Toni Niemi

Teollisuusnosturin PLC-koodin modernisointi

Metropolia Ammattikorkeakoulu

Insinööri (AMK)

Konetekniikan tutkinto-ohjelma

Insinöörityö
7.3.2022

Tiivistelmä

Tekijä:	Toni Niemi
Otsikko:	Teollisuusnosturin PLC-koodin modernisointi
Sivumäärä:	53 sivua + 2 liitettä
Aika:	7.3.2022
Tutkinto:	Insinööri (AMK)
Tutkinto-ohjelma:	Konetekniikan tutkinto-ohjelma
Ammatillinen pääaine:	Koneautomaatio
Ohjaajat:	Ohjelmistosuunnittelija, Olli Venho Lehtori, Maria Sjöholm

---

Opinnäytetyön tilaajana toimi Konecranes Finland Oy ja sen aiheena oli biomassan käsittelyyn tarkoitetun teollisuusnosturin PLC-koodin modernisointi. Alkuperäisen koodin alustana toimi Siemens Simatic S7-300 logiikkaohjain ja se oli kirjoitettu Simatic Manager -ohjelmointiympäristössä. Koodin modernisointi tehtiin käyttäen alustana Siemens Simatic S7-1500 logiikkaohjainta ja kirjoitettiin TIA Portal -ohjelmointiympäristössä.

Opinnäytetyö rajattiin koskemaan ainoastaan koodin automaatio-osuuksia. Näitä osuuksia oli kolme kappaletta ja niihin sisältyi yhteensä yli 100 funktiota. Työn päätavoitteiksi asetettiin koodin dynaamisuuden parantaminen sekä datan kulun selkeyttäminen. Lisäksi opinnäytetyössä tutustuttiin logiikkaohjelmoinnin teoriaan, hyviin käytäntöihin sekä eroihin alkuperäisessä ja uudessa projektissa käytettyjen logiikoiden ja ohjelmointiympäristöjen välillä. Opittuja tietoja käytettiin hyödyksi tavoitteiden saavuttamisessa.

Projektin aikatauluksi asetettiin 6 kuukautta, ja lähes kaikki funktiot ja niihin liittyvät datalohkot ehdittiin uudistamaan aikataulun puitteissa. Myös päätavoitteet onnistuttiin suurimmilta osin tavoittamaan, joskin käyttöliittymään liittyvien funktioiden ja datalohkojen suhteen jouduttiin tekemään kompromisseja. Tehtyjen muutosten ansioista nosturin projektikohtainen konfigurointi voidaan tehdä tulevaisuudessa nopeammin ja nosturin markkinointi asiakkaille voi olla helpompaa, sillä koodi taipuu joustavammin erilaisiin tarpeisiin ja projektikohtaisten muutosten teko on helpompaa.

Työ modernisoinnin parissa jatkuu edelleen, mutta ennen seuraavaan vaiheeseen siirtymistä on tarpeen tehdä päätös siitä, mitä nosturin varsinaisen pohjaohjelmiston suhteen halutaan tehdä. Vaihtoehtoina ovat pohjaohjelmiston päivittäminen kokonaan uuden sukupolven versioon tai alkuperäisen muokkaaminen tukemaan uutta mallia. Lisäksi on päätettävä, miten käyttöliittymän modernisoinnin suhteen halutaan edetä.

Avainsanat: PLC, Logiikkaohjain, Nosturi, Teollisuusnosturi, Siemens, S7-1500, S7-300, TIA Portal, Ohjelmointi

Abstract

Author:	Toni Niemi
Title:	Modernization of an Industrial Crane PLC Program Code
Number of Pages:	53 pages + 2 appendices
Date:	07 March 2022
Degree:	Bachelor of Engineering
Degree Programme:	Degree Programme in Mechanical Engineering
Professional Major:	Machine Automation
Supervisors:	Olli Venho, Software Designer Maria Sjöholm, Senior Lecturer

---

The thesis work was commissioned by Konecranes Finland Oy and the topic was the modernization of the PLC program code of an industrial crane used for biomass handling. The original code was made for Siemens Simatic S7-300 logic controller, using Simatic Manager as the programming environment. The modernization of the code was carried out using Siemens Simatic S7-1500 logic controller, with TIA Portal as the programming environment.

The thesis was limited to the sections of the code related to process automation. There were three of these sections, which comprised of over 100 functions in total. The main objectives for the thesis were to make the code more dynamic and to improve and clarify the data flow between functions. In addition, the thesis explored the theory and good practices of logic programming, and differences between the logic controllers and programming environments used in the original and the new project.

The project was set to take 6 months, and almost all the functions and related data blocks were re-written within the schedule. The main objectives were achieved for the most parts, although compromises had to be made on the functions and data blocks related to the user interface. As a result of the changes, project specific configuration of the crane can be done faster in the future, and it may also be easier to market the crane for customers, as the code is more flexible for different needs and it is easier to make project-specific changes.

Work continues with the modernization, but before moving on to the next stage of the project, it is necessary to decide on what to do with the base software of the crane. Alternatives are to upgrade the base software to a completely new next-generation version or to modify the current base software to support the new model designed for the automation-sections. In addition, it is necessary to decide how to proceed with the modernization of the user interface.

Keywords:	PLC, Programmable Logic Controller, Crane, Industrial Crane, Siemens, S7-1500, S7-300, TIA Portal, Programming

Sisällys

Lyhenteet

1 Johdanto 1

1.1 Yrityksestä 1
1.2 Työn aihe ja rajaus 1

2 Työn tavoitteet 2

2.1 Dynaamisuuden parantaminen 3
2.2 Datan kulun selkeyttäminen 4
2.2.1 Turhan datansiirtelyn vähentäminen 4
2.2.2 PLC datatyyppien käytön lisääminen 4
2.2.3 Parametrimuuttujien erittely ja absoluuttiarvojen poisto 4
2.2.4 Koodin luettavuuden parantaminen 5

3 Logiikkaohjelmointi 5

3.1 Logiikkaohjelmoinnin teoriaa 5
3.2 Ohjelmointikielet 6
3.2.1 Ladder (LAD) 7
3.2.2 Function Block Diagram (FBD) 8
3.2.3 Structured Control Language (SCL) 9
3.2.4 Statement List (STL) 12
3.2.5 Sekvenssikaavio (GRAPH) 13

4 Siemensin logiikat ja ohjelmointiympäristöt 15

4.1 Logiikat 15
4.2 Ohjelmointiympäristöt 16
4.2.1 Taulukoiden määrittely 16
4.2.2 Taulukoiden kutsuminen 17
4.2.3 PLC datatyyppien käyttö 18
4.2.4 Osoittajat 20

5 Toteutus 24

5.1 Nosturimalliin ja alkuperäiseen koodiin perehtyminen 24
5.2 Uuden projektin luominen 25

5.3 Migraatio S7-1500 alustalle 26
5.4 Funktioiden uudelleenkirjoittaminen 27
5.4.1 Master-tason datanpäivitykset 28
5.4.1.1 Liikennevalojen hallinta 28
5.4.1.2 Automaatioalueiden rajojen päivitys 29
5.4.1.3 Solualueen ja solujen koon määrittely 29
5.4.1.4 Solujen tilan päivitys 31
5.4.1.5 Solujen materiaalitasojen päivitys 33
5.4.1.6 Ulossyöttökohteiden tilojen ja kuormatietojen päivitys 34
5.4.2 Työsekvenssit ja niiden apufunktiot 35
5.4.2.1 Hälytykset ja tapahtumat 36
5.4.2.2 Sisääntulot 36
5.4.2.3 Kohteen tarkistus 37
5.4.2.4 Aloitusehtojen ja prioriteettien tarkistus 38
5.4.2.5 Sekvenssit 39
5.4.2.6 Ulostulot 42
5.4.3 Nosturin ajosekvenssit, reititys ja niiden apufunktiot 42
5.4.3.1 Rajojen ja reititysalueiden määrittely 43
5.4.3.2 Nosturin reititys 44
5.4.3.3 Aloitus- ja lopetusehdot 46
5.4.3.4 Monitorointi 46
5.4.3.5 Ajosekvenssit 47
5.4.3.6 Hälytykset 47
6 Pohdintaa 48
6.1 Tavoitteiden toteutuminen 48
6.1.1 Dynaamisuuden parantaminen 48
6.1.2 Datan kulun selkeyttäminen 49
6.1.2.1 Turhan datansiirtelyn vähentäminen 49
6.1.2.2 PLC datatyyppien käytön lisääminen 49
6.1.2.3 Parametrimuuttujien erittely ja absoluuttiarvojen poisto 50
6.1.2.4 Koodin luettavuuden parantaminen 51
6.2 Yhteenveto 51
Lähteet 52

Liitteet

Liite 1. Mallipohja SCL-kielellä toteutetulle tilakoneelle.

