Eräs näistä pyrkii varmistamaan, että ohjelmassa on varauduttu odottamattomia poikkeuksia vastaan, joita nimitetään esimerkiksi C# ja Java ohjelmointikielissä nimellä *unhandled exception*. Sen seurauksena ohjelma kaatuu hallitsemattomasti. Näiden torjumiseksi Longshaw et al. esittää artikkelissaan [28] tekniikkaa, jota hän nimittää *Big Outer Try Block* ja Haase nimellä *Safety Net* [29, s. 27]. Tekniikka mahdollistaa häiriöiksi johtaneiden virheiden järjestelmällisen kirjaamisen koko ohjelman elinkaaren aikana. Sen käytön on sanottu auttavan ohjelman hallittuun alasajoon, jolloin ohjelman kaatumiseen johtaneiden syiden kirjaaminen on mahdollista [28] [29, s. 27] [30].

Poikkeustenkäsittelijä toteutetaan sovellyksen ylimmälle tasolle, josta sovellus käynnistyy. Esimerkiksi C# ohjelmassa se on main metodi. Tavallisesti tälle tasolle päätyvät poikkeukset ovat ohjelman suorituksen kannalta sellaisia, että ohjelman ajoa ei voida enää jatkaa turvallisesti. Tämän vuoksi poikkeuksen syy ja sijainti ohjelmassa tallennetaan lokiin, jonka jälkeen käyttäjälle esitetään viesti, jossa ohjelman sulkeutumista pahoitellaan.

3.4.2 Tilan tallennus

Tilan tallennusta pidetään yleisimmin mainittuna keinona palauttaa ohjelma vikatilanteesta [31, s. 12] [32]. Tekniikkana se on yleiskäyttöinen, mikä mahdollistaa sen käytön usealla järjestelmän tasolla. Se soveltuu käytettäväksi myös odottamattomiin vikoihin.

Tilan tallennuksessa ohjelma pyrkii tallentamaan tilansa muistiin. Tallennus voi tapahtua tasaisin väliajoin tai tietyssä kohtaa ohjelman suoritusta. Jälkimmäisessä tilanteessa puhutaan dynaamisista tilan palautusmenetelmistä. Ohjelman suorituksen yhteydessä tapahtuva tilan tallentaminen voisi tapahtua esimerkiksi eräajossa, jossa siirretään tiedostoja verkon yli. Aina 10 tiedoston siirtymisen jälkeen ohjelma tallentaa tilansa talteen. Jos ohjelmassa tapahtuu siirron aikana vika, se voi palautua jatkamaan tiedostojen siirtoa sen sijaan, että siirto tarvitsee aloittaa alusta. Staattisesta tilan palautuksesta on kyse, kun ohjelman moduulin tila palautetaan ohjelman rakennusvaiheessa määrättyyn tilaan. Näitä tiloja voi olla useampia ja se mihin tilaan moduuli palautetaan riippuu vian havaitsemishetkellä olevasta ohjelman suoritusvaiheesta. [31, s. 12]

Ongelmana tilan tallentamisessa pidetään toimintoja, joista ei voi palautua. Tällaisia toimintoja voivat aiheuttaa järjestelmään liittyvät ulkopuoliset järjestelmät, sillä näiden suora hallinta ei ole mahdollista. Jos ulkopuoliselle järjestelmälle on mahdollista lähettää virheellinen pyyntö, voi sen peruminen olla vaikeaa tai mahdotonta. Ääriesimerkkinä tällaisesta toiminnasta voidaan pitää ohjuksen laukaisua. Sen aiheuttaman tuhon peruminen tapahduttuaan on tuskin mahdollista. [33, s. 149]

3.5 Virheiden poistaminen

Virheiden syntymistä ohjelmaan ei voida täysin estää. Jo syntyneitä virheitä voidaan kuitenkin pyrkiä poistamaan ja sitä kautta lisäämään ohjelman luotettavuutta. Perusperiaate virheiden poistamisessa on, että ohjelman toimintaa verrataan sille asetettuihin vaatimuksiin. Jos ohjelma toimii vaatimuksen vastaisesti, se sisältää virheen. On kuitenkin hyvä huomata, että myös ohjelmistovaatimus voi olla virheellinen, jolloin se tulee korjata.

Virheiden poistaminen voidaan toteuttaa ohjelmiston elinkaaren jokaisessa vaiheessa aina ohjelmiston määrittelystä sen käyttöönottoon. Boehm kuitenkin osoittaa tutkimuksessaan [34], että mitä aikaisemmassa vaiheessa virheet niiden luomisen jälkeen löytyvät, sitä halvempaa on niiden korjaaminen. Hänen mukaansa virheen korjaaminen aikaisessa vaiheessa on $50 - 200$ kertaa kustannustehokkaampaa kuin saman

virheen korjaaminen myöhemmässä vaiheessa.

Tekniikat joiden avulla virheitä löydetään, voidaan jakaa staattisiin ja dynaamisiin. Staattisissa tekniikoissa ohjelmaa tai järjestelmää ei suoriteta. Näihin tekniikoihin kuuluvat mm. järjestelmälle tehtävät tarkastukset, tietovuoanalyysi, kompleksisuusanalyysi ja kääntäjän tarkistukset.

Dynaamisissa tekniikoissa järjestelmä tai ohjelma on ajossa ja käytössä olevat tekniikat liittyvätkin yleensä ohjelman testaamiseen. Vaikka luotettavuutta ei voida testata osaksi ohjelmaa, sen avulla voidaan kuitenkin varmistua siitä, että ohjelmassa ei esiintyisi virheitä tai niiden esiintyminen olisi vähäisempää. Seuraavaksi keskitytään käymään läpi dynaamisten tekniikoiden muotoja. [23, s. 22-23]

3.5.1 Yksikkötestaus

Yksikkötestauksen kuten minkä tahansa testauksen muodon tavoitteena on suorittaa ohjelmaa, jotta siitä löytyisi virheitä. Yksikkötestauksessa keskitytään testaamaan, koko järjestelmän sijaan, pienempiä ohjelman rakenteita kuten ohjelman funktioita, luokkia tai moduuleita. Yksikkötestille voidaan esittää myös sen loogiseen tarkoitukseen pohjautuva määritelmä. Osherove määrittelee yksikkötestin kirjassaan seuraavasti:

> A unit test is an automated piece of code that invokes the unit of work being tested, and then checks some assumptions about a single end result of that unit. A unit test is almost always written using a unit testing framework. It can be written easily and runs quickly. It's trustworthy, readable, and maintainable. It's consistent in its results as long as production code hasn't changed. [35, s. 11]

Määritelmässä *unit of work* tarkoittaa Osheroven mukaan niitä toimenpiteitä, jotka tapahtuvat julkisen metodin kutsutuksi tulemisen ja testin lopputuloksen välissä. Se voi rajoittua yhteen metodikutsuun, luokkaan tai useaan luokkaan, jotka yhdessä pyrkivät toteuttamaan yhden loogisen tarkoituksen, joka on varmistettavissa. [35, s. 4]

Huonosti kirjoitetut yksikkötestit kuluttavat kehittäjältä enemmän aikaa kuin tuovat hyötyä, sillä niiden kirjoittamiseen menee aikaa ja pahimmassa tapauksessa ne hankaloittavat sovelluksen ylläpitoa. On siis tärkeä pystyä määrittelemään, millainen on hyvin kirjoitettu yksikkötesti. Osherove asettaa hyvälle yksikkötestille seuraavat vaatimukset

- suoritus täysin automatisoitavissa ja toistettavissa
- helppo toteuttaa
- käyttökelpoinen pidemmän ajan kuluttuakin
- jokaisen tulee pystyä suorittamaan se helposti
- suorituksen tulee olla nopea
- lopputuloksen tulee olla johdonmukainen (paluuarvo säilyy samana, jos suorituskertojen välissä ei muuteta mitään)
- sillä tulee olla täysi hallinta testattavana olevasta koodista
- sitä tulee voida ajaa eristetysti muusta ohjelmasta (myös muista yksikkötesteistä)
- kun se havaitsee virheen, virheen syy ja sijainti tulee olla helposti selvitettävissä. [35, s. 6-7]

Useat hyvän yksikkötestin tunnusmerkeistä liittyvät siihen, että testi on lukijalleen helposti ymmärrettävissä ja käytettävissä vielä useiden kuukausien päästä sen luontihetkestä. Tämän vuoksi Osherove esittääkin testien nimeämiselle käytäntöä [35, s. 181]. Testin ensimmäinen sana on testattavan metodin nimi. Tämän jälkeen tulevat ehdot, jotka testin ajon aikana toteutuvat ja viimeiseksi kerrotaan, mitä testin lopputuloksen tulisi olla. Myöhemmässä luvussa 6.1.3 on tämän käytöstä esimerkki.

Yksittäisen testin rakenne jakautuu kolmeen vaiheeseen, joiden käytölle on vakiintunut englannin kielinen kirjainlyhenne AAA eli arrange, act, assert. *Arrange* vaiheessa luodaan ja alustetaan testin tarvitsemat oliot niiden alkutilaan. *Act* vaiheessa testattavana olevan olion metodia kutsutaan. *Assert* vaiheessa tarkistetaan, että testille asetetut vaatimukset toteutuivat. [35, s. 27]

3.5.2 Järjestelmätestaus tutkivana testauksena

Järjestelmätestaus on testauksen muoto, jossa testataan täysin toteutettua järjestelmää, jonka toimintaa verrataan sille asetettuihin vaatimuksiin [1, s. 197]. Vaatimusten ja dokumentaation ollessa puutteellisia suositellaan testauksen suorittamista tutkivana testauksena [36, s. 72] [37]. Se soveltuu myös hankalien toiminnallisuuksien testaamiseen [38], joissa yksikkötestien kaltaisten automaattisten testien kirjoittaminen on aikaa vievää ja haasteellista. Sen nähdään olevan tyypillistä testausta,

jossa kirjoitetaan testitapauksia vaatimusten pohjalta, tehokkaampi keino virheiden löytämiseen [39] [40].

Tutkivassa testauksessa testitapaukset eivät ole ennalta määrättyjä. Testaaja tutkii ohjelman suoritusta testauksen avulla, joka ohjaa uusien testien tekemiseen aikaisempien testien pohjalta. Aiemmista testeistä saatu tieto, ohjaa uusien parempien testien tekemiseen. Williams määrittää tutkivan testauksen koostuvan seuraavista vaiheista:

1. selvitä tuotteen tarkoitus
2. selvitä sen toiminnallisuudet
3. selvitä epävakaat alueet, joissa todennäköisimmin esiintyy ongelmia
4. testaa toiminnallisuudet ja kirjaa ylös löydetyt ongelmat
5. kirjaa ylös ohjelman suoritusvaiheet, joilla ongelman voi toistaa. [36, s. 73]

Tutkivan testauksen ongelmaksi nähdään sen toistettavuuden hankaluus, sillä testejä ei ajeta ennalta kirjoitetuista skripteistä [36, s. 73]. Tämä tekee sen käytöstä mahdotonta tai erittäin aikaa vievää esimerkiksi regressiotestauksen kaltaisissa testeissä, koska testien ajoa ei voida automatisoida. Regressiotestauksessa järjestelmää tai ohjelmamoduulia testataan uudelleen, jotta saadaan selville noudattaako järjestelmä edelleen vaatimuksiaan, kun sitä muutetaan [1, s. 170].

3.5.3 Kuormitustestaus

Eräänä kuormitustestauksen muotona voidaan pitää testausta, jonka tarkoituksena on toistaa testitapauksia useita kertoja pyrkimyksenä simuloida ohjelman pitkää suoritusaikaa [41]. Englanniksi tätä nimitetään termillä *Long sequence testing*. Sitä pidetään tehokkaana keinona löytää virheitä, jotka johtuvat ajastukseen liittyvistä ongelmista, moduulien välisestä koordinoinnista tai resurssien liikakäytöstä. Tämän totesivat esimerkiksi McGee et. al. tapaustutkimuksessaan, jossa tutkittiin kuluttajamarkkinoilla tarkoitetun sulautetun ohjelman kehitystä [42].

Virheiden löytäminen perustuu ohjelman pitkään suoritusaikaan. Pitkän suorituksen aikana ohjelman erilaisten ajonaikaisten tilojen määrä kasvaa, jolloin ohjelmassa voi esiintyä häiriöitä, jotka eivät esiintyisi ohjelmassa, jos testien suoritus tapahtuisi yksittäin. [41]

3.6 Virheiden ennustaminen

Virheiden ennustamisessa pyritään arvioimaan nykyisten ja tulevien virheiden määrää, esiintymisen todennäköisyyttä sekä niistä seuraavien häiriöiden seurauksia. Tavoite on sama kuin virheiden poistamisessa, vastaako ohjelma sen määrittelyä. Tekniikoiden käyttö auttaa esimerkiksi selvittämään voidaanko ohjelman tehostetulla testaamisella saavuttaa tutkittavassa ohjelmassa parempaa luotettavuutta. Tekniikat voidaan jakaa kahteen ryhmään: luotettavuutta arvioivat ja luotettavuutta ennustavat tekniikat. [16, s. 11]

Luotettavuutta arvioivissa tekniikoissa pyritään selvittämään ohjelman nykyinen luotettavuustaso. Kuinka todennäköistä on, että ajossa olevassa ohjelmassa tapahtuu häiriö tietyn aikaikkunan sisällä [12, s. 539]. Arvioinneissa hyödynnetään ohjelman elinkaaren aikana kerättyä aineistoa ohjelmassa esiintyneistä virheistä, joita on kerätty systemaattisesti talteen mm. testauksen tai ohjelman normaalin käytön aikana [16, s. 11].

Ohjelman kehityksen aikaisemmassa vaiheessa tai tilanteessa, jossa ohjelmassa esiintyneitä virheitä ei ole systemaattisesti kerätty talteen, voidaan hyödyntää luotettavuutta ennustavia tekniikoita. Niiden avulla pyritään selvittämään ohjelman tulevaisuuden luotettavuustaso. Tekniikoissa hyödynnetään ohjelmistosta saatavaa metriikkaa sekä muuta mitattavaa tietoa sen mukaan mitä on saatavilla. [16, s. 11] Ohjelmistometriikka tarkoittaa funktiota, joka ottaa sisäänsä ohjelmasta saatavaa dataa ja antaa ulostulona yksittäisen numeroarvon, joka voidaan tulkita tarkoittavan, kuinka hyvin ohjelma toteuttaa kyseisestä laatuominaisuutta [1, s. 164]. Useita erilaisia ohjelmiston luotettavuutta parantavia metriikoita löytyy esimerkiksi IEEE:n julkaisemasta standardista [7]. Metriikoiden lisäksi kehitysprosessin alkuvaiheessa tietoa häiriöön johtavista virheistä voidaan saada esimerkiksi kehitysprosessia arvioimalla, kokeneilta kehittäjiltä tai toimiala-asiantuntijoilta [43, s. 23].

3.6.1 Koodikattavuus

Koodikattavuus on ohjelmistometriikka, jota voidaan pitää testauksen kattavuuden mittarina. Sen avulla voidaan selvittää, kuinka iso osa ohjelman lähdekoodista tulee suoritetuksi, kun ohjelmalle kirjoitetut testit ajetaan.

Lähdekoodin koodikattavuus voidaan mitata usealla eri tavalla, riippuen siitä mitä kattavuuden kriteeriä käytetään. Suoraviivaisin kriteeri on lausekattavuus. Siinä seurataan tulevatko kaikki ohjelman rivit suoritetuiksi. Esimerkiksi ohjelmassa 2

saadaan kaikki ohjelman rivit $1 - 7$ suoritettua, kun ohjelma ajetaan taulukon 1 mukaisilla parametrin arvoilla.

Taulukko 1 *Testitapaus, jolla saadaan 100% lausekattavuus ohjelmassa 2.*

fruit.Name	fruit.WeightInGrams
"Banana"	300

Sen avulla ei voida kuitenkaan osoittaa etenikö ohjelman suoritus kaikkien haarautumiskohtien kautta. Tätä varten on olemassa haarakattavuus, jossa lausekattavuuden lisäksi huomioidaan ohjelman ehtorakenteet. Jokainen ohjelmassa oleva ehto saa haarakattavuuden maksimoimisessa molemmat arvonsa. Esimerkkiohjelman 2 tapauksessa 100% haarakattavuus saavutetaan ajamalla ohjelma taulukon 2 mukaisilla parametrin arvoilla. Tällöin ohjelma saa suorituspolun $1 - 7$ lisäksi suorituspolun, jossa ajetaan ohjelman rivit 1, 2, 6 ja 7.

Taulukko 2 *Testitapaukset, joilla saadaan 100% haarakattavuus ohjelmassa 2.*

fruit.Name	fruit.WeightInGrams
"Banana"	300
"Orange"	300

Koodikattavuuden määrittäminen voi tapahtua tässä esitettyjen lisäksi muullakin tavalla, mutta niiden ääripäänä voidaan pitää polkukattavuutta. Siinä jokainen mahdollinen ohjelman suorituspolku tulee suoritetuksi. Tämä ei kuitenkaan ole tavallisesti mahdollista, sillä ohjelman suorituspolkujen määrä kasvaa ohjelman koon kasvaessa eksponentiaalisesti. [44, s. 121-123]

```
1 Console.WriteLine("Hello fruit gatherers.");
2 if (fruit.Name == "Banana" && fruit.WeightInGrams > 200)
3 {
4     Console.WriteLine("We got a huge banana!");
5 }
6 Console.WriteLine("Our fruit name is " + fruit.Name + ".");
7 Console.WriteLine("And it weights " + fruit.WeightInGrams + " grams.");
```

Ohjelma 2 *Koodikattavuuden määrittäminen esimerkkikoodista.*

Koodikattavuutta pidetään tutkijoiden keskuudessa hyvänä mittarina arvioida testauksen kattavuutta eli kuinka paljon ohjelmakoodista on vielä testaamatta. 100% koodikattavuus ei takaa, että ohjelmasta ei voisi löytyä virheitä, mutta tutkijoiden

mukaan korkea koodikattavuus kasvattaa ohjelmiston luotettavuutta ja laskee siinä esiintyvien virheiden määrää [45][46][47]. Ilmailualan kaupallisten lentokonejärjestelmä ohjelmistojen hyväksyminen perustuu *DO-178C* -dokumenttiin, jossa eräs lähdekoodilta vaadituista ominaisuuksista on 100% lause- ja haarakattavuus [48].

3.7 Toimintamallin muuttaminen

3.7.1 Ketterä ohjelmistokehitys

Perinteinen tapa ohjelmistokehitykselle on sisältänyt yksityiskohtaisen suunnitelman luonnin, jossa on pyritty määrittelemään viimeistä yksityiskohtaa myöten rakennettava tuote. Tästä on käytetty nimeä vesiputousmalli (*waterfall model*) ja se on soveltunut hyvin ohjelmien tekoon, joiden ohjelmistovaatimukset on voitu määritellä tarkasti jo projektin alussa. Eräs keskeinen syy mallin kyseenalaistamiselle oli seuraavan lainen. Päätöksenteko on tehokkaampaa juuri ennen kuin päätöksen tekoa ei enää voi etenemisen takia lykätä, sillä tässä vaiheessa ymmärrys kehitettävästä tuotteesta on kasvanut ja päätöksenteko on kasvaneen ymmärryksen myötä valistuneempaa.

Vuonna 2001 julkaistun ketterän ohjelmistokehityksen julistuksen *Manifesto for Agile Software Development* [49] kiihdyttämänä ohjelmistojen kehitysprosessit ovat siirtyneet etenevissä määrin kohti ketteriä ohjelmistokehitysmenetelmiä. Manifestin mukaan ketterät ohjelmistokehitysprosessit korostavat yksilöitä ja kanssakäymistä, toimivaa ohjelmistoa, asiakasyhteistyötä ja muutoksiin vastaamista. Ketteriin menetelmiin siirtyminen on tapahtunut mm. siitä syystä, että asiakkaat eivät useinkaan tiedä tarkkoja ohjelmistovaatimuksia ennen kuin näkevät toimivan ohjelman, jonka jälkeen he haluavat muuttaa vaatimuksia, joka johtaa uudelleen suunnitteluun, uudelleen kehitykseen ja uudelleen testaamiseen nostaen ohjelmistoprojektin kuluja. Ketterän ohjelmistokehityksen seurauksena on kehitetty useita toimintamalleja, jotka pyrkivät tehostamaan ohjelmien tekemistä ja takaamaan ohjelmien korkean luotettavuuden erityisesti nopean palautteen avulla, joka saavutetaan automatisoimalla prosesseja.

Ketterien ohjelmistokehitysmenetelmien käyttö erityisen korkeaan luotettavuuteen tähtäävien *mission-critical* ohjelmien toteuttamisessa on esitetty kritiikkiä, ja se on jatkuvan tutkimuksen alla [50], [51]. Toisaalta Humble et. al. mukaan ketterien ohjelmistokehityksen periaatteiden avulla on toteutettu avaruusaluksen ohjelmistoakin [13, s. 425], joten tilanne tämän osalta voi tulevaisuudessa muuttua jolloin tulemme näkemään nykyistä enemmän mission-critical ohjelmien kehitystä ketterän ohjelmistokehityksen keinoin. Joka tapauksessa ketterään ohjelmistokehitykseen pohjautuvia

menetelmiä hyödynnetään laajasti mission-critical järjestelmiä kevyemmässä ohjelmistokehityksessä, jossa tavoitteena on tuottaa käyttövarma ohjelma.

Seuraavat esiteltävät toimintatavat eivät ole yksittäisiä keinoja rakentaa käyttövarmaa ohjelmaa vaan vaativat suurimmillaan koko organisaatikulttuuria koskevan muutoksen. Ne pyrkivät mahdollistamaan käyttövarman ohjelman rakentamisen, joka on lähtöisin yrityksen toimintatapojen muutoksesta, jossa rikotaan perinteistä tapaa toimia.

