Tampereen yliopisto

Niko Rinko

AUTOMAATIOJÄRJESTELMÄN OHJELMISTOVIKOJEN KORJAUS

Kandidaatintyö
Tekniikan ja luonnontieteiden tiedekunta
Tarkastaja: Mikko Salmenperä
Maaliskuu 2021

TIIVISTELMÄ

Niko Rinko: Automaatiojärjestelmän ohjelmistovikojen korjaus
Kandidaatintyö
Tampereen yliopisto
Teknisten tieteiden kandidaattiohjelma
Maaliskuu 2021

Automaatiojärjestelmien suunnittelun ja päivitysten yhteydessä tapahtuu inhimillisiä virheitä, jotka johtavat ohjelmistovikoihin. Useimmat ovat kohdanneet ohjelmistovikoja IT-sovelluksia käyttäessään. Ohjelmistotekniikassa ohjelmistovikojen alkuperää ja korjausta on tutkittu paljon, mutta automaatiojärjestelmien kohdalla aiheesta kertovaa kirjallisuutta on vähemmän. Tässä työssä tutkitaan ohjelmistotekniikan viankorjaustapojen soveltuvuutta automaatioon. Työn teoriaosassa esitellään lyhyesti taustatietoja automaatio-ohjelmistojen erityispiirteistä, käytetyistä ohjelmointikielistä ja automaatiojärjestelmien yleisimpiä rakennehierarkioita. Lisäksi teoriaosuudessa tutustumaan hieman vikojen korjaamisesta kertovaan ohjelmistotekniikan kirjallisuuteen ja arvioimaan näissä esiteltyjä tuloksia ja käytäntöjä automaation näkökulmasta, sekä esitellään Valmet DNA-ohjausjärjestelmä yleisesti, sekä työssä käytetyt Valmet DNA:n suunnittelijatyökalut. Työn käytännön osana korjataan WinNovan opetuskäyttöön suunnitellun vesiprosessin automaatiojärjestelmän ohjelmistoviat ja arvioidaan korjauksen aikana käytettyjä tapoja, sekä verrataan niitä teoriaosuudessa esiteltyihin käytäntöihin.

Pohjimmiltaan viankorjauksen vaiheet, paikallistaminen, korjaus, testaus, ovat samat automaatiossa. Kuitenkin kehittyneemmät ja automatisoidut tavat vikojen korjaukseen olivat suunniteltu tekstipohjaisille ohjelmointikielille, jotka ovat hyvin yleisiä IT-ohjelmistokehityksessä. Näiden soveltaminen automaatiossa käytettäville graafisille ohjelmointikielille on vähintäänkin haastavaa, ellei jopa mahdotonta. Abstrakteja toteutuksesta riippumattomia viankorjaustapoja voi hyödyntää suoraan automaatiossa, mutta näistä saatu hyöty voi jäädä vähäiseksi.

WinNovan vesiprosessin ohjelmistovikojen korjaus onnistui hyvin, alkuvaikeuksista huolimatta. Toimimattomasta järjestelmästä saatiin jälleen toimintakykyinen, eikä työn lopussa enää havaittu toimintaa häiritseviä vikoja. Lisäksi vesiprosessin ohjelmiin lisättiin toimintoja ja toteutettiin parannuksia olemassa oleviin toimintoihin WinNovan toiveiden mukaisesti. Käytetyt tavat ja käytännöt vikojen korjaukseen olivat paikannuksen ja korjauksen osalta vastaavia, kuin kirjallisuudessa suositellut käytännöt. Testauksen osalta käytännöt eivät olleet optimaalisia, mutta työssä päästiin kuitenkin tavoiteltuun lopputulokseen.

Avainsanat: Valmet DNA, ohjelmistovika, automaatiojärjestelmä, vian korjaus

.

ALKUSANAT

Haluan kiittää vanhempiani tuesta ja erityisesti isääni avusta aiheen valinnassa, sekä käytännön työn toteutuksessa. Käytännön työssä ei olisi päästy yhtä hyvään lopputulokseen, jos työparinani olisi ollut joku muu.

Tampereella, 9.3.2021

Niko Rinko

SISÄLLYSLUETTELO

LYHENTEET JA MERKINNÄT

ana	Analoginen signaali Valmet DNA:n FBCAD-ohjelmassa
bin	Binäärinen signaali Valmet DNA:n FBCAD-ohjelmassa
BU	engl. Backup, varmuuskopio
CAD	engl. Computer-aided Design, tietokoneavusteinen suunnittelu
DCS	engl. Distributed Control System, hajautettu ohjausjärjestelmä
FBCAD	Valmet DNA:n suunnitteluohjelma, engl. Function Block Computer-aided design
FBD	engl. Function Block Diagram, funktiolohkokaavio ja ohjelmointikieli
GCC	engl. GNU Compiler Collection, kääntäjäkokoelma ohjelmointikielille
IEC	International Electrotechnical Commission, kansainvälinen sähköalan standardointiorganisaatio
IEEE	Institute of Electrical and Electronics Engineers, kansainvälinen tekniikan alan järjestö
IL	engl. Instruction List, ohjelmointikieli
int	Integer, kokonaisluku. Käytössä Valmet DNA:n FBCAD:issa
IO	engl. Input/Output, tiedonsiirto tietokoneiden laitteiston välillä
IT	engl. Information Technology, tietotekniikka
LD	engl. Ladder Diagram, tikapuulogiikka, ohjelmointikieli
NASA	engl. National Aeronautics and Space Administration, Yhdysvaltojen ilmailu- ja avaruushallintovirasto
PC	engl. Personal Computer, henkilökohtainen tietokone
PID	engl. Proportional-Integral-Derivative, säädintyyppi
PLC	engl. Programmable Logic Controller, ohjelmoitava logiikka
RAM	engl. Random Access Memory, tietokoneen keskusmuisti
RTSJ	engl. Real-Time Specification for Java, Java-ohjelmointikielen laajennus
SCADA	engl. Supervisory Control And Data Acquisition, valvomo-ohjelmistotyyppi automaatiossa
SFC	engl. Sequential function chart, ohjelmointikieli
ST	engl. Structured text, ohjelmointikieli
UML	engl. Unified Modeling Language, mallinnuskieli
Valmet DNA	engl. Valmet Dynamic Network of Application, automaatiojärjestelmä
WYSIWYG	engl. What You See Is What You Get, “mitä näet sitä saat”, Ohjelma, jossa ruudulla muokattaessa näkyvä sisältö vastaa lopputulosta

A	ampeeri.

1. JOHDANTO

Automaatiojärjestelmien ohjelmoinnissa näkee maailmalla paljon vikoja ja tehtaiden valvomomiehiltä kuuleekin usein sanat ”Me ajamme tätä käsiajolla, koska automaattiohjaus ei toimi”. Lisäksi järjestelmän koodissa on usein jo vuosia sitten käytöstä poistettuja laitteita edelleen ajossa. Eräs esimerkkitapaus löytyy Porista, Satakunnan ammattikoulu WinNovan prosessinhoitajien koulutukseen tarkoitetusta vesiprosessista. WinNovan vesiprosessi on siirretty uusiin tiloihin ja samalla päivitetty vanhasta Damatic XD-automaatiojärjestelmästä tuoreempaan Valmet DNA-automaatiojärjestelmään vuonna 2014. Päivityksen ja siirron myötä kuitenkin järjestelmä lakkasi toimimasta. Opetuskäyttöön tarkoitettua järjestelmää on jouduttu viimeiset kuusi vuotta ajamaan käsiajolla, koska automaattiohjaus ei ole toiminut. Tämän työn tutkimusongelmana on soveltaa ohjelmistotekniikassa käytössä olevia vian paikannus- ja -korjaustapoja automaatioympäristöön. Soveltavana käytännön osana työssä korjataan WinNovan vesiprosessin ohjelmistoviat.

Keskeisenä ongelmana automaatiojärjestelmän vikojen korjauksessa on vian paikallistaminen. Usein vian varsinaisen ratkominen tai vian poistavan muutoksen tekeminen on triviaalia verrattuna vian paikallistamiseen. Järjestelmän dokumentaatiosta on korvaamaton apu korjaustyössä, mutta dokumentaation ajantasaisuus ja laatu on otettava huomioon. Tässä työssä keskitytään automaatiojärjestelmän ohjelmistollisiin vikoihin. Työn toteutuksessa fyysinen laitteisto on kuitenkin niin vahvasti läsnä, että sitä ei voida täysin sivuuttaa.

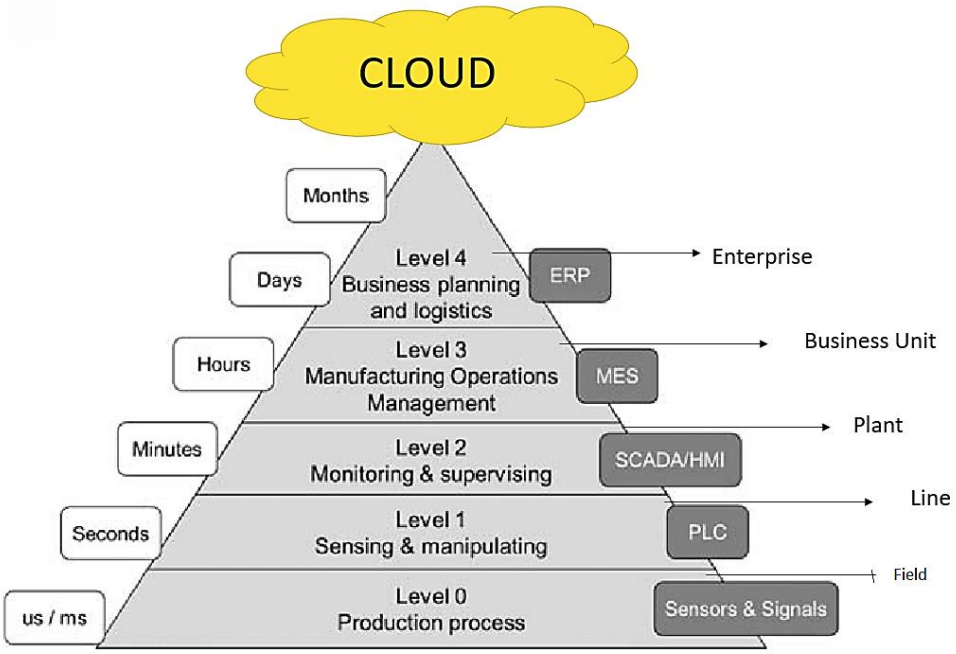
Toisessa luvussa käsitellään automaatio-ohjelmistoissa käytettäviä ohjelmointikieliä sekä järjestelmien karkeaa rakennetta. Luku pyrkii selvittämään millä kielillä automaatio-ohjelmat tyypillisesti kirjoitetaan. Kolmannessa luvussa tutkitaan ohjelmistokehityksen viankorjauskäytäntöjen soveltamista automaatiossa ja esitellään hyviä käytäntöjä dokumentaation päivitykseen, vikojen etsintään, sekä testaukseen. Neljäs luku koostuu Valmet DNA:sta, sen yleisrakenteesta ja tärkeimmistä työkaluista työn kannalta. Valmet DNA:n rooli on mittava tässä työssä, sillä WinNovan vesiprosessi on toteutettu käyttäen Valmet DNA:ta. Viidennessä luvussa käsitellään WinNovan vesiprosessin korjaustyön suorittamista ja kuudennessa luvussa kootaan lopputulokset ja arvioidaan työn onnistumista.

2. AUTOMAATIO-OHJELMISTOT

Termillä automaatio-ohjelmisto voidaan puhua käyttäjän näkökulmasta valvomokäyttöliittymästä tai automaatiosuunnittelijan näkökulmasta prosessiasemien tai ohjelmoitavien logiikkaohjainten (Programmable Logic Controller, PLC) suorittamista ohjelmista. Näillä eri ohjelmistotasoilla on merkittävä ero, ja tämän työn kannalta on merkittävää ymmärtää, millä tasoilla automaatiojärjestelmän ohjelmisto toimii.

2.1 Automaatiojärjestelmien rakenne ja tasot

Automaatiojärjestelmien karkeaa yleisrakennetta kuvataan usein tasopyramidilla. Kuvassa 1 on yksi verkosta poimittu esimerkki tasopyramidista.



Kuva 1 Automaation tasot [1]

Kuten kuvan 1 vasempaan reunaan on merkitty, tasoilla tapahtuviin päätöksiin on aikarajoja. Nämä rajat tietysti vaihtelevat sovelluskohteen mukaan, mutta perusidealtaan datan keruun ja toimilaitteiden ohjauksen on oltava erittäin nopeaa verrattuna tehtaan johdon tekemiin suunnitelmiin. Mitä alemmalla tasolla toimitaan, sitä nopeampaa toiminnan tulee olla. Alemmilla tasoilla tyypillisesti päätökset eivät ole kovin monimutkaisia ja dataa käytetään yksinkertaisesti lukemalla yksittäisiä arvoja. Ylemmillä tasoilla päätökset ovat monimutkaisempia ja dataa analysoidaan monimutkaisemmilla metodeilla. Pyramidin

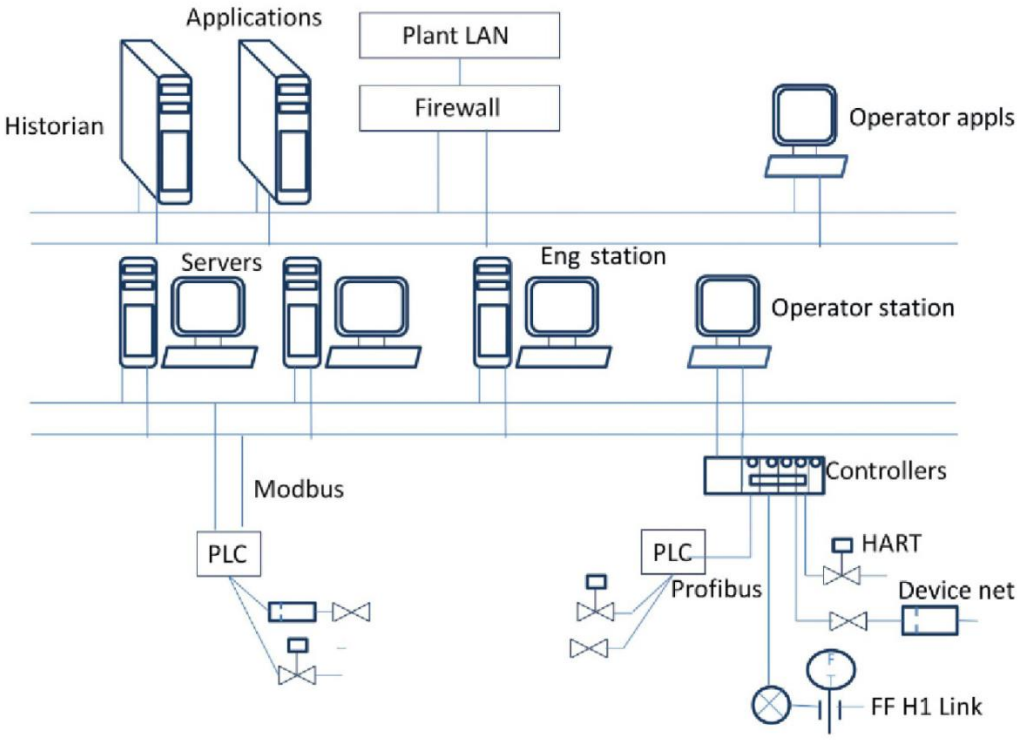
perusideana on datan kulkeminen pohjalta huipulle matkalla jalostuen ja päätösten kulkeutuminen huipulta pohjalle matkalla tarkentuen.

Kuvan 1 oikeassa reunassa on merkitty kullakin tasolla toimivat järjestelmän osat ja laajuusalue tehtaassa. Esimerkiksi tasolla 1 ajetaan ohjelmia, jotka säätävät ja ohjaavat toimilaitteita. Kuten kuvassa lukee, PLC on yksi esimerkki tämän tason toimijasta. Tämä työ keskittyy PLC:n tai muun automaatio-ohjaimen ajaman koodin vikoihin, joten tässä työssä keskitytään lähinnä tasoon 1. Ohjelmistot ovat toki käytössä tasosta 1 eteenpäin, mutta jo tason 2 jälkeen ohjelmat eivät enää vaadi automaation erityispiirteitä, vaan tavanomaisen PC-ohjelmiston ominaisuudet riittävät mainiosti.

Automaatiojärjestelmistä puhuttaessa usein käytetään termejä DCS (Distributed Control System), SCADA (Supervisory Control and Data Acquisition) ja PLC. Nämä ovat keskeisiä käsitteitä automaatiojärjestelmistä puhuttaessa ja myös tämän työn kannalta oleellisia.

2.1.1 DCS

DCS-järjestelmien rakenteessa nimensä mukaisesti hajautetaan ohjauslaitteet kentälle. Yksittäiset ohjaimet ovat kentällä sijaitsevat ohjaimet ja prosessiasemat ovat yhteydessä toisiinsa, valvomoihin ja palvelimiin kenttäväylien ja Ethernetin välityksellä. Kuva 2 havainnollistaa yksinkertaisen DCS-järjestelmän rakennetta.



Kuva 2 DCS-rakenne [2 s.82] (Lähteestä korjattu kirjoitusvirhe Engg->Eng)

DCS-järjestelmät ovat tyypillisesti helposti laajennettavissa ja kahdennettuja. Kahdennettu verkko lisää järjestelmän luotettavuutta. Verkkolaitteen vikaantuessa on todennäköistä, että toinen verkkolaite jatkaa toimintaansa ja voi korvata vikaantuneen välittömästi. DCS-järjestelmissä kentällä olevat ohjaimet ohjaavat oman vastuualueensa toimilaitteita ajamalla kyseisten piirien ohjelmia. Mehtan ja Reddyn [2 s. 75–88] kirjassa esitellään DCS-rakenteen etuja ja ominaisuuksia. Vaikka DCS-järjestelmät ovat alun perin suunniteltu jatkuva-aikaisille analogisille ohjauksille ja PLC:t digitaalisille diskreeteille ohjauksille, nykyisin DCS-järjestelmät usein sisältävät myös PLC-yksiköitä. Rakenne mahdollistaa myös ”automaation saarekkeet”, joka tarkoittaa sitä, että yhteyksien katketessa useat ohjaimet kykenevät jatkamaan toimintaansa. [2 s. 75–88]

2.1.2 PLC

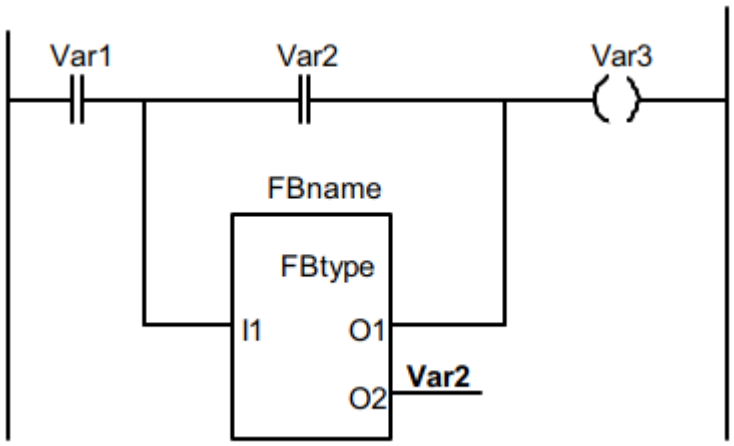
PLC-järjestelmissä ajettava ohjelmakoodi suunnitellaan tarkoitukseen soveltuvalla ohjelmointiympäristöllä ja ladataan sen jälkeen suoraan PLC:n RAM-muistipiirille [2 s.37–42][3]. Tämä on perusidealtaan vastaava, kuin DCS:n ohjainten toiminta. Ohjaimissa on kuitenkin eräs toiminnallinen ero. Alan kirjallisuudessa [2 s. 37–42][3] usein kerrotaan, että PLC:n ohjelmat ajetaan sykleissä niin, että kaikki IO-tulot luetaan ensin muistiin ja vasta sen jälkeen ajetaan ohjelma. Lopuksi kaikki ohjelman laskemat lähdöt luetaan IO-lähtöihin. [2 s. 37–42][3] Tämä eroaa DCS-järjestelmien tyypillisestä toiminnasta, jossa tuloja luetaan toisiin säätöpiireihin samalla, kun ohjelmaa suoritetaan toisessa.