Liite 2. Taulukoiden kopioimiseen suunniteltu funktio.

Lyhenteet

PLC:	*Programmable Logic Controller, eli suomennettuna "logiikkaohjain"* tai lyhyesti *"logiikka"*. Laite, jonka avulla voidaan esimerkiksi ohjata muita laitteita, kerätä tietoa antureilta ja lähettää tietoa muille laitteille.
HMI:	Human Machine Interface. Laite, jonka avulla voidaan välittää tietoa PLC:ltä käyttäjälle ja käyttäjän antamia käskyjä PLC:lle.
TIA Portal	Totally Integrated Automation Portal. Siemensin ohjelmointiympäristö ohjelmoitaville logiikoille.
STL	Statement List. Siemensin käyttämä ohjelmointikieli.
SCL	Structured Control Language. Siemensin käyttämä ohjelmointikieli.
LAD	Ladder Logic. Siemensin käyttämä ohjelmointikieli.
FBD	Function Block Diagram. Siemensin käyttämä ohjelmointikieli.
GRAPH	Siemensin käyttämä ohjelmointikieli sekvenssien luomiseen.
I/O	Input/Output, eli suomeksi Sisääntulo/Ulostulo.

1 Johdanto

1.1 Yrityksestä

Opinnäytetyön tilaajana toimii Konecranes Finland Oy. Yritykseen tuotevalikoimaan kuuluvat mm. siltanosturit, satamalaitteet, trukit ja työpistenosturit [Konecranes -a]. Yritys tarjoaa myös paljon erilaisia kunnossapitoon liittyviä palveluita, sekä varaosia niin Konecranesin omiin kuin muidenkin merkkien nostureihin [Konecranes -b].

1.2 Työn aihe ja rajaus

Työn aiheena on biomassan käsittelyyn tarkoitetun teollisuusnosturin PLC-koodin modernisointi ja tuominen uudemmalle PLC-alustalle. CXT Biomass (Kuva 1) on täysautomatiikalla varusteltu nosturi, joka on suunniteltu käytettäväksi esim. bioenergian tai biokaasun tuottamiseen tarkoitetuissa laitoksissa.

Kuva 1. CXT Biomass -nosturi [Konecranes nosturiesite 2016].

Nosturin tehtävänä on varmistaa polttoaineen syötön jatkuvuus prosessin aikana. Polttoaineena voidaan käyttää esimerkiksi puuhaketta tai kotieläintilojen ulostetta. Lisäksi nosturin avulla on mahdollista vapauttaa lattiatilaa ja kasvattaa varastointikapasiteettia verrattuna kuljettimien käyttöön biomassan liikuttamisessa. Mallista riippuen, nosturin tyypillinen jänneväli on 15–27 metriä, ja nostokapasiteetti voi vaihdella aina 3,2 tonnista 17 tonniin saakka. Nosturin vakio-ominaisuuksiin kuuluu mm. älykäs heilunnannosto ja raskaimpaan käyttöön suunnitelluissa malleissa myös esimerkiksi jarruenergian talteenotto on mahdollista. Bioenergialaitosten nosturitarpeen määrittämiseen vaikuttavat monet erilaiset asiat, kuten saapuvan biomassan määrä vuorokaudessa, massan tyyppi ja tiheys, laitoksen polttokapasiteetti sekä varastoalueen rakenne. [Konecranes -c.]

Työssä käsitellään syitä muutoksen takana, tutkitaan miten Siemensin poistuva logiikkamalli S7-300 eroaa uudemmasta S7-1500-logiikasta sekä mitä etuja päivitys Siemens TIA Portal -ohjelmointiympäristöön tuo tullessaan. Työssä tutustutaan myös logiikkaohjelmoinnin teoriaan ja hyviin käytäntöihin. Opinnäytetyö on rajattu koskemaan nosturikoodin automaatio-osuutta. Lisäksi työssä pyritään huomioimaan käyttöliittämän aiheuttamat rajoitteet (jotta myöhempi päivitys tapahtuisi sujuvammin), vaikka käyttöliittymän modernisointi itsessään ei sisälly opinnäytetyöhön.

2 Työn tavoitteet

Asiakkaan toiveena on, että koodi kirjoitetaan puhtaalta pöydältä, säilyttäen kuitenkin suurimmalta osin alkuperäisen koodin toiminnallisuus. Uusi koodi kirjoitetaan Siemensin TIA Portal -ohjelmointiympäristössä, käyttäen alustana Siemens S7-1500 ohjelmoitavaa logiikkaa. Työn päätavoitteina ovat dynaamisuuden parantaminen sekä datan kulun selkeyttäminen.

2.1 Dynaamisuuden parantaminen

Funktioiden ja datalohkojen dynaamiseksi päivittämisen tarkoituksena on helpottaa projektikohtaisten muutosten tekemistä. Alkuperäinen koodi on tehty Simatic S7-300 -alustan ja Simatic Manager -ohjelmointiympäristön rajoitteiden puitteissa. Yksi alkuperäisen koodin ongelmista on, että sen funktioihin ja datalohkoihin liittyvät resurssit on suurimmalta osin määritelty käyttäen paikallisia vakioarvoja. Tästä johtuen, mikäli johonkin projektikohtaiseen resurssiin on haluttu tehdä muutoksia, on muutos täytynyt käydä manuaalisesti tekemässä jokaisessa paikassa, jossa kyseistä resurssia on käytetty. Tällöin muutosten teko projektikohtaisiin resursseihin on ollut paitsi työlästä, myös altista virheille, mikäli joku ohjelmista tai datalohkoista on epähuomiossa jäänyt päivittämättä. Toinen osa alkuperäisen koodin ongelmaa on STL-kielen käyttö, jota alkuperäisessä koodissa on käytetty mm. silmukoiden ja tilakoneiden rakentamiseen. STL-kielen mahdollistama pointtereiden, eli muistiosoittimien käyttö sitoo ohjelman rakenteen ja datalohkon rakenteen kiinteästi toisiinsa, sillä ohjelma viittaa muuttujiin numeerisilla muistiosoitteilla. Datalohkon rakenteen muuttuessa käytetyt muistiosoitteet eivät enää päde, jolloin esimerkiksi silmukassa viitataan vääriin muuttujiin. Suorien muistiosoitteiden käyttö tekee myös ohjelmakoodin lukemisesta sekavaa.