3.7.2 Jatkuva integrointi

Perinteisessä ohjelmistokehityksessä ohjelma saattaa olla pitkiä aikoja tilassa, jossa se ei toimi tai ole käytettävissä. Syy on tyypillisesti se, että ohjelman ajaminen kokonaisuutena ennen kuin se on valmis on hankalaa, joka johtaa siihen, että ohjelmaa ei kehityksen aikana ajeta tuotantoympäristössä tai sitä vastaavassa ympäristössä. Tässä tilanteessa on tyypillistä, että ohjelman muodostamat moduulit integroidaan toisiinsa vasta projektin lopussa. Lopussa tapahtuva integrointivaihe voi aikaansaada virheitä, joiden olemassa olosta ei ennen tätä vaihetta tiedetty. Pahimmassa tapauksessa vaiheen jälkeen saattaa selvitä, että rakennettu ohjelma ei ollut tarkoitukseen sopiva. [13]

Jatkuva integrointi pyrkii ratkaisemaan edellä kuvatun kaltaisia ongelmia. Sen tarkoituksena on varmistua siitä, että ohjelma toimii jokaisen siihen tehdyn muutoksen jälkeen. Muutoksella tarkoitetaan versiohallinnan päähaaraan tuotuja muutoksia, joiden määrä voi kehitysryhmän koosta riippuen olla useita päivässä. Jos muutos rikkoo ohjelman, tästä saadaan välitön palaute, jolloin virheen korjaaminen voi tapahtua mahdollisimman pian sen esittelyn jälkeen, joka vähentää sen korjaamiseen kuluvaa aikaa [52] [34]. Kehitysryhmän jäsenten kanssa tuleekin sopia, että muutokset, jotka rikkovat ohjelman tulee korjata ensimmäisenä, jotta varmistuttaisiin siitä, että ohjelma on aina julkaistavassa tilassa. [13]

Jatkuva integrointi tarvitsee toimiakseen versiohallinta työkalun, joka mahdollistaa kaikkien projektin vaatimien tiedostojen säilyttämisen. Ohjelman kääntämisen tulee olla automatisoitavissa, jotta sen voi suorittaa komentoriviltä. Jotta jatkuva integrointi mahdollistaisi ilmoitukset, kun ohjelmaan on esitelty virhe, se vaatii toimiakseen kattavat automaattiset testit, joiden avulla ohjelman luotettavuus paranee virheiden vähenemisen myötä [52]. Testien puuttuessakin tiedetään, että ohjelmassa ei esiinny integrointivaiheessa olevia virheitä, koska ohjelman koostaminen onnistuu.

Testeihin kuuluvat määrällisesti suurimpana ja tärkeimpänä jo aiemmin luvussa

3.5.1 kuvatut yksikkötestit [13, s. 71], komponenttitestit (*component tests*) ja hyväksymistestit (*acceptance tests*). Komponenttitestit testaavat yhden ohjelmakomponentin toimintaa [24, s. 82] ja niiden ajoaika on tyypillisesti yksikkötestejä huomattavasti suurempi, sillä ne saattavat käyttää ohjelman ulkopuolisia palveluita kuten tietokantaa. Hyväksymistestit todentavat, että asiakkaan ohjelmalle asettamat sopimuksen mukaiset vaatimukset täyttyvät [24, s.5]. Näiden ajo kestää isoilla järjestelmillä jopa yli vuorokauden [13, s. 60].

3.7.3 Jatkuva toimitus ja -julkaisu

Jatkuva toimitus (*continuous delivery*) on laajennus jatkuvalle integroinnille ja sen nimi tulee suoraan ketterän ohjelmistokehityksen julistuksen ensimmäisestä periaatteesta [53], jonka mukaan ohjelmistokehityksen tärkein tehtävä on tyydyttää asiakas julkaisemalla nopeasti ja jatkuvasti arvokasta ohjelmaa. Sen avulla pyritään mahdollistamaan, että ohjelma voidaan julkaista usein ja nopeasti, jonka seurauksena ohjelmakehitysprosessi saa arvokasta asiakaspalautetta usein tapahtuvien julkaisujen jälkeen. Jatkuvaan integrointiin verrattuna merkittävin ero on juuri tässä. Nopea palaute tulee kehitysryhmälle automaattisten testien lisäksi myös ohjelman käyttäjältä.

Asiakaspalautteen avulla kehitysryhmä voi reagoida muuttuviin vaatimuksiin tehokkaasti. Nopea julkaisu mahdollistaa *fail fast* tyyppisen ajattelutavan, jossa esimerkiksi aiemmin haastavasti korjattavien ohjelmistovaatimuksien sisältämät virheet päästään korjaamaan nopeasti niiden syntymisen jälkeen [52]. Oletuksena tosin on, että asiakas huomaa virheen.

Riungu-Kalliosaari et. al. tutkimuksessa [52] haastateltu vanhempi ohjelmistokehittäjä ilmaisi kenties tärkeimpänä jatkuvan toimituksen hyötynä työskentelytapojen paranemisen ja työssäjaksamisen. Tällä hän viittasi siihen, että ohjelmistojulkaisut tapahtuvat usein, mutta ovat pieniä, jolloin isojen julkaisujen sisältämien virheiden korjaukseen kuluva aika pienenee yhdessä ohjelmistokehittäjän stressitason kanssa.

Jatkuva toimitus on mahdollista vain, jos ohjelmalle kirjoitetut automaattitestit ovat riittävän kattavat. Tätä pidetään suurimpana esteenä toimintamallin käytölle jo alkaneissa projekteissa, joissa testejä ei ole tai niiden kirjoittaminen on lukuisten ohjelmariippuvuuksien takia hankalaa [54]. Jotta yksikkötestien määrä saadaan riittävän suureksi Humble et. al. pitää ainoana vaihtoehtona testivetoisen kehityksen käyttöä (*test driven development TDD*). Sen pääperiaate on, että uuden ominaisuuden kehittäminen tai virheen korjaaminen aloitetaan ohjelmistovaatimuksen mukaisen yksikkötestin kirjoittamisella [13, s. 71].

Jatkuva julkaisu (*continuous deployment*) on esimerkki siitä, mitä tarkoittaa kun jatkuva toimitus toteutetaan äärimmilleen vietynä. Tämän seurauksena jokainen versiohallintaan tuotu muutos viedään tuotantoon asti, edellyttäen että ne läpäisevät automaattitestit. Yritykset kuten Amazon julkaisee uuden version ohjelmastaan *11,6* sekunnin välein [55]. Se ei olisi mahdollista ilman hyvin toteutettua automatisoitua julkaisuputkea (*deployment pipeline*), joka mahdollistaa sen, että lähdekoodit ja sitä pyörittävä infrastruktuuri ovat julkaistavassa kunnossa.

3.7.4 DevOps

DevOps koostuu sanoista ohjelmistokehitys *software development* ja operointi *software operation.* Termien yhdistämisellä viitataan siihen, että ohjelmistokehittäjät ja järjestelmän ylläpitäjät, jotka vastaavat ohjelman suoritusympäristöstä ja sen monitoroinnista, toimivat yhdessä. [56, s. 17] Tämä on sikäli merkittävää, että osapuolten tavoitteet ovat ristiriitaisia. Kehittäjät pyrkivät tekemään ohjelmaan muutoksia, jotka voivat rikkoa aiemmin toimineen ohjelman, kun taas ylläpitäjät pyrkivät varmistumaan siitä, että ohjelma toimii ja pysyy ajossa [56, s. 6].

Käytännön toimenpiteitä yhteistyön lisäämiseksi ovat Bass et. al mukaan se, että ylläpitäjien ongelmista tulee korkeimman prioriteetin ohjelmistovaatimuksia. Kehittäjien tulisi puolestaan osallistua tyypillisesti ylläpitäjille kuuluvien tehtävien tekemiseen kuten ohjelman käytön aikaisten virheiden selvittämiseen, jotta virheiden löytämisen ja korjaamisen välinen aika lyhenisi [57]. [56, s. 17] Yhteistyön lisäämisen nähdään parantavan osapuolien välistä kommunikointia, joka mahdollistaa tiedon ja kokemuksen paremman jakamisen osastojen välillä [52].

Informatiivisemman määritelmän DevOps sanalle antaa Bass et. al., jonka mukaan:

> DevOps is a set of practices intended to reduce the time between committing a change to a system and the change being placed into normal production, while ensuring high quality [57].

Määritelmästä nähdään, että DevOps sisältää ratkaisun samoihin ongelmiin mitä jatkuvalla integroinnilla ja -julkaisulle pyritään ratkaisemaan. Nämä toimintatavat yhdessä toimivatkin DevOpsin mahdollistajana, joten niistä saatava hyöty on osa DevOpsista saatavaa hyötyä. On kuitenkin hyvä huomata, että DevOpsin käyttö ei rajoitu vain luotettavien ohjelmien nopeaan julkaisuun, vaan ohjelma pyritään pitämään laadukkaana koko sen eliniän ajan. Käytännössä tämä tarkoittaa ohjelman

ajonaikaisten monitorointitekniikoiden hyödyntämistä yhteistyössä ylläpidon kanssa, jotta ohjelman tilasta saadaan tietoa myös sen käytön aikana. [57]

Puppet Labs järjesti vuonna 2015 ohjelmistotalojen kesken kyselyn, johon vastasi 4976 henkilöä. Sen mukaan henkilöt, jotka työskentelivät DevOpsia käyttävässä kehitysryhmässä saavuttivat 60 % vähemmän häiriöitä ohjelmajulkaisuissa, pystyivät julkaisemaan 30 kertaa useammin ja kykenivät palauttamaan kaatuneen ohjelman 168 kertaa nopeammin kuin henkilöt, jotka eivät työskennelleet DevOpsia käyttävässä kehitysryhmässä. [58]

4. TOTEUTUSYMPÄRISTÖ

4.1 PLC:n toimintaperiaate

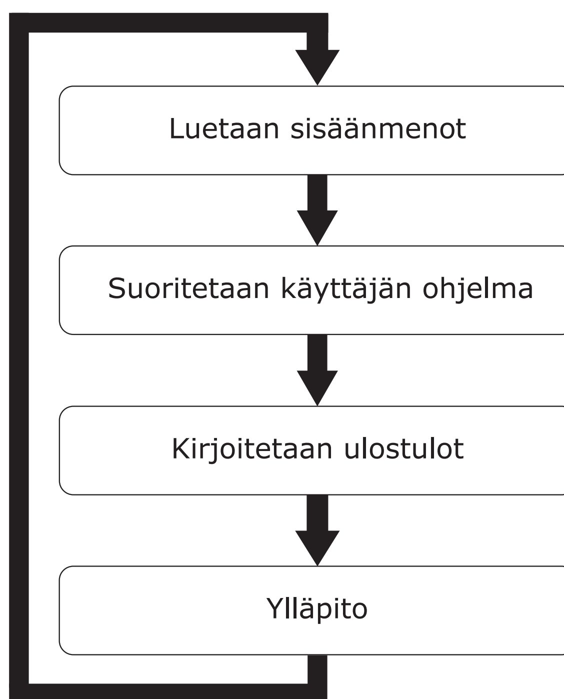
4.1.1 Skannaus sykli

Työn soveltavassa osuudessa käytetään PLC:tä. PLC tulee sanoista *Programmable Logic Controller* eli ohjelmoitava logiikka. Se on tietokone, joka on suunniteltu käytettäväksi teollisuusolosuhteissa laitteiden ohjaukseen. Tavallisesta tietokoneesta sen erottavat teollisuusympäristöön sopiva kotelointi, tuki lähinnä IEC 61131-3 mukaisille ohjelmointikielille ja monipuolisemmat I/O liitynnät.

Sisääntulot sekä ulostulot voivat olla esimerkiksi analogisia signaaleja, tcp/ip liikennettä tai kenttäväylältä saapuvaa liikennettä. Tilanne on yksinkertaisimmillaan, kun PLC:ssä on kiinni digitaalinen sisääntulo- ja ulostulokortti. PLC:n toiminta voidaan tällöin jakaa neljään tilaan, jotka on esitetty kuvassa 3. Kun PLC on kytketty ajotilaan (*run mode*), sen toiminta alkaa lukemalla ulkoiset sisääntulot. PLC tarkastaa jokaisen sisääntuloportin jännitetiedon. Jos portissa kulkee jännite, kirjoittaa PLC muistiinsa kyseisen portin kohdalle numeron 1 eli sähköpiiri on suljettu. Jos portissa ei ole jännitettä, kirjoittaa PLC kyseisen portin kohdalle numeron 0. [59, s. 80]

Seuraavaksi suoritetaan käyttäjän ohjelma. Ohjelma suoritetaan niillä sisääntulon arvoilla, jotka edellisessä vaiheessa luettiin. Käyttäjän kirjoittama ohjelmalogiikka on vastuussa siitä, mitä arvoja PLC:n ulostulot saavat. Kuten sisääntulojen tapauksessa myös ulostuloporttien arvot kirjoitetaan muistiin. Numero 1 vastaa suljettua sähköpiiriä ja 0 auki olevaa. [59, s. 80]

Kolmannessa vaiheessa ulostuloportteihin kytketään jännite päälle tai pois päältä sen mukaan, mitkä arvot ulostulot saivat edellisessä vaiheessa, kun käyttäjän ohjelma suoritettiin läpi. Vaikka fyysinen sisääntulo ehtisi muuttumaan toisen tai kolmannen vaiheen aikana, ei sillä ole vaikutusta ulostulojen arvoihin ennen seuraavan syklin alkamista. Tämä aiheuttaa ongelmia erityisesti silloin kun käyttäjän ohjelma on riippuvainen sisääntulosta, jonka tila vaihtuu nopeammin kuin PLC ehtii suorittamaan skannaussyklinsä läpi. [59, s. 80]



Kuva 3 *PLC:n yhden syklin aikaiset tehtävät.*

Viimeisessä vaiheessa PLC suorittaa ylläpidollisia toimenpiteitä, joihin kuuluvat muun muassa muistin, nopeuden ja toimintatilan tarkastelut sekä kommunikointiin liittyvien pyyntöjen palvelu. Kun viimeinen vaihe on suoritettu, alkaa suoritus jälleen alusta niin kauan, kunnes PLC kytketään pois ajotilasta. [59, s. 79]

Kaikkien vaiheiden kertaalleen suorittamista nimitetään *ohjelman skannaussykliksi* ja sen suorittamiseen kokonaisuudessaan kulunutta aikaa *skannaussyklin ajaksi.* Kulunut aika vaihtelee tyypillisesti $1 - 20ms$ välillä ja siihen vaikuttaa

- suorittimen nopeus
- käyttäjän ohjelman pituus
- suoritettavien käskyjen tyyppi sekä
- tulojen ja menojen todelliset arvot.

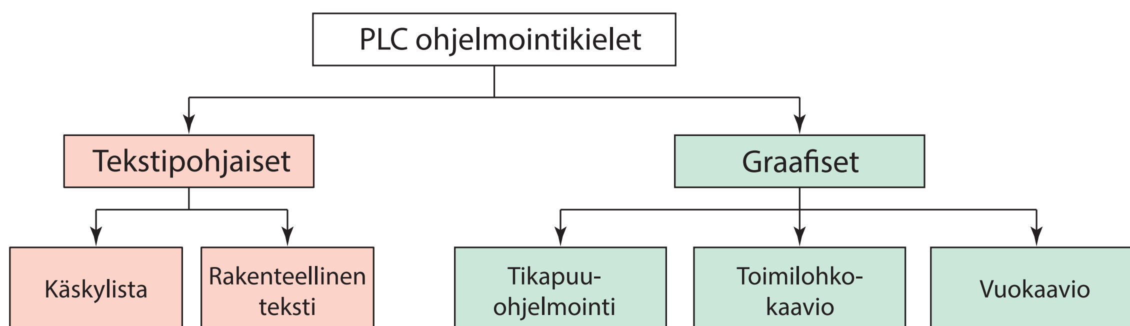
Todellinen skannaussyklin aika tallennetaan ylläpito vaiheessa laitteen muistiin ja sitä voidaan tarkastella ohjelmallisesti. [59, s. 79]

4.1.2 Ohjelmointikielet

PLC:n ohjelmoinnissa voidaan käyttää useaa eri ohjelmointikieltä, joista IEC 61131-3 standardi kuvaa 5 erilaista. Nämä ovat:

- IL (Instruction list) eli käskylista
- ST (Structured text) eli rakenteellinen teksti
- LD (Ladder logic) eli tikapuuohjelmointi
- FBD (Function block diagram) eli toimilohkokaavio
- SFC (Sequental function chart) eli vuokaavio.

Kielet on mahdollista jakaa tekstipohjaisiin- ja graafisiin kieliin. Kuvassa 4 esitetään tämä jako.



Kuva 4 *IEC 61131-3 standardin mukaiset PLC ohjelmointikielet. Suomennettu lähteestä [59, s. 81].*

Tikapuuohjelmointi on kielistä eniten käytetty [59, s. 81]. Sitä käytetään valmistajan toimesta muun muassa Motomanin robotin I/O:iden hallinnassa. Kielen rakenne pyrkii mukailemaan relelogiikalla tehtyä järjestelmää.

Toimilohkokaavion ajatuksena on paketoida ohjelma lohkoiksi, jotka yhdessä muodostavat tietyn toiminnallisuuden. Toisin sanoen ne pyrkivät abstrahoimaan olio-ohjelmoinnin tavoin monimutkaisia kokonaisuuksia lohkoiksi, jolloin lohkon sisäisen toiminnallisuuden ymmärtäminen ei ole välttämätöntä lohkon käyttämiseksi. Lohkot pitävät tyypillisesti sisällään ehtolauserakenteita, ajastimia ja laskureita. [59, s. 82]

Vuokaavio pyrkii mukailemaan prosessin työnkulkua. Se on suunniteltu käytettäväksi monimutkaisempien prosessien ohjaukseen. Kielen ajatuksena on pilkkoa ohjelma

tiloihin. Jokaisella tilalla on siirtymäehto, jonka täytyttyä tila aktivoituu. Tiloihin voidaan sitoa toimintoja, jotka suoritetaan, kun tila on aktiivinen.

Rakenteellista tekstiä verrataan usein korkeamman tason tekstipohjaisiin ohjelmointikieliin kuten BASIC, C tai PASCAL [59, s. 81]. Sen avulla voidaan toteuttaa asioita, jotka olisivat hankalia toteuttaa graafisilla kielillä [59, s. 83].

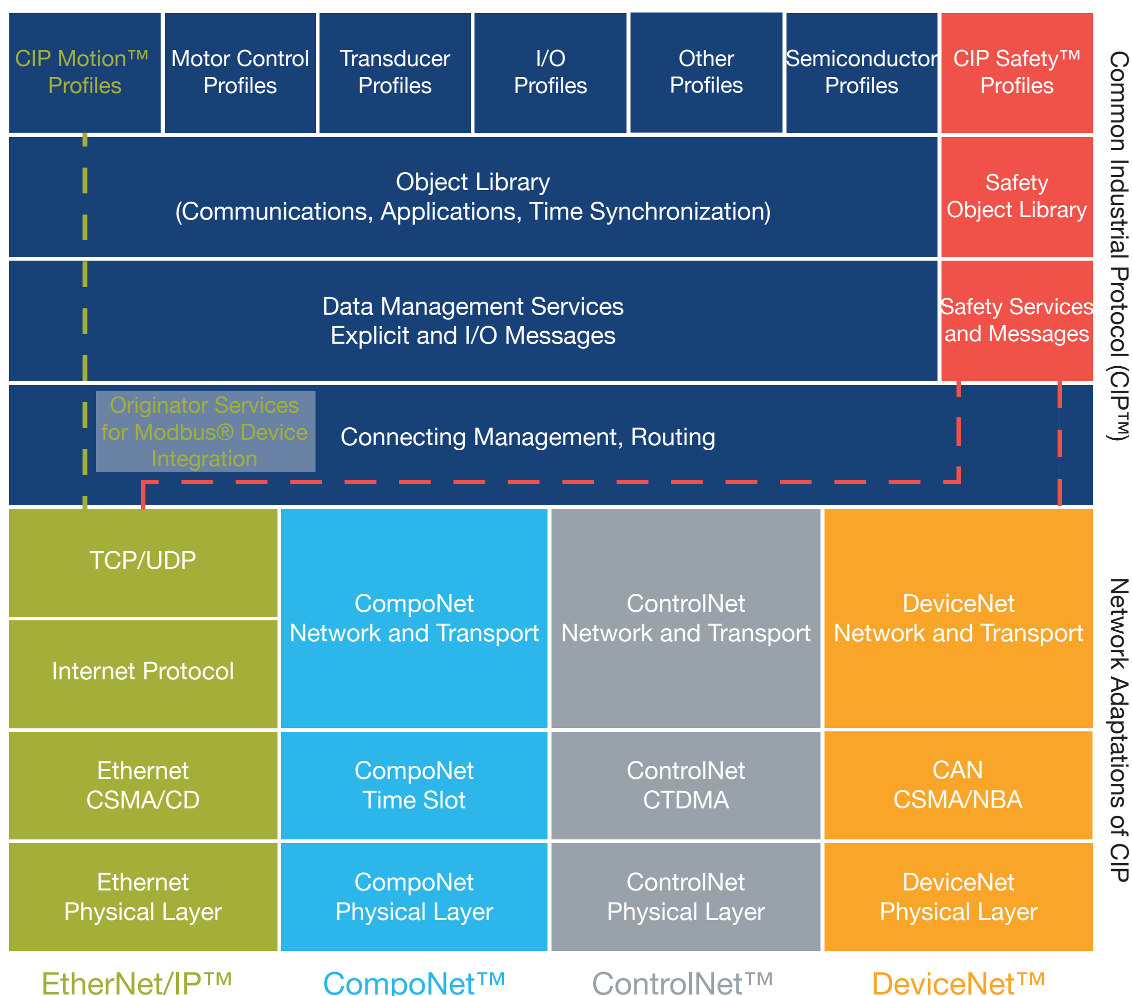
Käskylista on määritellyistä kielistä matalimmalla tasolla [59, s. 81]. Tekstipohjainen kieli käyttää muistisääntöjä (*mnemonic*) käskyrakenteissa. Boltonin mukaan sen voidaan ajatella olevan tikapuuohjelmoinnin tekstipohjainen versio [60, s. 119].