2.2 Automaatiossa yleisesti käytössä olevat ohjelmointikielet

PLC:stä on olemassa standardi IEC 61131. IEC:n verkkokaupasta [5] nähdään standardin olevan jaettu kymmeneen osaan. Osa 3, eli IEC 61131-3 sisältää PLC:n standardoidut ohjelmointikielet. [5] Standardissa [4] määritellään viisi ohjelmointikieltä PLC:lle: Kolme graafista ohjelmointikieltä ja kaksi tekstipohjaista. John ja Tiegelkamp toteavat [6 s.12] suurimman osan PLC valmistajista noudattavan IEC 61131-3 standardia. On hyvä kuitenkin muistaa, että standardi kattaa vain PLC-ohjaimet ja muitakin on olemassa. Esimerkiksi teollisuus-PC:t voivat ajaa tavallisemmilla ohjelmointikielillä kirjoitettuja ohjelmia. Näitä tavallisempia ohjelmointikieliä ovat esimerkiksi C++, C# ja Java.

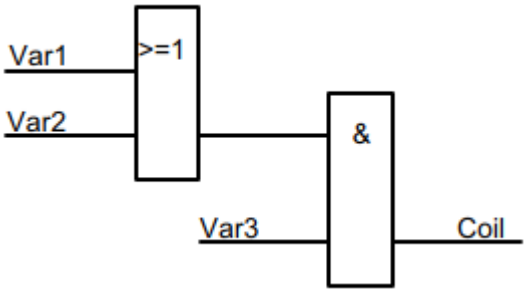
2.2.1 IEC 61131-3:n määrittelemät kielet

Standardin ehkä yleisimmin käytössä oleva kieli on ns. tikapuulogiikka (ladder diagram, LD). LD:ssä ohjelmat muodostetaan graafisesti vaakaviivoille. Jokaiselta vaakaviivalta saadaan yksi Boolen totuusarvo. Vaakaviivoilla alku- ja loppupisteen välillä voi olla useita ehtoja. Vaakaviivat ovat yhdistettynä molemmista päistä pystyviivoihin, jotka yhdessä muodostavat tikapuita muistuttavan kokonaisuuden, josta ohjelmointikielen nimi on peräisin. Ohjelmat suoritetaan ylhäältä alas, vaakaviiva kerrallaan. Vaakaviivan ehdot luetaan vasemmalta oikealle ja vaakaviivan päätyttyä siirrytään seuraavaan alla olevaan vaakaviivaan. [4][6] Kuvassa 3 on esimerkki LD:n ohjelmasyntaksista. Ohjelmoinnissa on käytetty apuna funktiolohkoa.



Kuva 3 Esimerkki LD:n syntaksista [6 s. 158]

Toinen standardin graafinen ohjelmointikieli on funktiolohkokaavio (Function Block Diagram, FBD). FBD on yleisesti DCS-järjestelmissä käytössä.[4] Tässä työssä korjattava automaatiojärjestelmä käyttää funktiolohkoihin perustuvaa ohjelmointikieltä. FBD:ssä perusideana on kapseloida osia ohjelmakoodista funktiolohkoiksi, joita voi tallentaa ja käyttää useissa eri projekteissa [4]. Esimerkiksi usein käytetty funktiolohko voisi olla PID-säätimen toteuttava lohko. Funktiolohkot yhdistetään toisiinsa viivoin, jotka alkavat lohkojen lähdöistä ja päättyvät tuloihin. Funktiolohkot ovat vähemmän organisoituja kuin LD:n vaakaviivat, mutta perussääntönä vasemmalta luetaan tulot ja oikealla lähetetään lähdöt. Välissä olevat lohkot voivat olla sekalaisessa järjestyksessä, kunhan viivat ovat kytkettynä oikeisiin tuloihin ja lähtöihin. Kuvassa 4 on yksinkertaistettu esimerkki FBD:n piirrostavasta.



Kuva 4 Esimerkki FBD:n syntaksista [6 s. 254]

Standardi määrittelee vielä kolmannen graafisen ohjelmointikielen, jonka nimi on Sequential Function Chart (SFC) [4]. Käännettynä suomen kielelle kuvaavin termi lienee sekvenssikaavio, mutta tätä ei pidä sekoittaa Unified Modeling Language:n (UML) sekvenssikaavioihin. Ramanathan [4] kirjoittaa, ettei SFC varsinaisesti ole ohjelmointikieli siinä missä muut IEC 61131-3:n määrittelemät kielet ovat. SFC:n perusideana on määritellä ohjelma tiloihin, joihin siirtymiseen edellytetään ehtojen täyttymistä. Tiloissa ollessa ajetaan tilalle määriteltyjä toimintoja, jotka määritellään muilla ohjelmointikielillä. [4] SFC:llä on kuitenkin kätevää laatia ohjelman yleisempi rakenne ja viitata muilla kielillä kirjoitettuihin yksityiskohtaisempiin toimintoihin oikeissa kohdissa.

Standardin tekstipohjaisista kielistä toinen on vanhentunut standardin kolmannesta versiosta lähtien [7]. Tämä kieli on Instruction List (IL), joka Ramanathanin [4] mukaan muistuttaa matalan tason ohjelmointikieltä Assembly. Kielen määrittelemät käskyt itsessään ovat varsin yksinkertaisia ja monimutkaisten järjestelmien ohjelmointi on siten haastavaa IL:llä. Sen etuina ovat kuitenkin nopeus ja alhainen muistin käyttö. IL:ää voi nähdä yhä käytössä monissa vanhemmissa PLC järjestelmissä erityisesti Euroopassa, jossa sen käyttö on ollut suosittua. [4]

Toinen tekstipohjainen ohjelmointikieli on Structured Text (ST), eli rakenteellinen teksti. Toisin kuin IL, ST on korkeamman tason ohjelmointikieli. Kieli muistuttaa C-kieltä, josta nykyisin laajassa käytössä oleva C++ on kehitetty laajentamalla. ST tukee monia PC-ohjelmoinnista tuttuja käsitteitä, kuten muuttujille arvojen tallentaminen, funktiot, silmukat ja ehtolauseet. ST:tä käytetään usein graafisten ohjelmointikielten rinnalla. [4]

2.2.2 Muita yleisiä kieliä

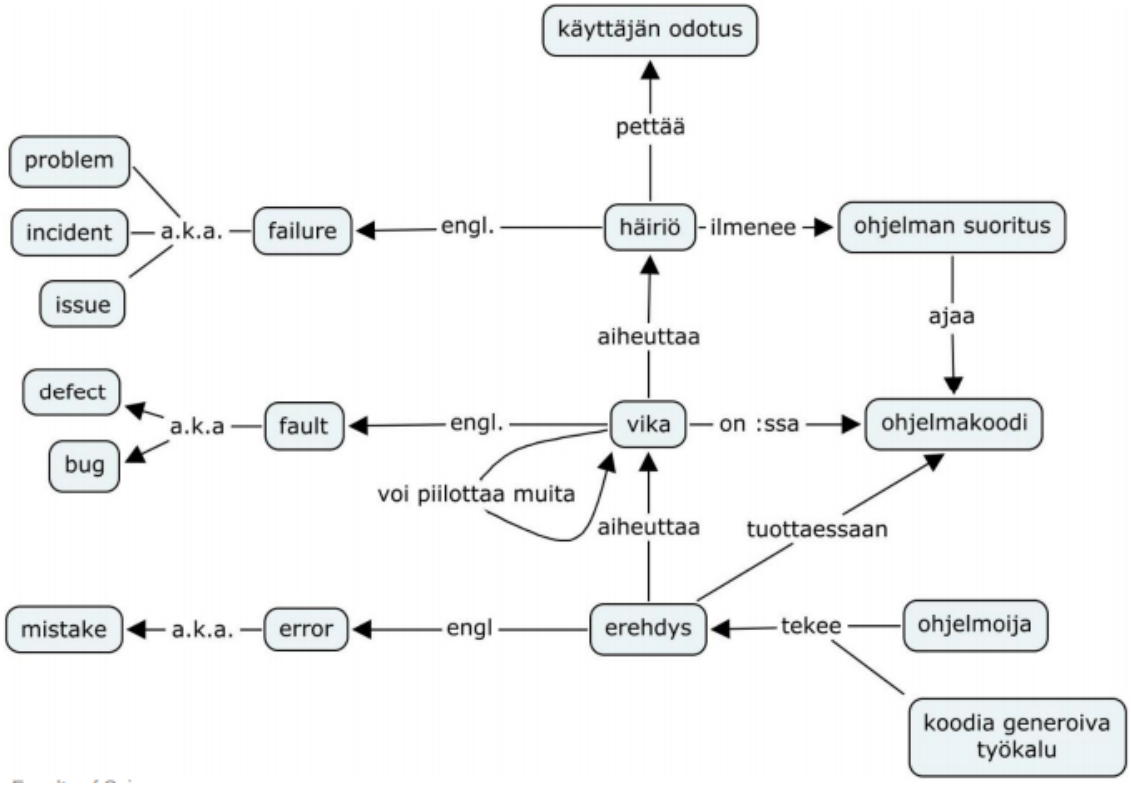
Kuten luvussa 2.2.1 mainittiin, FBD on yleisesti käytössä DCS-järjestelmissä. DCS-järjestelmille on kehitetty oma standardinsa IEC 61499, joka pohjautuu aiemmin mainitun IEC 61131-3:n FBD-kieleen ja on luotu sen laajennukseksi [8]. Thramboudilis [8] esittää nämä laajennukset parannuksina ja mainitsee standardin 61499 yhdistävän olio-ohjelmoinnin peruskäsitteitä aiempaan FBD-kieleen.

Automaatio-ohjelmistoja kirjoitetaan myös IT-ohjelmistoissa yleisesti käytössä olevilla kielillä, kuten Java, C++, C# ja Python. Usein tällöin kuitenkin kohdejärjestelmän vaatimukset ovat pienemmät. IT-ohjelmistojen kehitykseen tarkoitetut kielet eivät lähtökohtaisesti ole automaation kannalta reaaliaikaisia, mutta reaaliaikaisuus on toteutettavissa laajentamalla kieliä. Esimerkiksi Javalle on kehitetty laajennus RTSJ (Real-Time Specification for Java), joka mahdollistaa reaaliaikaisten ohjelmien kirjoittamisen Javalla [9].

3. OHJELMISTOSUUNNITTELUN KÄYTÄNNÖT AUTOMAATIOJÄRJESTELMÄN VIKOJEN KORJAAMISEEN

3.1 Vian määritelmä ja yleisimmät vikatyypit

Ohjelmistotekniikassa käytetään kolmea keskeistä termiä kuvaamaan väärin toimivaa ohjelmakoodia. Nämä termit ovat virhe (error), vika (fault) ja häiriö (failure). Helsingin Yliopiston luentodioissa [10] on näitä termejä hyvin havainnollistava kuva 5.



Kuva 5 Ohjelmistovikoihin liittyviä termejä [10]

IEEE:n standardissa 1044 vuodelta 2009 [11] määritellään termit error, fault ja failure. Termien määrittelyt ovat standardissa englanniksi ja pyrin kääntämään ne mahdollisimman tarkasti. Vika määritellään olevan virheen ilmentymä ohjelmassa. Virhe puolestaan on ihmisen toiminta, joka tuottaa virheellisen tuloksen. Häiriö määritellään tapahtumaksi, jossa järjestelmä tai sen osa ei kykene suorittamaan toimintoaan vaatimusten mukaisesti. [11] Häiriö on siis ulkoisesti havaittava poikkeama ohjelman toiminnassa, joka johtuu viasta. Standardissa lisäksi määritellään termit puute (defect) ja ongelma (problem). Puutteen määritelmä on tuotteen epätäydellisyys tai puutteellisuus (deficiency), jossa

tuote ei vastaa vaatimuksiaan ja se pitää joko korjata tai korvata. Ongelma määritellään negatiiviseksi tilanteeksi, josta pitää päästä yli. [11]

Hamill ja Goseva-Popstojanova [12] käyttävät samoja termejä, kuvaten niitä hieman eri sanoin. Hyvänä lisäyksenä IEEE:n standardin [11] määritelmiin Hamill ja Goseva-Popstojanova huomauttavat [12], ettei kaikista vioista seuraa häiriötä ja järjestelmissä on usein piileviä vikoja, jotka eivät esiinny häiriöinä. Vioilla voi olla ehtoja häiriön aiheutumiselle, jotka eivät koskaan käytännössä toteudu. Lisäksi Hamill ja Goseva-Popstojanova huomauttavat [12] kaikkien vikojen olevan puutteita, mutta kaikki puutteet eivät ole vikoja. Automaatiojärjestelmien kohdalla vioista puhuttaessa on käsiteltävä asiaa laajemmin. Automaatiossa viat voivat liittyä karkeasti jakaen ohjelmistoon, kenttälaitteiden vaurioihin tai kytkentöihin. Perusidealtaan termit ovat samoja kuin ohjelmistotekniikassa, mutta häiriön lähteenä oleva vika voi sijaita paljon laajemmalla alueella.

Tutkimuksessaan [12] Hamill ja Goseva-Popstojanova tutkivat tilastollisesti vikojen sijaintia ja jakaumaa eri ohjelmatiedostoihin, sekä tyypillisiä vikojen lähteitä. Tutkimuksen [12] tuloksena selvisi vikojen olevan hyvin paikallisia. Tarkoittaen, ettei vikaa korjatessa muutoksia joutunut tekemään kuin yhteen tai kahteen tiedostoon. Jopa yli 50 % vioista edellytti muutoksia vain yhteen tai kahteen tiedostoon. Lisäksi 80 % vioista johtui pienestä 20 % osasta tiedostoja. Häiriöihin johtavista vioista kolmannes oli peräisin vaatimusmäärittelystä, toinen kolmannes ohjelmakoodista ja 14 % dataongelmista. Nämä tulokset perustuivat NASA:n avaruuslentojen ohjelmistoihin ja yleisesti käytössä olevan C++ kääntäjän GNU Compiler Collection:in (GCC) vikatilastoihin. [12] Nämä tilastot eivät siis välttämättä vastaa automaatio-ohjelmistojen vikojen paikallisuutta ja syitä. Omien ohjelmointikokemusten perusteella ohjelmakoodin virheet johtuvat usein ehtolauseista. Esimerkiksi for-silmukoiden ja if-lauseiden ehdot menevät helposti väärin. Näitä kokemuksia tukee tutkimus [13] IT-ohjelmistojen viankorjauksista, jossa selvitettiin vikojen tyypin toistuvuutta. Pan et al. kirjoittavat yhteenvetona vikojen olevan yllättävän samankaltaisia projektista riippumatta ja noin puolet vioista oli kategorioitavissa mallien mukaisesti. Noin kolmannes vioista liittyi if-ehtolauseisiin ja toinen kolmannes metodien parametreihin. [13 p.312] Vaikka tutkimukset ovatkin tehty perinteisen IT-ohjelmistojen pohjalta, ihmisten ohjelmointitaidot ja ajattelumallit ovat kuitenkin pohjimmiltaan samat. Virheitä todennäköisesti syntyy automaatio-ohjelmistojen kehityksessä samankaltaisissa kohdissa.

Automaatioseuran kirjassa [14, sivut 6–7] kerrotaan automaatio-ohjelmistojen rakentuvan uuden ohjelmakoodin sijaan pitkälti valmiista lohkoista ja osakokonaisuuksista. Näiden yhteensopivuus voi olla kuitenkin määrittelemätöntä ja johtaa ohjelmistovirheisiin.

Ohjelmistot eivät kulu samalla tavalla kuin mekaaniset laitteet, vaan ohjelmistoviat johtuvat aina ohjelmistoon jääneistä virheistä. Nämä virheet puolestaan voivat johtua asiakasvaatimusten väärinymmärryksestä tai puutteellisista spesifikaatioista tai ohjelmointivirheestä. Viat olisi hyvä havaita projektin mahdollisimman varhaisessa vaiheessa, sillä tällöin niiden korjaus on helpompaa. Jo käyttöönotetun järjestelmän vikojen korjaus on kallista ja haastavaa. Näiden vikojen havaitsemiseksi projektin aikana suoritetaan paljon ja kattavasti erilaisia testejä, kuten yksikkötestejä ja integraatiotestejä.

3.2 Vian paikallistaminen

Jotta ongelman voi korjata, on ensin paikallistettava juurisyy. Esimerkiksi jos valvomon näytöltä klikatessa venttiilin kuvaketta, venttiili ei avaudukaan, on kyseessä häiriö. Häiriö johtuu viasta, mutta vian sijainti ja luonne on tuntematon. Tässä kappaleessa pyritään selvittämään hyviä käytäntöjä vikojen paikallistamiseen. Vian paikallistamisesta, korjaamisesta ja testauksesta on kirjoitettu ohjelmistotekniikan näkökulmasta kirja [15], jossa Butcher esittelee hyväksi havaittuja käytäntöjä ohjelmistojen korjaukseen. Näitä voidaan perusperiaatteiltaan soveltaa automaatioon, mutta lisäksi on huomioitava automaatiojärjestelmille ominaiset erityispiirteet. Butcher korostaa [15, Luku 1] vikojen korjausprosessin ensimmäisen vaiheen tärkeyttä, eli vian juurisyyn selvittämistä. Tämä pätee erityisesti automaatiossa, jossa aluksi ohjelmistovialta vaikuttava vika voikin johtua esimerkiksi tietoliikennekytkennöistä. On siis ensiarvoisen tärkeää ensin jäljittää vian alkuperä.

Jotta vika voidaan paikantaa, täytyy vika ensin saada luotettavasti ilmentymään. Jollei vikaa saada luotettavasti toistettua, tulee vian korjaamisen todistaminen olemaan hyvin hankalaa. Testaus on pääasiallinen ja ensisijainen tapa saada vikoja ilmentymään, sekä paikallistaa ne. Testaus on kuitenkin laaja käsite ja pitää sisällään useita tapoja testata. Testausta tehdään koko projektin ajan ja myös käyttöönoton jälkeen päivitysten yhteydessä. Testausta voidaan jakaa eri tasoille, joita esitellään luvussa 3.2.1.

3.2.1 Testauksen tasot

Tampereen Yliopiston kurssilla ”Automaation reaaliaikajärjestelmät” pidettiin luento automaatiojärjestelmien testauksesta, jolla avattiin testaamiseen liittyviä termejä. Luennolla esiteltiin testauksen tasoja, joita ovat yksikkötestaus, integraatiotestaus, järjestelmätestaus ja hyväksymistestaus. [16 s.14] Näistä tasoista yksikkö- ja integraatiotestaus ovat lähinnä käytössä vain järjestelmän kehitysprojektin aikana. Järjestelmätestausta voidaan helpommin käyttää myös jo käytössä olevalle järjestelmälle, joka ei ole toiminut täysin oikein. Testauksen tasoilla on tarkoitus testata sovellus kattavammin keskittyen eri tasoilla eri kohteisiin. Näistä kohteista yleisimpiä esitellään luvussa 3.2.2.