Tavoitteena onkin kirjoittaa alkuperäisen mallin ohjelmat uudelleen puhtaalta pöydältä, käyttäen tilanteen mukaan SCL-, LAD tai FBD-kieliä. Myös datalohkojen rakenteet on päivitettävä dynaamisiksi, ja kaikille projektikohtaisille resursseille on luotava taulukot, joiden rajat on määritelty globaaleilla vakioilla. Samoja globaaleja vakioita on käytettävä kaikissa samaan resurssiin liittyvissä datalohkoissa ja funktioissa. Tämän ansiosta globaalivakioihin tehdyt muutokset päivittyvät automaattisesti jokaiseen niitä käyttävään taulukkoon ja funktioon, joten yksi muutos koodissa riittää.

2.2 Datan kulun selkeyttäminen

2.2.1 Turhan datansiirtelyn vähentäminen

CXT Biomassan koodiin on jäänyt paljon historian painolastia, ja sen vuoksi dataa kopioidaan usein tarpeettomasti paikasta toiseen. Syynä on alkuperäisen PLC-koodin pohjautuminen erään toisen nosturimallin koodiin, jossa dataa on kommunikoitu usean eri PLC:n välillä. CXT Biomassa tukee kuitenkin vain yhtä nosturia ja sen koodi on muokattu toimimaan yhdellä PLC:llä. Tavoitteena onkin selkeyttää datan kulkua tunnistamalla turhat väliaikais-datalohkot ja poistaa koodista ylimääräiset datan siirtelyt paikasta toiseen.

2.2.2 PLC datatyyppien käytön lisääminen

Niissä tilanteissa, jolloin datan kopiointi kuitenkin on tarpeellista, tavoitteena on lisätä PLC datatyyppien käyttöä. PLC datatyyppien avulla voidaan saavuttaa samankaltaisia etuja kuin globaaleilla muuttujilla projektikohtaisissa resursseissa, eli mikäli samaa datarakennetta käytetään useammassa paikassa, päivittyvät PLC datatyyppiin tehdyt muutokset automaattisesti kaikkiin sitä käyttäviin datalohkoihin. PLC datatyyppien avulla voidaan myös vähentää funktioiden sisään/ulostulojen määrää, kun useita muuttujia voidaan tuoda funktioon yhdellä kertaa.

2.2.3 Parametrimuuttujien erittely ja absoluuttiarvojen poisto

Konecranesin koodauskäytäntöjen mukaan datalohkojen sisällön tyyppi tulisi olla määriteltynä jo niiden nimessä [Konecranes PLC programming guide 2020]. Esimerkiksi nosturin eri koneistojen maksiminopeuksia sisältävä datalohko voitaisiin nimetä ”Maksiminopeudet _P”, kun taas nosturin koneistojen sen hetkisiä nopeuksia tallentava datalohko voitaisiin nimetä ”KoneistojenNopeudet_V”. Alkuperäisessä koodissa tätä käytäntöä ei kuitenkaan aina ole noudatettu, ja tavoitteena onkin löytää ne datalohkot, joissa käytäntö ei toteudu, ja eriyttää parametrityyppiset muuttujat ja päällekirjoitettavat muuttujat omiin datalohkoihinsa.

Muutoksen ansiosta erityisesti projektikohtaisten räätälöityjen toimintojen ohjelmointi helpottuu, kun ohjelmoijan ei tarvitse tuhlata aikaa selvittääkseen voiko parametrejä sisältävän datalohkon muuttujan alkuarvoon luottaa, vai onko sen päälle kirjoitettu jossain vaiheessa jotain muuta.

Toisena ongelmana ovat ohjelmakoodiin kirjoitetut absoluuttiarvoilla kirjoitetut parametrit. Koska absoluuttiarvoille ei ole symbolisia vastineita, täytyy ne aina etsiä ohjelmakoodista erikseen. Kun absoluuttiarvoille luodaan omat muuttujansa, jotka sijoitetaan asiayhteydeltään sopivaan parametri-datalohkoon, saadaan käyttöönotossa projektikohtaisten muutosten tekoa nopeutettua.

2.2.4 Koodin luettavuuden parantaminen

Viimeisenä tavoitteena on parantaa koodin luettavuutta. Nykyisellään Biomassan koodista löytyy paljon asioita, jotka saattavat helposti hämmentää käyttäjiä. Esimerkiksi samasta asiasta saatetaan käyttää eri sijainneissa eri termejä ja datalohkoista löytyy myös paljon muuttujia, joita ei käytetä mihinkään. Koodista löytyy myös ominaisuuksia, joista käytetään väärää nimeä. Tavoitteena on tunnistaa nämä kohdat ja tehdä tarvittavat muutokset. Käyttämättömät muuttujat on poistettava datalohkoista ja funktioiden uudelleenkirjoittamisen yhteydessä myös koodin kommentointia on yleisesti parannettava. Terminologiaa on yhtenäistettävä siten, että kaikkialla samasta asiasta käytetään samaa nimitystä ja nosturin ominaisuuksia kutsutaan niiden oikeilla nimillä. Muuttujien nimeämisen suhteen tulee noudattaa ns. ”itsedokumentoitavuuden” periaatetta, eli muuttujan nimestä itsestään tulisi käydä mahdollisimman hyvin ilmi, mihin tarkoitukseen muuttujaa käytetään.

3 Logiikkaohjelmointi

3.1 Logiikkaohjelmoinnin teoriaa

Logiikkaohjelmointi on tietokoneohjelmoinnin suuntaus, jonka avulla voidaan luoda säännöt jonkin laitteen ohjaamista varten. Logiikkaohjelman voidaankin

katsoa olevan laitteen aivot, jonka perusteella se pystyy tekemään loogisia päätöksiä erilaisissa toimintatilanteissa. Logiikkaohjelma koostuu yksittäisistä ohjelmista, jotka puolestaan muodostuvat yksittäisistä ohjelmalausekkeista. Lausekkeet voivat olla tyypiltään sääntöjä tai toteamuksia. Säännöt koostuvat ehdosta ja seurauksesta, esimerkiksi ”jos muuttuja A on tosi TAI muuttuja B on tosi, suorita aliohjelma X”. Toteamuksissa ehtolausetta ei ole, esimerkiksi ”muuttuja X on tosi”. Logiikkaohjelmoijan vastuulla on varmistaa, että muuttujien väliset syy-seuraus-suhteet ovat loogisia keskenään, jotta tietokone pystyy suorittamaan halutut toimenpiteet. [Reed 2022.]

3.2 Ohjelmointikielet

Siemens S7-1500 tukee yhteensä viittä eri ohjelmointikieltä, jotka on lueteltu taulukossa 1. Taulukossa näkyy myös kunkin ohjelmointikielen tyyppi (graafinen vai tekstipohjainen), sekä kielien vastineet IEC 61131-3 standardissa. [Siemens 2013 -a.]

Nimi (lyhenne) - TIA Portal	**Tyyppi**	**Nimi (lyhenne) - IEC 61131-3**
Ladder Logic (LAD)	Graafinen	Ladder Diagram (FD)
Function Block Diagram (FBD)	Graafinen	Function Block Diagram (FBD)
Structured Control Language (SCL)	Tekstipohjainen	Structured Text (ST)
Statement List (STL)	Tekstipohjainen	Instruction List (IL)
Graph	Graafinen	Sequential Function Chart (SFC)

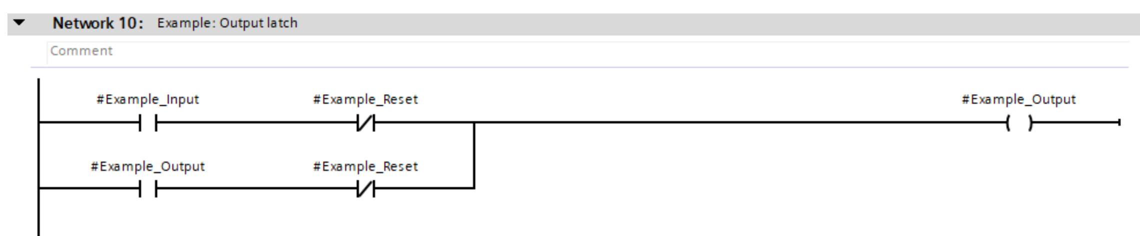
Taulukko 1. Ohjelmointikielet TIA Portalissa.