4.2 Tiedonsiirron tekniikat

4.2.1 DeviceNet

DeviceNet on Yhdysvaltalaisen Allen Bradley tuotemerkin alla kehitetty kenttäväyläteknologia. Sitä hallinnoi ODVA (*Open DeviceNet Vendor Association*) järjestö, jonka muodostavat maailman johtavat automaatioyhtiöt. DeviceNet:ille tyypillisiä piirteitä ovat samanaikainen tuki 64 eri laitteelle, joista jokaisen voi irrottaa virran ollessa päällä, ilman että tästä koituu haittaa muulle järjestelmälle. Yhdessä kaapelissa kulkee sekä virta että data. Tuetut nopeudet ovat 125, 250 ja 500 bps ja valintaan niiden välillä perustuu signaalin siirtomatkaan. Verkon laitteet ovat liitetty toisiinsa väylätopologialla. [61, s. 155-156]

DeviceNet käyttää hyväkseen autoteollisuudesta lähtöisin olevaa CAN (*Controller Area Network*) väylää datan siirtämisessä. Lisäksi se hyödyntää usean muun kenttäväylätekniikan kuten EtherNet/IP, CompoNet ja ControlNet kanssa CIP (*Common Industrial Protocol*) formaattiin paketoituja viestejä. Kuvassa 5 nähdään oranssilla värillä kuinka DeviceNet hyödyntää CAN -protokollan lisäksi CIP -protokollaa viestiessään muiden protokollien kanssa.



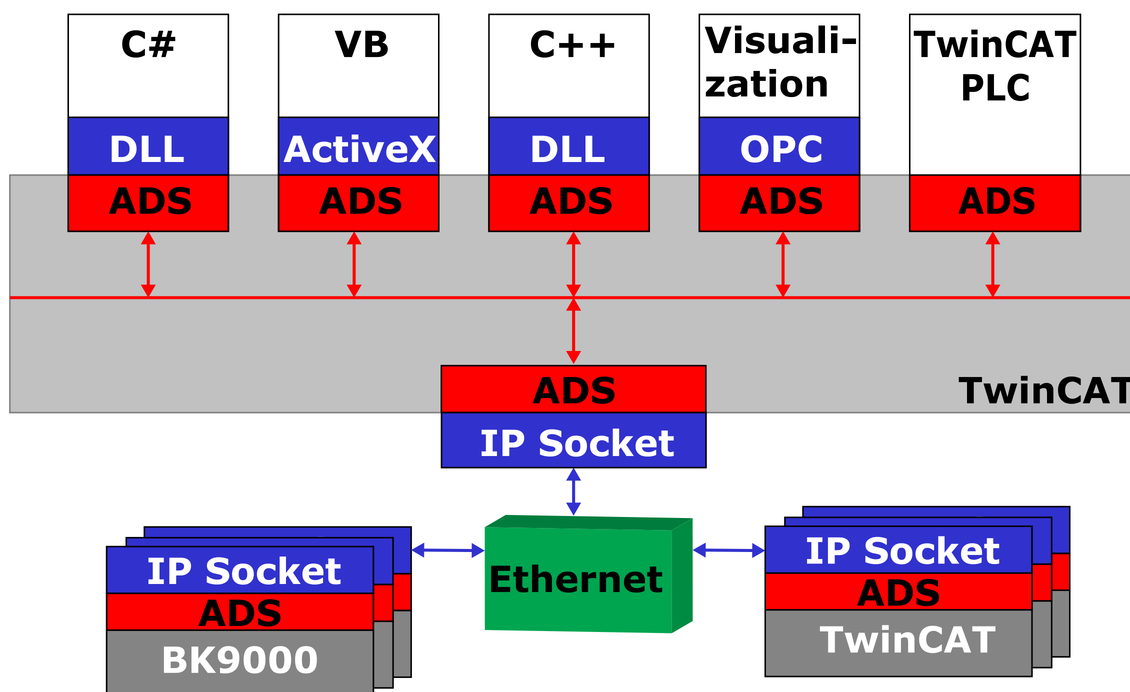
Kuva 5 *DeviceNet osana CIP OSI-mallia [62].*

CIP:n avulla yhdestä verkosta kuten DeviceNet lähetetty viesti voidaan vastaanottaa toisessa verkossa kuten EtherNet/IP. Tämä mahdollistaa eri verkkojen helpon integroinnin toisiinsa. [62] CIP:n tuomista muista hyödyistä voi lukea enemmän lähteestä [63].

4.2.2 TwinCAT ADS Communication Library

Beckhoffin kehittämä kirjasto on tarkoitettu saman nimisen TwinCAT ADS protokollan käyttöön. Kirjaston nimessä esiintyvä ensimmäinen sana TwinCAT koostuu sanoista *Total Windows Control and Automation Technology*. Kyseessä on ohjelmisto, joka jakaantuu kahteen kokonaisuuteen: *ajoympäristöön* ja *kehitysympäristöön* [64]. Ajoympäristön tarkoituksena on muuntaa Windows -pohjainen PC reaaliaikaiseksi PLC:ksi. Kuvassa 6 on harmaalla taustalla esitetty, kuinka eri ohjelmointikielillä (C#, VB, C++) kirjoitetut ohjelmat, OPC alustalla ajettava visualisointi sekä

PLC hyödyntävät TwinCAT ajoympäristöä välittäessään viestejä toisilleen. Kehitysympäristö mahdollistaa ajoympäristössä ajettavien ohjelmien kehittämisen IEC 61131-3 mukaisten ohjelmointikielien avulla. [65]



Kuva 6 *ADS laitteiden väliset tiedonsiirtokanavat. Muokattu lähteestä [66].*

ADS tulee sanoista *Automation Device Specification* [64]. TwinCAT ADS protokolla on kehitetty mahdollistamaan TwinCAT ekosysteemin ympärille rakennettujen laitteiden ja ohjelmien välisen tiedonsiirron. Protokolla pohjautuu OSI-mallin tasolla 3 olevaan *TCP/IP* pakettiliikenteeseen ja liikennöi kyseisen protokollan päällä [67]. Tämä on esitetty kuvan 6 keskiosassa harmaalla alueella. Siniset nuolet kuvan alaosassa kuvaavat TCP/IP kerroksessa tapahtuvaa liikennöintiä. ADS:n tarkoituksena on määritellä laite- ja kenttäväylä riippumaton liityntätapa muihin sitä käyttäviin laitteisiin. Sen reitittimenä toimivat kaikki TwinCAT ohjelmistoa pyörittävät tietokoneet sekä Beckhoffin BCxxxx sarjaan kuuluvat PLC:t [68]. [69]

Beckhoff on julkaissut yksityiskohtaisen teknisen dokumentaation protokollan toteutuksesta, jossa kuvataan tarkasti viestiliikenteen rakenne [67]. Dokumentaation on tarkoitus mahdollistaa protokollaa hyödyntävän asiakasohjelman toteutus Windowsin lisäksi myös muilla käyttöjärjestelmillä. Työn kirjoitushetkellä avoimen lähdekoodin toteutus löytyi ainakin macOS ja Linux käyttöjärjestelmille [70].

ADS -yhteyden hyödyntäminen tapahtuu kahdella tavalla. Asiakasohjelma voi suoraan lukea tai kirjoittaa haluamalleen toiselle ADS -laitteelle. ADS -laitteeksi ni-

mitetään laitetta, joka toteuttaa ADS -rajapinnan vaatiman toiminnallisuuden [67]. Lukeminen ja kirjoittaminen voivat kohdistua kerralla suurempaan muistialueeseen, jolloin suurenkin rakenteellisen tietosisällön käsittely tapahtuu kerralla. Toinen tapa hyödyntää ADS -yhteyttä on, että asiakasohjelma rekisteröityy kuuntelemaan toiselta ADS -laitteelta saapuvia ilmoituksia. Ilmoitukset voidaan asettaa saapuviksi silloin kun laitteen sisältämä data muuttuu tai tietyin väliajoin, mutta viive on aina vähintään 1 ms. [71]

4.2.3 FTP

FTP, joka tulee sanoista *File Transfer Protocol*, on internettiin liitettyjen koneiden väliseen tiedostonsiirtoon kehitetty sovellustason protokolla. Vaikka sen ensimmäinen versio *RFC 114* julkaistiin jo vuonna 1971, ei sen käyttö ole loppunut tänäkään päivänä. Internet liikenteeltään suurin [72] ohjelmointiin keskittyvä internet foorumi *Stack Overflow* listaa ftp aihetunnisteen kysymyksiä päivittäin. Viimeisin protokollan parannusehdotus *RFC 7151* [73] on julkaistu 2014. Sen tarkoituksena on lisätä FTP:hen tuki käsitellä yhteyksiä, jotka saapuvat samaan IP -osoitteeseen eri verkkotunnuksen (*domain*) takaa.

FTP:n tärkeimpiä ominaisuuksia on tarjota asiakasohjelmalle seuraavat palvelut:

1. tiedostojen lataus palvelimelta
2. tiedostojen siirto palvelimelle
3. kokonaisten tiedostohierarkioiden synkronointi asiakasohjelman ja palvelimen välillä sekä
4. interaktiivinen tiedostojen hallinta, jonka käyttökokemus muistuttaa unixin päätettä. [74, s. 318]

Koska FTP:n kehitys alkoi ennen kuin tietoturvalla oli merkitystä, löytyy sen kaikille tärkeimmille ominaisuuksille tänä päivänä myös muita vaihtoehtoja. Ominaisuuksien 1 ja 2 korvaavana tekniikkana käytetään HTTP -protokollaa, joka suojataan *TLS* -tekniikan avulla, jolloin protokollasta käytetään nimeä *HTTPS*. 3:mas ominaisuus toteutuu tehokkaammin esimerkiksi käyttämällä *rsync* tai *rdist* -työkaluja, jotka siirtävät ainoastaan muuttuneen osuuden tiedostohierarkiaa laitteelta toiselle, jolloin rajattua verkkoresurssia ei kuormiteta turhaan. Koska FTP yhteydet toimivat suojaamattomina niin sisällön kuin käyttäjätunnuksen ja salasanan siirtämisessä, suositellaan ominaisuuden 4 toteutukseen esimerkiksi suojatun *SFTP* -yhteyden

käyttöä. Se ei nimestään huolimatta liity FTP -protokollaan vaan hyödyntää toteutuksessaan *SSH* -yhteyttä [74, s. 312]. [74, s. 318]

4.2.4 High Speed Ethernet Client (HSE)

High Speed Ethernet Client (HSE) -tekniikan käyttö on mahdollista Yaskawan DX100-robottiohjaimesta eteenpäin valmistetuissa Motoman robottiohjaimissa. Kyseessä on valmistajan kehittämä protokolla, jonka avulla robottiohjaimia voidaan hallita, varmuuskopioida ja toteuttaa kokonaisvaltaista kommunikointia niiden ja kolmannen osapuolen ohjelmien välillä. [75]

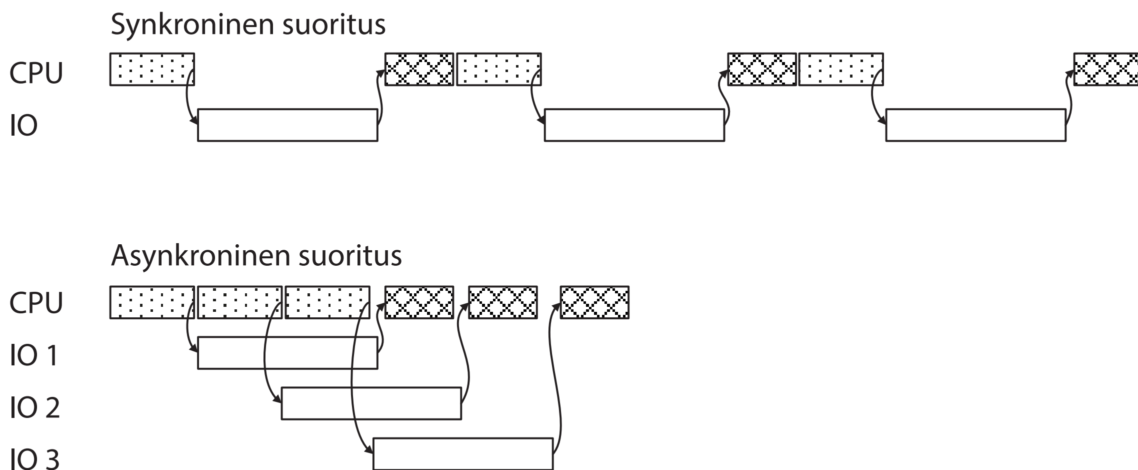
Protokollaa on tarkoitus käyttää valmistajan toteuttamalla High Speed Ethernet Client -kirjastolla, jonka käyttö vaatii lisenssin ostamista. Kirjasto on toteutettu *.NET framework 2.0* -ohjelmistokehyksellä. [75]

HSE toimii TCP/IP kerroksen päällä, tarkemmin sanottuna kyseessä on Yaskawan kehittämä OSI -mallin sovelluskerroksen protokolla [76]. Se tarjoaa mahdollisuuden käsitellä PLC:lle tyypillisten binääri -tyyppisten arvojen lisäksi rakenteellista tietoa kuten robotin paikkatietoa. [75]

4.3 Asynkroninen ohjelmointi

Blewett et. al. määrittelee asynkronisen ohjelmoinnin seuraavasti “Kun kirjoitetaan ohjelmistoa, jonka tarkoituksena on tehdä enemmän kuin yhtä asiaa kerrallaan.” [77, 1 s.]. Asynkronisella ohjelmoinnilla tavoitellaan ohjelman parempaa responsiivisuutta. Sen sijaan, että ohjelma pysähtyisi odottamaan aikaa vievän laskutoimituksen vastausta, se jatkaa suoritusta ja säilyttää ominaisuutensa reagoida käyttäjän syötteisiin. [78, 3 s.]

Kuvassa 7 esitetään synkronisen ja asynkronisen suorituksen eroavaisuus, kun suoritin käsittelee I/O:ta. Kuvassa voidaan olettaa olevan käyttöliittymäsäie, joka suorittaa käyttäjän toimesta verkon yli tapahtuvia pyyntöjä.



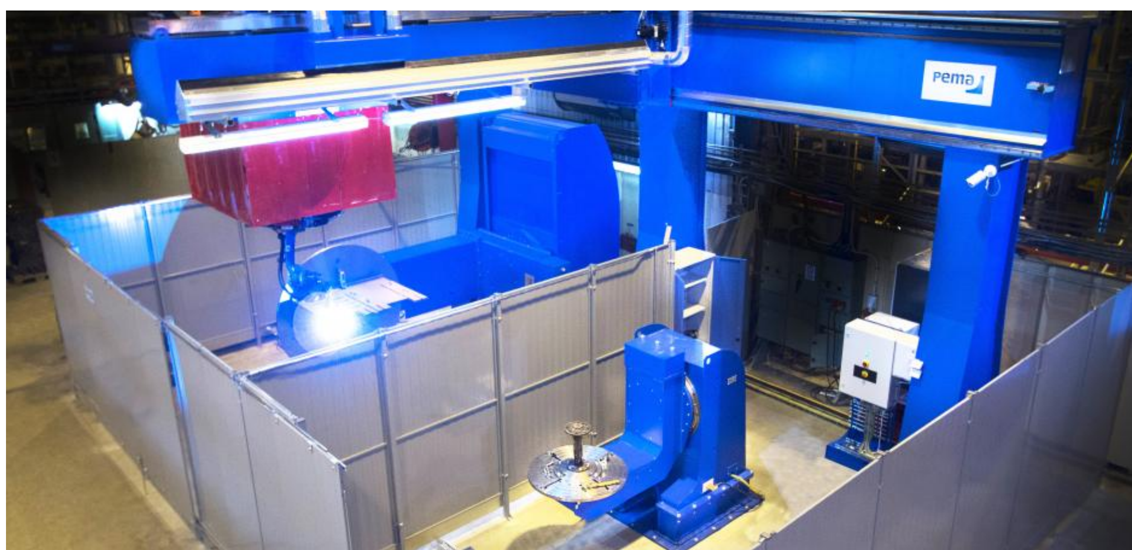
Kuva 7 *Synkronisen- ja asynkronisen suorituksen vertailu. Muokattu lähteestä [79].*

Synkronisessa suorituksessa suoritin jää odottamaan verkkokortin vastausta. Tänä aikana suoritin ei suorita muita ohjelman ajoon liittyviä toimintoja, jolloin käyttöliittymäsäie ei reagoi käyttäjän syötteisiin ja käyttöliittymä vaikuttaa pysähtyneen. Asynkronisessa suorituksessa ohjelma käynnistää verkkopyyntöjä, mutta ei jää odottamaan niiden valmistumista. Suoritus palaa takaisin käyttöliittymäsäikeeseen ja mahdollistaa siten käyttäjän syötteet. Kun verkon yli tapahtuvat pyynnöt valmistuvat, ne käsitellään käyttöliittymäsäikeessä. [79]

5. TYÖN KOHTEENA OLEVA ROBOTTISOLU

5.1 Rakenne

Soluohjaimen tarkoituksena on mahdollistaa kuvassa 8 esitetyn kahden työaseman robottisolun hallinta. Työasemien määrän muuttuminen on huomioitu ohjelmistoarkkitehtuurissa, ja soluohjainta on mahdollista käyttää myös robottisoluissa, joissa työasemien määrä voi vaihdella yhden ja kolmen välillä. Operaattori hallitsee solua teollisuus-PC:n kosketusruudulta, joka jää esitetyssä kuvassa piiloon kuvan etualalle oikeaan alanurkkaan.



Kuva 8 *Robottisolu, jossa on käytössä työn soluohjain.*

Soluohjain tarjoaa videokuvan enintään kolmelle teollisuuskameralle. Robottisolun kuvasta 8 käy hyvin ilmi, minkä takia robotin työskentelyalueelta halutaan saada videokuvaa. Kirkas hitsausliekki vaatii korkean turva-aidan, joten robotin työskentelyalueelle näkeminen ilman kameroita ei onnistu. Kamerat on kiinnitetty vaakatasossa olevan ulkoisen akselin molempiin päätyihin kuvaamaan sitä lähimpänä olevaa pyörityspöytää, jossa kappaleen hitsaaminen tapahtuu.

Molemmille työalueille pääsyä on rajoitettu turva-PLC:llä toteutetulla valvonnalla.

Turvaovet sallivat pääsyn työalueelle, jossa robotti ei työskentele. Jos ovi avataan alueelle, jossa robotti työskentelee, pysäytetään robotti välittömästi. Näin tapahtuu myös tilanteessa, jossa robotti on vaihtamassa työaluetta alueelle, jonka turvaovi on avattuna.

5.2 Soluohjaimen vaatimusten määrittely

Käyttötapauskaaviossa 9 esitetään soluohjaimelta vaadittavat ominaisuudet korkealla tasolla. Näistä jokaisesta on kirjoitettu tarkempi käyttötapauskuvaus, ja niiden perusteella on laadittu matalimman tason ohjelmistovaatimukset, joiden perusteella ominaisuudet on voitu toteuttaa. Harmaalla korostetut käyttötapaukset liittyvät robotin liikuttamiseen ja niiden toteuttamisessa priorisoidaan muita vaatimuksia enemmän luvussa 3 esitettyjä luotettavuutta lisääviä keinoja. Vaatimukset ovat muodostuneet työn tilaajan kanssa käytyjen palaverien pohjalta ja ne ovat muuttuneet sekä tarkentuneet työn kirjoituksen aikana.

Kuva 9 *Järjestelmän käyttötapaukset.*

Ohjelmistovaatimusten muuttumista on helpottanut niiden hallinnassa käytetty *Visual Studio Team Services* -palvelu, joka sisältää vaatimustenhallintaohjelman lisäksi versiohallinta -palvelimen. Sen käyttö on mahdollistanut useimpien luvussa 3.3.1 kuvattujen hyvien käytäntöjen hyödyntämisen vaatimustenhallinnassa. Näihin ovat kuuluneet mm. vaatimusten:

- priorisointi
- tärkeyden perustelu kommentoinnin avulla
- ryhmittely

- validoinnin seuraaminen tilatiedon avulla
- viittaukset rajapintadokumentaatioihin, kuviin ja kaavioihin sekä
- ohjelmistovaatimusten nopea muokkaaminen.

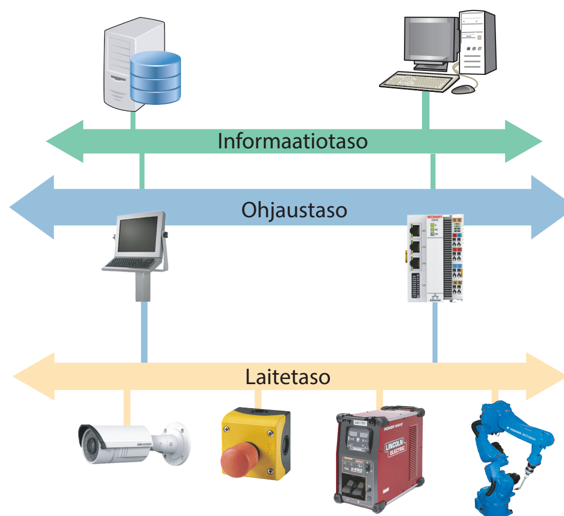
Soluohjain toimii pehmeässä reaaliajassa. Sille ei ole asetettu erillisiä reaaliaikavaatimuksia. Tämä johtuu siitä, että soluohjaimen aikaansaama viive on merkityksetön sen rinnalla, kuinka kauan varsinaisen kappaleen käsittelyssä kuluu aikaa. Käsittelyajat ovat tyypillisesti useita tunteja. Tämän vuoksi sovelluksen käyttövarmuus pitkäkestoistenkin ajojen aikana on reaaliaikavaatimuksia tärkeämpää.

Kuvassa 9 esitetään käyttäjänä robottisolun operaattori ja oikeassa reunassa olevat ominaisuudet liittyvät välillisesti soluohjaimen toimintaan, joihin operaattorilla ei ole suoraa pääsyä. Käyttövarmuuden kasvattamisen lisäksi toteutuksessa pyritään löytämään ratkaisu luvussa 2.5 esitettyihin ongelma-alueisiin.