Yksikkötestien tarkoituksena on puoliautomaattisesti testata ohjelman osa, joka on jaettavissa omaksi kokonaisuudekseen. Esimerkiksi olio-ohjelmoinnissa tällaisia osia ovat tyypillisesti luokat. Yksiköt eivät yleensä ole itsessään ajettavia kokonaisuuksia, joten yksikkötestauksessa tarvitaan tyypillisesti erilaisia tynkiä ja apuluokkia, jotka toteuttavat testattavan yksikön vaatiman rajapinnan. Yksikkötesteillä on helppoa testata funktioita, jotka suorittavat laskutoimituksia. Tällöin yksikkötestille määritellään oikea tulos, syöte ja laskutoimituksen tekevä funktio. Jos funktion antama arvo täsmää testin oikean tuloksen kanssa, funktio toimii kyseisellä syötteellä oikein. Yksikkötesteistä puhuttaessa useimmiten viitataan perinteisillä IT-maailman ohjelmointikielillä kirjoitettuihin ohjelmiin ja niiden testaukseen. Automaatio-ohjelmistoillekin voidaan kuitenkin tehdä yksikkötestejä. Jamro artikkelissaan [17] esittelee funktiolohkoille soveltuvan yksikkötestiajurin CPTest+:n, joka tukee kaikkia IEC 61131-3:n määrittelemiä ohjelmointikieliä. Näin ollen kyseinen testiympäristö soveltuu hyvin PLC-ohjelmien testaukseen. CPTest+ soveltuu useammalle testauksen tasolle: yksikkötestaukseen, integraatiotestaukseen ja järjestelmätestaukseen [17]. Automaatio-ohjelmistojen yksikkötestauksesta ei kuitenkaan ole yhtä helposti materiaalia saatavilla, kuin IT-ohjelmistojen yksikkötestauksesta. Näin ollen voidaan olettaa, ettei automaatio-ohjelmistojen yksikkötestaus, tai ainakaan testien automatisointi, ole yhtä yleistä IT-ohjelmistoihin verrattuna.

Integraatiotestauksessa keskitytään testaamaan yksikkötestauksessa testit läpäisseiden moduulien yhteistoimintaa. Toisin sanoen integraatiotestauksen tarkoituksena on testata moduulien rajapinnat, kun yksikköjen sisäinen toiminnallisuus on testattu. Integraatiotestaus suoritetaan yksikkötestauksen jälkeen, mutta ennen koko järjestelmän laajuisia testejä. Luennolla [16 s.20] esiteltiin kaksi päätapaa koota integraatiotestin kohde, Big bang -integrointi ja jatkuva integrointi. Big bang -integroinnissa kaikki osat yhdistetään kerralla ja testataan kerralla. Tällöin virheiden paikallistaminen on haastavampaa, mutta pienten ohjelmien kohdalla vielä realistisesti mahdollista. Jatkuvassa integroinnissa kokonaisuus kootaan yksikkö kerrallaan ajaen jokaisen liitoksen yhteydessä testit uudelleen. [16 s.20] Tällöin virheen sisältävä yksikkö on helpommin paikannettavissa, mutta testien ajoon kuluu merkittävästi enemmän aikaa. Automaatio-ohjelmistot koostuvat usein moduuleista, jotka lähtökohtaisesti soveltuvat hyvin integraatiotestaukseen.

Järjestelmätestauksessa testattavana kohteena on koko automaatiosovellus. Poiketen edellisistä tasoista, järjestelmätestauksessa ei hyödynnetä tietoa toteutuksesta [16 s.22]. Tällöin testit kohdistuvat vertailemaan ohjelman toimintaa määrittelydokumentaatioon. Tavoitteena on simuloida mahdollisimman kattavasti todellisia käyttötilanteita ja mahdollisia poikkeustilanteita. Järjestelmätestien kattava suunnittelu on kuitenkin vaikeaa ja kaikkea ei mitenkään voida testata [16 s. 22 ja 26].

Hyväksymistestauksen suorittaa tyypillisesti asiakas ja varmistaa ohjelmiston vastaavan vaatimuksia [16 s.14]. Hyväksymistestauksen kohdalla ohjelmistoviat tulisi olla jo korjattu, mutta välttämättä näin ei ole. Tämän tason testauksessa ei kuitenkaan enää aktiivisesti etsitä ohjelmointivirheitä, vaan lähinnä varmistetaan, että ohjelma täyttää kaikki asiakkaan vaatimukset. Näin ollen hyväksymistestaus ei enää ole tämän työn kannalta merkittävä.

3.2.2 Testitapausten valinta

Tässä luvussa esitellään joitain yleisimpiä kohteita, jotka tulisi testata. Testitapaukset ovat erilaisia eri tasoilla. Testausluennolla [16 s.17] esiteltiin parametrien testauksessa virhealttiita pisteitä, joita ovat erityisesti rajatapaukset. Jos funktio sallii vain positiivisia arvoja, mitä tapahtuu, jos funktiolle syöttää nollan? Rajatapausten välissä olevat arvot toimivat tyypillisesti samalla tavalla ja niiden testausta voidaan optimoida ekvivalenssiluokkien avulla [16 s.27]. Testaamalla ekvivalenssiluokka kerrallaan, testattavien syötteiden määrä pienenee ja on loogisempi. Lisäksi kannattaa kiinnittää huomiota erityisesti kaikkiin ohjelman kohtiin, jossa suoritus haaraantuu. Tällaisia ovat esimerkiksi ehtolauseet. Edellä mainitut testikohteet ovat lähinnä yksikkötestien osa-aluetta. Muita luennolla [16 s.17] esiteltyjä yksikkötestien kohteita ovat parametrien ja paluuarvojen tulkinta, tietorakenteiden toiminta ja poikkeustilanteet.

Integraatiotestauksessa testattavia kohteita ovat yksiköiden väliset rajapinnat. Esimerkiksi voidaan testata käyttöliittymäyksiköstä tulevan syötteen tulkintaa toisessa yksikössä. Integraatiotestauksesta on vaikea antaa yleispätevää kuvausta, sillä ohjelmien moduulien toiminnat ovat aina tapauskohtaisia. Automaation näkökulmasta integraatiotestauksessa voidaan testata esimerkiksi anturin tuottaman mittausdatan ja säätimen yhteistoimintaa.

Järjestelmätestauksessa testataan järjestelmän käyttötapaukset. Näiden testien automatisointi voi olla haastavaa [16 s.22]. Verkkosivun [18] mukaan järjestelmätestauksesta on yli 50 tyyppiä, joista ohjelmistokehityksessä yleisimpiä ovat käytettävyystestaus, rasitustestaus, luotettavuustestaus ja toiminnallisuustestaus. Parhaassa teoreettisessa tapauksessa kaikki mahdolliset testit suoritettaisiin järjestelmälle, mutta käytännössä tämä ei ole mahdollista ajan ja rahan säästämiseksi. Järjestelmätestauksessa testien suunnittelu on hankalaa, sillä testattavana on koko järjestelmä kaikkine osineen. Tekemällä kattavat testaukset yksikkötestausten tasolla, voidaan saavuttaa korkeampaa laatua ja välttyä projektin loppuvaiheessa ikäviltä yllätyksiltä.

Kaikilla tasoilla testien tulisi kattaa vähintään turvallisuuteen vaikuttavat osat, kuten suojalukitukset. Lukitusten puuttuminen ei näy helposti loppukäyttäjälle, mutta voi aiheuttaa

vaaratilanteita. Lisäksi lukitukset tulisi testata kattavasti, jotta niiden toimivuus voidaan varmistaa.

3.2.3 Simulointi

Simulointi on eräs tapa testata järjestelmiä ja ohjelmia. Simulaattoreita on monenlaisia. Jotkin simulaattorit auttavat testaamaan ohjelmakoodia simuloimalla esimerkiksi PLC-ohjaimen tuloja ja lähtöjä. Toinen simulaattorityyppi soveltuu paremmin säätimien viritykseen, esimerkiksi Simulink-malleja voidaan käyttää mallintamaan prosessin käyttäytymistä ja testata simulaattorin avulla säätimen parametreja. Tässä työssä tärkeässä roolissa olevassa Valmet DNA:ssa on sisäänrakennettuna virtuaalinen ajoympäristö, jota käyttämällä voidaan prosessin tilaa simuloida ja tarkkailla ohjelman suoritusta ilman todellista laitteistoa suoraan suunnitteluasemalta. Virtuaalinen ajoympäristö auttaa testauksessa, sillä fyysinen ympäristö ei usein ole valmiina testiajoja varten. Virtuaaliympäristö ei kuitenkaan paljasta koodin vikoja, jotka ilmentyvät vain fyysisen laitteiston kanssa. Esimerkkitapauksena olen kuullut ohjelmasta, joka ohjasi sähköisesti ohjattavaa venttiiliä auki ja kiinni niin nopeasti, ettei kukaan huomannut muutosta tapahtuvan. Näennäisesti järjestelmä toimi oikein. Venttiili jouduttiin kuitenkin vaihtamaan viikoittain, sillä magneettikelat sulivat kuormituksesta. PLC-pohjaisille järjestelmille on myös virtuaalisia ajoympäristöjä. Simuloimalla voi testata myös poikkeustapauksia. Rösch ja Vogel-Heuser artikkelissaan [19] esittelevät tavan simuloida vikoja ja testata näin ohjelman toimintaa eri tilanteissa. Tämä tekniikka on kätevä tapa tarkistaa ohjelman toiminta useissa poikkeustilanteissa ja havaita näiden poikkeustilanteiden käsittelyssä mahdollisesti piilevät viat. Tässä työssä vikoja paikannettiin lähinnä simuloimalla kenttälaitteilta IO-kortille tulevaa datasignaalia. Näin voitiin analysoida ohjelman toimintaa erilaisilla anturisyötteillä ja varmistua onko kyseessä ohjelmistovika. Datasignaalia voidaan simuloida milliampeerisimulaattorilla, josta voi syöttää järjestelmään virtaa. Tätä tekniikkaa voidaan käyttää laitteille, jotka viestivät perinteisin 4-20mA:n signaalein. Lisäksi joissain IO-korteissa voi olla mahdollisuus pakko-ohjata syötettä. Tämän työn kohteessa IO-kortit mahdollistivat tämän vain binääridatalle.

3.3 Vian korjaus

Tässä luvussa käsitellään varsinaista vian korjausta, eli muutosta ohjelmakoodiin, jonka tavoitteena on poistaa vika. Jos vika on paikallistettu hyvin, itse korjaaminen voi olla hyvin suoraviivaista ja helppoa. Esimerkiksi WinNovan vesiprosessissa korjasin vian, joka aiheutti järjestelmän kaikkien pumppujen, venttiilien ja muiden toimilaitteiden samanaikaisen käynnistyksen. Vika vaikutti aluksi hyvin kaoottiselta, sillä se vaikutti kaikkiin järjestelmän laitteisiin. Järjestelmässä on valmiudet ohittaa Valmet DNA:n säätöpiirit ja

käyttää Siemensin S7-järjestelmää ohjaukseen. Tätä S7-järjestelmää ei kuitenkaan ollut kytketty ja ohjelmakoodiin oli asetettu oletusohjaukseksi bitti 1. Näin ollen asettaessa ohjauksen S7-järjestelmälle, puuttuvan järjestelmän sijaan ohjelma lähettää kaikille laitteille ohjaukseksi bitin 1, joka useimmissa tapauksissa tarkoittaa käynnistystä tai avausta. Vian sai siis korjattua yhdellä bitin vaihdolla. Vaihtoehtoisesti bitin kääntö olisi voitu tehdä jokaiselle toimilaitteelle erikseen.

Tavallisen PC-ohjelman kohdalla muutoksen jälkeen ohjelma käännetään ja testataan vian osalta. Tässä apuna toimii testien automatisointi. Automatisoimalla mahdollisimman paljon testejä säästetään ohjelmoijan aikaa ja pyritään vähentämään testauksen laadun vaihtelua. Tyypillisesti ainakin yksikkö- ja integraatiotestit voidaan automatisoida helposti. Tietokone ajaa itse testitapaukset jokaisen käännöksen ohella ja ilmoittaa, jos jokin testi epäonnistui. Kattavasti automatisoidut testitapaukset auttavat havaitsemaan, mikäli ohjelmakoodin muutos aiheutti uusia vikoja. Samaa voidaan soveltaa automaatio-ohjelmistojen kehityksessä, kuten Jamro [17] on tehnyt. Pan et al. tutkimuksessaan esittelevät valmiita korjauksia tyypillisimpiin vikoihin, jotka noudattavat tiettyä mallia. Pohjimmiltaan korjauksia on kolmea eri tyyppiä, koodia voi muuttaa, lisätä tai poistaa. [13 p. 290] Mielestäni ohjelmakoodille voi tehdä vain kahdenlaisia korjauksia. Koodia voi lisätä ja poistaa. Näiden yhteisvaikutuksena saadaan illuusio, että koodia on muutettu. Kyseessä on kuitenkin vain pieni muotoiluero. Näiden valmiiden mallien soveltaminen automaatiossa ei ainakaan funktiolohkojen osalta toimi suoraan ja muiden automaatio-ohjelmissa käytettävien ohjelmointikielten osalta on suositeltavaa olla tarkkana valmiiden vikakorjausten käytössä.

Butcherin mukaan vikojen korjaus koostuu kolmesta tavoitteesta: korjaa ongelma, vältä taantuman synty, ylläpidä tai paranna koodin yleistä laatua. Eräs Butcherin neuvo näiden tavoitteiden saavuttamiseksi on korjata juurisyy, eikä muuttaa ohjelmaa niin että häiriötä ei enää ilmene. [15 kpl 4] Vaikka vian juurisyy olisikin paikallistettu hyvin, voi vika sijaita niin syvällä koodissa, että sen korjaus vaatisi suuren työn. Erityisesti näissä tapauksissa on houkuttelevaa tehdä helppo muutos, joka piilottaa juurisyyn. Näissäkin tapauksissa olisi ohjelmiston elinkaaren kannalta parempi korjata ongelman juurisyy ja toteuttaa korjauksesta seuraavat ylimääräiset muutokset.

Järjestelmän tulevaisuutta ajatellen on syytä varmistaa dokumentaation ajantasaisuus aina muutosten yhteydessä. Erityisen tärkeää on kirjata tehty muutos järjestelmädokumentteihin, jotta niitä lukiessa vältytään vanhentuneen tiedon levittämiseltä. Myös vanhat versiot on säilytettävä niin kauan kuin voi olla tarpeen selvittää järjestelmän historiasta jotain [14 s.22]. Ohjelmistotekniikan puolella on tutkittu dokumenttien päivityksen automatisointia. Lee et al. tutkimuksessaan [20] kehittivät ”FreshDoc” nimisen työkalun, joka

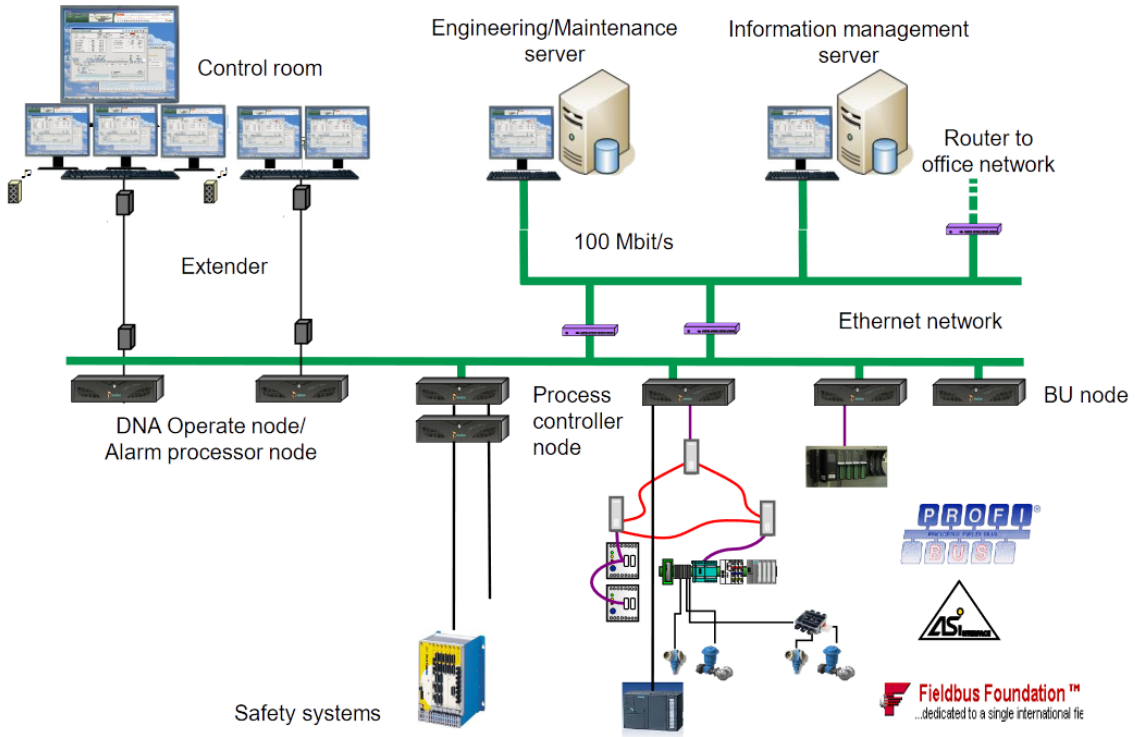
tunnisti vanhentuneet rajapintametodit dokumentista 79 % tarkkuudella ja osasi päivittää 31 % tapauksista automaattisesti. Koska tutkimus [20] perustui avoimen lähdekoodin IT-ohjelmiin, ei ole takeita toimisiko työkalu automaatiojärjestelmien kohdalla, joiden ohjelmointikielissä on usein hyvin erilainen syntaksi. Vastaavasti kuin ohjelmistovikojen kohdalla, myös dokumentaation päivityksessä virheen paikantaminen on oleellisempaa ja usein haastavampaa kuin itse korjaus. Dokumentaation päivitys on huomattavasti helpompaa, kun se päivitetään heti järjestelmämuutosten yhteydessä. Tämän työn käytännön kohteessa dokumentaatio oli vanha, eikä enää vastannut järjestelmän tilaa. Dokumentaation päivitys ajantasaiseksi oli suurimpia haasteita työssä. Nämä haasteet olisi ollut vältettävissä päivittämällä dokumentteja aktiivisesti sen sijaan, että tieto on yhden henkilön muistin varassa.

4. VALMET DNA

Valmet DNA on Valmetin automaatio-ohjausjärjestelmä, joka on käytössä erityisesti prosessiteollisuudessa. Valmet DNA:lla on pitkä historia ja tuotteen nimi on muuttunut vuosien varrella useampaan kertaan samalla kun tuotetta on päivitetty ajantasaisemmaksi. Valmetin DNA:n yleisesityksessä esitetään tuotteen historiaa vuodesta 1979 lähtien. Tuotteen nimi on vaihtunut ensin Damaticista Damatic XD:ksi, sitten Damatic XDi:ksi, Metso DNA:ksi ja 2015 Valmet DNA:ksi.[22] Nykyisin Valmet DNA on muutakin kuin pelkkä ohjausjärjestelmä [21]. Järjestelmästä on tehty laaja-alainen ja esimerkiksi mittauksista kerättyä dataa hyödynnetään prosessin optimoinnissa ja tuotannon suunnittelussa. Valmet DNA on hajautettu ohjausjärjestelmä (DCS), jota Valmetin mukaan voidaan käyttää myös PLC- tai SCADA-järjestelmänä [21].

4.1 Valmet DNA järjestelmähierarkia

Valmet DNA- järjestelmän rakenne on peruspiirteittäin alla olevan kuvan 6 mukainen.