Standardissa määriteltyjen kielien määrittelyistä vastaa PLCOpen-organisaatio, joka on perustettu vuonna 1992 [PLCopen -a]. Organisaatio myöntää erilaisia sertifikaatteja sen perusteella, miten hyvin laitevalmistajien PLC-ohjelmointiympäristöissä käytetyt kielet ovat linjassa standardin määritysten kanssa [PLCopen -b].

3.2.1 Ladder (LAD)

Ladder-logiikka on graafinen ohjelmointikieli, jonka esitystapa muistuttaa monilta tavoin sähkölaitteiden asennuksissa käytettäviä sähkökaavioita. Joskus ladder-logiikasta puhuttaessa saatetaan käyttää suomenkielistä termiä ”relekaavio”, joka kuvastaa hyvin kieleen liittyvää historiaa, sillä alun perin ladder-logiikkaa käytettiin kuvastamaan prosesseja ohjaavien releiden kytkentöjä erilaisiin antureihin ja toimilaitteisiin. PLC-tietokoneiden keksimisen myötä relekaavioiden pohjalta luotiin kieli niiden ohjelmointiin. Ladder-logiikasta tulikin nopeasti suosittu ohjelmointikieli, sillä esitystapa oli ennestään tuttu alan ihmisten keskuudessa, eikä yritysten täten tarvinnut kuluttaa suuria määriä rahaa henkilöstön kouluttamiseen. Ladder-logiikka soveltuu parhaiten binääristen operaatioiden kuvastamiseen ja se on vielä tänä päivänäkin yksi suosituimmista PLC-tietokoneiden ohjelmoinnissa käytetyistä kielistä. [Cleverism.]

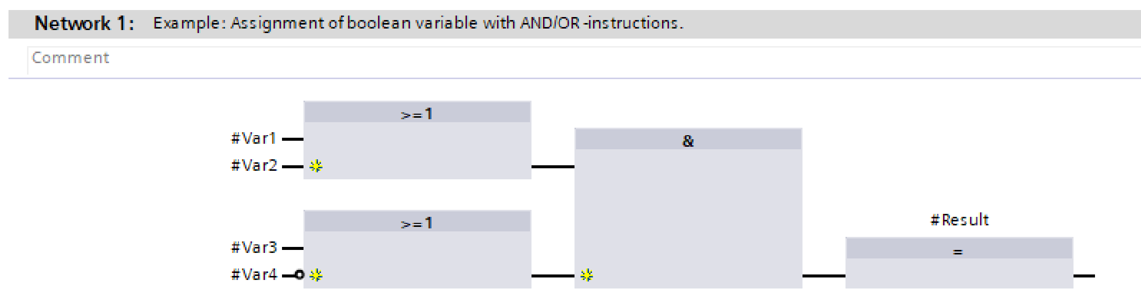
Kuten sähkökaavioissa, myös ladder-logiikassa käytetään erilaisia symboleita kuvastamaan tehtäviä operaatioita. Esimerkiksi avoimet ja suljetut kontaktorit (open / closed contact) kuvastavat ehtolauseita ja käämeillä (coil) kuvataan ehtolauseiden seuraamuksena aktivoitavia muuttujia. Koodi koostuu askelmista, joita luetaan ensin vasemmalta oikealle ja sitten ylhäältä alas. Peräkkäin asetetuilla ehtolauseilla muodostetaan JA-ehdot, kun taas allekkain asetetuilla ehdoilla voidaan muodostaa TAI-ehtoja. Esitystapojen voidaan huomata muistuttavan sähkökaavioiden sarjaan- ja rinnankytkentöjä. Kuvan 2 esimerkissä demonstroidaan yksinkertaista menetelmää, jolla ulostulo on mahdollista lukita tiettyyn tilaan LADDER-kielen TAI-ehtoa käyttäen. Kuvassa ulostulon nollaus (reset) tapahtuu ”Example_Reset” -muuttujan avulla. [Peter 2017.]



Kuva 2. Ulostulon lukitus LADDER-kielellä.

3.2.2 Function Block Diagram (FBD)

Toinen IEC 61131-3 standardissa määritelty graafinen kieli on nimeltään Function Block Diagram. Sen toiminta perustuu erilaisten toimintalohkojen yhdistelemiseen, joille on määritelty tietyt sisääntulot ja ulostulot riippuen sen suorittamasta toiminnasta. Yleisimpiä toimintalohkoja ovat JA tai TAI-lohkot, joiden toimintaa demonstroidaan kuvassa 3. Kuvassa muuttujan ”Result” tila asetetaan muuttujien ”Var1” – ”Var4” tilojen perusteella. Esimerkissä muuttujat Var1 ja Var2 on asetettu yhden TAI-operaation taakse ja Var3 ja Var4 toisen TAI-operaation taakse. Kummankin TAI-operaation lopputulos on yhdistetty JA-lohkoon, eli muuttuja ”Result” saa tilan ”tosi” vain mikäli molemmat TAI-operaatiot toteutuvat. Esimerkistä voidaan myös huomata, että muuttujan ”Var4” perään on asetettu pieni pallo, joka tarkoittaa muuttujan negaatiota, eli muuttujan tila käännetään sen vastakohdaksi (tosi → epätosi, epätosi → tosi).



Kuva 3. JA- ja TAI-lohkojen käyttö FBD-kielellä.

Joskus FBD-kieltä (tai vastaavasti edellisessä kappaleessa mainittua LAD-kieltä) voidaan käyttää myös pelkkään ohjelman kutsumiseen. Esimerkiksi käyttäjän itse kirjoittama funktio voi olla kirjoitettu tekstipohjaisella SCL-kielellä, mutta sitä kutsutaan FBD-lohkon sisältä toimintalohkona, jolloin eri funktioiden väliset prioriteetit ja järjestykset voidaan esittää helposti ymmärrettävässä graafisessa muodossa. [Peter 2018.]

3.2.3 Structured Control Language (SCL)

Structured Control Language, lyhyesti SCL, on Siemensin vastine IEC 61131-3 -standardin mukaiselle Structured Text (ST) -ohjelmointikielelle. Se on tekstipohjainen ohjelmointikieli, joka perustuu Pascal-ohjelmointikieleen, ollen syntaksiltaan hyvin samankaltainen.

TIA Portal tarjoaa käyttäjälle kaksi erilaista mahdollisuutta SCL-kielen käyttöön. Funktio on mahdollista määritellä käyttämään SCL-kieltä joko funktion luontihetkellä (jolloin koko kyseinen funktio on kirjoitettava SCL-kielellä) tai vaihtoehtoisesti käyttäjä voi lisätä LAD, FBD tai STL-kielillä luotuihin funktioihin yksittäisiä SCL-komentolohkoja (Network). Yksittäisten komentolohkojen käyttö on perusteltua esimerkiksi silloin, kun muutoin yksinkertaisista binäärioperaatioista koostuvassa ohjelmassa ilmenee tarve yksittäisen silmukan luomiselle. Tällöin koodista voidaan saada kokonaisuutena selkeämpää käyttämällä binäärioperaatioihin esimerkiksi LAD- tai FBD-kieltä ja kirjoittamalla silmukka yksittäiseen SCL-komentolohkoon.