Toteutuksen pääpaino tulee olemaan teollisuus-PC:n ja robotin välisessä tiedonsiirrossa, sillä luvussa 2.5 kuvattujen ongelmien ratkaisun kannalta näiden alueiden merkitys korostuu. Teollisuus-PC:lle rakennettavasta soluohjaimesta löytyy myös muita käyttötapauskaavion ulkopuolelle jääviä toiminnallisia vaatimuksia, joita saatetaan sivuuttaa työssä. Näiden tarkempi tarkastelu rajataan kuitenkin pois työn sisällöstä, koska ne eivät tuo käyttövarmuuden kasvattamisen näkökulmasta lisäarvoa työlle.

5.3 Soluohjain

Työssä esitellyistä laitteista kolme tärkeintä ovat robotti, PLC ja teollisuus-PC. Niiden välinen viestien koordinointi on soluohjaimen toteutuksen kannalta keskeisessä roolissa. Kuvassa 10 esitetään laitteiden sijoittuminen tyypillisen teollisuusverkon eri hierarkiatasoille.



Kuva 10 *Soluohjaimen laitteiden sijoittuminen teollisuusverkon eri hierarkiatasoille. Suomennettu ja muokattu lähteestä [59, s. 318].*

Soluohjain ei sisällä työn kirjoitus hetkellä kommunikointia informaatiotasolle, jonne esimerkiksi myynti- ja toiminnanohjausjärjestelmät kuuluvat. PLC ja teollisuus-PC, jossa soluohjainsovellusta ajetaan sijoittuvat kuvassa ohjaustasolle, sillä molemmat pyrkivät ohjaamaan robottisolun toimintaa. Ne myös kommunikoivat keskenään toteuttaessaan rooliaan osana soluohjainta.

Tyypillisesti robottiohjain sijoitettaisiin nimensä mukaisesti ohjaustasolle, mutta tässä sovelluksessa sen toiminta muistuttaa enemmän passiivista laitetta, joka vastaanottaa soluohjainsovellukselta tulevia ohjauskäskyjä. Tämän vuoksi sen voidaan ajatella kuuluvan alimmalle laitetasolle. Samassa kuvassa esitetty punainen laite on hitsauksessa käytettävä virtalähde, joka tarjoaa informaatiota hitsauksessa käytettävistä suureista kuten jännitteen ja virran arvoista. Lisäksi ohjaustasolla on näkyvillä turvalaitteiden ryhmää kuvastava hätäpysäytyspainike sekä valvontakamera, joka välittää kuvaa hitsausalueelta.

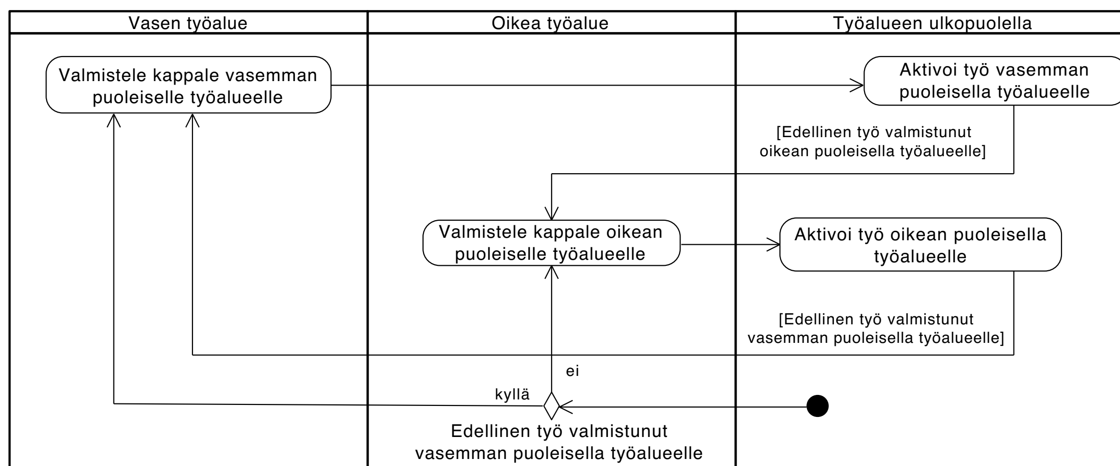
PLC:n vastuualueena on mahdollistaa robotin ja PLC:n välinen I/O:seen perustuva viestien välitys, sekä välittää teollisuus-PC:ltä tulevat komennot eteenpäin. Kaikki robotille päin menevä ohjaukseen liittyvä viestiliikenne kulkee PLC:n kautta. Se pitää yllä robottisolun tilatietoa, jonka avulla tiedetään reaaliajassa mitä robottisolussa minäkin hetkenä tapahtuu. Kokonaiskuva erilaisista tiloista on nähtävillä myöhemmin käsiteltävässä tilakaaviossa 16. Lisäksi PLC vastaa robottisolun turvallisuudesta hallinnoimalla mm. turvarajakytkimiä ja valoverhoja, mutta niiden käsittely jää tämän työn ulkopuolelle.

Robotti suorittaa varsinaisen työkappaleen käsittelyn. Se odottaa PLC:ltä saapuvia viestejä kuten työn käynnistyskomentoa ja tarjoaa rajapinnat, joiden avulla uusia työtiedostoja voidaan siirtää sen muistiin. Robotti antaa tarvittaessa myös hälytyksiä, jotka voivat liittyä mm. käsivarren törmäykseen tai hitsauslangan loppumiseen. Lisäksi robotti kertoo PLC:lle omasta tilastaan kuten milloin se on tietyllä työasemalla tai koska työ on aktivoitunut.

Teollisuus-PC:llä on ajossa ohjelma, jota nimitetään tässä työssä soluohjainsovellukseksi. Se tarjoaa solun operaattorille kokonaisvaltaisen työkalun hallita robotin töiden ajoa. Ohjelman tehtävänä on välittää operaattorilta tulevat käskyt PLC:lle tai tiedostojen hallinnan osalta suoraan robotille. Lista sen erilaisista tehtävistä on esitelty aiemmin läpi käydyssä käyttötapauskaaviossa 9.

5.3.1 Hallittavan robottisolun työkierto

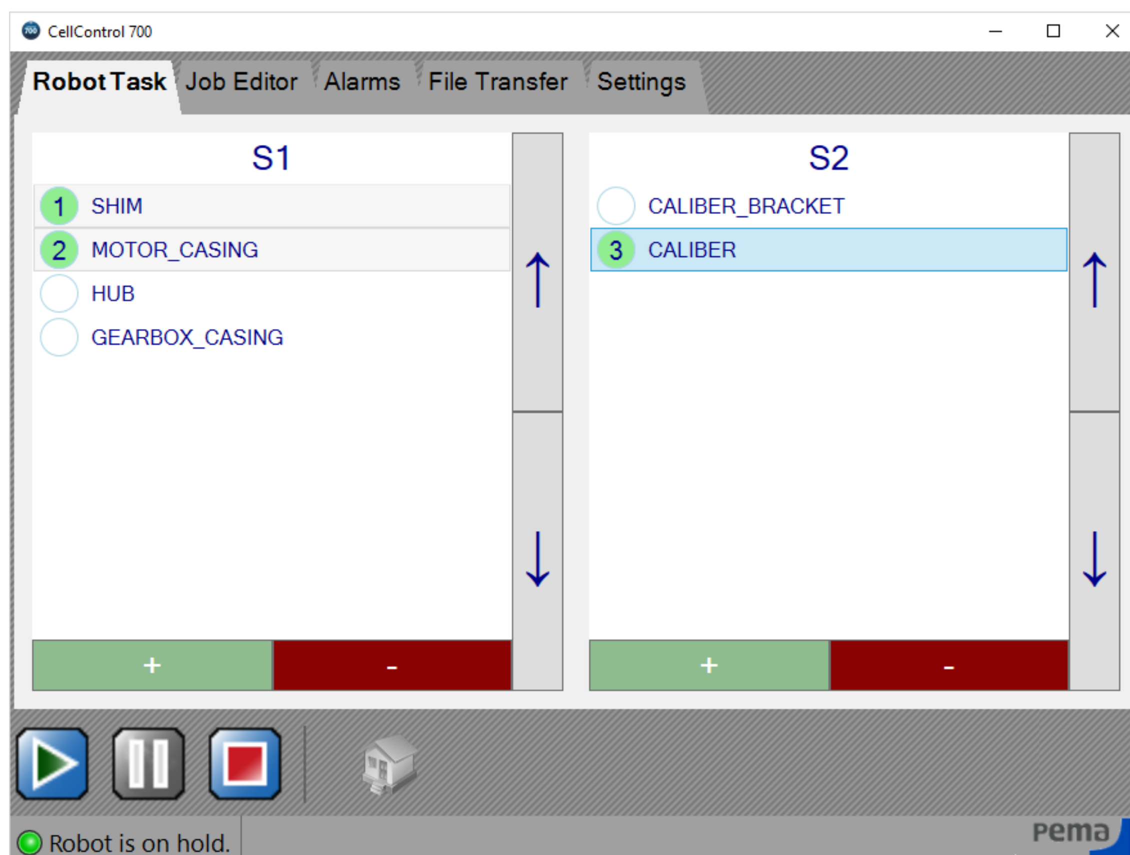
Kahden työpisteen robottisolun työkierto on esitetty aktiviteettikaaviossa 11. Kun operaattori saapuu asemalle, hän tarkastaa kumpi asemista on vapaa käytettäväksi. Tilanteessa, jossa vasemmanpuoleinen asema on vapaa, hän valmistelee asemalla alkavan hitsaustyön asettamalla työstettävän kappaleen pyörityspöydälle. Tämän jälkeen hän siirtyy pois robotin työskentelyalueelta aseman ulkopuolella olevalle teollisuus-PC:lle, jossa soluohjainsovellus on käytössä.



Kuva 11 *Operaattorin työkierto robottiasemalla.*

Operaattori aktivoi kuvassa 12 esitetystä soluohjainsovelluksen näkymästä vasemman puoleiselle työalueelle yhden tai useamman hitsaustyön. Hitsaustyöt ovat ennestään opetettu robotille tai ne luodaan niitä generoivalla robotin etäohjelmointiin

tarkoitetulla ohjelmalla. Jos robotti ei ole vielä käynnissä, operaattori käynnistää soluohjaimesta työn käyntiin. Tämän jälkeen robotti suorittaa operaattorin valitsemat hitsaustyöt kappaleelle.



Kuva 12 *Soluohjainsovelluksen töiden hallintanäkymä.*

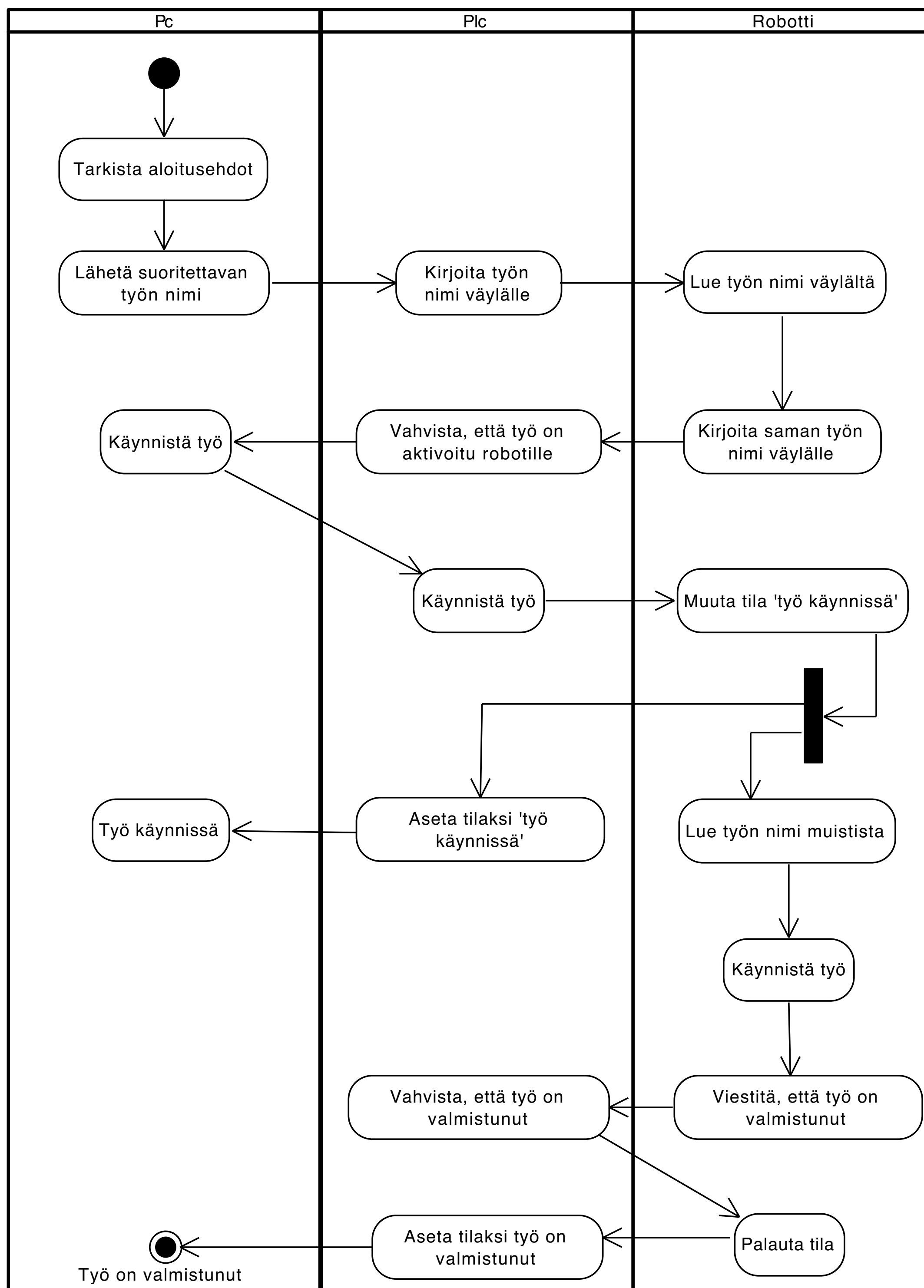
Samaan aikaan kun robotti työskentelee vasemman puoleisella työasemalla, operaattori voi käydä valmistelemassa oikean puoleisen työaseman valmiiksi. Tämän jälkeen hän voi aktivoida oikean puoleisen aseman hitsaustyöt. Kun robotti nyt lopettaa työskentelyn vasemmalla työasemalla, siirtyy se automaattisesti suorittamaan töitä oikean puoleiselle työasemalle ilman, että se tarvitsee operaattorilta erillistä käskyä.

Operaattorille vapautuu erityisesti pitkien hitsaustöiden välissä aikaa tehdä muita asioita kuin valvoa robotin työskentelyä. Sen lisäksi robotin käyttöaste saadaan pysymään korkeana, kun työasemalta toiselle siirtyminen tapahtuu heti edellisen työn loputtua automaattisesti. Näin ollen uusi hitsaustyö toisella asemalla pääsee alkamaan ilman viivettä.

5.3.2 Työn suorittaminen

Kun operaattori käynnistää soluohjaimesta robotilla ajettavan työn käyntiin, käynnistyy tapahtumaketju, jonka tilakaavio on esitetty kuvassa 13. Työn suorituksen monivaiheisuus johtuu siitä, että se toteutetaan 3 erillisen laitteen välisenä kommunikointiketjuna, jossa laitteet lähettävät synkronoidusti viestejä toisilleen ja samalla jäävät odottamaan toisilta saapuvia viestejä.

Työn käynnistäminen alkaa aloitusehtojen tarkistuksella, jotka ovat esitetty myöhemmin läpi käytävässä tilakaaviossa 16. Aloitusehdon epäonnistuminen johtaa siihen, että operaattorille esitetään soluohjaimessa ohjeet, kuinka hänen on mahdollista aloittaa työn suorittaminen.



Kuva 13 *Robotille käynnistettävän työn suorituksen vaiheet.*

Aloitusehtojen täytyttyä, soluohjain lähettää suoritettavan työn nimen PLC:lle hyödyntäen TwinCAT ADS protokollaa. PLC vastaanottaa tiedon nimestä ja kirjoittaa sen DeviceNet-väylälle. Robotti lukee nimen ja kuittaa paluuviestinä takaisin

PLC:lle.

Soluohjain vastaanottaa PLC:ltä tiedon, että robotti on vastaanottanut uuden suoritettavan työn nimen. Tämän jälkeen soluohjain käskyttää PLC:tä uudella viestillä käynnistämään työn. PLC välittää viestin robotille väylää pitkin.

Kun robotti saa viestin käynnistää työ, se ilmoittaa ensimmäisenä takaisin PLC:lle työn olevan käynnissä. PLC välittää tiedon soluohjaimelle, jossa käynnissä olevan työn nimi esitetään operaattorille. Robotti jatkaa tämän jälkeen suoritusta lukemalla käynnistettävän työn nimen muistista, jonka jälkeen sen varsinainen suoritus käynnistyy. Robotin osuus työn käynnistämisestä on kuvattu yksityiskohtaisemmin luvussa 6.3.2.

Työn valmistuttua robotti ilmoittaa tästä PLC:lle kättelynä, jossa molemmat osapuolet varmistuvat viestin perille menosta. Työn suorituksen päättää PLC:n ilmoitus soluohjaimelle, että työ on valmis. Valmiit työt esitetään operaattorille soluohjaimessa.

6. LUOTETTAVUUDEN RAKENTAMINEN OSAKSI SOLUOHJAINTA

6.1 Teollisuus-PC

6.1.1 Ohjelmistoarkkitehtuuri

Ohjelman arkkitehtuurilla voidaan vaikuttaa virheiden määrään ohjelmassa. Modulaarinen rakenne mahdollistaa vian eristämisen koskettamaan pienempää osaa ohjelmaa, jolloin vikaantunut osa ei vaikuta koko ohjelman toimintaan. Modulaarisuuden saavuttamisessa tärkeintä on minimoida ohjelman moduulien välisiä riippuvuuksia. Hyvin toteutetussa arkkitehtuurissa vikaantuminen aiheuttaa poikkeuksen, jolloin viat eivät jää piileviksi. [14] Arkkitehtuurin avulla voidaan mahdollistaa myös ohjelman yksikkötestaaminen ja siten virheiden poistaminen ohjelmasta.

Soluohjainsovelluksen arkkitehtuurirungoksi otetaan käyttöön yleisesti käytössä oleva kerrosarkkitehtuuri, jossa ohjelman rakenne on jaettu kuvan 14 mukaisesti eri abstraktiotasojen mukaan kerroksiin [80, s. 17]. Jokainen kerroksista piilottaa itseään ylemmiltä kerroksilta alempien kerrosten toteutuksen. Kuvassa soluohjainsovellus käyttää alusta-abstraktiokerroksen palveluita, joka puolestaan käyttää robotin, PLC:n ja virtalähteen palveluita, mutta soluohjainsovellus ei ole tietoinen robotin, PLC:n tai virtalähteen olemassaolosta. Sen riippuvuudet rajoittuvat alusta-abstraktiokerrokseen.

Kerrosarkkitehtuurin käytöstä seuraa useita hyötyjä, jotka tekevät ohjelman jatkokehittämisesti helpompaa. Sovelluskehittäjän ei tarvitse ymmärtää kuin sen kerroksen toiminnallisuus, jonka parissa hän työskentelee. Esimerkiksi soluohjainsovelluksen jatkokehittäminen ei vaadi syvällistä ymmärrystä robotin ohjelmasta tai siitä, kuinka virtalähteen hitsausparametrit saapuvat soluohjainsovellukselle, sillä alusta-abstraktiokerros piilottaa näiden toteutusten yksityiskohdat kehittäjältä.

Kerrosten käyttö mahdollistaa myös yksittäisten kerrosten korvaamisen toisella kerroksella, ilman että ylempiin kerroksiin täytyy tehdä muutoksia. [80, s. 17] Jos myöhemmin kävisi ilmi, että I/O:hon pohjautuva viestiliikenne laitteistolle haluttaisiin

Soluohjain

Alusta-abstraktiokerros (PAL)

Robotti | Plc | Virtalähde | Myöhemmin lisättävä laite

Kuva 14 *Soluohjainohjelmiston arkkitehtuuri.*

korvata luvussa 6.5 esitellyllä vaihtoehtoisella toteutuksella, ei soluohjainsovellukseen tarvitsisi tehdä muutoksia. Muutokset koskettaisivat tällöin ainoastaan uuden komponentin sovittamista olemassa olevaan alusta-abstraktiokerroksen rajapintaan, jossa voidaan hyödyntää sovitin -suunnittelumallia (*adapter*), joka muuntaa yhden rajapinnan kutsut toiseen rajapintaan sopiviksi [81, s. 139].

Yksikkötestaamisen mahdollistamiseksi laitteisto- ja alustariippuvainen osuus koodista on pyritty eristämään kuvassa 14 esitetyn alusta-abstraktiokerroksen taakse [35, s. 55]. Kerros piilottaa sovellukselta näkyvyyden varsinaiseen laitteistoon, jolloin laitteiston hallintaan osallistuva osuus koodista voidaan korvata testien ajon aikana testisijaisella. Testisijainen on kehittäjän hallittavissa oleva riippuvuus, jonka avulla korvataan ohjelman todellinen riippuvuus kuten ohjelman riippuvuus laitteistoon. Testisijaisen avulla kehittäjä voi esimerkiksi päättää, mitä syötteitä robotti palauttaa, kun sitä pyydetään kuittaamaan hälytykset. [35, s. 50]

Jos alusta-abstraktiokerros poistettaisiin, olisi soluohjainsovellus testattavissa ainoastaan oikealla laitteistolla. Se tekisi testien ajamisesta hankalaa, kun kehittäjällä tulisi olla testien ajon aikana koko laitteisto saatavilla [82, s. 147]. Lisäksi testien kesto kasvaisi, kun testit liikuttaisivat oikeaa laitteistoa. Osheroven mukaan näiden rajoitteiden seurauksena testejä ajettaisiin harvoin, mikä tarkoittaisi, että testien paljastamien ongelmien löytäminen viivästyisi ja ohjelmistovikojen löytämisestä tulisi hankalampaa ja aikaa vievää [35, s. 9]. Testit alkaisivat hiljalleen menettää merkitystään.