Kuva 6 Valmet DNA arkkitehtuuri [23]

Kuvassa 6 solmuiksi (node) ovat merkittynä asemat, joissa tapahtuu varsinaisen ohjelman suoritus. Prosessiasema tai prosessisolmu suorittaa ohjelmakoodia ohjelmasykleissä, jotka ovat tyypillisesti pituudeltaan noin 100ms-1000ms. Nämä prosessiasemat

ovat kytkettynä Ethernetillä varmuuskopiopalvelimelle (BU-node), josta on puolestaan yhteys insinööripalvelimelle. Varmuuskopiopalvelin huolehtii ohjelmien jaosta hajautetuille prosessiasemille, hälytysasemille ja operointiasemille. Insinööriasemilta on oikeudet editoida ohjelmakoodia ja muokata muita järjestelmän asetuksia. Insinööriasemalta nämä tehdyt muutokset ladataan varmuuskopiopalvelimelle käyttäen Ethernetia. Insinööriaseman kanssa samalla tasolla voi sijaita myös tiedonkeruupalvelin, joka vastaa historiatietojen säilytyksestä ja käsittelystä. Toimilaitteiden suuntaan prosessiasemat ovat kytkettynä kenttälaitteisiin IO-korttien ja kenttäväylien avulla.

Valvomoille on omat operointi- ja hälytysasemansa, jotka vastaavat käyttäjän ja järjestelmän rajapinnasta. Operointiasema ajaa graafista käyttöliittymää ja hoitaa käyttäjän syötteiden välityksen prosessiasemille.

Hälytysaseman vastuulla on valvoa järjestelmän mittauksia ja verrata näitä annettuihin hälytysrajoihin. Ehtojen täyttyessä valvomoissa laukaistaan hälytys.

Prosessiasemien ajamat ohjelmat koostuvat Valmetin omasta ohjelmointikielestä, joka rakentuu graafisesti funktiolohkoja ja moduuleita käyttäen. Tätä ohjelmointikieltä kutsutaan Valmet DNA:n käyttöoppaassa automaatiokieleksi. Moduuleita on erilaisia ja niillä on kaikilla oma roolinsa järjestelmässä. Moduuleita on mm. säätö- tai mittauspiiri, kokonainen sekvenssiohjelma, operaattorin käyttöliittymä. [24] Lisäksi on olemassa pienempiä moduuleita, joista edellä mainitut koostuvat. Näitä käydään tarkemmin luvussa 4.2.1.

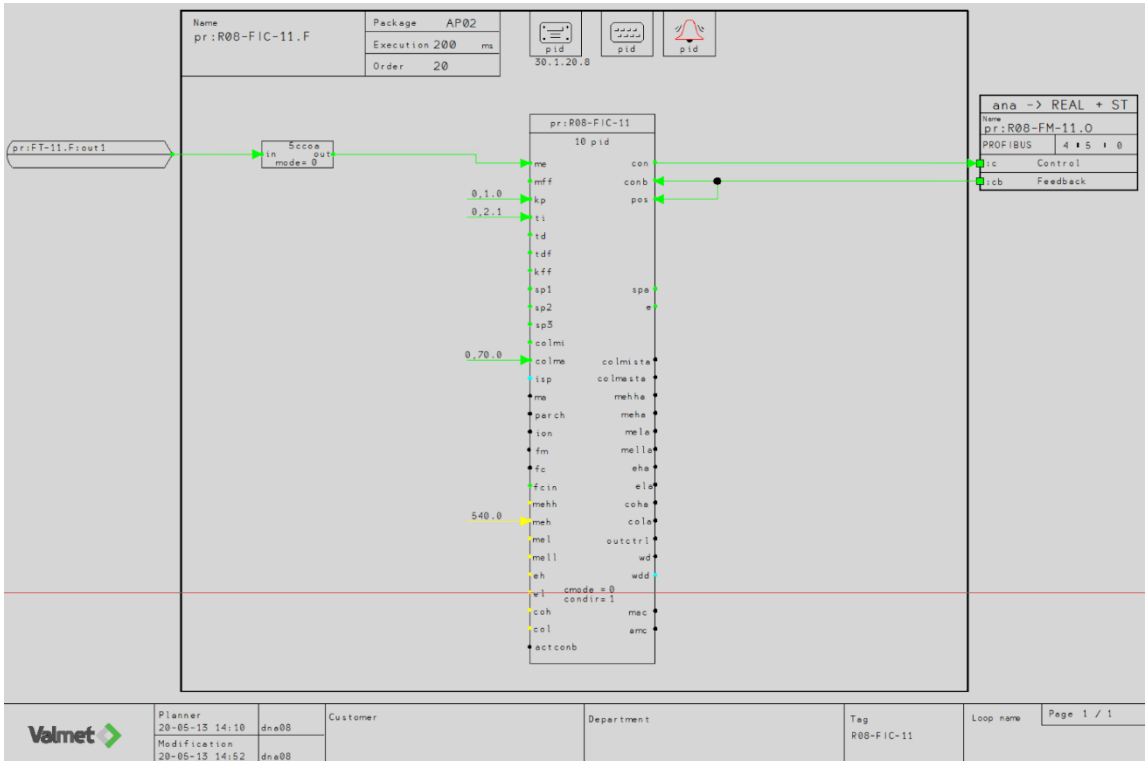
4.2 Suunnittelijatyökalut

Tässä luvussa esitellään perusperiaatteet tässä työssä käytettävistä Valmet DNA -suunnittelijatyökaluista. Valmet DNA:ssa on paljon muitakin työkaluja, mutta ne eivät olleet tarpeellisia tässä työssä.

4.2.1 Function Block CAD

Function Block CAD (FBCAD) on yksi tärkeimmistä työkaluista Valmet DNA:ssa automaatiosuunnittelijan kannalta. FBCAD:lla suunnittelija/insinööri luo ja muokkaa automaatiomoduuleita, eli käytännössä ohjelmia, joita prosessiasemat lopulta ajavat. Nämä automaatiomoduulit koostuvat useista pienemmistä moduuleista, kuten aiemmin luvussa 4.1 mainittiin. Moduulit yhdistetään toisiinsa graafisilla viivoilla, joiden väri kertoo kyseisen tiedon tyypin. Näitä tietotyyppejä ovat esimerkiksi ana, bin ja int. Lyhenteet tarkoittavat analogista signaalia, binääritietoa ja kokonaislukuja. Tyypillisesti anturien tuottama mittausdata on tyypiltään analogista tai binääristä jos kyseessä on esimerkiksi rajakytkin.

Alla on osittainen kuva FBCAD:in käyttöliittymästä. Kuva 7 on Tampereen yliopiston tislauskolonni-laboratoriotyön yhteydessä otettu näyttökaappaus.



Kuva 7 FBCAD

Kuvan ohjelma on merkittävästi yksinkertaisempi kuin esimerkiksi WinNovan vesiprosessissa käytettävät automaatiomoduulit ovat. Kuvasta on näin ollen helpompi havainnollistaa FBCAD:in päämoduuleita.

FBCAD:issa vasemmassa reunassa sijaitsevat ulkoiset tulot, eli mittaukset ja mahdolliset muut arvot, joita käytetään muokattavana olevan toimilaitteen ohjauksessa. Esimerkiksi pakko-ohjauskäsky tuodaan tyypillisesti täältä toimilohkolle.

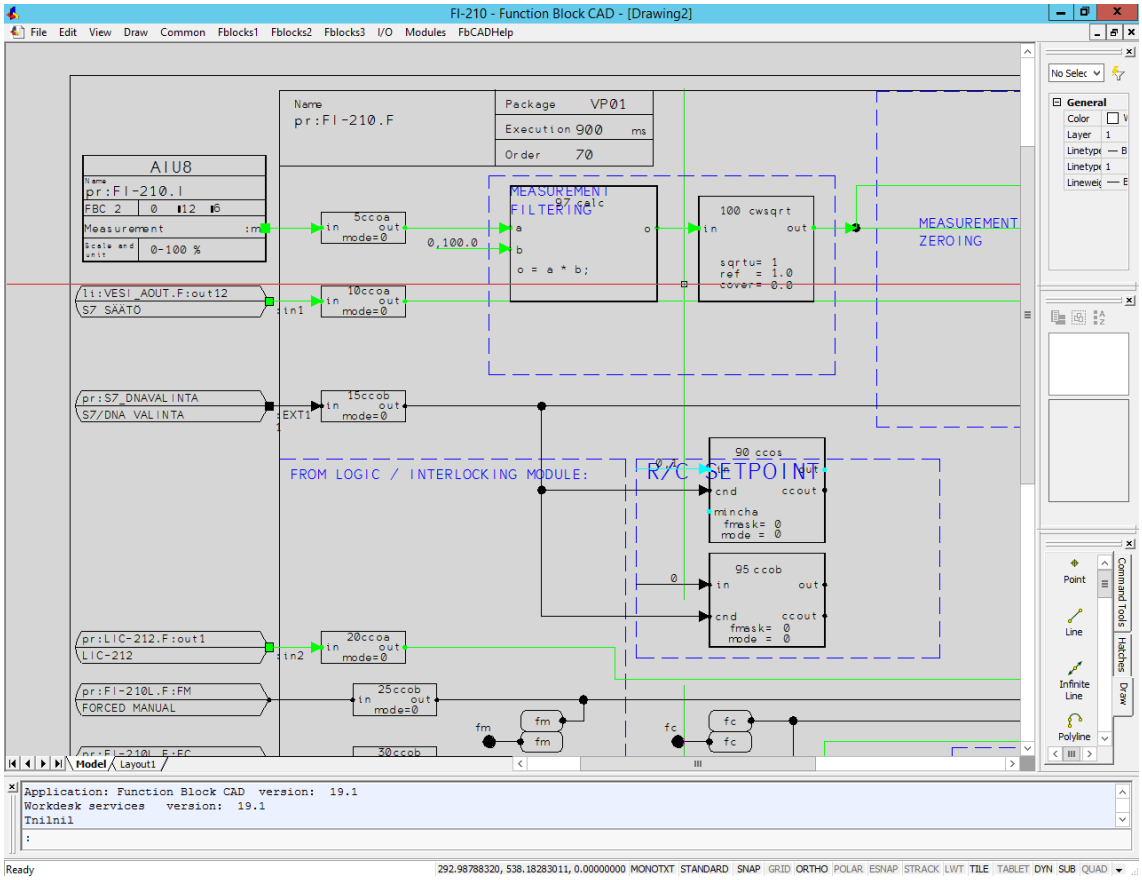
Keskellä rajattuna on toimintamoduuli, joka pitää sisällään useita lohkoja kytkettynä toisiinsa viivoin [24]. Yllä olevassa kuvassa toimintamoduuliin kuuluvat PID-säädinlohko, sekä CCOA-kopiointilohko. Toimintamoduuli sisältää kaiken sen logiikan, jolla tulevaa signaalia käsitellään lähtösignaalin aikaansaamiseksi. Yksinkertaisena esimerkkinä tulona on esimerkiksi paineen mittaus. Mittaussignaali viedään PID-säätimelle, jossa on säätöparametrit asetettuna kohdilleen. PID-säädin lähettää puolestaan ohjaussignaalin, joka johdetaan lähtönä pumpulle.

PID-säädinlohkon yläpuolella sijaitsevat operointi-, positio- ja tapahtumamoduulit [24]. Nämä moduulit kytkevät positiotunnuksella määriteltävän lohkon käyttöliittymään. Operointilohkon kautta voi esimerkiksi kuvan PID-säätimen parametrejä muuttaa tai estää

niiden muuttamisen operointikäyttöliittymästä. Operointilohkon omilla parametreillä voidaan säätää, mikä on operointikäyttöliittymästä sallittua. Positiomoduulin asetuksista voidaan vaikuttaa, mistä operaattorin toimista jää merkintä tapahtumalokiin ja muita positiotoimintoja. Tapahtumamoduulin asetuksista voidaan määrittää, mitkä tapahtumat aiheuttavat hälytyksen valvomoon. Lähes jokainen automaatiomoduuli sisältää nämä kolme moduulia.

Oikealla reunalla sijaitsevat ulkoiset lähdöt, eli kyseisen automaatiomoduulin lähtöjen rajapinta. Tästä osiosta lähtee esimerkiksi ohjaussignaali IO-kortille. Ulkoiseksi lähdöksi, tai tarkemmin sanottuna rajapintaportiksi voidaan myös määrittää mikä tahansa ohjelman käsittelemä arvo. Tällöin muut moduulit voivat käyttää tätä arvoa omana ulkoisena tulonaan.

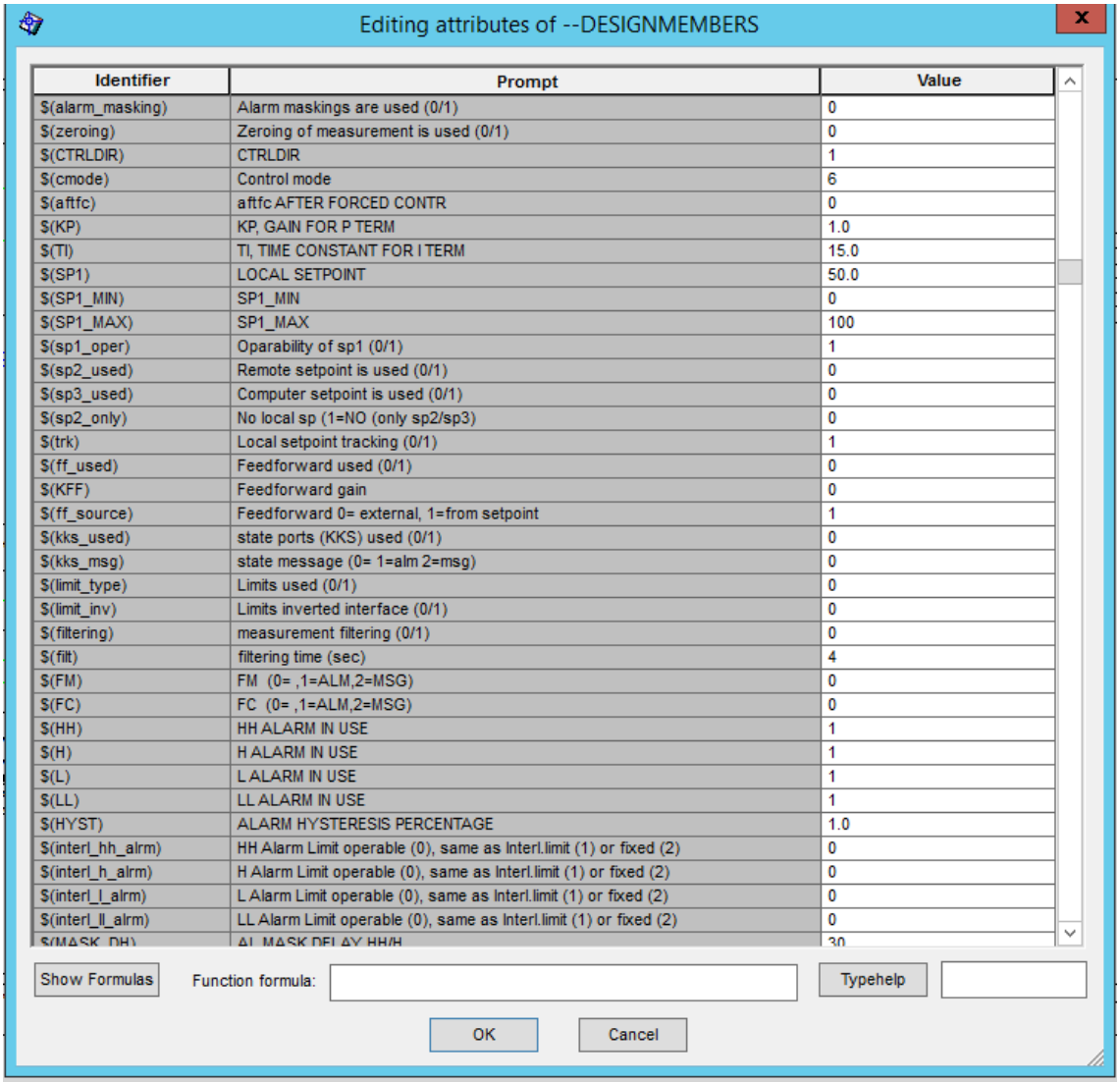
FBCAD:in käyttöliittymä WinNovan vesiprosessilla näytti kuvan 8 mukaiselta.



Kuva 8 Kuvankaappaus osasta WinNovan vesiprossessin erään virtausmittarin ohjelmaa

Ohjelmakuvat olivat niin suuria, ettei niistä saanut järkevästi kuvankaappauksia. Kuvasta 8 kuitenkin näkee miltä ohjelma käytännössä näyttää. Vesiprosessin automaatiomoduuleissa oli laajalti käytössä suunnittelujäseniä, eli ohjelmapohjaan kaavoitettuja muuttuja-

arvoja. Näillä arvoilla voidaan helposti muuttaa parametreja vaikkei kyseisiin parametreihin viittaava lohkoa löytäisikään ohjelmalohkojen seasta. Suunnittelujäseniä muutettiin erikseen avattavasta ikkunasta Edit->Design Members, joka näytti kuvan 9 mukaiselta.