Yksittäisten SCL-komentolohkojen käytössä on kuitenkin joitain rajoitteita, jonka vuoksi funktion määrittely kokonaan SCL-kieliseksi on ainoa tapa, jolla kaikkia SCL-kielen ominaisuuksia päästään hyödyntämään. Yksi tällaisista hyödyistä on mahdollisuus sisällysluettelon luontiin alueiden (REGION) avulla. Alueita merkitään koodissa REGION ja END_REGION-komentojen väliin, ja alueita on myös mahdollista jäsennellä sisäkkäin. Merkatut alueet ilmestyvät näkyviin SCL-funktion vasemmalle puolelle sisällysluettelon tapaan. Luettelon eri kohtia klikkaamalla käyttäjä siirtyy suoraan kyseiseen kohtaan koodissa. Alueita on mahdollista luoda myös yksittäisiin SCL-komentolohkoihin, mutta yksittäiset komentolohkot eivät generoi alueiden perusteella sisällysluetteloa, joka allekirjoittaneen mielestä on niiden tärkein hyöty. Sisällysluettelon avulla onkin mahdollista selkeyttää huomattavasti monimutkaisten funktioiden sisällä navigointia. Alueiden käytössä on kuitenkin otettava huomioon, että ne on määriteltävä tietyllä tapaa muihin komentoihin nähden. Aluetta ei voi esimerkiksi aloittaa kes-

ken IF-lauseen ja lopettaa IF-lauseen jälkeen, vaan se on sijoitettava joko kokonaan IF-lauseen ulkopuolelle, tai kokonaan IF-lauseen sisään. CASE-lauseen kanssa taas yksittäisiä tapauksia (case) ei voi laittaa alueiden sisään, vaan alueet on sijoitettava tapauksien sisään. Tätä demonstroidaan esimerkkikoodeissa 1 ja 2. Koodeista jälkimmäinen aiheuttaisi syntaksivirheen, sillä CASE-lause vaatii toimiakseen vähintään yhden tapauksen, jota se ei tässä tapauksessa pystyisi löytämään.

```
CASE some_case OF

#CASE_1:
REGION First_Region
; // Regions within cases work
END_REGION

#CASE_2:
REGION Second_Region
; // Still working
END_REGION

END_CASE;
```

Esimerkkikoodi 1. Alueiden käyttö CASE-lauseen sisälllä. Oikea tapa.

```
CASE some_case OF
REGION First_Region
CASE_1:
; (* Case within region will
     result in Jump label instead
     of a proper case *)
END_REGION

REGION Second_Region
CASE_2:
; // Still not working
END_REGION

END_CASE;
```

Esimerkkikoodi 2. Alueiden käyttö CASE-lauseen sisälllä. Väärä tapa

Kokonaan SCL-kielellä kirjoitetut funktiot mahdollistavat myös ns. hyppyotsikoiden (Jump label) ja hyppykomennon (GOTO) käytön, joka yksittäisissä SCL-komentolohkoissa on estetty. Hyppyotsikoiden ja hyppykomentojen käytössä (ohjelmointikielestä välittämättä) tulee kuitenkin käyttää harkintaa, sillä väärin käytettynä se voi helposti johtaa vaikeasti luettavaan ja ylläpidettävyyden kannalta huonoon koodiin. Hyppykomentojen käyttö jakaakin vahvasti mielipiteitä, ja monien ohjelmoijien mielestä niitä ei tulisi käyttää lainkaan. Tämä johtaa juurensa aina vuoteen 1968, jolloin Edgar Dijkstra julkaisi vaikutusvaltaisen esseen otsikolla ”Go To Statement Considered Harmful”. Hyvänä nyrkkisääntönä voidaankin pitää, että goto-käskyllä liikuttaisiin koodissa aina vain eteenpäin, ei koskaan taaksepäin, jotta vältetään riski loputtoman silmukan luomisesta. Silmukoiden rakentamiseen kannattaa käyttää vain sisäänrakennettuja komentoja kuten ”FOR” ja ”WHILE”, jotka pystyvät sisäänrakennettujen virheenetsintämekanismiensa perusteella havaitsemaan (joskaan eivät täydellisesti) koodista loputtomia silmukoita, ja estämään ohjelman kääntämisen, kunnes virhe on korjattu. Mitkä sitten voisivat olla tilanteita, joissa hyppykomentojen käyttö on perusteltua? Tällaisiin voisi lukeutua esimerkiksi virhetilanteita sisältävän funktion yhtenäinen virheenhallinta, jolloin kustakin mahdollisesta virhetilanteesta voitaisiin hypätä aina samaan sijaintiin funktion lopussa. Toiseksi esimerkiksi voitaisiin ottaa moniulotteisesta silmukasta poistuminen, jota demonstroidaan esimerkkikoodeissa 3 ja 4. Esimerkkikoodi 3 demonstroi silmukoista poistumista ehdollisten EXIT-komentojen kanssa, kun taas esimerkkikoodi 4:ssä poistumiseen käytetään GOTO-komentoa. Koodinpätkiä vertaamalla voidaan huomata, että tässä tapauksessa GOTO-komentoa käyttämällä voidaan saavuttaa lyhyempi ja selkeämpi lopputulos. Toki kolmantena vaihtoehtona olisi sijoittaa silmukka omaan, pääfunktiosta käsin kutsuttavaan funktioonsa, jolloin silmukasta olisi mahdollista poistua RETURN-komennolla. [Cross 2020.]

```
FOR #loop_X := #LowLimit_X TO #HighLimit_X DO
     FOR #loop_Y := #LowLimit_Y TO #HighLimit_Y DO
          FOR #loop_Z := #LowLimit_Z TO #HighLimit_Z DO
               IF #ArrayOfBool[#loop_X , #loop_Y , #loop_Z] THEN
                    #ResultVar := TRUE;
                    #TargetFound := TRUE;
                    EXIT;
               END_IF;
          END_FOR;
          IF #TargetFound THEN
               EXIT;
          END_IF;
     END_FOR;
     IF #TargetFound THEN
          EXIT;
     END_IF;
END_FOR;
```

Esimerkkikoodi 3. Moniulotteisesta silmukasta poistuminen käyttäen IF-lauseella ehdollistettuja EXIT-komentoja.

```
FOR #loop_X := #LowLimit_X TO #HighLimit_X DO
     FOR #loop_Y := #LowLimit_Y TO #HighLimit_Y DO
          FOR #loop_Z := #LowLimit_Z TO #HighLimit_Z DO
               IF #ArrayOfBool[#loop_X , #loop_Y , #loop_Z) THEN
                    #ResultVar := TRUE;
                    GOTO EndOfLoops;
               END_IF;
          END_FOR;
     END_FOR;
END_FOR;

// Program continues here after loop condition has been fulfilled:
EndOfLoops:
```

Esimerkkikoodi 4. Moniulotteisesta silmukasta poistuminen GOTO-komennolla.

3.2.4 Statement List (STL)

Statement List on Siemensin vastine IEC 61131-3-standardin mukaiselle Instruction List -kielelle. Se on tekstipohjainen ohjelmointikieli, joka on ladder-logiikan ohella yksi vanhimmista PLC-tietokoneiden ohjelmointiin käytetyistä kielistä. Se on niin sanottu alhaisen tason ohjelmointikieli, eli se muistuttaa luonteeltaan hyvin paljon tietokoneen prosessoreiden käyttämää konekieltä. Kuten kielen nimestä voidaan päätellä, jokainen käsky suoritetaan omalla rivillään ja toiminta

perustuu niin sanottujen akkumulaattoreiden, eli väliaikaisten muistipaikkojen, ja osoiterekisterien käyttöön. [Collins 2019.]

Muistipaikkojen käyttöä demonstroidaan kuvassa 4, jossa vertaillaan kahta 16-bittistä kokonaislukua. Esimerkissä muuttuja ”Int1” ladataan muistipaikkaan ACC1, jonka jälkeen sama tehdään muuttujalle ”Int2”. Toista muuttujaa ladatessa, siirretään ACC1:ssä valmiiksi oleva muuttuja (tässä tapauksessa ”Int1”) automaattisesti muistipaikkaan ACC2. Tämän jälkeen suoritetaan vertailuoperaatio, joka vertaa onko ACC2:n sisältämä muuttuja suurempi kuin ACC1:n sisältämä. Lopuksi operaation lopputulos (tosi/epätosi) sijoitetaan muuttujaan ”ComparisonResult”.