Koska arkkitehtuuri paketoi laitteistoa hallinnoivan osan koodista omaksi komponentikseen, se mahdollistaa laitteistoa käskyttävän komponentin käytön myös kokonaan toisessa sovelluksessa. Käyttö vaatii, että sovellus pystyy hyödyntämään *.NET assemblyjä*. Toisaalta komponentin toteutusta voidaan jatkaa paketoimalla se esimerkiksi mikropalveluarkkitehtuurin mukaisen *REST* -rajapinnan sisälle, jolloin sen käyttö onnistuu millä tahansa ohjelmalla, joka mahdollistaa HTTP -protokollan

käytön.

6.1.2 Asynkroninen laiterajapinta

Microsoft esitteli .NET 4.5 -version myötä asynkronista ohjelmointia helpottavan suunnittelumallin *Task-based Asynchronous Pattern (TAP)*. Mallin käyttöön osana soluohjaimen ohjelmistoarkkitehtuuria päädyttiin Microsoftin suosituksen takia. [83]. Mallin käytöllä tavoiteltiin responsiivisempaa käyttöliittymää, hyödyntämällä sitä kaikissa niissä rajapintametodeissa, jotka osallistuvat PLC:n ja soluohjainsovelluksen väliseen viestiliikenteeseen. Lisäksi se on käytössä FTP -yhteyden käsittelyssä. Pitkäkestoisimpiin kutsuihin kuten tiedostojen latauspyyntöihin on lisäksi lisätty latausanimaatioita, joiden avulla käyttäjälle viestitään toiminnan pidemmästä kestosta.

Ohjelmassa 3 kuvataan erään rajapinnasta löytyvän metodin asynkroninen toteutus. Metodin tarkoituksena on hakea robotilta uusin hälytys ja lisätä se hälytysten hallinnasta vastaavan komponentin esitettäväksi. Ohjelman lähdekoodi muistuttaa rakenteeltaan synkronista koodia, mutta metodin nimen edessä oleva *async* -avainsana kertoo, että metodi on asynkroninen. Avainsanan avulla kääntäjä saa tiedon, että metodista tulee rakentaa tilakone, jonka avulla sen suorituksesta saadaan asynkroninen. Tilakoneesta ja muista toteutuksen yksityiskohdista voi lukea enemmän lähteistä [77] [78].

```
1        private async Task AlarmStepsAsync()
2        {
3            var alarm = await _RobotReadOnlyAccess.GetAlarm();
4
5            _AlarmManager.AddAlarm(alarm.Number, alarm.Name);
6        }
```

Ohjelma 3 *Asynkroninen metodi, joka hakee robotilta uusimman hälytyksen.*

Toteutuksen tärkeä yksityiskohta on *await* -avainsanan käyttö rivillä 3 ennen kuin robotilta pyydetään uusia hälytyksiä. Sen avulla ohjelman suoritus palaa takaisin metodin kutsujalle, eikä jää odottamaan robotilta saapuvaa vastausta. Metodia kutsunut käyttöliittymäsäie ei näin ollen jää odottamaan vaan säilyy responsiivisena. Kun robotti lopulta vastaa uudella hälytyksellä, suoritus palaa takaisin metodin sisälle riville 5 lisäten hälytyksen näkyville käyttöliittymään.

Asynkronisuuden lisääminen pitkäkestoisempiin toimintoihin oli sovelluksen käyttöön liittyvänä muutoksena erityisesti ohjelman käyttäjälle yksi näkyvimmistä. Oh-

jelman käyttö tuntuu nopeammalta, ja käyttöliittymä ei pysähdy odottamaan toimintoja vaan säilyttää responsiivisuutensa. Luotettavuuden näkökulmasta voidaan ajatella, että sovelluksen käyttäjä ei enää erehdy luulemaan, että sovellus olisi kaatunut kun se suorittaa pitkäkestoista operaatiota säilyttäen responsiivisuutensa.

Vaikka asynkronisuuden toteutus onnistui koodin luottavuuden kannalta selkeillä muutoksilla, muutokset kuitenkin lisäsivät toteutuksen monimutkaisuutta. Sen voidaan nähdä heikentävän sovelluksen käyttövarmuutta, jos muutoksen seurauksena ohjelmaan esiteltiin lisää virheitä. Ohjelman käytettävyyden parantumisesta saatava hyöty on kuitenkin suurempi kuin käyttövarmuuden mahdollisesta heikentymisestä johtuva haitta.

6.1.3 Töiden hallinnan yksikkötestaaminen

Sovelluksen yksikkötestaamisen teki mahdolliseksi luvussa 6.1.1 kuvatun arkkitehtuurin käyttöönotto, jonka seurauksena ohjelman rakenteesta tehtiin myös muilta osin modulaarisempi. Soluohjaimen töiden hallintanäkymä, joka on nähtävillä aiemmin esitetyssä kuvassa 12, on eräs niistä komponenteista, jonka toimintalogiikka erotettiin sen käyttöliittymästä. Tämän seurauksena sille päästiin kirjoittamaan yksikkötestejä, jotka pyrkivät todentamaan sen toimintaa.

Yksikkötestit keskittyvät testaamaan näkymässä esitettyjen robotin töiden prioriteetin asettamista, sillä tällä on suuri merkitys siihen, missä järjestyksessä työt ajetaan. Työn kirjoitushetkellä töiden hallintanäkymää testasi 24 yksikkötestiä, joiden avulla pyrittiin varmistamaan komponentin toimivuuden lisäksi sen helpompi jatkokehitys.

Jatkokehityksen voidaan sanoa muuttuneen helpommaksi ainakin tämän komponentin osalta, sillä työn kirjoituksen aikana sen sisäistä rakennetta on muutettu uusien toiminnallisuuksien lisäämisen vuoksi useampaan kertaan. Muutoksien tekeminen helpottui, koska yksikkötestit valvoivat komponentin toiminnallisuuden säilymistä ja mahdollistivat virheiden korjaamisen heti niiden ilmenemisen jälkeen.

Ohjelmassa 4 on esitetty eräs ohjelmassa käytetyistä yksikkötesteistä. Sen tarkoituksena on testata töiden prioriteetin kasvamista, kun käyttäjä aktivoi 3 työtä töiden hallintanäkymästä. Prioriteetilla tarkoitetaan sitä järjestysnumeroa, missä järjestyksessä työt suoritetaan robotilla.

```
[Test]
public void Add_Adds3Jobs_ShouldIncreasePriorityOfEach()
{
    // Arrange
    var jobRepository = new JobRepository();

    var jobWithPriority1 = new Job("JOB_1", WorkStation.S1);
    var jobWithPriority2 = new Job("JOB_2", WorkStation.S1);
    var jobWithPriority3 = new Job("JOB_3", WorkStation.S1);

    // Act
    jobRepository.Add(jobWithPriority1);
    jobRepository.Add(jobWithPriority2);
    jobRepository.Add(jobWithPriority3);

    // Assert
    jobRepository.Get(jobWithPriority1.Id).Priority.ShouldEqual(1);
    jobRepository.Get(jobWithPriority2.Id).Priority.ShouldEqual(2);
    jobRepository.Get(jobWithPriority3.Id).Priority.ShouldEqual(3);
}
```

Ohjelma 4 *Töiden prioriteetin kasvamista testaava yksikkötesti.*

Testien nimeäminen pyrkii mukailemaan Osheroven nimeämiskäytäntöä, joka on tarkemmin kuvattu luvussa 3.5.1. Nimeämiskäytäntö helpotti testien hallintaa ja ylläpitoa erityisesti siinä vaiheessa, kun niiden määrä alkoi lisääntyä.

6.1.4 Töiden hallinnan koodikattavuus

Ohjelman yksikkötestien lausekattavuuden selvittämisessä hyödynnettiin *dotCover* -työkalua, joka on *Visual Studio* kehitysympäristöön asennettava lisäosa. Sen avulla ohjelman eri osille saadaan laskettua lausekattavuus.

Kuvassa 15 esitetään robotin töiden valitsemiseen tarkoitetun luokan lausekattavuus. Sen käyttövarmuus on lopullisen ohjelman toiminnan kannalta oleellista, minkä vuoksi sen lausekattavuus halutaan pitää korkeana. Kuva paljastaa, että 3 metodia on jäänyt yksikkötestien ulkopuolelle, sillä ne ovat saaneet lausekattavuudekseen 0%. Metodit ovat kuitenkin tarkoituksella jätetty testaamatta. Tämä johtuu

siitä, että metodeista kahta käytetään hitaaseen I/O:seen, jonka seurauksena ne soveltuvat huonosti testattavaksi yksikkötesteissä, joiden suorituksen tulee tapahtua nopeasti. Viimeistä metodia käytetään ainoastaan käyttöliittymien suunnitteluun tarkoitetussa työkalussa, eikä se tule osaksi lopullista ohjelmaa.

Symbol	Coverage (%)	Uncovered/Total Stmts.
Total	86%	31/222
WpfUiComponents	86%	31/222
WpfUiComponents.JobControl.ViewModel	86%	31/222
JobControllerVm	86%	31/222
SetHomeArrows(int,int)	100%	0/20
GetNextUserSelectedJob()	100%	0/17
GetNext()	100%	0/14
FirstJobChanged()	100%	0/12
StationControllerOnJobUnSelected(object,Jo	100%	0/12
ResetPriorities(Nullable<int>)	100%	0/10
ClearArrows()	100%	0/9
Remove(string)	100%	0/9
UnSelectAllJobs()	100%	0/9
SetPriorityOneJobSelectionToFalse()	100%	0/7
StationControllers	100%	0/5
StationControllerOnJobSelected(object,Job)	100%	0/4
HasActiveJob()	100%	0/3
JobControllerVm(IRobotJobAccess,IRobotPo	94%	1/16
GetJobThatMovesRobotToWorkStationHom	91%	1/11
CheckIfFirstJobChangedAsync(Nullable<int>	89%	1/9
InitStationController(string,WorkStation,strin	88%	1/8
Add(Job)	86%	1/7
GetIndex(Nullable<WorkStation>)	83%	2/12
IsEnabled	80%	1/5
Save(string)	0%	9/9
Load(string)	0%	9/9
JobControllerVm()	0%	5/5

Kuva 15 *dotCover -työkalun esittämä lausekattavuus robotin työn valinta moduulin lähdekoodeista.*

Lausekattavuuden käytön avulla selviää hyvin, että kaikki oleelliset osat sovellusta tulevat katetuiksi yksikkötesteillä. Lisäksi sen avulla nähdään missä kohdissa testauksen määrää tulisi vielä lisätä, jotta ohjelmaan jäävien virheiden määrää saadaan pudotettua.

Lausekattavuuden käyttö jäi virheitä ennustavista menetelmistä ainoaksi, sillä useat tämän ryhmän keinoista vaativat kehitysprosessilta tarkempaa statistiikan keräämistä. Keinot olisivat vaatineet mm. ohjelmasta löytyvien virheiden esiintymisestä tarkan tilastoinnin, jonka avulla voidaan mallien avulla laskennallisesti arvioida ohjelmassa jäljellä olevien virheiden määrää ja päätellä tästä tarvitseeko ohjelmaa testata vielä lisää. Muiden käyttövarmuuden metriikoiden käytöstä on kerrottu kattavasti

lisää kirjassa [84].

6.1.5 Vikasietoisuuden parantaminen poikkeustenkäsittelyllä

Luvussa 3.4.1 kuvatun ylimmän tason poikkeustenkäsittelijän (Big Outer Try Block) tehtäväksi tuli käsitellä ohjelman odottamattomat poikkeukset, joihin ohjelman arkkitehtuurissa ei olla osattu varautua. Ilman ylimmän tason poikkeuksenkäsittelijää, osa ohjelmaan jäävistä virheistä kaataisi ohjelman siten, että kaatumisesta ei jäisi tietoa mihinkään. Vaikka ohjelman käyttäjä raportoisi ohjelman kaatumiseen johtaneen virheen, olisi sen korjaaminen ilman lokitietoa erittäin hankalaa, sillä virhe voi sijaita missä tahansa kohdassa ohjelmaa. Jokainen poikkeus, joka aikaansaa ohjelman suorituksen siirtymisen tälle tasolle on merkittävä virhe, jonka aiheuttaja tullaan korjaamaan osana ohjelman kehitystyötä.

Tekniikan käyttö olisi tosin entistä tehokkaampaa, jos robottisolut olisivat jatkuvassa yhteydessä internettiin ja jakaisivat kirjatut virheet eteenpäin. Tällä hetkellä virhetiedot saadaan talteen vain pyynnöstä erikseen avattavan etäyhteyden kautta.

Ohjelmaan toteutettiin ylimmän tason poikkeustenkäsittelijän lisäksi myös muita poikkeustenkäsittelijöitä, joiden tarkoituksena oli reagoida ohjelman ajonaikaisiin tunnettuihin vikoihin. Keinot reagoida näihin vaihtuvat ohjelmassa sen mukaan minkälainen palautusmekanismi tietynlaisesta viasta palautumiseen soveltuu. Esimerkiksi robotille lähetettävän käskyn päättyessä aikakatkaisuun, sitä yritetään lähettää poikkeuksenkäsittelijässä vielä kaksi kertaa uudelleen, jonka jälkeen käyttäjälle esitetään viesti epäonnistuneesta komennosta robotille.

Ohjelma sisältää myös poikkeustenkäsittelijöitä odottamattomia poikkeuksia vastaan, joissa ohjelman toimintaa ei keskeytetä, vaan vika kirjataan ylös myöhempää tarkastelua varten. Näiden käyttö on rajattu ohjelman osiin, joissa tiedetään, että suorituksen jatkaminen vian jälkeen on mahdollista ilman, että siitä seuraa suurta haittaa ohjelman käyttäjälle.

6.1.6 Odottamattomasta viasta toipuminen

Ohjelmaan rakennettiin dynaamiseen tilan tallennukseen perustuva palautusmenetelmä, jolla pyrittiin suojautumaan erityisesti odottamattomia vikoja vastaan. Mekanismi rakennettiin osaksi luvussa 6.1.5 kuvattua ylimmän tason poikkeustenkäsittelijää.

Tarkoituksena oli suojautua ohjelman suorituksen aikaisilta vioilta, joiden käsittelyyn ei olla varauduttu. Näissä tapauksissa ohjelman suoritus siirtyy ylimmän tason poikkeustenkäsittelijälle, joka vian ylös kirjaamisen jälkeen käynnistää ohjelman nykyisen tilan tallennuksen. Ohjelma tallentaa:

- tärkeimpien moduulien tilan
- ohjelman käyttäytymiseen vaikuttavat parametrit sekä
- käytössä olevat ohjelman asetukset.

Kun ohjelma käynnistetään uudelleen, tiedot ladataan ohjelman käyttöön. Näin mahdollistetaan, että käyttäjä voi jatkaa siitä mihin oli ennen vian ilmenemistä jäänyt.

Vaikka dynaamisen tilan tallennukseen perustuvan palautusmekanismin aktivoituminen tulisi olla viimeinen keino suojautua ohjelmassa tapahtuneita häiriöitä vastaan, niin sen olemassa olon koettiin nopeuttavan huomattavasti ohjelman käytön jatkamista, jos tällainen häiriö ilmeni. Käyttäjä pystyi sovelluksen uudelleen käynnistymisessä kuluvan ajan jälkeen jatkamaan siitä mihin oli jäänyt.

6.2 PLC

6.2.1 Tilakoneanalyysi robotin ohjaimelle

Järjestelmässä tunnistettiin olevan useita erilaisia tilanteita, joiden aikana robotin ajon tuli olla estetty. Esimerkki tällaisesta tilanteesta on, kun robotin työn yrittää käynnistää, silloin kun turvapiiri on jäänyt kuittaamatta. Näiden tilanteiden tunnistettiin olevan eräs luvussa 2.5 kuvattujen ongelmien aiheuttaja, sillä käyttäjälle ei annettu riittävästi tietoa toimia näissä tilanteissa, koska tietoa ei välitetty soluohjainsovellukselle saakka.

Robotilla on lisäksi valmistajan määrittelemiä tiloja, joita ovat muun muassa: remote, run, hold ja play -tilat. Remote -tilassa robottia on mahdollista ohjata ulkopuolelta, joten tilan tulee olla aktiivinen, kun soluohjainta halutaan käyttää. Run -tilassa robotti suorittaa aktiivisena olevaa työtä. Hold -tilassa aktiivisena oleva työ on keskeytetty siten, että sen jatkaminen on mahdollista. Play -tila ei ole työn kannalta oleellinen, sillä se on tarkoitettu robotin töiden ajamiseen käsiohjaimesta.

Järjestelmästä tunnistetut ajon estävät tilat ja robottivalmistajan määrittelemät tilat ohjasivat kehitystä siten, että PLC:n ohjauslogiikan suunnittelussa hyödynnettiin luvussa 3.3.2 kuvattua tilakoneanalyysiä. Analyysi aloitettiin mallintamalla kaikki tilat tilakaavioksi, jonka jälkeen tilakaaviota muokattiin useassa iteraatiossa, kun siitä tunnistettiin yhdessä toimiala-asiantuntijan kanssa vaaratilanteita. Muokkausten seurauksena tiloja uudelleen järjesteltiin ja uusia tiloja luotiin, kun ne koettiin tarpeelliseksi. Näin tehtiin esimerkiksi, kun haluttiin mahdollistaa työn jatkaminen lisäämällä tilakaavion välitila *CONTINUE-JOB*. Tilakaavion muokkauksilla pyrittiin varmistumaan siitä, että robottia ei voida käynnistää tilanteessa, jossa todennäköisen törmäyksen riski oli olemassa. Lopullinen versio tilakaaviosta on esitetty kuvassa 16.

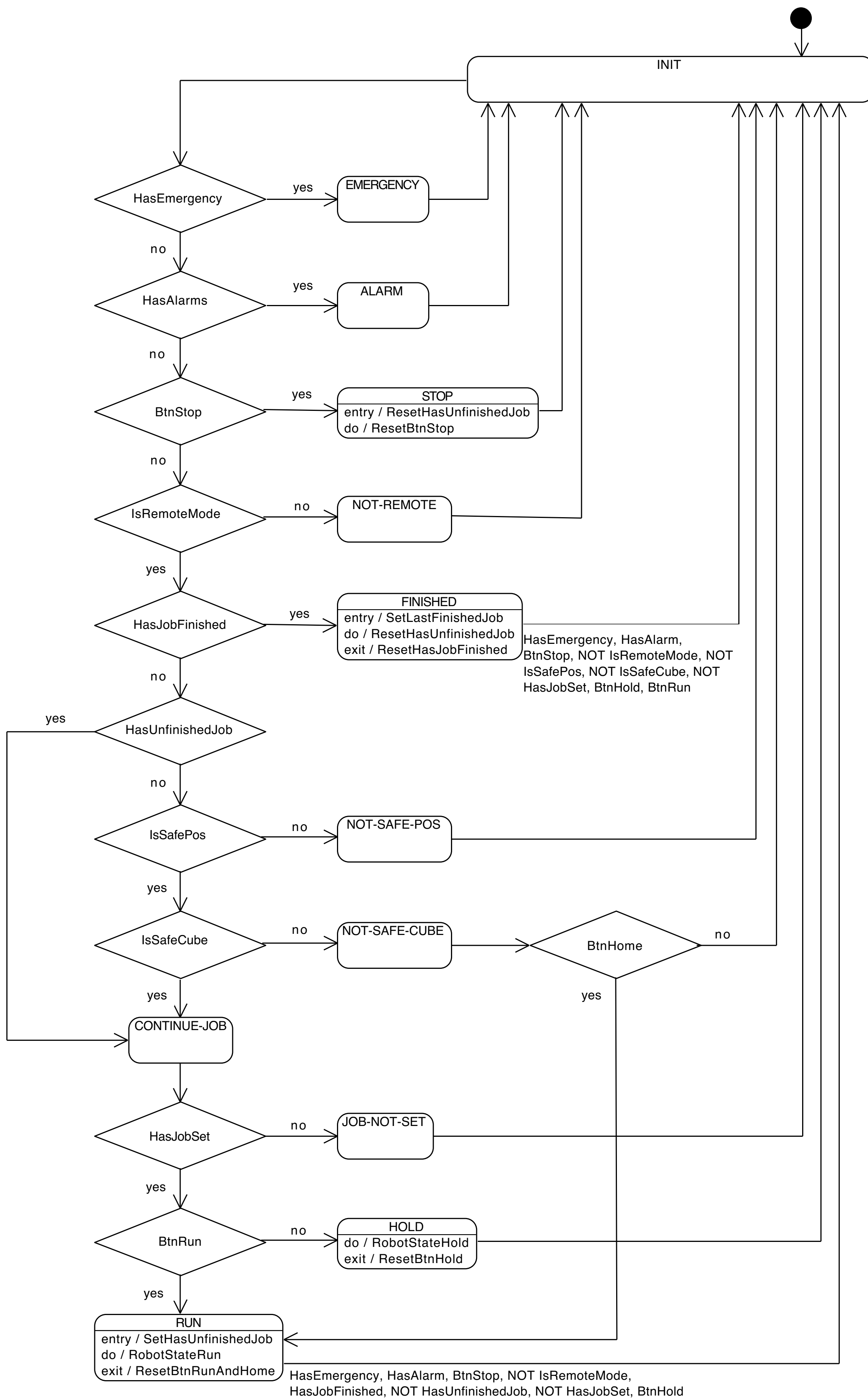
Luvussa 4.1.2 on kerrottu, että SFC soveltuu ohjelmointikielenä hyvin tilallisten järjestelmien mallintamiseen. Tämän vuoksi tilakaavio päädyttiin kääntämään sellaisenaan PLC:n SFC -ohjelmaksi. Ohjelman tekeminen tilakaavion pohjalta oli suoraviivaista, ja kaikki toiminnallisuus saatiin osaksi SFC -ohjelmaa. Ohjelman tarkoituksena oli ohjata robottia sekä välittää järjestelmästä tunnistetut tilatiedot eteenpäin soluohjainsovellukselle.