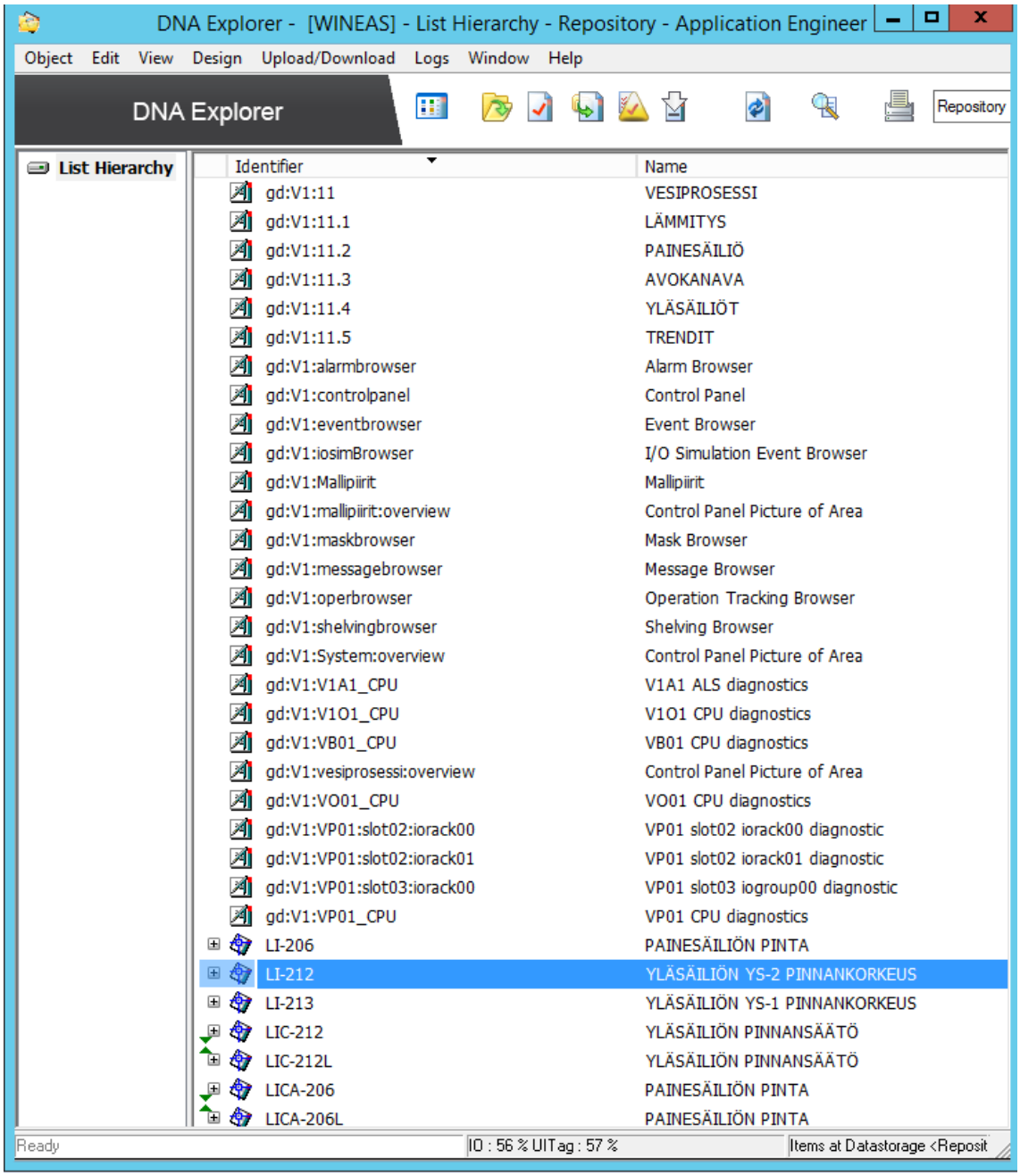


Identifier	Prompt	Value
$(alarm_masking)	Alarm maskings are used (0/1)	0
$(zeroing)	Zeroing of measurement is used (0/1)	0
$(CTRLDIR)	CTRLDIR	1
$(cmode)	Control mode	6
$(aftfc)	aftfc AFTER FORCED CONTR	0
$(KP)	KP, GAIN FOR P TERM	1.0
$(TI)	TI, TIME CONSTANT FOR I TERM	15.0
$(SP1)	LOCAL SETPOINT	50.0
$(SP1_MIN)	SP1_MIN	0
$(SP1_MAX)	SP1_MAX	100
$(sp1_oper)	Oparability of sp1 (0/1)	1
$(sp2_used)	Remote setpoint is used (0/1)	0
$(sp3_used)	Computer setpoint is used (0/1)	0
$(sp2_only)	No local sp (1=NO (only sp2/sp3)	0
$(trk)	Local setpoint tracking (0/1)	1
$(ff_used)	Feedforward used (0/1)	0
$(KFF)	Feedforward gain	0
$(ff_source)	Feedforward 0= external, 1=from setpoint	1
$(kks_used)	state ports (KKS) used (0/1)	0
$(kks_msg)	state message (0= 1=alm 2=msg)	0
$(limit_type)	Limits used (0/1)	0
$(limit_inv)	Limits inverted interface (0/1)	0
$(filtering)	measurement filtering (0/1)	0
$(filt)	filtering time (sec)	4
$(FM)	FM (0= ,1=ALM,2=MSG)	0
$(FC)	FC (0= ,1=ALM,2=MSG)	0
$(HH)	HH ALARM IN USE	1
$(H)	H ALARM IN USE	1
$(L)	L ALARM IN USE	1
$(LL)	LL ALARM IN USE	1
$(HYST)	ALARM HYSTERESIS PERCENTAGE	1.0
$(interl_hh_alrm)	HH Alarm Limit operable (0), same as Interl.limit (1) or fixed (2)	0
$(interl_h_alrm)	H Alarm Limit operable (0), same as Interl.limit (1) or fixed (2)	0
$(interl_l_alrm)	L Alarm Limit operable (0), same as Interl.limit (1) or fixed (2)	0
$(interl_ll_alrm)	LL Alarm Limit operable (0), same as Interl.limit (1) or fixed (2)	0
$(MASK_DH)	AL MASK DELAY HH/H	30

Kuva 9 FBCAD Design Members -ikkuna

4.2.2 DNA Explorer

Kaikki automaatiomoduulit, operointikuvat, sekvenssit, väylälogiikat ja muut suunnitteluoliot tallennetaan tietokantoihin. Näitä tietokantoja voidaan hallita DNA Explorer -ohjelmalla. Explorerissa voidaan hallita työalueita ja järjestellä ja jäsennellä suunnitteluobjekteja. Explorerista valmiit ohjelmat voidaan myös ladata varmuuskopiopalvelimelle ja sitä kautta aitoon järjestelmään. DNA Explorer osaa avata useita eri suunnittelutyökaluja tiedostotyypin mukaan ja toimiikin suunnittelijalle linkkinä eri suunnitteluohjelmien ja tiedostojen välillä. [24] Kuvassa 10 on näyttökaappaus DNA Explorer:in käyttöliittymästä.

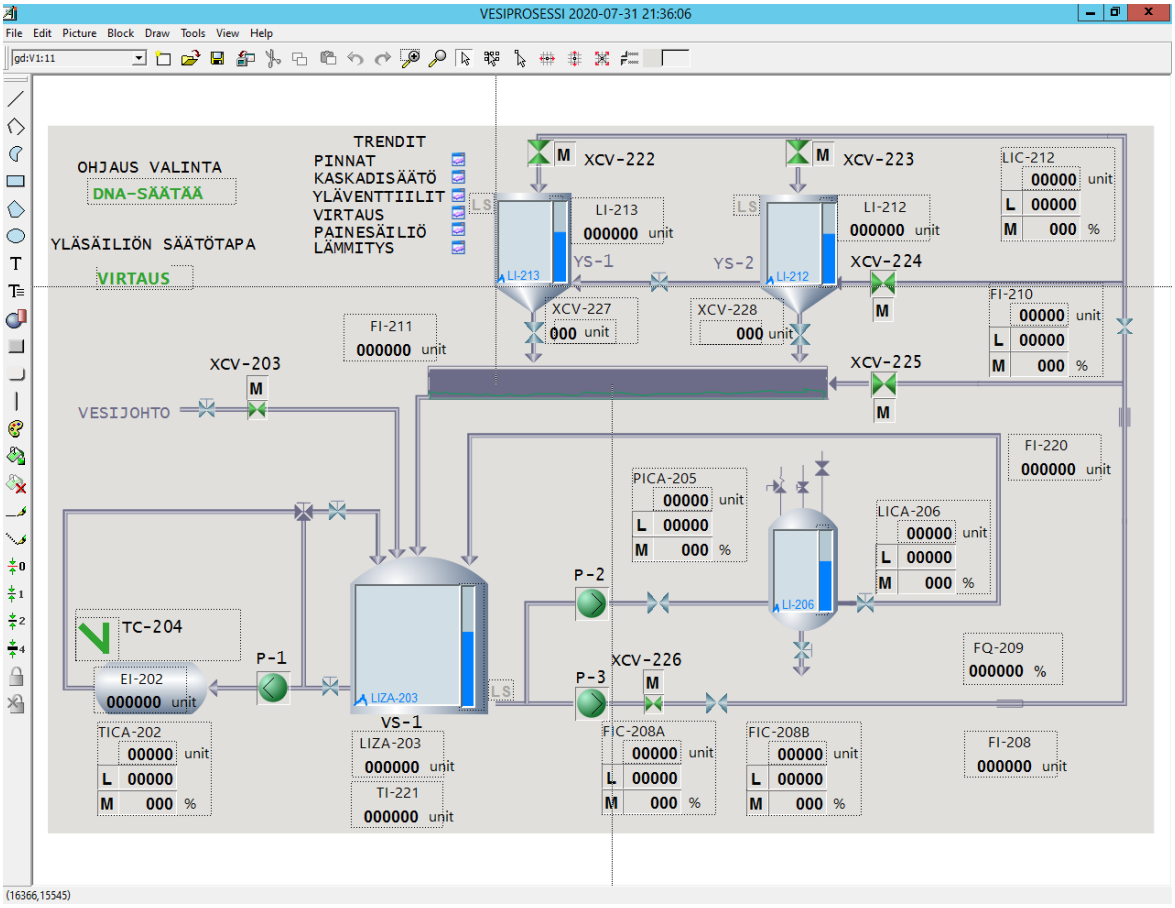


Kuva 10 DNA Explorer kuvankaappaus

4.2.3 Picture Designer

Valvomoiden operointiasemien käyttöliittymien suunnittelussa apuna on Picture Designer -ohjelma. Tällä ohjelmalla voidaan luoda graafisesti valmiita kuvamoduuleita käyttäen prosessia kuvaava käyttöliittymä, josta operaattorin on helppo ajaa prosessia. Ohjelma toimii WYSIWYG-periaatteella (What You See Is What You Get), eli koodin kirjoittamisen sijaan kuva rakennetaan graafisesti ja lopputuloksen näkee suoraan jo muokatessa. Ohjelmassa voi myös piirtää vapaasti, mutta tässä työssä valmiita piirrosmerkkejä käyttäen saatiin kaikki tarpeellinen tehtyä. Kopioitua valmiin piirrosmerkin kirjastosta kuvaan, suunnittelija avaa piirrosmerkin parametrit ja säätää ne kohdilleen. Vähintäänkin

ohjattavan toimilaitteen positiotunnus on määriteltävä kuvaan. Kuvassa 11 on näyttökuva Picture Designeristä, jossa muokattavana on WinNovan vesiprosessin operaattorinäkymän pääikkuna.



Kuva 11 Kuvankaappaus Picture Designerin käyttöliittymästä

5. WINNOVAN VESIPROSESSIN KORJAUS

Tässä luvussa esitellään käytännön esimerkki automaatiojärjestelmän ohjelmointivikojen korjauksesta. Tämä käytännön työ on tehty kesällä 2020 ja tällöin en ollut vielä tutustunut tarkemmin vikojen paikallistamisen ja korjauksen menetelmiin. Tämän takia työssä kaikkea ei tehty yhtä hyvin kuin olisi voitu. Tämä projekti toteutettiin yhteistyössä WinNovan kanssa. Vastuullani oli järjestelmän ohjelmistovikojen korjaus ja osana korjausprosessia vikojen paikallistaminen. Laitteiston tai kytkentöjen vikojen korjaus ei ollut enää vastuualueellani. Lisäksi työssä tehtiin WinNovan opettajan toiveiden mukaisesti parannuksia järjestelmän ohjelmointiin lisäten uusia toimintoja ja parantaen olemassa olevia.

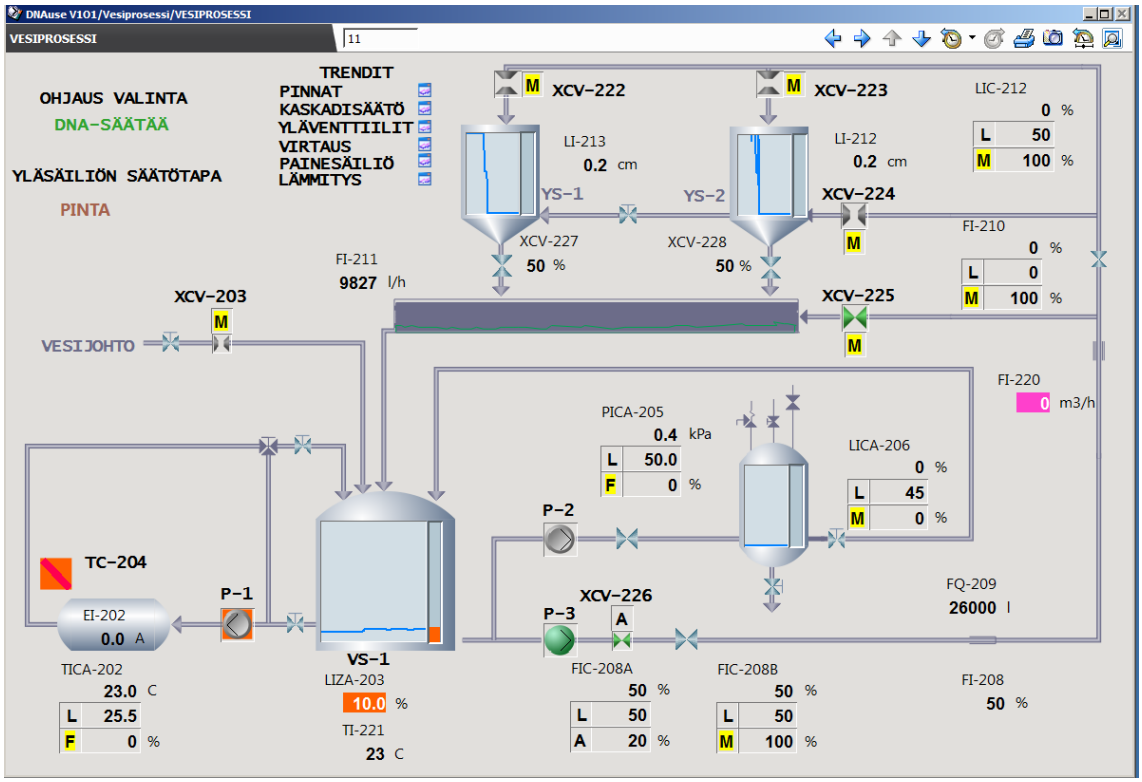
5.1 Järjestelmän esittely

Työn kohteena on WinNovan Porin toimipisteellä sijaitseva vesiprosessi. Vesiprosessi on alun perin suunniteltu ja rakennettu vuonna 1997 prosessialan koulutuskäyttöön. Toimintakuvauksesta [25] tiivistäen vesiprosessin perustoiminta on seuraava. Vesiprosessissa kierrätetään varastosäiliön VS-1 vettä prosessin eri osiin kolmen pumpun avulla. Pumpulla P-1 vettä voidaan kierrättää vedenlämmittimen VL-1 läpi lämmittäen näin varastosäiliön vettä. Pumpulla P-2 vettä voidaan ajaa painesäiliöön PS-1, jossa painetta voidaan säätää paineilman ja kuristusventtiilien avulla. Pumpulla P-3 vettä voidaan pumpata yläsäiliöihin YS-1 ja/tai YS-2, sekä avokanavaan AK-1. [25] Nämä kolme prosessin osaa toimivat omina kokonaisuuksinaan kytkeytyen vain varastosäiliön kautta toisiinsa. Lämmityskierrosta ja painekierrosta vesi palautuu suoraan varastosäiliöön, joka on avoin. Myös yläsäiliöistä ja avokanavasta vesi palautuu varastosäiliöön.

Toimintakuvauksessa kuvataan myös nämä kolme kiertoa tarkemmin. Lämmityskierrossa lämmittimelle tulevan ja lähtevän veden lämpötilaa mitataan ja lämmitintä säädetään lämmittimeltä lähtevän veden lämpötilan mukaan. Vedestä osa palautetaan varastosäiliöön VS-1 kolmitieventtiilin kautta. Painesäiliön PS-1 kierrossa pumppu P-2 toimii vakionopeudella ja painetta säädetään rajoittamalla säiliöön tulevan veden virtausta. Painesäiliön pinnankorkeus säädetään paineilman avulla. Painesäiliön vesi poistuu takaisin varastosäiliöön VS-1. [25] Yläsäiliöiden osalta ajotapoja on useampia. Pumpun tehoa voidaan säätää taajuusmuuttajalla ja lisäksi virtausta voidaan säätää kuristusventtiilillä. Yläsäiliöiden pinnankorkeutta voidaan säätää YS-2 pinnan mukaan, tai virtauksen mukaan tai kolmantena vaihtoehtona kaskadisäädöllä hyödyntäen molempia säätimiä.

Lisäksi käyttäjä voi valita, ajaako vettä avokanavaan, YS-1:een tai YS-2:een. Kaikki näiden kombinaatiot ovat mahdollisia. YS-1 ja YS-2 välillä on käsiventtiilillä varustettu putkilinja, josta säiliöt voidaan erottaa tai yhdistää.

Prosessin ohjauksen käyttöliittymässä (kuva 12) on havainnollistettu prosessin rakennetta.



Kuva 12 Vesiprosessin operoinnin päänäkymä

Vesiprosessin automaatio-ohjelmistona on aiemmin ollut käytössä Damatic XD ja Siemens S7 yhteistoiminnassa. 2015 järjestelmä siirrettiin ja päivitettiin Valmet DNA-järjestelmäksi, mutta päivitys ei onnistunut. Ohjelmamoduulit, sekä Siemens S7-järjestelmä lakkasivat toimimasta päivityksen yhteydessä. Lisäksi laitteiston uudelleenrakennuksen yhteydessä IO-kytkennät tehtiin väärin, joka aiheutti lisää ongelmia.

5.2 Työn alku

Alussa minulla ei ollut konkreettista käsitystä järjestelmän toiminnasta, taikka aiempaa kokemusta Valmet DNA:n ohjelmoinnista. Eräällä Tampereen Yliopiston opintojaksolla harjoitustyössä hieman harjoiteltiin Valmet DNA:n ohjelmointia, mutta olin jo ehtinyt unohtaa oleellisia asioita aiheesta tämän projektin alkaessa. Harjoitustyössä ei kuitenkaan työskennelty aidon laitteiston parissa, joten projektissa oli minulle lähtökohtaisesti paljon uutta. Järjestelmän toiminta alkutilanteessa oli varsin kaoottinen, suuri osa vent-

tiileistä oli pysyvästi lukittuna ja automaattiohjaus ei toiminut. Käsiajolla vettä pystyi kierrättämään tiettyä yksittäistä reittiä pitkin, mutta järjestelmä ei täyttänyt tarkoitustaan opetuskäytössä. Järjestelmästä oli kattavat dokumentaatiot vuosilta 1997–1998, mutta ne eivät enää kuvanneet järjestelmää todenmukaisesti. Dokumenteista oli kuitenkin merkittävä apu verratessa järjestelmän ohjelmamoduuleita vanhoihin ohjelmiin, jotka WinNovan mukaan toimivat oikein ennen päivitystä ja siirtoa.

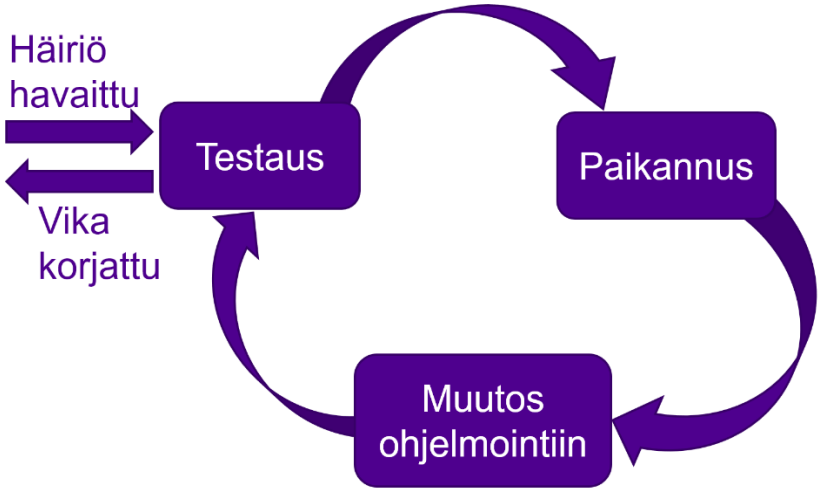
Aivan ensimmäiseksi sovimme aloitustapaamisen Valmetin työntekijän kanssa, joka neuvoi DNA Explorerin ja FBCAD:in käyttöä, jotta pääsisimme aloittamaan. WinNovalla oli 2017 laadittu alustava vikalista [26], mutta se osoittautui lopulta puutteelliseksi. Tässä vikalistassa oli mainittuna noin kymmenen havaittua häiriötä ja lopulta järjestelmään teimme yli 90 muutosta. Näitä muutoksia ei kuitenkaan tässä työssä käydä läpi yksitellen. Muutoksia ja vikoja esitellään esimerkkeinä työssä käytetyille paikannus, korjaus ja testaustavoille. Ensimmäinen vaihe aloitustapaamisen jälkeen oli varmistua, että järjestelmän ohjelmointi voidaan pahimmassakin tapauksessa palauttaa lähtötilaan. Työssä ei käytetty kovin kehittynyttä versionhallintaa, vaan DNA Explorerissa tehtiin uusi työalueen, johon kopioitiin senhetkiset ohjelmamoduulit. Lisäksi kopioimme kaikkien ohjelmamoduulien CAD-tiedostot erikseen suunnitteluaseman paikalliselle kiintolevylle omaan kansioonsa. Näin lähtötilanne oli riittävän hyvin turvattu. Varmuuskopioita olisi kenties ollut hyvä ottaa enemmänkin, jotta ongelmien ilmetessä ainoa varmuuskopio ei olisi paluu alkutilanteeseen. Työssä suuria ongelmia ei kuitenkaan tullut ja koko työn aikana jouduttiin turvautumaan varmuuskopioon vain yhden tiedoston kohdalla, jonka positiotunnusta muutettiin.