Network 3: Example: Comparing two 16-bit integers

Comment

```
1    L    #Int1                // Load variable into ACC1
2    L    #Int2                // Load variables into ACC1, move existing ACC1 variable into ACC2
3    >I                        // Compare if variable in ACC2 is greater than variable in ACC1
4    =    #ComparisonResult     // Result of the comparison
5
```

Kuva 4. Kahden kokonaisluvun vertailu STL-kielellä.

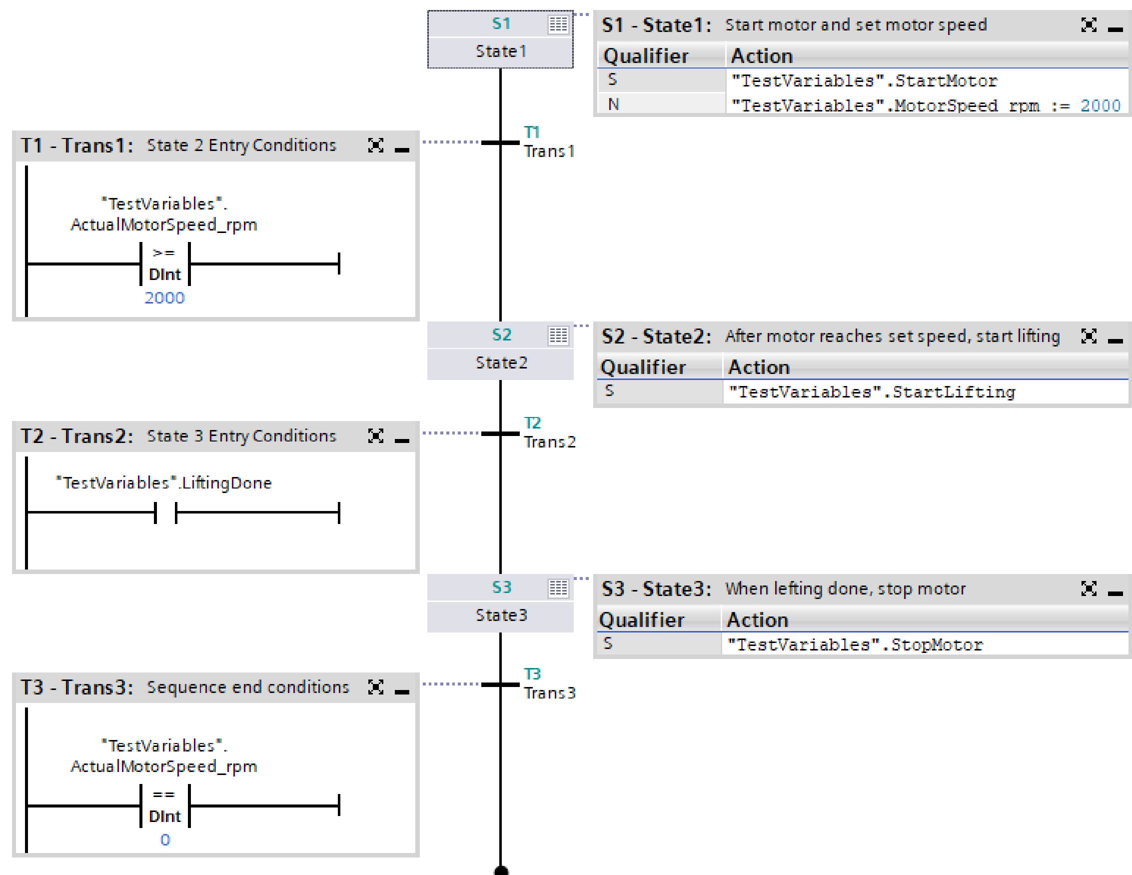
Kielen etuna on hyvä suorituskyky ja tehokas muistinkäyttö, josta syystä sitä on menneisyydessä käytetty paljon mm. silmukoiden rakentamiseen. Sen huonona puolena on kuitenkin moniin muihin kieliin verrattuna vaikeampi luettavuus, josta syystä mahdollinen vianselvitys voikin olla huomattavasti hankalampaa, varsinkin mikäli kielen syntaksi ei ole ennestään tuttu. Tästä syystä kieli onkin IEC-standardin uusimmassa versiossa merkitty vanhentuneeksi, eikä sen käyttöä enää suositella. Tästä syystä onkin mahdollista, että tulevaisuudessa laitevalmistajat saattavat poistaa tuen kyseiselle kielelle kokonaan. [Collins 2019.]

3.2.5 Sekvenssikaavio (GRAPH)

Siemensin alustoilla sekvenssikaavioita kutsutaan nimellä GRAPH. Se on sekvenssien rakentamiseen tarkoitettu graafinen ohjelmointikieli, joka vastaa IEC 61131-3 standardissa määriteltyä Sequential Function Chart (SFC) -kieltä.

GRAPH ohjelmat rakennetaan askelten (step) ja siirtymien (transition) ympärille. Askelten sisään määritellään kussakin ohjelman vaiheessa tehtävät operaatiot. Siirtymiin taas määritellään ehdot niiden välillä siirtymiseen. Sekvensseille on myös mahdollista määritellä yhtäaikaisia tai vaihtoehtoisia polkuja.

Kuvan 5 esimerkkiohjelmassa demonstroidaan askelten ja siirtymien käyttöä. Esimerkkisekvenssin ensimmäisessä askeleessa moottori käynnistetään ja sille asetetaan pyörimisnopeus. Seuraavaan askeleeseen siirtymisehtona on, että moottorin on saavutettava sille määritelty pyörimisnopeus. Kun pyörimisnopeus on saavutettu, aloitetaan nosto-operaatio. Noston ollessa valmis (LiftingDone) moottori sammutetaan. Ennen sekvenssin päättämistä, funktio varmistaa, että moottori on täysin pysähtynyt.



Kuva 5. GRAPH-kielellä tehty esimerkkisekvenssi.

4 Siemensin logiikat ja ohjelmointiympäristöt

Tässä kappaleessa esitellään lyhyesti alkuperäisessä ja uudessa koodissa alustoina käytetyt logiikat sekä niiden ohjelmointiin käytetyt alustat, ja käydään läpi eroavaisuuksia niiden välillä.

4.1 Logiikat

Simatic S7-300 on Siemensin vuonna 1994 julkaisema teollisuuskäyttöön tarkoitettu logiikkaohjain, ja se on historiallisesti yksi käytetyimmistä logiikoista ympäri maailman [OEM]. S7-300-tuotesarjan aktiivimyynnin on ilmoitettu jatkuvan vuoteen 2023 asti, mutta niiden myynti varaosiksi jatkuu aina vuoteen 2033 asti [Siemens -a].

Simatic S7-1500 taas edustaa Siemensin uusinta PLC-tietokoneiden sukupolvea, ja se julkaistiin vuonna 2013 [Siemens 2013 -b]. Päältäpäin katsoessa kenties näkyvin ero logiikkojen välillä on S7-1500:aan lisätty fyysinen käyttöpaneeli, joka sisältää pienikokoisen näytön ja nuolinäppäimistön valintanappeineen. Sen avulla on mahdollista tehdä perustason diagnostiikkaa ja konfigurointeja, kuten esimerkiksi määritellä IP-osoitteita, ilman tarvetta erillisille ohjelmointityökaluille. [Realpars.]