Tilakaaviosta 16 on oleellista huomata, että siirtymien järjestyksellä on suuri merkitys siihen, mikä tiloista on aktiivinen. Jos robotin ajotila on muutettu käsiohjaimella pois *remote* -tilasta, tulee *NOT-REMOTE* -tila aktiiviseksi. Jos jokin nyt laukaisee turvapiirin esimerkiksi aktivoimalla valoverhon, *EMERGENCY* -tila muuttuu aktiiviseksi, sillä siirtymä siihen tapahtuu ennen *NOT-REMOTE* -tilaa. Toisin sanoen tilojen järjestys tilakaaviossa asettaa niiden keskinäisen prioriteetin siitä, mikä tila on aktiivisena.

Tilakaavion tiloista *FINISHED* ja *RUN* ovat poikkeuksellisia sen osalta, että siirtymä näistä tiloista seuraavaan vaatii toteutuakseen herätteen. *RUN* -tilan osalta herätteenä toimivat erilaiset virhetilanteet, jotka keskeyttävät robotin ajon, työn valmistuminen sekä soluohjaimelta tulevat komennot. *FINISHED* -tilasta pois siirtyminen tapahtuu niin ikään virhetilanteiden ja soluohjaimelta tulevien komentojen kautta. *FINISHED* -tilan siirtymäehtojen puuttumisen huomattiin järjestelmätestauksen aikana aiheuttavan tilanteen, jossa soluohjainsovellus ei koskaan vastaanottanut tietoa työn valmistumista. Tämä johtui siitä, että tila oli aktiivisena niin vähän aikaa, että ohjelma ehti kirjoittaa soluohjainsovellukselle tämän tilan sijasta seuraavana aktivoituvan tilan. Myös *STOP* -tilassa havaittiin sama ongelma, mutta se ratkaistiin pakottamalla tilan aktiivisena ololle minimi aika, jolloin tieto tilan vaihtumisesta ehti saavuttaa soluohjainsovelluksen.

Kun tilakoneen lopullinen muoto oli valmis, sen pohjalta ohjelmoitiin robotin hallintalogiikka. Tilakone muuntui sellaisenaan IEC 61131-3 standardin mukaiseksi SFC ohjelmointikieleksi. Muunnos tapahtui niin suoraviivaisesti, että se saattaisi mahdollistaa automaattisen koodigeneraattorin toteuttamisen tätä käyttötilannetta varten, jolloin PLC:lle kirjoitettavan koodin sijasta voitaisiin hyödyntää tilakaavioita. SFC ohjelman tilatieto välitettiin sovellusohjainsovellukselle, mikä mahdollisti sen, että käyttäjälle voitiin esittää merkityksellisiä virheviestejä, jos robotti ei lähtenyt käyntiin. Viestit sisälsivät ohjeet, kuinka käyttäjä pääsee jatkamaan robotin työn suoritusta.

SFC ohjelmointikielen käytössä havaittiin myöhemmässä vaiheessa ongelmia koodin siirrettävyyden kanssa. Kävi ilmi, että työkaluna käytetty *TwinCAT 2* ei kyennyt siirtämään SFC ohjelmia sellaisenaan projektista toiseen, vaan järjesteli tilasiirtymät uudelleen säilyttäen kuitenkin niiden välisen logiikan. Näin ollen ohjelman luettavuus kärsi pahasti. Tämä tarkoitti, että jos ohjelman rakenteeseen haluttiin tehdä myöhemmin muutoksia, tulisi niiden tekeminen olemaan tilasiirtymien sekoittumisen myötä työlästä.



Kuva 16 *PLC:n robottia ohjaava toteutus tilakaaviona.*

6.2.2 Käyttötapausten toteutuminen

PLC osallistuu osaan kuvassa 9 esitetyistä käyttötapauksista joko toteuttamalla käyttötapauksen täysin tai olemalla jossakin sen vaiheessa mukana. Seuraavaksi käydään läpi, miten PLC mahdollistaa eri käyttötapausten toteuttamisen.

Järjestelmän palauttaminen alkutilanteeseen tapahtuu, kun käyttäjä antaa soluohjaimelta käskyn, jonka PLC vastaanottaa signaalina *BtnStop*. Tilakaaviosta 16 käy ilmi, että PLC käy tilassa *STOP*, jonka aikana nollataan mahdollisuus jatkaa mahdollisesti aiemmin aktiivisena olleen työn suoritusta. Signaalin *BtnStop* tila palautetaan tilasta poistuttaessa.

Ilmoitukset järjestelmän tilasta tapahtuvat siten, että suurimmasta osasta tilakaavion 16 tilojen aktivoitumisesta lähetetään tieto teollisuus-PC:lle. Ainoastaan tiloista *INIT* ja *CONTINE-JOB* ei lähetetä tilatietoa eteenpäin. *CONTINE-JOB* -tilan välittäminen olisi turhaa ja *INIT* -tilatiedon lähettäminen jopa haitallista, sillä tilatieto välitettäisiin vuoron perään sillä hetkellä toisen aktiivisena olevan tilatiedon kanssa.

Kotiasemaan ajaminen on toteutettu mahdollistamalla työn käynnistäminen turvakuution ulkopuolelta. Tämä on esitetty kaaviossa 16 *BtnHome* -napin aktivoidulla signaalilla, jossa siirtymä ohittaa turvakuution ja siirtyy suoraan *RUN* -tilaan. Kotiasentoon ajamisen esiehtona ovat virhetilanteiden puuttuminen ja robotin turvallinen asentotieto.

Työn jatkaminen PLC:n osalta tapahtuu siten, että *RUN* -tilassa aktivoidaan lippu, joka kertoo työn olevan aktiivisena. Työn keskeytyessä, esimerkiksi robotilta saapuvaan hälytykseen, sitä päästää jatkamaan lipun takia suoraan. Se mahdollistaa työn jatkamisen siitä kohdasta, johon robotti keskeytystilanteessa oli jäänyt. Tilakaaviossa 16 on esitetty tilasiirtymä, joka ohittaa robotin työn turvallisen käynnistämisen asentotiedot siirtymällä suoraan *CONTINUE-JOB* -tilaan.

Työn pysäyttäminen tapahtuu PLC:n ollessa *RUN* -tilassa. Soluohjaimelta peräisin oleva signaali *BtnHold* aktivoi siirtymän *RUN* -tilasta aina *HOLD* -tilaan asti, sillä *RUN* -tilasta poistuttaessa *BtnRun* ja *BtnHome* -signaalit nollataan. Heti kun *RUN* -tilasta on poistuttu, robotin työn suoritus pysähtyy.

Työn käynnistäminen onnistuu, kun tilakaavion 16 esiehdot toteutuvat ja soluohjaimelta on tullut signaali *BtnRun*. Tämä aktivoi *RUN* -tilan, jolloin robotti käynnistää aktiivisena olevan työn.

6.3 Robotti

6.3.1 Liitynnät muuhun järjestelmään

Robotti on yhteydessä kahteen muuhun järjestelmän laitteeseen: teollisuus-PC:seen ja PLC:hen. Robotille saapuvat komennot tulevat PLC:ltä. Kommunikointi on toteutettu DeviceNet-väylällä. Väylältä on varattu 2 kappaletta 100 BYTE:n kokoisia taulukoita. Niitä liikutetaan väylän hyötykuormana (*payload*) käyttämättä kuvassa 5 esitettyä *CIP* -protokollaa. PLC:ltä robotille menevän tiedon välitykseen varattu taulukko on esitetty sisältöineen kuvassa 17. Robotilta PLC:lle saapuvan tiedon välitykseen varattu taulukko on puolestaan esitetty kuvassa 18. Molempien taulukoiden sisältö on linkitetty robotin omiin muistipaikkoihin.

	1	2	3	4	5	6	7	8	9	0
0	Työn hallinta		Kuit-taukset							
1			Robotilla suoritettavan työn nimi ASCII kirjaimilla							
2										
3										
4										
5										
6										
7										
8										
9										

Kuva 17 *DeviceNet-väylää pitkin PLC:ltä robotille menevä 100 BYTE:n taulukko.*

Taulukoita yhdistää alue 'robotilla suoritettavan työn nimen kirjoittamiseen ASCII kirjaimilla', sillä se on esitettynä molemmissa taulukoissa. Tämä johtuu siitä, että PLC:n ohjelmalogiikkaan on kirjoitettu varmistusehto, jossa työn siirto katsotaan onnistuneeksi robotille vasta siinä vaiheessa, kun robotti on peilannut työn nimen takaisin PLC:lle.

Kommunikaatio ei tarvitse tässä vaiheessa kuin pienen osan taulukon alkioista, joten taulukkoon on jätetty varaa myöhemmille laajennuksille. Nämä näkyvät kuvissa tyhjinä valkoisina neliöinä. Värjätyt neliöt kuvaavat käytössä olevia taulukon alkioita. Niissä on lyhyt kuvaus siitä, minkä tyyppistä tietoa taulukon alkion avulla välitetään.

Jokainen BYTE kuvaa esitettävästä tietotyypistä riippuen eri määrän informaatiota. Yhdellä BYTE:llä voitaisiin esittää $2^8 = 256$ eri tilaa, mutta yksinkertaisuuden vuoksi sillä esitettävien tilojen määrä on 8, jolloin jokainen kahdeksasta bitistä voi esittää kahta tilaa. Näin on tehty esimerkiksi kuvassa 17 esitetyn kolmannen vihreän BYTE:n osalta, jossa sillä välitetään 4 eri robotin kuittaustietoa, jotka ovat auki esitettynä kuvassa 19.

	1	2	3	4	5	6	7	8	9	0
0	Robotin tila		Sijainti	Turvapiirit						
1			Robotilla suoritettavan työn nimi ASCII kirjaimilla							
2										
3										
4			Päätason hälytyskoodi		Alatason hälytyskoodi					
5										
6										
7										
8										
9										

Kuva 18 *DeviceNet-väylää pitkin robotilta PLC:lle saapuva 100 BYTE:n taulukko.*

Robotin ja teollisuus-PC:n välinen yhteys toimii ethernet kaapelia pitkin FTP -yhteyden avulla. Yhteyden tarkoituksena on tarjota mahdollisuus siirtää teollisuus-PC:ltä ".jbi" -päätteisiä työtiedostoja robotille. Työtiedostot sisältävät kaiken tiedon, jota robotti tarvitsee esimerkiksi tietynlaisen kappaleen hitsaamiseksi. Toisin sanoen se sisältää liikepisteet, joihin robotin tulee liikkua, hitsauksen käynnistys- ja sammutuskomennot ja muuta työn suorituksen kannalta oleellista informaatiota,

joka on tarkemmin kuvattu Motomanin käyttäjän manuaalissa [85].

Yhteys toimii myös toiseen suuntaan, jolloin tiedostoja voidaan ladata robotilta teollisuus-PC:lle. Tiedostoja, joita FTP-yhteyden avulla voidaan robotilta ladata ovat muun muassa robotin asetustiedostot, joista voidaan ottaa ajastettuja varmuuskopioita. Ne sisältävät tietoa kuten hitsauksen aloitus- ja lopetusarvo parametreja ja robottiaseman nollapisteen sijainnin. Näiden tietojen häviäminen pysäyttää aseman käytön pidemmäksi aikaa, sillä niiden uudelleen hakeminen vie aikaa. Varmuuskopiot nopeuttavat palautusprosessia.

6.3.2 Käyttötapausten toteutuminen

Robotti osallistuu suurimpaan osaan kuvassa 9 esitetyistä käyttötapauksista joko toteuttamalla käyttötapauksen täysin tai olemalla jossakin sen vaiheessa mukana. Seuraavaksi käydään läpi, miten robotti mahdollistaa eri käyttötapausten toteuttamisen.

Uuden työn siirtäminen robotille on toteutettu luvussa 6.3.1 kuvatun kaltaisesti.

Työn jatkaminen ja -pysäyttäminen sekä hälytysten kuittaaminen onnistuvat välittämällä DeviceNet-väylää pitkin robotille ohjaustietoa. Tiedon välitystapa on näissä tilanteissa saman kaltainen ja ainoastaan muistipaikat muuttuvat. Keskitytään tämän vuoksi tarkastelemaan hälytyksen kuittaamista. Se tapahtuu PLC:n ohjelmassa välittämällä kuvassa 17 vihreällä värillä esitetyssä 3:essa BYTE:ssä tieto, että robotilla oleva hälytys halutaan kuitata. Kuvassa 19 on esitettynä kyseisen BYTE:n auki kirjoitettu binäärimuoto. Siitä selviää, että BYTE:een tulee kirjoittaa binääriesityksenä arvo **0 1 0 0 0 0 0 0**. Toisena olevaan bittiin, joka siis kuvasi hälytysten kuittaamista, on kirjoitettu arvoksi 1 ja muut bitit on jätetty arvoon 0. Vastaavasti kirjoittamalla samaan kohtaan binääriesitys **0 1 1 0 0 0 0 0** kuitattaisiin hälytykset sekä ilmoitettaisiin, että robotti on työasemalla S1.

3
0 Kuit-taukset

nosturi	häly-tykset	työase-malla S1	työase-malla S2				
1	2	3	4	5	6	7	8

Kuva 19 *Robotille menevän taulukon 3:nnen BYTE:n binääriesitys.*

Motomanin robotti vastaanottaa väylältä saapuvan taulukon. Robotille on kirjoitettu PLC:n tapaan I/O:iden tiloja hallinnoiva logiikkaohjelma tikapuuohjelmoinnilla. Ohjelmaan on lisätty kohta, jossa väylää pitkin saapuva yksittäinen bitti aktivoi robotin ulkopuolisen hälytyksen kuittaamisen. Oikean rekisteripaikan löytämiseksi käytetään apuna valmistajan tarjoamaa manuaalia [86].

Tilatiedon välitys, robotilla olevien hälytysten- ja nykyisen työaseman hakeminen sekä työn valmistumisesta ilmoittaminen ovat toteutukseltaan saman kaltaisia toimintoja. Ne poikkeavat yllä esitetystä tapauksesta ainoastaan väylällä liikkuvan taulukon osalta. Niissä käytetään kuvan 18 mukaista taulukkoa eli tietoliikenteen suunta on robotilta PLC:lle päin. Toinen eroavaisuus löytyy ‘nykyisen työaseman hakemisesta’, sillä sitä varten robotille on toteutettu molempia työasemia varten omat kuutiot. Kuutio on käytännössä servomoottorien pulssitietojen avulla rajattu alue. Jos esimerkiksi ulkoisen akselin servomoottorin pulssitiedon arvo ylittää raja-arvon, tiedetään että robotti on siirtynyt kyseisen kuution ulkopuolelle. Kun robotin sijainti on kuution sisällä, kirjoittaa robotti väylällä olevaan bittiin tiedon, että robotti on kyseissä työasemassa. Molemmille työasemille löytyy oma tilatieto.

Työn aktivointi kuluttaa robotin prosesseista eniten väylän kapasiteettia, sillä siinä tieto liikkuu sekä PLC:ltä robotille, että robotilta PLC:lle varaten 15 BYTE:n edestä muistia. Hälytyksen kuittaamisen kaltaisesti PLC aloittaa kirjoittamaan robotin työn nimeä ASCII kirjain kerrallaan kuvan 17 taulukkoon. Työn nimi voi sisältää enintään 15 merkkiä, sillä taulukon koko nimen osalta on rajattu. Robotti vastaanottaa työn nimen väylää pitkin ja kirjoittaa tämän jälkeen kuvan 18 taulukkoon juuri vastaanottamansa työn nimen. Toisin sanoen robotti kaiuttaa saamansa työn nimen takaisin PLC:lle. Näin tehdään, jotta PLC voi varmistua siitä, että robot-

ti vastaanottaa nimen sellaisena kuin se oli PLC:n lähetysvaiheessa. PLC suorittaa robotilta saamansa vastauksen perusteella tarkastuksen, jossa se vertaan lähetettyä nimeä vastaanotettuun ja mahdollistaa työn ajamisen vasta tämän jälkeen.

Työn käynnistäminen voi tapahtua vasta työn aktivoinnin jälkeen kuten kuvasta 13 käy ilmi. Esiehtona on, että käynnistettävän työn nimi on kirjoitettuna robotin muistiin. Kuten hälytyksen kuittaamisessa, robotti saa väylältä tiedon, että työn käynnistäminen tulee aloittaa. Tämän seurauksena robotti suorittaa pääohjelmakutsun, jolloin liitteessä A esitetty ohjelma käynnistyy. Heti alussa robotin tila muuttuu osoittamaan, että työ on käynnissä, jonka jälkeen se ilmoittaa takaisin PLC:lle, että työ on käynnistetty. Tämän jälkeen robotti lukee aiemmin väylää pitkin saapuneen aktivoidun työn nimen muistista kirjain kirjaimelta. Työn nimen lukemisen suorittaa liitteessä B esitetty aliohjelma. Kun työn nimi on kokonaisuudessaan luettu, suoritetaan sen varsinainen käynnistäminen. Kun käynnistetty työ tulee suoritetuksi loppuun, robotti ilmoittaa PLC:lle, että työ on valmistunut ja jää odottamaan PLC:ltä kuittausta. Kun kuittaus saapuu, robotti palautuu alkutilanteeseen.

6.4 Testaus

6.4.1 Järjestelmätestaus

Järjestelmätestauksen lähtötilanteessa järjestelmä vastasi toimituskuntoista robottiasemaa. Testaus suoritettiin aseman omalta PC:ltä samalla tavoin kuin tuleva operaattori tulisi asemaa käyttämään. Testauksen tarkoituksena oli todentaa, että järjestelmä vastasi sen määrittelyä. Testaus suoritettiin tutkivana testauksena, jossa keskityttiin yhdellä kertaa testaamaan yhtä ohjelman kokonaisuutta, mutta käyden kuitenkin läpi ohjelman kaikki toiminnot.

Järjestelmätestausta suoritettiin pääsääntöisesti isompien ohjelmamuutosten jälkeen testausasemalla, joka vastasi ominaisuuksiltaan asiakkaalle toimitettavaa asemaa. Tämän lisäksi järjestelmätestausta suoritettiin aina ennen kuin uusi asema toimitettiin asiakkaalle. Isona erona näiden tilanteiden välillä oli, että testiaseman ja asiakkaalle toimitettavan aseman robottien konfiguraatiot saattoivat erota toisistaan. Muutokset johtuivat erilaisten antureiden ja laitteiden lisäyksestä, jonka seurauksena näiden hallintaa varten täytyi tehdä muutoksia robotin I/O-aluetta hallinoivaan tikapuuohjelmaan, jossa sijaitsi myös rajapinta, jota PLC hyödynsi robotin kanssa kommunikointiin. Testauksen aikana kävi ilmi, että toisinaan nämä muutokset rikkoivat osittain rajapinnan toiminnan. Tämän seurauksena järjestelmätestauksessa löytyi virheitä, jossa robotti ei esimerkiksi vastannut sille soluohjainsovelluk-

selta saapuvaan työn pysäytys -komentoon. Tämä johti järjestelmätestauksen tarkistuslistojen käyttöönottoon, joissa listattiin rajapinnan käytön kannalta oleellisia toimintoja, joiden tuli kaikissa tilanteissa toimia.

Testauksessa pyrittiin myös löytämään sellaisia käyttötapauksia, joita suunnitteluvaiheessa ei ole osattu huomioida. Nämä käyttötapaukset voivat olla mahdollisia ohjelman virhelähteitä. Eräs tunnistamaton käyttötapaus, joka johti lopulta virheen löytämiseen, selvisi seuraavasti. Ohjelmaa oli testattu ajamalla useita eri kestoisia robotin töitä, joita oli ajoittain pysäytetty ja jatkettu. Uutta työtä lisätessä testaajalle tuli mieleen lisätä sama työ ajettavaksi peräkkäin. Tilanne oli täysin mahdollinen, ja ohjelman tuli kyetä suorittamaan se ilman häiriöitä. Näin ei kuitenkaan käynyt, vaan töiden suoritus pysähtyi ensimmäisen työn valmistuttua häiriöön. Ohjelmassa käyttöönotetun luvussa 6.1.5 esitellyn poikkeustenhallinan seurauksena häiriö ei kuitenkaan päässyt etenemään ohjelmassa pitkälle, vaan siitä esitettiin käyttäjälle virheilmoitus. Myöhemmin suoritettava tarkempi ohjelman tutkiminen osoitti, että käytössä oleva tapa aktivoida työ sisälsi virheen, jonka seurauksena samaa työtä ei pystytty suorittamaan useampaa kertaa peräkkäin. Edellä kuvatun kaltaista tilannetta, jossa edellinen testi vaikuttaa seuraavan testin suoritukseen, pidetään tutkivan testauksen vahvuutena. Se mahdollistaa virheiden löytämisen myös määrittelyn ulkopuolelta, joka tuli osoitetuksi myös tätä ohjelmaa testatessa.

6.4.2 Kuormitustestaus

Soluohjainsovelluksen työlista täytettiin sadoista lyhytkestoisista robotin töistä. Tämän jälkeen töiden ajo käynnistettiin. Robotti ajoi useita tunteja töitä läpi. Kun ajoon laitettavien töiden lukumäärä oli ennalta tiedossa, voitiin ohjelmaan lisättävän laskurin avulla todentaa, että ohjelma oli todella suorittanut tarpeellisen määrän töitä. Huomiota kiinnitettiin siihen, että suorituksessa ei hypätty töiden yli tai kokonaan keskeytetty työn suoritusta. Vain täysin suoritetut työt kasvattivat laskurin arvoa. Samalla seurattiin heittikö ohjelma suorituksen aikana odottamattomia poikkeuksia, jotka tallentuivat ohjelman lokiin.

Aikaa vievän kuormitustestauksen käyttö oli ohjelman kehityksen alkuvaiheessa paljon käytössä, sillä sen avulla löydettiin virheitä, jotka johtuivat viestien ajoituksesta. Ohjelman käyttövarmuus kasvoi näiden virheiden korjauksen myötä erityisesti peräkkäin suoritettavien töiden osalta. Kun ohjelma alkoi selvitä testiajosta ongelmitta, kuormitustestauksen käyttö väheni tapahtuvaksi lähinnä uuden aseman käyttöönoton yhteyteen.