Seuraavaksi ajankohtaiseksi tuli ensimmäisen vian korjaus ja ohessa ohjelmointityökalujen käytön opettelu. FBCAD:in käyttö oli aluksi erittäin kankeaa ja ensimmäiset päivät kuluivat käyttöopasta [24] selaten. Toimintakuvauksessa [25] oli liitteenä lukituskaaviot, jotka päätimme käydä läpi järjestyksessä vertaillen Valmet DNA:n lukituslogiikoita dokumentin lukituskaavioihin. Aloitimme lukituskaavioista, sillä vikalistassa [26] isoimmaksi ongelmaksi oli merkitty virheelliset lukitukset. Virheelliset lukitukset näkyivät selkeästi myös operointi-ikkunassa, jossa useat venttiilit olivat virheellisesti lukittuna kiinni.

5.3 Vian korjaussykli

Tässä kappaleessa esitellään työssä käytettyjä tapoja ja periaatteita vikojen paikallistamiseen, ohjelmakoodin muutoksiin, testaukseen ja dokumentointiin. Työssä käytetyt tavat eivät kaikin puolin ole hyviä, mutta järjestelmän kokoluokassa näillä tavoilla saavu-

tettiin lopulta merkittävä parannus järjestelmän laatuun ja toimivuuteen. Lisäksi analysoidaan miksi kyseinen tapa toimi kyseisessä työssä ja joistain tavoista esitetään parannusehdotuksia. Pääasiassa vikojen korjaus toimi kuvan 13 syklin mukaisesti.



Kuva 13 Vian korjaussykli

Sykliin siirtyminen ja syklistä poistuminen tapahtuu aina testauksen kautta. Pääasiallinen syy sykliin siirtymiseen on häiriön havaitseminen. Havainto testataan ja erityisesti pyritään löytämään systemaattinen tapa häiriön aiheuttamiseksi. Testauksesta siirrytään vähitellen paikantamaan vikaa, joka usein tapahtuu testauksen ohella. Vian paikannuksen jälkeen siirrytään toteuttamaan muutokset. Muutosten jälkeen kohde testataan uudelleen. Jos häiriötä ei enää ilmene, voidaan vika olettaa korjatuksi ja poistua syklistä. Jos häiriö ilmentyy yhä, on korjauksessa jokin mennyt pieleen ja sykli kierretään uudelleen. Syklin aikana on yleistä havaita uusia vikoja tai häiriöitä, mutta syklissä on tavoitteena korjata yksi vika kerrallaan. Uudet havaitut viat kannattaa kuitenkin kirjata muistiin heti havaintohetkellä, jotta ne eivät unohdu.

5.3.1 Vian paikallistaminen

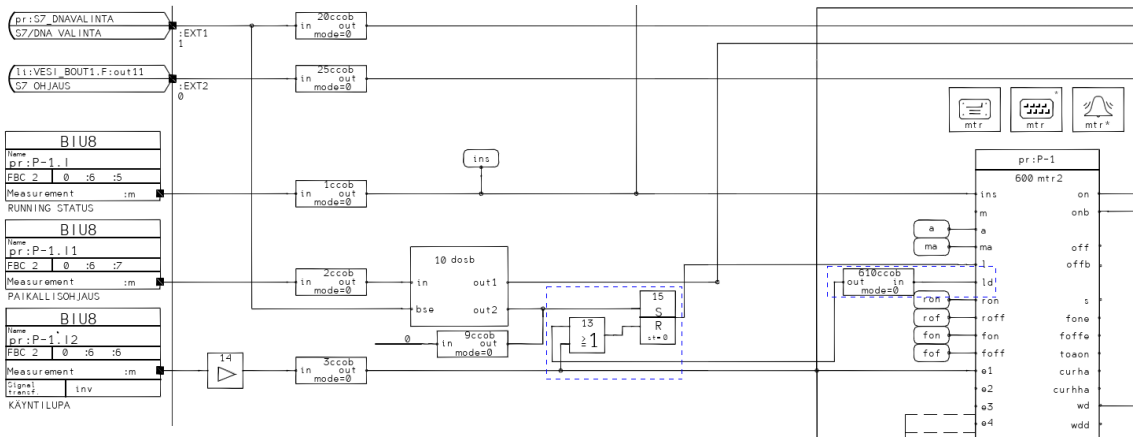
Vian paikallistamisessa tässä työssä käytettiin ohjelmakoodin manuaaliista lukemista, kenttäsimulointeja, sekä FBCad:in Function Test-toimintoa. Usein aloitimme vian paikallistamisen lukemalla itse ohjelmakoodia pyrkien selvittämään, kuinka kyseisen moduulin ohjelmalogiikka toimii. Huomiota kiinnitettiin erityisesti rajapintoihin, eli moduulin tuloihin ja lähtöihin. Valmet DNA:ssa tulot vastaavat mittauksia tai syötteitä muista moduuleista. Lähdöt vastaavasti ovat ohjauksia tai syötteitä muihin moduuleihin. Saatuamme alustavan käsityksen moduulin toiminnasta, vertasin ohjelmakoodia vanhan dokumentaation ohjelmakaavioon. Vanhojen ohjelmien kaaviot olivat huomattavasti yksinkertaisempia, sillä nykyisissä on merkittävästi enemmän hyödynnetty valmiita monipuolisia pohjia.

Näissä pohjissa on valmiuksia hyvin monille erilaisille toiminnoille ja rakenteille, mutta samalla ne hankaloittavat ohjelmakoodin luentaa. Joissain tapauksissa havaittiin vanhoihin kaavioihin vertaamalla eroavaisuuksia, joihin kiinnitettiin erityisesti huomiota seuraavassa vaiheessa. Ohjelmakoodia lukemalla ja vertaamalla pääasiallinen tarkoitus oli tutustua ohjelman toimintaan, havaita mahdollisia ongelmakohtia ja valita kiinnostavia tarkkailupisteitä testejä varten. Tapa toimi varsin hyvin, mutta liiallinen ohjelmakoodin luenta on aikaa vievää, eikä suoraan edistä työtä. Tässä tapauksessa aiempaa kokemusta järjestelmästä ei ollut, joten työssä käytettiin paljon aikaa ohjelmakoodien lukemiseen. Moduulin toiminnasta saa nopeammin käsityksen, jos sen toimintaa voi tarkkailla virtuaalisen tai todellisen ajon aikana. Järjestelmän todelliset ajot sijoittuivat projektin loppupuolelle, jolloin suurin osa vioista oli jo korjattu. Virtuaalisia ajoja ei käytetty tämän työn yhteydessä, mutta ne olisivat mahdollisesti nopeuttaneet ongelmakohtien havaitsemista.

Tyypillisesti ohjelmakoodin luennan jälkeen seurasi moduulin testaus. Valvomosta asetettiin FBCAD Test-tilaan, jolloin ruudulle päivittyy reaaliajassa arvoja ohjelmakaavion eri kohdista. Näin voidaan seurata ohjelman toimintaa kiinnostavista testipisteistä samalla, kun syötteitä simuloidaan. Suunnitteluaseman vieressä sijaitsi operointiasema, jonka operointinäkymän ruudunkaappaus on aiemmin esitelty kuvassa 12, ja hälytysnäkymä, johon järjestelmä listaa hälytykset aikaleimoineen. Tarkkaillen näitä näyttöjä, pyrimme selvittämään vian sijaintia samalla kun kyseisen laitteen syötteitä simuloitiin. Simulointi toteutettiin lähes kokonaisuudessaan aidolta mittauslaitteelta hyödyntäen milliampeerisimulaattoria. Simuloimme kentältä järjestelmään syötteitä ja simulaattorin syöte ilmoitettiin valvomoon. Analysoimme järjestelmän vastetta signaaliin ja ohjelmasta etsittiin kohtaa, jossa toiminta muuttuu vääräksi. Tyypillisesti viat liittyivät lukitusten binääridataan. Etsimme kohtaa, jossa bitti vaihtui, vaikka sen ei pitäisi tai toisinpäin. Usein näin löydettiin uusia kiinnostavia testipisteitä ja valvomosta pyydettiin kentältä tiettyjä syötteitä hypoteesien testaamiseksi. Testaus suoritettiin tarkalla ja kattavalla tavalla yksittäisten laitteiden osalta, mutta suuremmissa järjestelmissä tämä työtapa voi olla liian työläs ja hidas. Lisäksi järjestelmissä, jotka ovat ajossa kokonaan tai osittain, tämä tapa voi olla jopa vaarallinen. Simuloidut syötteet voivat laukaista vahingossa turvamekanismeja ja aiheuttaa prosessin alasajon. Järjestelmät, joita ei olla vielä edes rakennettu, tämä vian paikallistamistapa on liki mahdoton toteuttaa. Kuitenkin tämän työn osalta kenttäsimuloinnit toimivat loistavasti, koska kohde oli suhteellisen pieni, valmiiksi rakennettu ja kohde ei ollut aktiivisesti ajossa.

Toisinaan tässäkään työssä kenttäsimulointi ei ollut mahdollista ja tällöin käytettiin FBCAD:in Test-tilaa virtuaalisessa ajoympäristössä ja syötettiin ohjelmalle syötteiden ar-

voja suoraan FBCAD:ista. Myös tällä tavalla voitiin ohjelma testata yhtä lailla, mutta tällöin testauksen ulkopuolelle jäi laitteisto ja kytkennät. Automaatiojärjestelmien yhteydessä on muistettava aina tarkistaa myös laitteiston ja kytkentöjen toiminta. Paikalliskäyttökytkinten kohdalla vika oli puolittain ohjelmistossa, puolittain kytkennöissä. Mikäli tässä tapauksessa olisi tarkistettu vain ohjelmisto, järjestelmä ei olisi toiminut päivityksen jälkeenkään. Ohjelmakaavioista puuttui paikalliskytkinten osalta toiminnan mahdollistavat lohkot. Lisäksi paikalliskytkinten sisäiset kytkennät olivat väärin niin, ettei käyntilupaa ollut. Kuvassa 14 on sinisellä katkoviivalla merkittynä osiot, jotka laadittiin paikalliskytkinten toiminnallisuuden toteuttamiseksi.



Kuva 14 Pumpun P-1 paikalliskytkimen logiikka

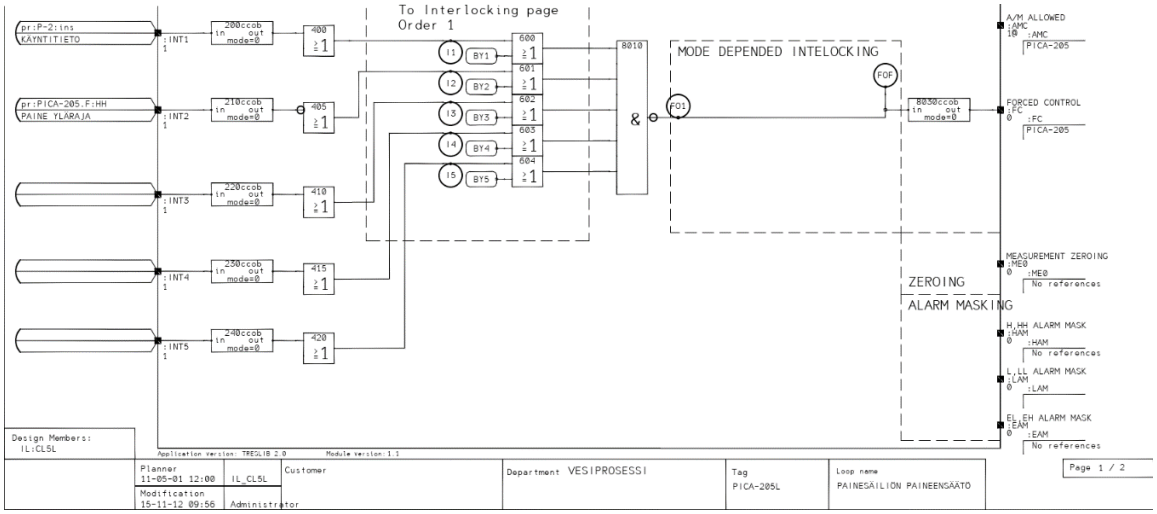
Kuvan 14 logiikassa PAIKALLISOHJAUS-niminen syöte on kytkimen START-asennon liipaisusignaali, tämä kuljetetaan desb-lohkolle, joka ohjaa syötteen out2:een mikäli DNA-järjestelmä ohjaa ja out1:een jos ohjaavaksi järjestelmäksi on valittu S7. Myös KÄYNTILUPA-niminen syöte tulee kytkimeltä. Jos kytkin on asetettu ON-asentoon, käyntilupa on myönnetty. Sinisellä katkoviivalla on merkitty uusi laadittu piiri, joka käynnistysliipaisun jälkeen pitää moottorin toimilohkolle syötteenä 1, eli pitää moottorin käynnissä. Moottorin sammutus tapahtuu pääasiallisesti poistamalla käyntilupa. Vahinkojen välttämiseksi piiriin lisättiin ”väärinpäin” piirretty lohko, joka lähettää reset-lohkolle käskyn nollata syöte kun ohjaustavaksi asetetaan paikallisajo. Vaikka moottoritoimilohkon kytkentäpiste ld onkin tulo, kyseinen kytkentäpiste toimii myös lähtönä. Metso DNA käyttöohjeessa [24] kirjoitetaankin tulo-tyyppisen kytkentäpisteen lukevan tietoa tai sekä lukevan että kirjoittavan tietoa. Käyttäen tätä kytkentäpistettä

lähtönä, voidaan tietoa ohjaustavan vaihdosta hyödyntää muualla logiikassa. Näin vanha ajokäsky ei välittömästi käynnistä moottoria vaihdettaessa paikallisajolle.

5.3.2 Ohjelmakoodin muutokset

Edellisen luvun paikalliskytkimiin liittyvä tapaus on eräs esimerkki työssä tehdyistä ohjelmakoodin muutoksista. Tässä työssä paikalliskytkimet olivat yksittäisiä harvoja tapauksia, joissa vaadittiin uusien lohkojen luontia ”käsin” lohko ja viiva kerrallaan. Pääasiassa vaaditut muutokset olivat yksinkertaisia lohkojen konfigurointeihin liittyviä muutoksia. Tämän tyyppisiä muutoksia olivat mm. bitin inversio, lukituskaavion positiotunnusten syöttö, raja-arvojen asetus ja operointikuvien päivitys. Tässä luvussa esitetään lyhyesti erilaisia tehtyjä muutoksia ja arvioidaan ohjelmamuutosten merkitystä järjestelmän toiminnan ja laadun kannalta.

Valtaosa tehdyistä muutoksista liittyi ohjelman lukituksiin. Näin ollen yleisin muutos tässä työssä oli lukituskaavion täyttö. Vielä enemmän lukituskaavioita tarkastettiin, mutta tämä on enemmän vian paikallistamiseen liittyvä toimenpide. Järjestelmän lukituskaaviot olivat kuvan 15 tapaisia.



Kuva 15 Vesiprosessin PICA-205 lukituskaavio

Lukituskaaviot olivat toteutettu Valmetin valmiiden pohjien avulla, jotka tekivät lukituskaavioiden muokkauksesta hyvin helppoa Design Members-ikkunan avulla, joka esiteltiin aiemmin kuvassa 9. Design Members-valikkoon syötettiin positiotunnus ja pohjan automaattiset kaavat sijoittivat syötetyn positiotunnuksen mukaisen tulon lukituskaavioon. Nämä tulot ovat binääritietoja, jotka muuttuvat esimerkiksi lukitusrajojen ylittyessä. Tyypillisesti lukituskaavion tulot saapuvat muiden laitteiden ylä- ja alarajoista, sillä esimerkiksi säiliön täyttyessä yli ylärajan, tulee pumppu ja venttiilit sulkea. Näin esimerkiksi pumpun lukituskaaviossa sijaitsisi tulona säiliön ylärajan mittaus. Tässä työssä lukitusten

muutokset olivat yksinkertaisia ja helppoja. Yhden moduulin kohdalla tarvittavaa ylärajatietoa ei ollut valmiiksi asetettu saataville, joten laadittiin käsin lohkoja, jotka mahdollistivat kyseisen tiedon lukemisen muista moduuleista, tässä tapauksessa toisen laitteen lukituskaaviosta. Lohkojen laatiminen käsin moduuleissa, jotka käyttävät valmiita pohjia, ei ole hyvää ohjelmointityyliä. Jos järjestelmää joudutaan päivittämään uudelleen tulevaisuudessa, aiheuttavat käsin piirretyt lohkot hankaluuksia pohjien käytön kanssa. Pohjista ei kuitenkaan aina löytynyt tarvittua toiminnallisuutta ja tällöin laadimme lohkokaavion käsin. Parempi tapa olisi ollut laajentaa pohjia kattamaan tarvitsemani toiminnon, työtä tehdessä tämä osoittautui liian vaikeaksi suhteessa käytettävään aikaan ja työmäärään. Näitä tapauksia oli kuitenkin harvoin, vain noin 3 kpl koko projektissa, joten en usko järjestelmän ylläpidettävyyden kärsineen pahasti. Yleisesti ottaen lukitusten läpikäynti paransi järjestelmän toimintaa ja luotettavuutta merkittävästi, kyseessä oli entuudestaan tiedossa oleva ongelmakohta ja suurin osa järjestelmän vioista liittyi lukituksiin.

Seuraavaksi yleisin muutos järjestelmän ohjelmiin oli yksittäisten bittien muutokset, sekä vastaavasti säädinten toimintasuunnan muutos. Nämä muutokset olivat toteutettavissa yksittäisen inversion lisäyksellä/poistamisella oikeasta kohdasta. Kuten aiemmin on mainittu, vian paikallistaminen on tärkein vaihe. Tämä korostuu yhden bitin muutoksissa. Korjaus on tehtävä oikeaan kohtaan, jotta vältytään uusien ongelmien luomiselta. Mikäli havaitsee kääntävänsä samaa bittiä yhä uudelleen ja uudelleen, kannattaa tarkistaa voisiko saman saada aikaan yksittäisellä käännöllä jossakin toisessa kohdassa. Bittimuutosten yhteydessä kattava testaus korostuu. Pienen muutoksen aiheuttamat seuraukset voivat olla suuret, eikä kaikkia seurauksia ole aina helppo ennustaa. Työssä eräs pieni muutos aiheutti pumpun pysäytyksen eston. Ohjelman osa, joka vastasi sammutuskäskystä, suoritettiin aina ennen ajokäskyä, joka oli pitopiirissä kokoaikaisesti päällä. Näin ajokäsky kumosi aina sammutuskäskyn, eikä pumppua voinut sammuttaa paikallisesti. Tapaus oli opettavainen. Pienet asiat kuten suoritusjärjestys ja bittien käännöt voivat aiheuttaa odottamattomia seurauksia. Säätimen toimintasuunnan käänteisyydelle, eli säädin avasi venttiiliä, kun olisi pitänyt sulkea, löytyi looginen selitys. Vanhassa Damatic XD järjestelmässä 0- ja 1-bitit merkitsivät eri toimintasuuntia kuin uudemmassa Valmet DNA:ssa. Tämäkin on hyvä esimerkki pienestä muutoksesta, jolla on suuri merkitys.