Ohjelmallisia eroja mietittäessä, yksi suurimmista muutoksista S7-300- ja S7-1500-logiikoiden välillä on niiden tapa käsitellä funktioita. Vanhalla S7-300-alustalla esimerkiksi LAD- ja FBD-ohjelmat käännettiin ensin STL-kielelle, ja vasta sen jälkeen konekieliseksi, kun taas S7-1500-logiikalla kaikki kielet käännetään suoraan konekielelle, jolloin suorituskykyä ei tuhlata välissä tapahtuviin käännöksiin. [Siemens 2018, s. 11–12.]

4.2 Ohjelmointiympäristöt

Tässä kappaleessa käsitellään eroavaisuuksia alkuperäisessä koodissa käytetyn Simatic Manager -ohjelmiston ja uuden TIA Portal -ohjelmiston välillä. Yksi suurimmista myyntiargumenteista TIA Portaliin päivittämisen puolesta on, että uuden sukupolven S7-1200- ja S7-1500-logiikoiden ohjelmointi on mahdollista ainoastaan TIA Portalin avulla. Myös vanhempien S7-sarjan logiikkaohjainten (kuten S7-300) ohjelmointi on mahdollista TIA Portalista käsin. [Siemens -b.]

TIA Portalin avulla on myös mahdollista hallinnoida kokonaisvaltaisemmin kaikkia automaatioon liittyviä osa-alueita, sillä esimerkiksi käyttöliittymän suunnittelu tai Siemensin omien taajuusmuuttajien konfigurointi onnistuu suoraan TIA Portalista, ilman tarvetta muiden ohjelmistojen asentamiselle [Siemens -c].

Kuten aiemmin mainittu, TIA Portal tukee oletuksena SCL-kieltä, siinä missä Simatic Managerin kanssa kieli oli saatavilla vain maksullisena lisäpakettina [Siemens 1998, s. 5]. Tämä oli yksi syistä, jonka vuoksi asiakkaan alkuperäisessä koodissa SCL-kielen käyttöä oli vältetty.

4.2.1 Taulukoiden määrittely

Tässä kappaleessa käsitellään mitä etuja TIA Portal tuo taulukoiden määrittelyyn. Taulukot (array) ovat tiedostorakenne, joka koostuu useasta samantyyppisestä datatyypistä. Sen jokaiselle elementille on määrätty oma indeksi, jolla taulukkoon kuuluvia jäseniä voidaan käsitellä. Elementit voivat olla joko PLC:n perusdatatyyppejä, paikallisesti määriteltyjä rakenteita (Struct) tai käyttäjän itse määrittelemiä PLC datatyyppejä. [Siemens 2018, s. 69.]

Taulukoiden luonti oli mahdollista myös S7-300:n ja Simatic Managerin avulla, mutta niiden rajojen määrittely oli mahdollista vain paikallisesti. TIA Portalissa sen sijaan on versiosta V13 SP1 (Service Pack 1) lähtien ollut mahdollista määritellä taulukoiden rajat käyttäen globaaleja vakioita [Denilson Pegaia 2015].

4.2.2 Taulukoiden kutsuminen

Toinen S7-1500 alustan mahdollistama etu taulukoiden käsittelyssä on mahdollisuus kutsua niitä dynaamisesti. Tämä tehdään lohkon kutsurajapinnassa käyttämällä tähti-symbolia taulukon rajojen määrittämiseen. Alla olevassa kuvassa (Kuva 6) esitellään erilaisia tapoja määritellä taulukon rajat funktion kutsurajapinnassa.

	Name	Data type	Def...	Comment
1	Input			
2	ExampleArray_1	Array[1..15] of DInt		Array with fixed limits
3	ExampleArray_2	Array[*] of DInt		Array with variable limits
4	ExampleArray_3	Array["gc_LowLimit".."gc_HighLimit"] of DInt		Array with global constants as limits
5	ExampleArray_4	Array[1.."gc_HighLimit"] of DInt		Array with global constant as high limit only.
6	<Add new>			
7	Output			
8	<Add new>			
9	InOut			
10	<Add new>			

Kuva 6. Taulukon kutsuminen funktion sisään- ja ulostulojen rajapinnassa.

Yhtenä dynaamisen kutsun hyötynä on, että se mahdollistaa silmukoiden rakentamisen ilman riskiä muistirajojen ylittämisestä. Tämä on mahdollista TIA Portalin ”LOWER_BOUND” ja ”UPPER_BOUND” -komennoilla, joiden avulla voidaan lukea dynaamisesti kutsutun taulukon ylä- ja alarajat. Kutsutun taulukon lisäksi komennoille on annettava parametrina se taulukon ulottuvuus, josta rajat halutaan selvittää. Esimerkkikoodissa 5 demonstroidaan kyseisten käskyjen toimintaa SCL-kielellä. Koodin alussa taulukon rajatiedot tallennetaan väliaikaisiin ”v_LowLimit” ja ”v_HighLimit” -muuttujiin, joita sittemmin hyödynnetään koodin ensimmäisen FOR-silmukan luomisessa. Rajatietoja ei kuitenkaan ole pakko tallentaa väliaikaismuuttujiin, vaan vaihtoehtoisesti ne voi syöttää suoraan FOR-silmukan määrittelyehtoihin. Tätä tapaa demonstroidaan esimerkkikoodin 5 toisessa FOR-silmukassa. Kumpikin tapa tuottaa saman lopputuloksen, ensimmäisessä tavassa kuitenkin voimme todeta koodin olevan lukemisen kannalta jokseenkin selkeämpää. Viimeisenä vaihtoehtona on käyttää silmukan määrittelyssä samoja globaaleita vakiota, joita on käytetty taulukon määrittelyssä. Mikäli funktio on tarkoitettu yleiskäyttöiseksi, on suositeltavaa käyttää ensimmäistä tai

toista vaihtoehtoa, jotta funktion kanssa voidaan käyttää erikokoisia taulukoita. Jos kuitenkin kyse on yksittäisestä toiminnallisuudesta, jonka kanssa käytetään aina samaa taulukkoa, voi olla järkevämpää käyttää vaihtoehtoa numero 3, koska tällöin koodi saadaan kirjoitettua kompaktimmin, ja lisäksi osuvasti nimettyjen globaalien vakioiden käyttö myös havainnollistaa ohjelman toimintaa ja tekee siten koodista selkeämmän lukea.

```
// Read array limits:
#v_LowLimit := LOWER_BOUND(ARR := #IQ_FirstArray, DIM := 1);
#v_HighLimit := UPPER_BOUND(ARR := #IQ_FirstArray, DIM := 1);

// Loop 1: Define loop with pre-defined temp variables
FOR #v_loop_index := #v_LowLimit TO #v_HighLimit DO
     // Add I_Value to the value of Array-element:
     #IQ_FirstArray[#v_loop_index] += #I_Value;
END_FOR;

// Loop 2: Define loop and read limits at the same time:
FOR #v_loop_index := LOWER_BOUND(ARR := #IQ_SecondArray, DIM := 1)
                  TO UPPER_BOUND(ARR := #IQ_SecondArray, DIM := 1)
DO
     // Add I_Value to the value of Array-element:
    #IQ_SecondArray[#v_loop_index] += #I_Value;
END_FOR;

// Loop 3: Define loop limits with global constants:
FOR #v_loop_index := gc_ResourceStartNbr TO gc_MaxNbrOfResource DO
     // Add I_Value to the value of Array-element:
     #IQ_ThirdArray[#v_loop_index] += #I_Value;
END_FOR;
```

Esimerkkikoodi 5. Silmukan muodostaminen ja dynaamisesti kutsutun taulukon rajojen selvittäminen