6.5 Vaihtoehtoinen toteutus - High Speed Ethernet Client

6.5.1 Muutoksen vaikutus

Nykyisessä toteutuksessa teollisuus-PC:ltä PLC:lle lähtevät viestit eivät suoraan sisällä tukea paluuarvolle. I/O:seen perustuvassa viestiliikenteessä paluuarvon käsite on rakennettu tärkeimpiin viesteihin erillisenä toteutuksena. Näin on toimittu esimerkiksi robotin työn käynnistämisen osalta kuten luvussa 5.3.2 kuvataan. HSE tarjoaa tuen paluuarvoille osana valmista rajapintaa. Jokainen viesti palauttaa *RecStatus* olion, joka sisältää tiedon siitä hyväksyikö robotti sille välitetyn käskyn. Jos robotti hylkää viestin, paluuarvona on syy viestin hylkäämiselle. Lisäksi viesteissä on tuki aikakatkaisulle, jolloin kirjastoa käyttävä asiakasohjelma voi informoida käyttäjäänsä, että yksittäisen viestin perillemeno on estynyt. Verrattuna nykyiseen toteutukseen HSE mahdollistaisi paluuarvojen ja aikakatkaisun käytön kaikessa PC:n ja robotin välisessä viestiliikenteessä. Yksinkertaisempi ja yhtenäisempi viestirajapinta tulisi todennäköisesti kasvattamaan sovelluksen käyttövarmuutta. [75]

Kirjaston käyttöönottaminen yksinkertaistaa soluohjaimen arkkitehtuuria, sillä viestiliikenne robotille ei enää kulkisi PLC:n kautta, vaan tieto siirtyisi suoraan robotille ethernet kaapelia pitkin. Tämä ei kuitenkaan tarkoita, että DeviceNet -väylästä voidaan luopua kokonaan, sillä PLC:n ja robotin välistä väylää tarvitaan edelleen turvalogiikan toteutuksessa, johon soluohjain ei osallistu. Soluohjaimen arkkitehtuuri yksinkertaistuu myös tiedostojen käsittelyn osalta, sillä tarve toteuttaa tätä tarkoitusta varten FTP -rajapinta poistuu. FTP:n käyttö töiden hallinnassa voidaan korvata HSE:n palveluilla. Suuntaus on sama mitä luvussa 4.2.3 kerrottiin FTP:n korvaamisesta HTTP -protokollalla.

Suurimpia etuja I/O:seen perustuvan viestiliikenteen korvaamisessa HSE:llä on käyttövarmuuden kasvaminen. Robotin I/O-rajapinnan herkkyys muutoksille on käynyt selväksi erityisesti järjestelmätestauksen aikana kuten luvussa 6.4.1 kuvataan. Nykyisen rajapinnan käyttöönotto uudessa robottisolussa on virheherkkää, sillä pienikin muutos väylällä liikkuvan hyötykuorman rakenteeseen saattaa osittain tai täysin rikkoa yhteyden. HSE:n käyttö mahdollistaa sen, että robotin I/O-alueen muutokset eivät vaikuta rajapinnan toimintaan yhtä laajasti kuin ennen, sillä suurin osa vaadittavasta toiminnallisuudesta saadaan sen avulla toteutettua siten, että riippuvuus robotin I/O-alueeseen poistuu. Robotin kuutiotietojen välitys tulee kuitenkin edelleen tapahtumaan I/O-pohjaisesti, sillä HSE ei tarjoa tukea tämän tiedon hakemiseen. Yhteysvirheet ovat vanhassa toteutuksessa HSE:tä todennäköisempiä, sillä yhteys voi olla poikki erinäisistä syistä teollisuus-PC:n ja PLC:n välillä tai PLC:n ja robotin välillä.

HSE:n käyttö tulee parantamaan soluohjaimen ylläpidettävyyttä. Rajapinnan laajentaminen uudella toiminnallisuudella on nykyisessä toteutuksessa hidasta, sillä muutosten tekeminen tapahtuu teollisuus-PC:n ohjelman lisäksi PLC:lle ja robotille. Kehittäjän tulee toisin sanoen ymmärtää näiden kaikkien sisäinen toteutus halutessaan laajentaa rajapintaa. Uudessa toteutuksessa osa muutoksista saadaan toteutettua suoraan rajapintaa hyödyntämällä pelkästään teollisuus-PC:n ohjelmaan, ja hankalimmassakin tapauksessa muutokset koskettavat tämän lisäksi vain robotin I/O:ta hallinnoivaa tikapuuohjelmaa.

Vasteaikojen (*response time*) osalta ei voida antaa varmoja tuloksia ilman mittauksia, mutta tästä huolimatta PLC:n pois jääminen viestiliikenne -ketjusta voidaan olettaa tiputtavan vasteaikoja merkittävästi. Syynä tähän on erityisesti se, että yksittäisen viestin ei tarvitse kuvassa 13 esitetyn tavoin kulkea usean laiterajapinnan läpi, joista jokainen lisää oman osuutensa vasteaikaan. Soluohjaimen käyttäjän kannalta vasteaikojen pienentyminen näkyy soluohjaimen nopeampana käytettävyytenä.

6.5.2 Käyttötapausten toteutuminen

Taulukkoon 3 on kerätty käyttötapauskaavion 9 käyttötapaukset, ja High Speed Ethernet Client -kirjastosta löytyvät rajapintametodit, joiden avulla käyttötapausten vaatima toiminnallisuus saadaan osaksi soluohjainta. Tärkeintä on huomata, että taulukko sisältää kaikki käyttötapauskaavion tilanteet, mikä tarkoittaa, että kaikki nykyinen toiminnallisuus voidaan toteuttaa uuden rajapinnan avulla.

Taulukko 3 *Vasemmalla esitetään työn kannalta merkittäviä käyttötapauksia, jotka sovelluksen täytyy toteuttaa. Oikealla puolella on esitetty, mitä rajapintametodeja kutsumalla High Speed Ethernet Client -kirjasto toteuttaa käyttötapauksen.*

Käyttötapaus	Toteutuksessa tarvittava(t) rajapintametodi(t)
Siirrä uusi työ robotille	`RecStatus FileLoad(string Filename, byte[] Data)`
Poista työ robotilta	`RecStatus FileDelete(string Filename)`
Lataa työ robotilta	`RecStatus FileSave(string Filename, out byte[] Data)`
Hae robotilla olevat työt	`RecStatus FileList(string Filter, out List<string> Filenames)`
Pysäytä työ	`RecStatus Hold(int OnOff)`
Käynnistä työ	`RecStatus StartJob()`
Aktivoi työ	`RecStatus JobSelect(int Type, string JobName, uint LineNumber)`
Hae nykyinen työasema	`RecStatus IORead(int LogicalNumber, out byte Data)`
Aja kotiasemaan	Toiminnallisuus yhdistämällä: "Hae nykyinen työasema" ja "Käynnistä työ"
Näe järjestelmän hälytykset	`RecStatus ReadAlarmData(int AlarmNumber, out Alarm_Data AD)`
Kuittaa hälytykset	`RecStatus AlarmReset()`
Ilmoitukset järjestelmän tilasta	`RecStatus StatusInformationRead(out Status_Information AD)` ja `HSEClient.StatusNotification`
Hae valmistuneen työn nimi	`RecStatus ExecutingJobInformationRead(int Task, out ExecutingJobInfo EJI)`
Muodosta työjono	Soluohjaimen tehtävä
Palauta alkutilanteeseen	Ei käyttöä

Suurimpaan osaan käyttötapauksista löytyy kyseistä tarkoitusta varten tehty metodi rajapinnasta, jota käyttämällä robotin tikapuuohjelmaan ei tarvitse tehdä muutoksia. Näistä löytyy lisätietoja teknisestä dokumentaatiosta [75]. Poikkeuksena on "Hae nykyinen työasema" -käyttötapaus, jota ei sellaisenaan löydy rajapinnasta. Tämä tarkoittaa, että toteutus tulee rakentaa joiltakin osin osaksi robotin tikapuuohjelmaa. Käytännössä tämä tapahtuu samalla tavalla kuin luvussa 6.3.2 kuvataan kohdassa "Tilatiedon välitys" eli robotin ohjelmaa muutetaan kirjoittamaan haluttu arvo muistiin, josta se luetaan rajapinnan tarjoamalla metodilla *IORead*. Tässä tapauksessa haluttu arvo on tieto siitä, että robotti on työasema kuution sisällä.

Taulukossa esitetty kohta "Palauta alkutilanteeseen" tulisi jäämään uudessa toteutuksessa pois, sillä sen olemassaolon tarkoituksena on varmistua, että PLC:n ja robotin välinen synkronoitu viestiliikenne saadaan palautettua toimintaan, jos se jostain syystä lukkiutuu. Koska tämä osuus viestiliikenteestä poistuu uuden toteutuksen myötä, poistuu myös tarve käyttötapaukselle.

7. ARVIOINTI

Työn tarkoituksena oli selvittää kuinka ohjelmistokehityksen keinoin voidaan rakentaa ohjelma, jonka käyttövarmuusaste on korkea. Keinojen tuli soveltua käytettäväksi keskeneräiseen projektiin, jonka kehittäminen tapahtui ketterien ohjelmistokehityksen periaatteiden mukaisesti. Työn 3 luvussa tähän etsittiin vastausta kirjallisuustutkimuksena ja luvussa 6 kirjallisuudesta löytyneitä keinoja sovellettiin käytäntöön. Seuraavaksi pohditaan kuinka keinojen käyttö lisäsi ohjelman käyttövarmuutta ja kuinka tämä on verrattavissa kirjallisuudessa esitettyyn.

Virheiden välttämiseen kuuluvat keinot osoittautuivat erityisen tehokkaaksi tavaksi lisätä ohjelman käyttövarmuutta. Tilakoneanalyysin käyttö auttoi hahmottamaan ohjauslogiikkaan liittyviä luvussa 2.5 kuvattuja ongelma-alueita sekä löytämään näihin vaihtoehtoisia ratkaisuja. Analyysin käyttö nopeutti kehitystyötä, kun muutosten tekeminen tapahtui PLC -ohjelmakoodin sijasta malliin. Sen käyttö auttoi kommunikoimaan ohjauslogiikan toiminnasta muiden osallisten kanssa, sillä tilakoneen syntaksin selvittäminen niille, jotka eivät olleet sitä aiemmin käyttäneet oli suoraviivaista. Suurin osa virheistä, jotka olisivat vaikuttaneet järjestelmän käyttövarmuuteen, kyettiin korjaaman yhdessä toimiala-asiantuntijoiden kanssa jo mallia kehitettäessä. Havainnot analyysimenetelmän käytöstä tukivat kirjallisuudesta löytyviä havaintoja, joiden mukaan analyysin kerrotaan soveltuvan vaihtoehtoisten ratkaisujen löytämiseen yhdessä toimiala-asiantuntijoiden kanssa. Sovellettaessa menetelmää huomattiin, että sen lopputuloksena syntyvä tilakaavio kääntyi erinomaisesti SFC -ohjelmaksi, mikä osaltaan vähentää virheiden syntymistä tässä vaiheessa. Tilakoneen muuttamista automaattisen koodigeneraattorin avulla ST -ohjelmaksi on tutkittu [87], mutta sen käyttö keskittyy testiympäristöihin.

Virheitä poistavista keinoissa työssä hyödynnettiin ohjelmistotestaamista sen eri muodoissa. Projektin alussa järjestelmätestauksen aikana paljastui useita virheitä, joiden määrä kuitenkin väheni selvästi järjestelmätestauskertojen määrän mukana, jolla oli suora vaikutus ohjelman käyttövarmuuteen. Havaittujen virheiden korjauksen lisäksi tähän on osaltaan saattanut vaikuttaa aseman parissa työskennelleiden henkilöiden kokemuksen lisääntyminen siten, että heidän tekemiensä virheiden määrä uusissa tuotantojulkaisuissa on vähentynyt.

Järjestelmätestauksessa koettiin ajoittain vaikeaksi keksiä uusia tapoja löytää ohjelmasta virheitä, sillä ajettavat testitapaukset alkoivat usein muistuttaa toisiaan. Lisäksi järjestelmän sisäisen toteutuksen tuntemisesta oli ajoittain haittaa, koska testaaja ei nähnyt järjestelmässä kaikkia niitä ongelmia, joita järjestelmää tuntematon operaattori olisi nähnyt. Kirjallisuudessa varoitetaan, että testauksen puolueettomuus kärsi sen seurauksena, että ohjelmaa testaa sen kehittäjä, eikä esimerkiksi ulkopuolinen testausorganisaatio [88, s. 14]. Asia tuli osoitettua käytännössä.

Ajastukseen liittyvien virheiden poistaminen pitkäkestoisen kuormitustestauksen avulla osoittautui hyväksi keinoksi, sillä sen avulla löydettiin ohjelmasta tiloja, jotka johtivat sen lukkiutumiseen. Virheet olivat vaikeasti paikannettavia, minkä takia ne olisivat saattaneet jäädä helposti huomaamatta muilla menetelmillä. Tämä tukee luvussa 3.5.3 esitettyä, jossa kirjallisuudessa on käytetty Long sequence testing -tyyppistä testauksen muotoa ajoitusvirheiden löytämiseksi.

Yksikkötestien käyttö sovelluksen kriittisemmissä komponenteissa vaikutti selvästi näistä komponenteista löytyvien virheiden määrään. Havaittiin, että komponenteista, jotka sisälsivät vähemmän tai eivät ollenkaan yksikkötestejä löytyi useammin virheitä järjestelmätestausta suoritettaessa. Sen lisäksi näiden komponenttien jatkokehityksen koettiin olevan helpompaa, koska testit valvoivat toiminnallisuuden säilymistä ohjelmistovaatimusten mukaisina muokkausten aikana, koska testejä ajettiin automaattisesti osana ohjelman käännösprosessia. Luvussa 3.7.2 todetaan, että jatkuvan integraation toteuttamisen edellytys on automaattisesti ajettavat yksikkötestit, joiden kerrotaan vähentävän ohjelmassa esiintyvien virheiden määrää nopeasti saatavan palautteen avulla. Tässä työssä käytetyt yksikkötestit osoittivat saman.

Yksikkötestien kirjoittaminen koko ohjelman lähdekoodille ei kuitenkaan olisi ollut järkevää, koska niiden kirjoittaminen vaati ohjelmakoodin muokkaamista siten, että se soveltui yksikkötestattavaksi. Tämä tarkoitti esimerkiksi ylimääräisten riippuvuuksien poistoa sitomalla ohjelman muodostavien luokkien riippuvuudet konkreettisten toteutusten sijaan abstrakteihin luokkiin ja rajapintoihin, joka puolestaan mahdollisti testisijaisten käytön ja siten yksikkötestien kirjoittamisen. Näin ollen yksikkötestejä keskityttiin kirjoittamaan uusille luokille sekä ohjelman kriittisimmille alueille. Myös kirjallisuudessa tiedostetaan sama ongelma, sillä luvussa 3.7.3 todetaan tämän olevan yksi suurimmista syistä, jotka estävät jatkuvan toimituksen käyttöönoton jo alkaneissa projekteissa.

Ohjelman käyttövarmuuden mittaamisessa on mahdollista hyödyntää virheiden ennustamiseen perustuvia keinoja, joista tässä työssä käytettiin ohjelmistometriikoihin kuuluvaa koodikattavuutta yksikkötestien kattavuuden arvioinnissa. Tähän tarkoi-

tukseen se soveltui erinomaisesti. Tosin sen vaikutuksen arviointi ohjelman käyttövarmuuteen on hankalaa. Tästä huolimatta metriikat, joiden laskenta tapahtuu osana kehitysympäristöä ohjelmoinnin edetessä ovat vaivattomia käyttää. Tästä syystä ei ole syytä olla käyttämättä näiden metriikoiden antamaa informaatiota ohjelman käyttövarmuuden parantamiseksi osana tavallista kehitystyötä. Tämän tärkeyttä korostaa se, että IBM on patentoinut koodikattavuuden metriikan käytön yhdessä ohjelman kehitysympäristön (*integrated development environment, IDE*) kanssa [89]. Lisäksi koodikattavuuden käytöllä nähtiin olevan positiivinen vaikutus yksikkötestien määrään, sillä sen käyttö lisäsi innokuutta kirjoittaa yksikkötestejä.

Hankalammin käyttöönotettavista metriikoista ei saatu tässä työssä tuloksia, mutta erityisesti luotettavuutta arvioivien tekniikoiden käyttö vaikuttaisi lupaavalta. Ne edellyttävät ohjelmistokehitysprosessin muotoilua siten, että se mahdollistaisi systemaattisen tavan kerätä ohjelmassa esiintyviä virheitä ylös ja hyödyntää saatua dataa käyttövarmuuden mallin rakentamisessa (*reliability growth model*). Mallien käyttöä ohjelman käyttövarmuuden kasvattamiseksi tutkitaan aktiivisesti [90] [91]. Malleista saatavat hyödyt ovat, että testaukseen tarvittava ajankäyttö olisi ennalta tiedossa, eikä siihen kulutettaisi ylimääräistä aikaa yrittäen poistaa virheitä, jotka luotettavuuden arvion mukaan epätodennäköisesti ovat olemassa. Se toimisi päätöksenteon tukena siitä, mitä käyttövarmuuden näkökulmasta tulisi tehdä seuraavaksi.

Kaikkien ohjelmassa esiintyvien virheiden poistaminen ei ole mahdollista jo pelkästään siksi, että ohjelman koon kasvaessa siinä esiintyvien suorituspolkujen määrä kasvaa eksponentiaalisesti. Tämän vuoksi ohjelmaan jäävien vikojen hallinnassa hyödynnetään vikasietoisuuden keinoja. Erityisen hyödyllisenä koettiin ylimmän tason poikkeustenkäsittelijän käyttö, joka mahdollisti häiriöiksi johtaneiden virheiden järjestelmällisen kirjaamisen koko ohjelman elinkaaren aikana. Mitä pidempään ohjelma on käytössä, sitä enemmän virheitä tällä tekniikalla tullaan löytämään. Tämä tukee kirjallisuudesta löytyviä näkemyksiä tekniikan käytöstä, joissa sen käytön on sanottu auttavan ohjelman hallittuun alasajoon, jolloin ohjelman kaatumiseen johtaneiden syiden kirjaaminen on mahdollista kuten luvun 3.4.1 kirjallisuudessa on kuvattu.

8. YHTEENVETO

Työssä kehitettiin teollisuusrobottisolun hallintaan tarkoitettu soluohjainohjelmisto, joka mahdollistaa robottisolun hallinnan teollisuus-PC:n kosketusnäytöltä. Soluohjaimelta vaadittiin korkeaa käyttövarmuutta, jonka vuoksi diplomityössä pyrittiin löytämään ohjelmistokehityksen keinoja, joiden hyödyntäminen soluohjaimen kehitystyössä mahdollistaisi tavoitteeseen pääsyn. Näitä keinoja pyritään hyödyntämään myös uusissa alkavissa projekteissa.

Useaan kertaan projektin aikana suoritettu järjestelmätestaus osoitti, että soluohjaimen käyttövarmuuden kannalta sen heikoimmaksi kohdaksi jäi robotin puoleinen osuus kommunikointirajapinnan toteutuksesta eli robotin I/O alue. Oli tyypillistä, että se saattoi uusien toimitusten yhteydessä muuttua erilaisen robottisolun kokoonpanon myötä.

Robotin rajapintatoteutuksen muuttuminen toimitusten välissä on suurimpia syitä, miksi luvussa 6.5 esitettyä vaihtoehtoista kommunikointitapaa robotille suositellaan kokeiltavaksi soluohjaimessa. Uusi kommunikointitapa ei ole riippuvainen robotin I/O-alueesta, jolloin I/O-alueen muutokset eivät aiheuta virheitä sen toiminnassa. Koska soluohjaimen laitteistosta riippuva osuus on luvussa 6.1.1 kuvatun kaltaisesti eristetty abstraktion taakse, pitäisi toteutuksen vaihtaminen onnistua ilman suuria muutoksia nykyiseen koodikantaan.

Seuraavaksi käydään lyhyesti läpi asioita, joiden käyttöä uusissa projekteissa tulisi käyttövarmuuden kasvattamisen takia suosia. Tilakoneanalyysin käyttöä suositellaan kokeiltavaksi niissä projekteissa, joissa on tarkoitus toteuttaa PLC:llä ohjausta. Analyysin käytön nähtiin alentavan kehitykseen kuluvaa aikaa ja virheiden esittelyä ohjelmaan. Kirjalliset ohjelmistovaatimukset ovat edellytyksenä ohjelman testaamiselle, joten niiden johdonmukainen käyttö uusissa projekteissa on tärkeää. Järjestelmätestauksen tehokkuutta olisi mahdollista parantaa ulkopuolisen testaajan käytöllä, jolloin testauksen aikana löydettävien virheiden määrä tulisi todennäköisesti kasvamaan. Ohjelmistometriikat, joita ohjelman kehitysympäristö tarjoaa valmiina olisi syytä ottaa käyttöön. Niiden käyttö ei aiheuta lisätyötä, vaan motivoi ohjelmistokehittäjiä parantamaan esimerkiksi ohjelman testikattavuutta kuten kävi

koodikattavuus metriikan käytön aloittamisen yhteydessä.

DevOpsin käytön aloittamiselle tulisi varata uusissa alkavissa projekteissa aikaa, sillä siitä saatavat hyödyt kuten ohjelman käyttövarmuuden kasvaminen, julkaisunopeuden paraneminen ja nopean palautteen hyödyntäminen ovat suuria. Erityisesti nopean palautteen hyödyntäminen tuotekehityksessä, jossa ohjelmistovaatimukset muuttuvat usein toisi todennäköisesti parhaimman hyödyn.

Käyttöönottoa ei tarvitse aloittaa ottamalla käyttöön kaikkea kerralla, vaan liikkeelle voisi lähteä pyrkimällä aluksi ottamaan käyttöön jatkuvan integraation, mikä tarkoittaisi testaamisen huomioimista ohjelman rakenteessa ja ohjelman automaattisen kääntämisen mahdollistamista. Saavutettavia hyötyjä olisivat, että osa ohjelmaan esitellyistä virheistä saataisiin korjattua nykyistä nopeammin ja sovelluksesta olisi tuotantoympäristöä vastaavalla asemalla aina käytössä uusi toimiva versio, jonka perusteella sen kehityksestä voisi antaa palautetta ohjaamaan sen jatkokehitystä.