Ohjelmakoodien kolmas muutostyyppi liittyi hälytysten ja lukitusten raja-arvoihin. Kaikissa moduuleissa nämä raja-arvot olivat määritelty mittauslaitteen ilmoittaman maksimialueen mukaan. Tästä seurasi hölmöjä hälytysrajoja, kuten veden lämpötilan alaraja -40C°. Kaikkia hälytysrajoja ei käyty läpi, mutta pyrimme tarkistamaan kaikki käytössä olevat lukitusrajat ja muokkaamaan rajoja tarvittaessa. Rajojen muokkaus oli yksinker-

taista Design Members-valikon kautta. Oikeilla raja-arvoilla on merkitystä erityisesti tehtaissa käyttäjäkokemuksen kannalta. Ajoissa ja merkitykselliset hälytykset ovat avainasemassa ajoissa toimimiseen. Tuotannon työkokemusten pohjalta tiedetään, ettei koko prosessia voi valvoa kattavasti kaiken aikaa ja valtaosa toimista tehdään hälytyslistan pohjalta. Tässä kohteessa hälytyksillä on vain pedagogisesti arvoa ja siksi niiden määrittelyyn ei käytetty paljoa aikaa.

Viimeinen tässä luvussa esiteltävä muutos on operointikuvien päivitys. Osa operointinäkymän mittauksista oli määritelty virheellisesti, joten asetimme näiden positiotunnukset oikeiksi. Operointinäkymät linkittyvät kätevästi FBCAD:in lohkokaavioihin, joten suurin osa toiminnallisuuden ja arvojen määrittelyistä tehdään FBCAD:in puolelta. Picture Designerissa keskitytään ulkoasuun ja ainoa toiminnallisuuteen liittyvä määrittely on kytkös positiotunnukseen. Operointikuviin toteutettiin useita muutoksia. Muutokset olivat luonteeltaan tyypillisesti mittausten yksiköiden ja mittausalueiden skaalausta.

5.3.3 Testaus

Tässä luvussa esitellään työssä käytetyt testaustavat vikojen korjaussyklissä, eli ennen kuin kaikki havaitut viat on korjattu. Vikojen korjauksen jälkeisestä järjestelmätestauksesta on oma lukunsa 5.5. Ohjelmakoodin muutosten jälkeen testataan, onko vika korjaantunut. Koska järjestelmä oli laajalti ajokelvoton alkutilanteessa, turvauduimme samoihin testausmenetelmiin kuin vian paikallistamisessa. Näitä olivat kenttäsimuloinnit ja FBCAD:in Function Test-toiminto. Testaus ei näin ollen ollut kovin kattavaa. Moduulin toimintaa muiden moduulien kanssa testattiin vain vähän käyttäessä kenttäsimulointeja. Järjestelmä oli kuitenkin kokonsa puolesta testattavissa suurella järjestelmätestillä, joka toteutettiin havaittujen vikojen korjausten jälkeen. Testaus on vian korjaussyklissä ainoa tie ulos syklistä. Jos vikaa ei enää havaita, oletetaan vika korjatuksi ja voidaan siirtyä käsittelemään seuraavaa vikaa. Testaus aloittaa ja päättää vian korjauksen.

5.3.4 Muutosloki

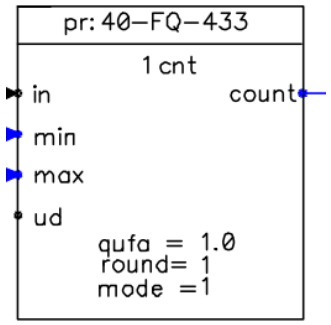
Testauksen jälkeen kirjasimme havaitun vian korjatuksi. Muutoslokina käytimme perinteistä kynää ja paperia listaamaan havaitut viat ja merkitsemään näitä korjatuksi. Koko projektin päätteeksi muutosloki kirjoitettiin puhtaaksi ja dokumentit päivitettiin vastaamaan nykytilannetta. Yksittäisten vikojen kohdalla emme muokanneet järjestelmän dokumentteja, vaan säästimme kaikki muutokset projektin loppuun. Tässä tavassa hyvänä puolena on, ettei dokumentteja tarvitse kirjoittaa uudelleen kuin kerran. Haittapuolena on kirjoitusmäärän suuruus ja muutokset eivät enää ole tuoreessa muistissa, joten paperilla olevan muutoslokin tarkkuus korostuu. Nopeasti työn ohella kirjattu muutos ei välttä-

mättä kuukauden päästä tunnu enää yhtä selkeältä ja yksiselitteiseltä kuin kirjoitushetkellä. Parempi tapa olisi käyttää joka työpäivänä esimerkiksi tunti dokumentaation päivittämiseen, niin muutokset olisivat tuoreessa muistissa, ja päivitystyö ei tuntuisi yhtä suurelta.

5.4 Ylimääräiset parannukset ohjelmaan

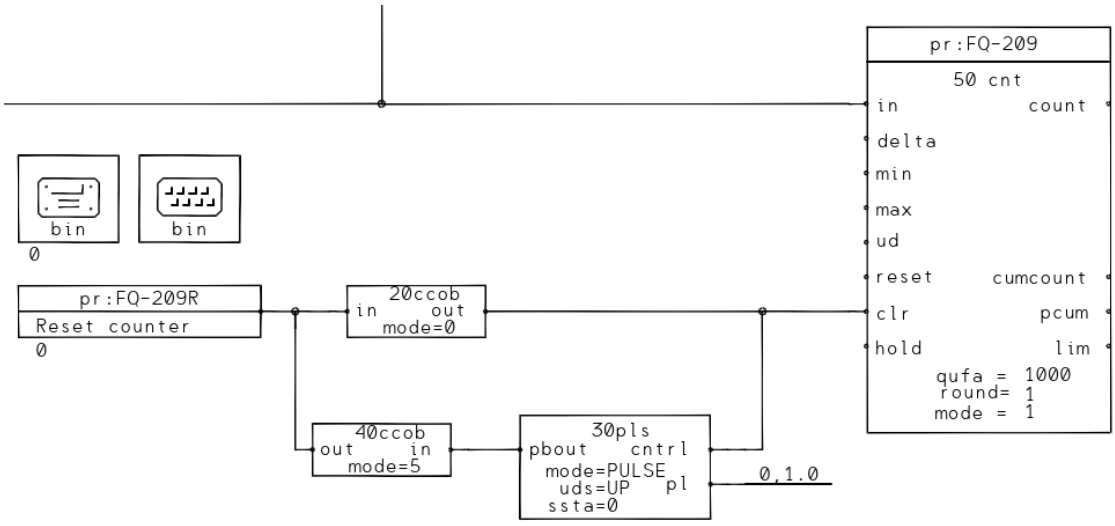
Vikojen korjauksen lisäksi järjestelmään tehtiin keskustelujen yhteydessä ilmenneiden toiveiden pohjalta pieniä uusia toimintoja. Uusina toimintoina toteutettiin veden virtauslaskurin nollauspainike ja uusia trendikuvia. Lisäksi paranneltiin DNA Help-sovelluksen lukitusnäkymiä. Nämä parannukset toteutettiin Help:in lukitusnäkymiä lukuun ottamatta vikojen korjausten jälkeen, jolloin taidot DNA-järjestelmän ohjelmoinnista olivat kehittyneet.

Virtauslaskurin nollauspainikkeen toteutus vaati joitain uusia lohkoja, sekä laskurilohkon uudelleen luonnin. Aiempi laskurilohko oli vastaava, kuin käyttöoppaassa [24] esitelty kuvan 16 lohko.



Kuva 16 Laskurilohko ilman reset-ominaisuutta [24]

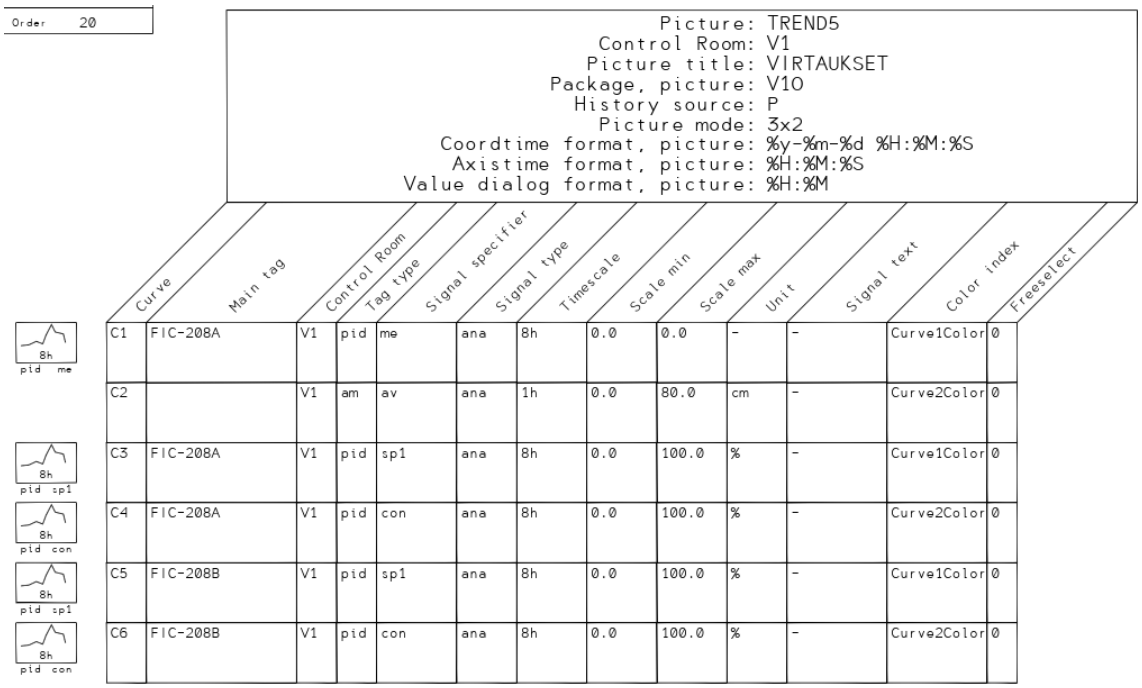
Tästä lohkosta puuttui clr-tulo, jolla laskurin saa nollattua aloitusarvoonsa. Tässä tapauksessa aloitusarvona oli oletusarvo 0. Uusi positiotunnus luotiin käyttöliittymää varten, joka nimettiin FQ-209R. Laskurin positiotunnuksena on FQ-209, joten perään lisättiin kirjain R merkitsemään reset-toimintoa. Nimeämiskäytäntöä ei tarkistettu mistään, vaan nimi perustuu täysin omaan arviooni. Käyttöliittymän puolelle tehtiin yksinkertainen painike, josta painamalla kuvan 17 lohko pr: FQ-209R lähettää eteenpäin arvon 1.



Kuva 17 Uusi laskurilohko ja reset-logiikka

Jotta arvo palautuisi takaisin nollaksi painamisen jälkeen ja näin ollen painike ei olisi kertakäyttöinen, luotiin pulssipiiri, joka aktivoituu samalla kun laskuri nollautuu. Laskurilohko on vaihdettu uuteen, enemmän toimintoja sisältävään laskurilohkoon. Asetukset ovat asetettu samoiksi kuin vanhassa lohkossa, mutta vaihtoehtoisia tuloja ja lähtöjä on enemmän. Käyttöoppaan [24] mukaan reset-tulo nollaa lähdön count lisäksi myös lähdön cumcount, mutta clr nollaa vain lähdön count. Koska järjestelmässä ei käytetty lähtöä cumcount mihinkään, molemmat nollaukset tekevät käytännössä saman. Nollaukseen käytimme tuloa clr, jotta myös toista laskurin arvoa voi käyttää tulevaisuudessa helpommin.

Toinen ja työllistävin parannus järjestelmään oli trendikuvien luonti. Trendikuvien kohdalla muusta DNA-ohjelmointikokemuksesta ei juurikaan ollut apua, sillä tapa luoda trendikuvia poikkesi FBCAD:in lohkoista. Trendikuvia yritettiin aluksi luoda operointikuvien muokkaukseen tarkoitetulla Picture Editor:lla, mutta se osoittautui lopulta niin työlääksi ja haastavaksi että aloimme etsiä vaihtoehtoista ratkaisua. Lopulta tähänkin tarkoitukseen löytyi valmis pohja FBCAD:ista ja uudet trendikaaviot saatiin luotua. Kuvassa 18 on esimerkkinä virtaussäädinten trendikuvien määrittelypohja.



Order 20

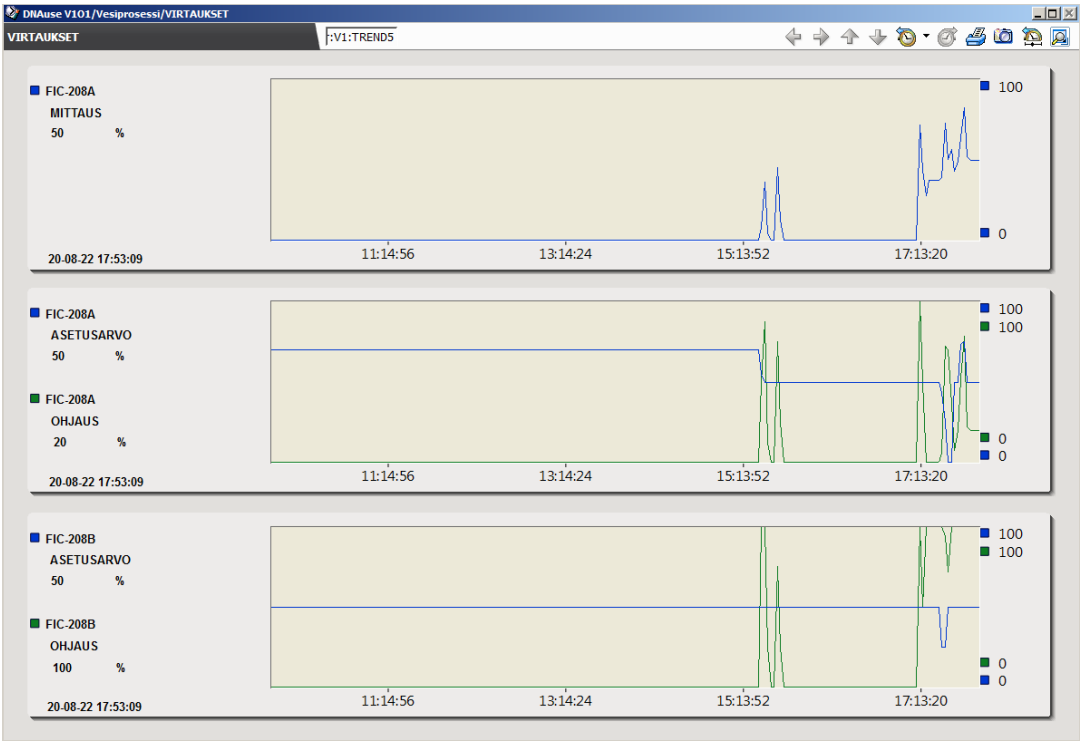
Picture: TREND5
Control Room: V1
Picture title: VIRTAUKSET
Package, picture: V10
History source: P
Picture mode: 3x2
Coordtime format, picture: %y-%m-%d %H:%M:%S
Axistime format, picture: %H:%M:%S
Value dialog format, picture: %H:%M

	Curve	Main tag	Control Room	Tag type	Signal specifier	Signal type	Timescale	Scale min	Scale max	Unit	Signal text	Color index	Freeselect
8h pid me	C1	FIC-208A	V1	pid	me	ana	8h	0.0	0.0	-	-	Curve1Color	0
	C2		V1	am	av	ana	1h	0.0	80.0	cm	-	Curve2Color	0
8h pid sp1	C3	FIC-208A	V1	pid	sp1	ana	8h	0.0	100.0	%	-	Curve1Color	0
8h pid con	C4	FIC-208A	V1	pid	con	ana	8h	0.0	100.0	%	-	Curve2Color	0
8h pid sp1	C5	FIC-208B	V1	pid	sp1	ana	8h	0.0	100.0	%	-	Curve1Color	0
8h pid con	C6	FIC-208B	V1	pid	con	ana	8h	0.0	100.0	%	-	Curve2Color	0

Kuva 18 Virtaustrendikuvan määrittelypohja

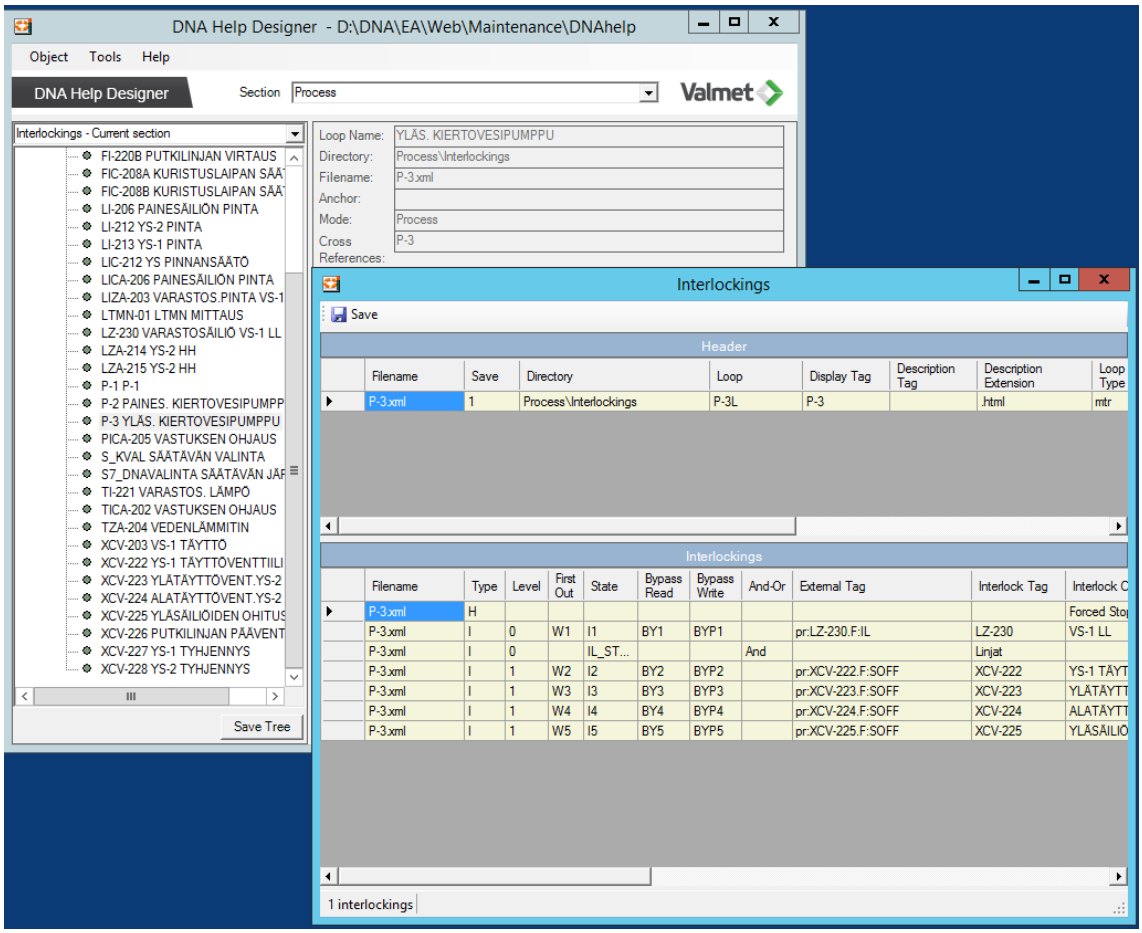
Pohjan täytön jälkeen haasteena oli saada uudet trendikuvat näkyviin operointinäkymässä. Havaittiin isojen ja pienten kirjainten merkitsevän kuvan tunnuksessa. Suurin osa haasteista ratkesi, kun tarkastimme tunnuksen oikeinkirjoituksen. Kuvat sai näkyviin yksinkertaisesti syöttämällä tunnuksen operointi-ikkunan osoitekenttään. Tämän jälkeen

loimme vielä päänäkymään linkkejä, jotta trendikuvia pääsee katsomaan ilman osoitteen kirjoitusta. Kuvassa 19 on laaditut trendikuvat virtaussäätimistä.