Uusi soluohjain otettiin käyttöön ensimmäisen asiakkaan tiloissa elokuussa 2017. Sen jälkeen soluohjain on käyttöönotettu yhteensä 8 asiakkaalla, joista kahden käyttöönotto on vielä kesken. Käyttöönoton yhteydessä koko toimitettavalle robottisolulle suoritetaan hyväksymistestaus, jossa asiakas on mukana. Näissä testeissä soluohjaimen toimintaa on kuvattu ongelmattomaksi. Työn kirjoitushetkellä yhteensä mitattu tuotantoaika soluohjaimella on hiukan yli 2 vuotta. Tänä aikana ei ole asiakkaiden toimesta raportoitu ongelmista, jotka olisivat aiheuttaneet luvussa 2.5 kuvatun kaltaisia häiriöitä. Ohjelman käyttö ja sen kehitys jatkuvat edelleen, mutta tähänastisen palautteen perusteella soluohjaimen käyttövarmuus on saavuttanut hyväksyttävän tason sen käyttäjien keskuudessa.

LÄHTEET

[1] ”IEEE Standard Computer Dictionary: A Compilation of IEEE Standard Computer Glossaries”. In: *IEEE Std 610* (Jan. 1991), pp. 1–217.

[2] L. Vanhala. *Näin VR sotki lippujärjestelmänsä – Miksi it-projektit epäonnistuvat?* Suomen Kuvalehti. Tammikuu 2012. URL: https://suomenkuvalehti.fi/jutut/kotimaa/nain-vr-sotki-lippujarjestelmansa-miksi-it-projektit-epaonnistuvat/ (viitattu 01. 12. 2017).

[3] S. Korhonen. *VR myöntää: it-ongelmat olisi pitänyt tunnistaa etukäteen.* Tivi. Marraskuu 2011. URL: https://www.tivi.fi/CIO/2011-10-05/VR-my%C3%B6nt%C3%A4%C3%A4-it-ongelmat-olisi-pit%C3%A4nyt-tunnistaa-etuk%C3%A4teen-3187161.html (viitattu 01. 12. 2017).

[4] *Pemamek yrityksenä.* Pemamek Oy. URL: http://www.pemamek.com/fi/pemamek (viitattu 14. 06. 2017).

[5] *Referenssit.* Pemamek Oy. URL: http://www.pemamek.com/fi/referenssit (viitattu 14. 06. 2017).

[6] J.-C. Laprie. *Dependability: Basic Concepts and Terminology.* Springer Vienna, 1992. ISBN: 9783709191705. DOI: 10.1007/978-3-7091-9170-5.

[7] ”IEEE Guide for the Use of IEEE Standard Dictionary of Measures to Produce Reliable Software”. In: *IEEE Std 982.2-1988* (1989). DOI: 10.1109/IEEESTD.1989.122630.

[8] A. Avizienis, J. C. Laprie, B. Randell, and C. Landwehr. ”Basic concepts and taxonomy of dependable and secure computing”. In: *IEEE Transactions on Dependable and Secure Computing* 1.1 (Jan. 2004), pp. 11–33. ISSN: 1545-5971. DOI: 10.1109/TDSC.2004.2.

[9] W. Heimerdinger and C. Weinstock. *A Conceptual Framework for System Fault Tolerance.* Tech. rep. CMU/SEI-92-TR-033. Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 1992.

[10] T. Anderson and P. Lee. *Fault tolerance, principles and practice.* Prentice/Hall International, 1981, p. 369. ISBN: 9780133082548.

[11] M. Kaâniche, J.-C. Laprie, and J.-P. Blanquart. "A framework for dependability engineering of critical computing systems". In: *Safety Science* 40.9 (2002), pp. 731–752. ISSN: 0925-7535.

[12] M. R. Lyu. *Handbook of Software Reliability Engineering*. Hightstown, NJ, USA: McGraw-Hill, Inc., 1996. ISBN: 0-07-039400-8.

[13] J. Humble and D. Farley. *Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation*. 1st. Addison-Wesley Professional, 2010. ISBN: 0321601912, 9780321601919.

[14] M. R. Lyu. "Software Reliability Engineering: A Roadmap". In: *Future of Software Engineering, 2007. FOSE '07*. May 2007, pp. 153–170. DOI: 10.1109/FOSE.2007.24.

[15] K. Wiegers and J. Beatty. *Software Requirements*. 3rd ed. Redmond, WA, USA: Microsoft Press, 2013. ISBN: 0735679665.

[16] L. L. Pullum. *Software Fault Tolerance Techniques and Implementation*. Norwood, MA, USA: Artech House, Inc., 2001. ISBN: 1-58053-137-7.

[17] L. Rierson. *Developing Safety-Critical Software: A Practical Guide for Aviation Software and DO-178C Compliance*. CRC Press, 2017. ISBN: 9781439813690.

[18] T. Schäfer, A. Knapp, and S. Merz. "Model Checking UML State Machines and Collaborations". In: *Electronic Notes in Theoretical Computer Science* 55.3 (2001). Workshop on Software Model Checking (in connection with CAV '01), pp. 357–369. ISSN: 1571-0661.

[19] M. Balser, S. Bäumler, A. Knapp, W. Reif, and A. Thums. "Interactive Verification of UML State Machines". In: *Formal Methods and Software Engineering*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 434–448. ISBN: 978-3-540-30482-1.

[20] *Review Of Techniques To Support The Eatmp Safety Assessment Methodology*. Tech. rep. 1. European Organisation for Safety of Air Navigation, Jan. 2004, p. 117.

[21] N. G. Leveson. *SafeWare: System Safety and Computers*. Computer Science and Electrical Engineering Series. Addison-Wesley, 1995. ISBN: 9780201119725.

[22] N. G. Leveson. *Engineering a Safer World.* MIT University Press Group Ltd, 2012, p. 560. ISBN: 0262016621.

[23] H. Harju. *Ohjelmiston luotettavuuden kvalitatiivinen arviointi.* Valtion teknillinen tutkimuskeskus (VTT), 2000. ISBN: 9513857670.

[24] "ISO/IEC/IEEE International Standard - Systems and software engineering–Vocabulary". In: *ISO/IEC/IEEE 24765:2017(E)* (Aug. 2017), pp. 1–541. DOI: 10.1109/IEEESTD.2017.8016712.

[25] J. Cooling. *Software Engineering for Real-Time Systems.* Addison-Wesley, 2002. ISBN: 0201596202.

[26] M. Rintala. *Techniques for Implementing Concurrent Exceptions in C++.* Tampere University of Technology. Publication. Tampere University of Technology, Nov. 2012. ISBN: 978-952-15-2915-3.

[27] B. Stroustrup. *The C++ Programming Language, Third Edition.* 3rd. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1997. ISBN: 0201889544.

[28] A. Longshaw and E. Woods. "Patterns for Generation, Handling and Management of Errors." In: *Proceedings of the 9th European Conference on Pattern Languages of Programms (EuroPLoP '2004).* Irsee, Germany, Jan. 2004, pp. 27–52.

[29] A. Haase. "Java Idioms - Exception Handling." In: *Proceedings of the 7th European Conference on Pattern Languages of Programms (EuroPLoP '2002).* Irsee, Germany, Jan. 2002, pp. 41–70.

[30] C.-T. Chen, Y. C. Cheng, C.-Y. Hsieh, and I.-L. Wu. "Exception handling refactorings: Directed by goals and driven by bug fixing". In: *Journal of Systems and Software* 82.2 (2009), pp. 333–345.

[31] W. Torres-Pomales. *Software Fault Tolerance: A Tutorial.* Tech. rep. NASA Langley Research Center, Oct. 2010, p. 66.

[32] J. Hursey, J. M. Squyres, T. I. Mattox, and A. Lumsdaine. "The Design and Implementation of Checkpoint/Restart Process Fault Tolerance for Open MPI". In: *2007 IEEE International Parallel and Distributed Processing Sym-*

posium. Long Beach, CA, USA, Mar. 2007, pp. 1–8. DOI: 10.1109/IPDPS.2007.370605.

[33] P. A. Lee and T. Anderson. *Fault Tolerance*. Springer Vienna, 1990. ISBN: 978-3-7091-8992-4.

[34] B. W. Boehm and P. N. Papaccio. "Understanding and controlling software costs". In: *IEEE Transactions on Software Engineering* 14.10 (Oct. 1988), pp. 1462–1477. ISSN: 0098-5589. DOI: 10.1109/32.6191.

[35] R. Osherove. *The Art of Unit Testing*. Manning, 2013, p. 266. ISBN: 9781617290893.

[36] W. E. Lewis. *Software Testing and Continuous Quality Improvement, Third Edition*. Auerbach Publications, 2008. ISBN: 1420080733.

[37] K. Frajtak, M. Bures, and I. Jelinek. "Model-Based Testing and Exploratory Testing: Is Synergy Possible?" In: *2016 6th International Conference on IT Convergence and Security (ICITCS)*. Sept. 2016, pp. 1–6. DOI: 10.1109/ICITCS.2016.7740354.

[38] J. Itkonen and K. Rautiainen. "Exploratory testing: a multiple case study". In: *2005 International Symposium on Empirical Software Engineering, 2005.* Nov. 2005, p. 10. DOI: 10.1109/ISESE.2005.1541817.

[39] J. Itkonen, M. V. Mäntylä, and C. Lassenius. "The Role of the Tester's Knowledge in Exploratory Software Testing". In: *IEEE Transactions on Software Engineering* 39.5 (May 2013), pp. 707–724. ISSN: 0098-5589. DOI: 10.1109/TSE.2012.55.

[40] B. Wood and D. James. "Applying Session-Based Testing to Medical Software". In: *Medical Device & Diagnostic Industry* 25.5 (May 2013), p. 90.

[41] D. J. Berndt and A. Watkins. "High Volume Software Testing using Genetic Algorithms". In: *Proceedings of the 38th Annual Hawaii International Conference on System Sciences*. Jan. 2005, 318b–318b. DOI: 10.1109/HICSS.2005.296.

[42] C. Kaner, W. P. Bond, and P. McGee. "High Volume Test Automation". In: *International Conference on Software Testing Analysis and Review (STAR)*. Orlando, Florida, May 2003.

[43] A. K. Pandey and N. K. Goyal. *Early Software Reliability Prediction: A Fuzzy Logic Approach*. Springer Publishing Company, Incorporated, 2013. ISBN: 8132211758, 9788132211754.

[44] R. Patton. *Software Testing*. Sams, 2000. ISBN: 0-672-31983-7.

[45] J. R. Horgan, S. London, and M. R. Lyu. "Achieving software quality with testing coverage measures". In: *Computer* 27.9 (Sept. 1994), pp. 60–69. ISSN: 0018-9162. DOI: 10.1109/2.312032.

[46] P. G. Frankl and E. J. Weyuker. "An applicable family of data flow testing criteria". In: *IEEE Transactions on Software Engineering* 14.10 (Oct. 1988), pp. 1483–1498. ISSN: 0098-5589. DOI: 10.1109/32.6194.

[47] S. Rapps and E. J. Weyuker. "Selecting Software Test Data Using Data Flow Information". In: *IEEE Transactions on Software Engineering* SE-11.4 (Apr. 1985), pp. 367–375. ISSN: 0098-5589. DOI: 10.1109/TSE.1985.232226.

[48] *RTCA DO-178C, Software Considerations in Airborne Systems and Equipment Certification*. RTCA, Inc. Dec. 2011.

[49] *Manifesto for Agile Software Development*. URL: http://agilemanifesto.org/ (visited on 05/14/2018).

[50] J. D. Arthur and J. B. Dabney. "Applying standard independent verification and validation (IV V) techniques within an Agile framework: Is there a compatibility issue?" In: *2017 Annual IEEE International Systems Conference (SysCon)*. Apr. 2017, pp. 1–5. DOI: 10.1109/SYSCON.2017.7934770.

[51] V. Rantala, K. Könnölä, S. Suomi, M. Isomäki, and T. Lehtonen. "Agile Embedded System Development Versus European Space Standards". In: *Int. J. Inf. Syst. Soc. Chang.* 8.1 (Jan. 2017), pp. 1–23. ISSN: 1941-868X. DOI: 10.4018/IJISSC.2017010101.

[52] L. Riungu-Kalliosaari, S. Makinen, L. E. Lwakatare, J. Tiihonen, and T. Männistö. "DevOps Adoption Benefits and Challenges in Practice: A Case Study". In: Nov. 2016, pp. 590–597. ISBN: 978-3-319-49093-9.

[53] *Principles behind the Agile Manifesto*. URL: http://agilemanifesto.org/principles.html (visited on 05/15/2018).

[54] G. Kim, P. Debois, J. Willis, and J. Humble. *The DevOps Handbook: How to Create World-Class Agility, Reliability, and Security in Technology Organizations*. IT Revolution Press, 2016. ISBN: 1942788002, 9781942788003.

[55] *Amazon Deploys To Production Every 11.6 Seconds*. Dec. 2013. URL: http://joshuaseiden.com/blog/2013/12/amazon-deploys-to-production-every-11-6-seconds/ (visited on 05/15/2018).

[56] M. Hüttermann. *DevOps for Developers*. 1st. Berkely, CA, USA: Apress, 2012. ISBN: 1430245697, 9781430245698.

[57] L. Bass, I. Weber, and L. Zhu. *DevOps: A Software Architect's Perspective*. SEI Series in Software Engineering. Pearson Education, 2015. ISBN: 9780134049878. URL: https://books.google.fi/books?id=fcwkCQAAQBAJ.

[58] *2015 State of the DevOps Report*. Puppet Labs. URL: https://www.devops.ch/wp-content/uploads/2016/12/AST-0147237_2015-state-of-devops-report.pdf (visited on 05/16/2018).

[59] F. Petruzella. *Programmable Logic Controllers*. McGraw-Hill Education; 2016, p. 432. ISBN: 978-0-07-337384-3.

[60] W. Bolton. *Programmable Logic Controllers, Fourth Edition*. Newnes, 2006, p. 416. ISBN: 0-7506-8112-8.

[61] S. Mackay, E. Wright, D. Reynders, and J. Park. *Practical Industrial Data Networks*. Elsevier Science & Technology, 2004, p. 448. ISBN: 075065807X.

[62] *CIP on CAN Technology*. Tech. rep. PUB00026R4. ODVA, Mar. 2016, p. 7. URL: https://www.odva.org/Portals/0/Library/Publications_Numbered/PUB00026R4_Tech-Adv-Series-DeviceNet.pdf.

[63] *Common Industrial Protocol (CIP) And The Family Of CIP Networks*. Tech. rep. PUB00123R1. ODVA, Feb. 2016, p. 133. URL: https://www.odva.org/Portals/0/Library/Publications_Numbered/PUB00123R1_Common-Industrial_Protocol_and_Family_of_CIP_Networks.pdf.

[64] *TwinCAT System - Overview*. Beckhoff Automation GmbH & Co. KG. URL: https://infosys.beckhoff.com/content/1033/tcsystem/html/tcsystem_intro.htm?id=7233122095836903011 (visited on 06/16/2017).

[65] *TwinCAT Overview*. Beckhoff Automation GmbH & Co. KG. URL: https://infosys.beckhoff.com/content/1033/tcoverview/html/default.htm?id=4049411600652121908 (visited on 06/16/2017).

[66] *ADS Communication*. Beckhoff Automation GmbH & Co. KG. URL: https://infosys.beckhoff.com/english.php?content=../content/1033/bc9000/html/bt_ethernet%20ads%20potocols.htm&id=2350280922589392761 (visited on 12/06/2017).

[67] *TwinCAT ADS/AMS-Specification - Overview*. Beckhoff Automation GmbH & Co. KG. URL: https://infosys.beckhoff.com/content/1033/tcadsamsspec/html/tcadsamsspec_intro.htm?id=1197268624309806635 (visited on 06/25/2017).

[68] *ADS introduction*. Beckhoff Automation GmbH & Co. KG. URL: https://infosys.beckhoff.com/content/1033/tcadscommon/html/tcadscommon_intro.htm?id=898081192215463875 (visited on 06/25/2017).

[69] A. Airto. *TWINCAT ADS TO PLC - Getting started step-by-step*. 1.0. Beckhoff Automation Oy. PL 23 Kankurinkatu 4-6 05801 Hyvinkää, Apr. 2014, p. 24.

[70] *Beckhoff / ADS*. URL: https://github.com/Beckhoff/ADS (visited on 06/25/2017).

[71] *Command Overview*. Beckhoff Automation GmbH & Co. KG. URL: https://infosys.beckhoff.com/english.php?content=../content/1033/tc3_adscommon/html/tcadsamsspec_adscmds.htm&id=2307577238379460642 (visited on 03/12/2018).

[72] *stackoverflow.com Traffic Statistics*. Alexa. URL: https://www.alexa.com/siteinfo/stackoverflow.com (visited on 03/30/2018).

[73] P. Hethmon, H. Brothers, and R. McMurray. *RFC 7151 - File Transfer Protocol HOST Command for Virtual Hosts*. RFC. RFC Editor, Sept. 2014, p. 24. URL: https://www.rfc-editor.org/rfc/rfc7151.txt.

[74] J. Goerzen, T. Bower, and B. Rhodes. *Foundations of Python Network Programming*. Apress, 2011, p. 368. ISBN: 978-1-4302-3004-5.

[75] J. Elofsson and A. Leopold. *High Speed Ethernet Client manual*. 1.0. Yaskawa Motoman. Oct. 2013, p. 29.

[76] *DX200 Options Instructions For High-speed Ethernet Server Function*. Manual No. HW1481977, Part Number: 165304-1CD, Revision: 2. Yaskawa Motoman, p. 111. URL: `https://www.motoman.com/hubfs/downloads/documentation/165304-1CD.pdf`.

[77] R. Blewett and A. Clymer. *Pro Asynchronous Programming with .NET*. Apress, 2013, p. 352. ISBN: 978-1430259206.

[78] S. Cleary. *Concurrency in C# Cookbook*. O'Reilly UK Ltd., 2014, p. 190. ISBN: 978-1449367565.

[79] *Asynchronous IO and Boost::ASIO*. URL: `https://honghongcs.wordpress.com/2011/12/30/asynchronous-io-and-boostasio-1/` (visited on 09/07/2017).

[80] M. Fowler. *Patterns of Enterprise Application Architecture*. Addison Wesley, 2002, p. 560. ISBN: 0321127420.

[81] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Prentice Hall, 1995, p. 395. ISBN: 978-0201633610.

[82] J. W. Grenning. *Test Driven Development for Embedded C*. O'Reilly UK Ltd., 2011, p. 352. ISBN: 1-934356-62-X.

[83] *Task-based Asynchronous Pattern (TAP)*. Microsoft Corporation. URL: `https://docs.microsoft.com/en-us/dotnet/standard/asynchronous-programming-patterns/task-based-asynchronous-pattern-tap` (visited on 09/11/2017).

[84] N. Fenton and J. Bieman. *Software Metrics: A Rigorous and Practical Approach, Third Edition*. 3rd. Boca Raton, FL, USA: CRC Press, Inc., 2014. ISBN: 1439838224, 9781439838228.

[85] *DX200 - Operator's Manual*. Manual No. HC1400020, Part Number: 165832-1CD, Revision: 0. Yaskawa Motoman, p. 111. URL: `https://www.motoman.com/hubfs/downloads/documentation/165832-1CD.pdf`.

[86] *DX200 Options Instructions For Concurrent IO*. Manual No. RE-CKI-A465, Part Number: 165294-1CD, Revision: 2. Yaskawa Motoman, p. 359. URL:

https://www.motoman.com/hubfs/downloads/documentation/165294-1CD.pdf.

[87] D. Darvas, E. Blanco Vinuela, and I. Majzik. "A Formal Specification Method for PLC-based Applications". In: (2015), WEPGF091. 4 p. URL: http://cds.cern.ch/record/2213506.

[88] G. J. Myers, C. Sandler, and T. Badgett. *The Art of Software Testing*. Wiley John & Sons, 2011. 240 pp. ISBN: 1118031962.

[89] C. L. Bates and J. M. Santosuosso. *Code coverage with an integrated development environment*. US Patent 7,089,535. Aug. 2006.

[90] K. Honda, H. Washizaki, Y. Fukazawa, M. Taga, A. Matsuzaki, and T. Suzuki. "Empirical Study on Recognition of Project Situations by Monitoring Application Results of Software Reliability Growth Model". In: *2017 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*. Oct. 2017, pp. 169–174. DOI: 10.1109/ISSREW.2017.53.

[91] V. Kumar, V. B. Singh, A. Garg, and G. Kumar. "Selection of Optimal Software Reliability Growth Models: A Fuzzy DEA Ranking Approach". In: *Quality, IT and Business Operations: Modeling and Optimization*. Ed. by P. Kapur, U. Kumar, and A. K. Verma. Singapore: Springer Singapore, 2018, pp. 347–357. ISBN: 978-981-10-5577-5. DOI: 10.1007/978-981-10-5577-5_28.

A. ROBOTIN PÄÄOHJELMA

```
1  NOP
2  'Clear the JOB stack
3  CLEAR STACK
4  'Signal PLC that a 'job has started'
5  DOUT OT#(34) ON
6  'Execute the job
7  CALL JOB:CREATEPRGNAME
8  'Signal PLC that the 'job has finished'
9  DOUT OT#(36) ON
10 'Wait for PLC to acknowledge
11 WAIT IN#(35)=ON
12 'Reset 'job has finished'
13 DOUT OT#(36) OFF
14 'Reset 'job has started'
15 DOUT OT#(34) OFF
16 END
```

B. APUOHJELMA ROBOTIN UUDEN TYÖN LUOMISEEN

```
1  NOP
2  'Initialize variable to hold job's name
3  SET S000 ""
4
5  'Loop through memory locations 11-28
6  FOR LI003 START= 11 TO 28
7          'Get an 8 bit character from memory and store it to LB000
8          DIN LB000 IG#(LI003)
9          'Type conversion from byte to char
10         CHR$ LS000 LB000
11         'Append the new character to job's name
12         CAT$ S000 S000 LS000
13 NEXT LI003
14
15 'Execute the job that name was just read from memory
16 CALL S000
17 END
```