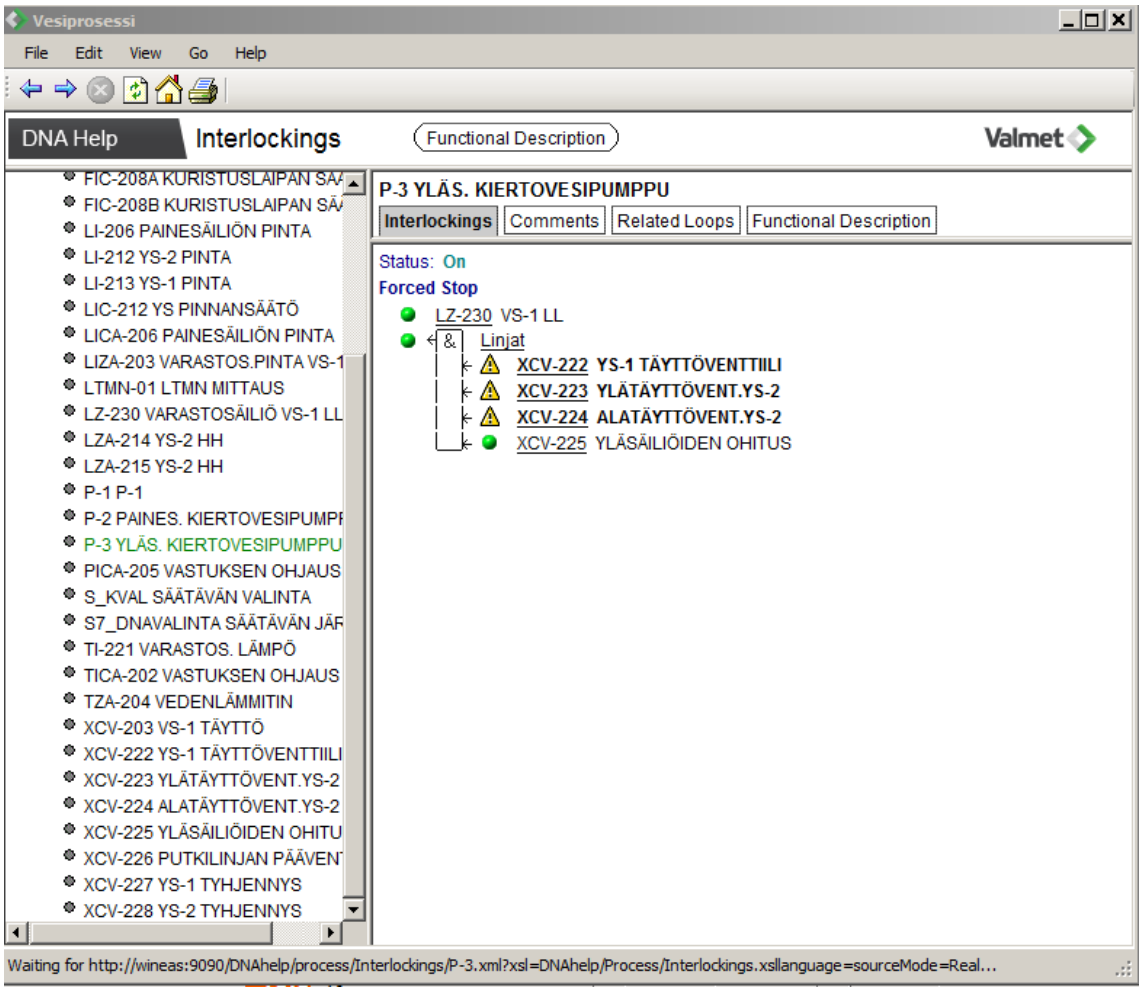
Kuva 19 Trendikuva virtaussäätimistä

Kolmas parannus toteutettiin vikojen korjauksen ohella. Tapana oli tarkastaa Help-sovelluksen lukitustietojen oikeellisuus jokaisen moduulin kohdalla. Pumpun P-3 kohdalla Help-ikkunan lukitukset olivat epäselvästi yhdessä listassa, vaikka pumpun lukitukseen vaaditaan kaikki yläsäiliöiden venttiilit kiinni tai varastosäiliön pinnan alaraja. Muokkasimme näkymää kuvaamaan lukituksia paremmin. Näkymiä muokattiin ohjelmalla Help Designer. Pumpun P-3:n lukitusten muokkausnäkymä on kuvassa 20.



Kuva 20 Help Designerin lukitusnäkymän muokkaus

Kuvan 20 mukaiset lukitusnäkymät näyttävät käyttäjän näkökulmasta kuvan 21 mukaisilta.



Kuva 21 P-3 Lukitusnäkymä

Myös monet muut Help-näkymät päivitettiin, mutta vain tässä tapauksessa näkymän rakennetta muutettiin. Kaikkien muiden moduulien kohdalla ajoimme automaattisen päivityksen, joka haki lukituskaavioista oikeat ajantasaiset lukitukset.

5.5 Järjestelmätestaus

Tässä luvussa esitellään projektissa käytetyt tavat järjestelmätestaukseen ja arvioidaan niiden toimivuutta. Kokeilimme ajaa järjestelmää sekä oikein, että väärin. Ajamalla järjestelmää väärin, testaus kohdistuu enemmän lukituksiin, joissa suurin osa vioista sijaitsi. Emme suunnitelleet järjestelmätestejä tarkasti etukäteen, vaan testasimme satunnaisesti toimintoja, jotka eivät toimineet ennen muutoksia.

Aloitimme järjestelmätestit ajamalla järjestelmää oikein. Kohteen kontekstissa tällä tarkoitetaan oikeiden venttiilien avausta, pumppujen käynnistystä ja järkeviä asetusarvoja säätimille. Havaitsimme tällöin erään säätimen toimintasuunnan olevan väärä ja palasimme vian korjaussykliin korjaaman tämän vian. Toimimme samoin muiden järjestelmätesteissä ilmenneiden vikojen kohdalla.

Kun vikoja ei enää ilmentynyt ajamalla järjestelmää järkevästi, siirryimme ajamaan järjestelmää väärin syöttein. Yhtenä esimerkkinä asetimme pumpun maksimiteholle ja avasimme reitin vain yhteen yläsäiliöön avaten venttiilit maksimille tarkoituksenamme testata järjestelmän vastetta ylivuotoon. Havaitsimme, että järjestelmän lukitukset olivat liian hitaita reagoimaan pienen säiliön täyttymiseen maksimaalisella virtauksella. Tällöin pääsimme kuivaamaan lattiaa ja pohtimaan voisiko vastetta nopeuttaa. Kyseinen vika johtui osittain yläraja-anturin puuttumisesta, jonka vuoksi lukituksen reunaehto johdettiin pinnanmittauksesta, jossa oli viivettä. Poistimme ylimääräisen viiveen ohjelmasta, mutta anturin asennus ei kuitenkaan kuulunut enää osuuteeni. Lukitukset kuitenkin toimivat oikein ja pääasiassa riittävän nopeasti. Löysimme myös muita uusia vikoja, kuten mahdollisuuden jättää paikalliskytkimien kautta ohjelmalle muistiin ajokäskyn. Tällöin ohjelma käynnistää pumpun välittömästi vaihdettaessa ohjaus paikallisajolle. Tämä vika korjattiin kuvassa 14 näkyvällä reset-lohkolla, kuten luvussa 5.3.2 esitellään. Tämä vika kuitenkin havaittiin vasta järjestelmätesteissä.

Järjestelmätestauksen olisi voinut suorittaa paljon systemaattisemmin. Satunnaisella testauksella on merkittävä riski, että jotain jää testaamatta ja siten vikoja jää järjestelmään. Olisi parempi käytäntö suunnitella testaukset kattavasti etukäteen ja pyrkiä varmistamaan vähintään kaikki kriittiset toiminnot. Hyvänä käytäntönä puolestaan on testata järjestelmän toimintaa sekä oikeilla, että väärillä syötteillä. Näidenkin osalta olisimme kuitenkin voineet mennä testeissä pidemmälle ja testata kattavammin järjestelmää. Jälleen järjestelmän koko antaa huonoja käytäntöjä anteeksi. Suuren kohdejärjestelmän kohdalla näin suuri järjestelmätestien painotus ja epäjärjestelmällisyys johtaisi huonoon testikattavuuteen ja siten järjestelmän laatua ei voitaisi varmistaa. Näin pienessä kohteessa suuri painotus järjestelmätesteille on vielä hallittavissa, mutta olisi tehokkaampaa käyttää laajemmin muita testauksen tasoja.

Kohteen laatu parantui kuitenkin merkittävästi työn tuloksena, eikä vikoja viimeisissä järjestelmätesteissä enää ilmentynyt. Järjestelmä oli työn lopussa jälleen valmis opetuskäyttöön. Järjestelmästä löytyi työtä tehdessä myös useita vikoja kenttälaitteiden ja kytkentöjen osalta. Näitä vikoja ei ole esitelty tämän työn yhteydessä.

6. YHTEENVETO

Työssä tavoitteena oli korjata WinNovan vesiprosessin ohjelmakoodit, sekä tutkia ohjelmistotekniikan viankorjaus tapojen soveltuvuutta automaatiossa. Kirjallisuusmateriaali oli pääasiassa kirjoitettu ohjelmistotekniikan näkökulmasta ja pyrin arvioimaan kirjallisuudessa esitettyjä tapoja vikojen korjaamiseen automaation näkökulmasta. Työn käytännön osuus suoritettiin ennen teoriaan ja vikojen korjaamisen käytäntöihin tutustumista ja siten poikkesi tyypillisestä opinnäytetyön etenemisestä. Samaisesta syystä kaikkea työssä ei tehty niin hyvin kuin olisi ollut mahdollista, mutta toisaalta tämä lähestymistapa toi uuden ja syvällisemmän näkökulman teoriaan. Työn teoria avautui käytännön työskentelyn jälkeen uudella tavalla ja testauksen tasojen merkitykset selkenivät.

Luvussa 2 esiteltiin pintapuolisesti automaatiojärjestelmien yleisiä rakenteita ohjelmiston näkökulmasta, sekä yleisiä ohjelmointikieliä automaatiossa. Luku koostui pääasiassa pintapuolisesta taustatiedosta, joka oli työn kohdejärjestelmän ymmärtämisen kannalta oleellista. Lisäksi automaatiojärjestelmien ohjelmointikielet usein poikkeavat merkittävästi perinteisemmistä IT-ohjelmointikielistä.

Luvussa 3 arvioitiin lähinnä ohjelmistotekniikan tapoja vikojen korjaukseen automaation näkökulmasta. Koska ohjelmistotekniikassa on paljon kirjallisuutta aiheeseen liittyen, olisi hyödyllistä, jos tätä kirjallisuutta voisi hyödyntää automaatiossa. Automaation ohjelmointikielet ovat kuitenkin niin erilaisia, että vain abstraktit vian korjauksesta kertovat lähteet olivat suoraan käyttökelpoisia automaatiossa. Näistä oli kuitenkin tunnistettavissa samoja perusperiaatteita, kuten vian paikallistaminen ja testaus. Tämä kertoo, että pohjimmiltaan samat käytännöt ovat käyttökelpoisia sekä IT-ohjelmistojen, että automaatio-ohjelmistojen kanssa. Yksityiskohtiin menevät kirjallisuuslähteet eivät enää olleet automaatio-ohjelmistoihin suoraan sovellettavissa.

Neljännessä luvussa esiteltiin kohteessa käytettävä automaatiojärjestelmä, Valmet DNA, ja sen suunnittelijatyökaluja niiltä osin, joita työssä käytettiin. Lähes kaikki käytäntö työssä tehtiin käyttämällä Valmet DNA:n suunnittelijatyökaluja apuna.

Viidennen luvun aiheena oli käytännön työ, eli WinNovan vesiprosessin ohjelmointivikojen korjaus. Luvussa esiteltiin työssä käytetyt tavat ja käytännöt ja arvioitiin näiden toimivuutta yleisesti, sekä kohteen kannalta. Toistuvasti arvioitiin kohteen koolla olevan merkitystä käytettyjen tapojen toimivuuteen. Epäjärjestelmällinen lähestymistapa voi toimia pienessä kohteessa, mutta suuressa kohteessa satunnaisesti testaaminen ei ole enää hallittavissa ja tärkeitä osia jää helposti testaamatta.

Työ oli opettavainen, mutta vaati paljon perehtymistä Valmet DNA:han. Lähes kaikki työssä vaaditut tiedot Valmet DNA:sta olivat peräisin Valmet DNA:n käyttöoppaasta [24]. Teoriaosuus työssä voi vaikuttaa hieman irralliselta, sillä teoriaosuudessa tutkittiin vian-korjausperiaatteita käytännön työn suorituksen jälkeen. Käytännön osuuden tapoja vertailtiin näihin teoreettisiin tapoihin ja tavoista löytyi pääasiassa yhtäläisyyksiä, mutta myös eroavaisuuksia. Suurin yhtäläisyys teoriaosuuden ja käytännön työn välillä oli vian paikallistaminen. Viat paikallistettiin työssä tarkasti oikeaan moduuliin ja moduulin sisällä oikeaan lohkoon, joka mahdollisti vikojen korjauksen helposti. Suurimmat eroavaisuudet hyviin käytäntöihin löytyivät järjestelmätestauksesta. Käytännön työssä käytetyt tavat järjestelmätestaukseen olivat huonoja, sillä testit eivät olleet systemaattisesti etukäteen suunniteltuja, vaan testit toteutettiin suunnitellen testejä testauksen aikana.

Vian korjaussykli toimi automaatiojärjestelmälle varsin hyvin, vaikka sykli pohjautuikin ohjelmistotekniikan tapoihin. Syklissä ei kuitenkaan suoraan oteta kantaa tapauksiin, joissa vika ei sijaitsekaan ohjelmistossa. Näitä on automaatiossa paljon ja samalla ohjelmistotekniikassa hyvin vähän. Näiden tapausten osalta luvussa 5.3 esitelty sykli ei toimi sellaisenaan, sillä laitteistoon kohdistuva vika edellyttää usein laitehankintoja tai huoltoa. Laitteistosta johtuvat viat voivat näkyä myös harhaanjohtavalla tavalla käyttöliittymässä, joka hankaloittaa paikannusta. Luvun 5.3 sykliä tulisikin yleisempää käyttöä varten laajentaa kattamaan myös laitteiston. Tässä työssä sykliin ei myöskään sisällytetty dokumentaation päivittämistä, vaikka se on tärkeä osa korjausprosessia. Muutosten yhteydessä muutokset kirjattiin paperille nopeasti ylös tarkoituksena päivittää dokumentaatio kerralla työn suorituksen jälkeen.

7. LÄHTEET

[1] R. Kumar, What is the five layer automation pyramid?, Medium, 2019, Saatavissa (viitattu 10/2020): https://medium.com/world-of-iot/92-what-is-the-five-layer-automation-pyramid-d0ccc1b903c3

[2] YJ. Reddy, BR. Mehta, Industrial process automation systems: design and implementation, Oxford, England, Butterworth-Heinemann, 2016.

[3] The Basics of PLC Operation, Technology Transfer, verkkosivu Saatavissa (viitattu 24.11.2020): https://www.techtransfer.com/blog/basics-plc-operation

[4] R. Ramanathan, "The IEC 61131-3 programming languages features for industrial control systems," 2014 World Automation Congress (WAC), Waikoloa, HI, 2014, pp. 598-603.

[5] IEC 61131-3:2013, IEC Webstore, verkkosivu Saatavissa (viitattu 24.22.2020): https://webstore.iec.ch/publication/62427

[6] K. John and M. Tiegelkamp, IEC 61131-3: Programming Industrial Automation Systems, 2nd Ed. Springer Inc., 2010.

[7] Status IEC 61131-3 standard, PLCopen, verkkosivu Saatavissa (viitattu 24.11.2020): https://plcopen.org/status-iec-61131-3-standard

[8] K. Thramboulidis, IEC 61499 as an Enabler of Distributed and Intelligent Automation: A State-of-the-Art Review-A Different View, Journal of Engineering 2013, 2013.

[9] B. Goetz and R. Eckstein, An Introduction to Real-Time Java Technology: Part 1, The Real-Time Specification for Java (JSR 1), Oracle, 2008, Saatavissa (viitattu 4.1.2021): https://www.oracle.com/technical-resources/articles/javase/jsr-1.html

[10] A-P. Tuovinen, Ohjelmistotestauksen periaatteita – Luento 1, Helsingin Yliopisto, 12.3.2018, Saatavissa (viitattu 8.1.2021): https://courses.helsinki.fi/sites/default/files/course-material/4544968/Ohj_testaus_2018_1.pdf

[11] "IEEE Standard Classification for Software Anomalies," in IEEE Std 1044-2009 (Revision of IEEE Std 1044-1993), vol., no., pp.1-23, 7 Jan. 2010, doi: 10.1109/IEEESTD.2010.5399061.

[12] M. Hamill and K. Goseva-Popstojanova, "Common Trends in Software Fault and Failure Data," in IEEE Transactions on Software Engineering, vol. 35, no. 4, pp. 484-496, July-Aug. 2009, doi: 10.1109/TSE.2009.3.

[13] K. Pan et al., Toward an understanding of bug fix patterns, Empirical software engineering: an international journal. 2008;14(3):286–315.

[14] Suomen Automaatioseura ry, Laatu automaatiossa: Parhaat käytännöt, 2001, s. 245.

[15] P. Butcher, Debug It, 1st ed, Pragmatic Bookshelf, 2009.

[16] M. Salmenperä, Automaation reaaliaikajärjestelmät – Luento 7, 3.11.2020, luento opintojaksolla Automaation reaaliaikajärjestelmät, Tampere, Tampereen Yliopisto

[17] M. Jamro, POU-Oriented Unit Testing of IEC 61131-3 Control Software, IEEE transactions on industrial informatics, 2015 Oct;11(5):1119–29.

[18] What is System Testing? Types & Definition with Example, Guru99, verkkosivu, Saatavissa (viitattu 26.1.2021): https://www.guru99.com/system-testing.html

[19] V-H. Rösch, A Light-Weight Fault Injection Approach to Test Automated Production System PLC Software in Industrial Practice, Control engineering practice, 2017 Jan;58:12–23.

[20] S. Lee et al., Automatic Detection and Update Suggestion for Outdated API Names in Documentation, IEEE transactions on software engineering, 2019 Feb 22;1–1.

[21] Valmet, Valmet DNA automation system, verkkosivu Saatavissa (viitattu 23.9.2020): https://www.valmet.com/automation/control-systems/valmet-dna/.

[22] Valmet DNA short overview presentation, Valmet, 2015

[23] J.Ruotsalainen, UX in Automation, Valmet, 11.2.2020, vierailuluento opintojaksolla Johdatus automaation tietotekniikkaan, Tampere, Tampereen yliopisto.

[24] Valmet DNA Manuals 2015, Valmet Automation, 2015

[25] PORTEK vesiprosessi: Prosessi- ja instrumentointikuvaus, Porin tekniikkaopisto, 1998

[26] Vesiprosessin vikalista, WinNova, 2017