4.2.3 PLC datatyyppien käyttö

PLC datatyyppi (PLC data type) on datarakenne, joka koostuu käyttäjän itse määrittelemistä erilaisista datatyypeistä. Aikaisemmassa Simatic Manager -ohjelmointiympäristöissä PLC datatyyppejä kutsuttiin nimellä ”User Data Type”, eli lyhyesti UDT. PLC datatyypin ja Struct -datatyypin ainoa eroavaisuus on, että Struct määritellään aina paikallisesti, kun taas PLC datatyyppi määritellään koko projektin laajuisesti ja siihen tehdyt muutokset päivittyvät automaattisesti kaikkiin sitä käyttäviin funktioihin ja datalohkoihin. Yksi S7-1500:n ja TIA Portalin

mahdollistama uusi ominaisuus on PLC datatyyppien käyttäminen niin kutsutuilla ”Tag Table” -alueilla, eli PLC:n I/O-muistialueilla. Tämän ansiosta PLC datatyyppejä voidaan käyttää esimerkiksi mallipohjana jonkun tietyn laitteen liitännöille. Tällä voidaan vähentää virheiden määrää esimerkiksi sellaisten laitteiden tapauksessa, joiden määrä vaihtelee eri projektien välillä, sillä kerran toimivaksi todettua pohjaa voidaan helposti käyttää uusien laitteiden kanssa. PLC datatyyppien käytössä I/O-alueilla on kuitenkin oltava tietoinen siitä, minkälainen niiden datarakenne oikeasti on Siemensin logiikan sisällä. Kaikki rakenteelliset datatyypit (Struct, Array ja PLC datatyypit) nimittäin koostuvat aina vähintään kahden tavun, eli yhden WORD-datatyypin mittaisista muistipaloista, joka täytyy ottaa huomioon, mikäli PLC datatyyppiä aikoo käyttää I/O-muistialueella. Toisin sanoen, esimerkiksi kahden bitin pituinen taulukko vie PLC:n muistissa aina vähintään kaksi tavua, vaikka teoriassa taulukosta käytetäänkin vain kaksi bittiä. [Siemens 2018, s. 72–74.]

Test_TagTable

	Name	Data type	Address
1	Example_PLC_data_type_WRONG	"_Test_IO_Field_Wrong"	%I1000.0
2	Alarm_1	Bool	%I1000.0
3	Alarm_2	Bool	%I1000.1
4	Mode	Struct	%I1002.0
5	Mode_Bit1	Bool	%I1002.0
6	Mode_Bit2	Bool	%I1002.1
7	Mode_Bit3	Bool	%I1002.2
8	StatusBits	Array[1..3] of Bool	%I1004.0
9	StatusBits[1]	Bool	%I1004.0
10	StatusBits[2]	Bool	%I1004.1
11	StatusBits[3]	Bool	%I1004.2
12	SomeByteInfo	Byte	%IB1006
13	Example_PLC_data_type_RIGHT	"_Test_IO_Field"	%I1008.0
14	Alarm_1	Bool	%I1008.0
15	Alarm_2	Bool	%I1008.1
16	Mode_Bit1	Bool	%I1008.2
17	Mode_Bit2	Bool	%I1008.3
18	Mode_Bit3	Bool	%I1008.4
19	Status_Bit1	Bool	%I1008.5
20	Status_Bit2	Bool	%I1008.6
21	Status_Bit3	Bool	%I1008.7
22	SomeByteInfo	Byte	%IB1009

Kuva 7. PLC datatyyppien käyttö I/O-muistialueella.

Kuvasta 7 voimme nähdä käytännössä, miten tämä ominaisuus vaikuttaa muuttujien muistiosoitteisiin PLC Tag Tablessä. Kuvan ylempi esimerkki demonstroi väärää tapaa määritellä I/O-alueella käytettävä PLC datatyyppi ja alempi oikeaa tapaa. Vaikka kumpikin rakenne sisältää periaatteessa saman määrän dataa, kuluttaa ylempi datatyyppi 8 tavua muistia (osoitealue I1000.0 – I1007.7), alemman datatyypin kuluttaessa vain kaksi tavua (osoitealue I1008.0 – I1009.7). On myös huomioitavaa, että PLC datatyyppi voi käyttää vain yhtä muistialuetta kerrallaan, eli saman datatyypin sisällä ei voi olla samanaikaisesti sisääntulo ja ulostuloalueiden muuttujia, vaan kummatkin on tässä tapauksessa määriteltävä erikseen.

4.2.4 Osoittajat

Kuten jo aiemmin mainittu, muistiosoittimien (tästä eteenpäin: pointer) käyttö oli alkuperäisessä koodissa tapa esimerkiksi silmukoiden ja tilakoneiden rakentamiseen. Tämä on kuitenkin huono käytäntö, sillä muistialueen sisällön muuttuessa, eivät viittaukset enää päde. Parempi toimintatapa onkin asettaa halutut muuttujat taulukkoon ja kutsua kyseistä taulukkoa indeksien avulla, jolloin ohjelmakutsut eivät hajoa, vaikka datalohkon rakenteeseen tehtäisiin muutoksia.

Yksi S7-1500 alustan uudistuksista on Variant-datatyyppi, jonka tehtävänä on korvata S7-300 alustalla käytetyt ANY-pointterit. Myös Variant on luonteeltaan pointer, mutta ANY-pointteriin verrattuna Variant-datatyypille tehdään ajohetkellä suoritettava tyyppitarkastus, jota hyödynnetään esimerkiksi useiden logiikkaohjainten välisessä kommunikoinnissa. TIA Portalista löytyy useita erilaisia sisäisiä komentoja, joita voidaan käyttää hyödyksi Variant-datatyypin kanssa. Esimerkiksi TypeOf-komennon avulla voidaan testata ja verrata Variantin datatyyppiä, tai vaikka määritellä CASE-lausekkeita riippuen datatyypistä. Tyyppiä on mahdollista verrata suoraan tiettyyn datatyyppiin tai vertailu voidaan tehdä toista Variant-tyyppistä muuttujaa kohtaan. Variant-datatyypin käyttöä ei suorituskykyyn liittyvistä syistä suositella käytettäväksi silmukoiden sisällä. Siemens kannustaakin käyttäjiä hyödyntämään taulukoita ja sisäänrakennettuja käskyjä (kuten MOVE, MOVE_BLK tai MOVE_BLK_VARIANT), joiden avulla voidaan

kopioida yksittäisten elementtien sijasta kokonaisia alueita kerrallaan. [Siemens 2018, s. 26–33.]

Esimerkkikoodissa 6 demonstroidaan TypeOf-, sekä VariantGet- ja VariantPut-komentojen käyttöä. Ohjelma ottaa Variant-datatyypin avulla sisääntulossaan erilaisten hypoteettisten laitteiden ajureita, ja suorittaa CASE-lausekkeen avulla erilaiset toimenpiteet riippuen valitusta laitteesta. Koodiesimerkissä CASE-lauseke on asetettu IF-lauseen sisään, eli ennen operaatioiden suorittamista ohjelma varmistaa, että sisääntulossa ja ulostulossa olevien rakenteiden datatyypit ovat identtiset. CASE-lauseessa määritellyt ”Device1” ja ”Device2” kuvastavat itse määriteltyjen PLC datatyyppien nimiä. VariantGet-komennolla voidaan hakea tiedot, joihin Variant osoittaa ja sijoittaa ne muuttujaan/rakenteeseen, jonka tyyppi tiedetään. Vastaavasti VariantPut-komennon avulla operaatio voidaan tehdä toiseen suuntaan, eli tiedetyn datatyypin sisältö yritetään siirtää Variantin osoittamaan muuttujaan, jonka tyyppi tarkistetaan vasta koodia ajettaessa.

```
// Check that input and output have same datatype
IF TypeOf(#I_FromDevice) = TypeOf(#Q_ToDevice) THEN

     // Choose correct operation for the selected device:
     CASE TypeOf(#I_DeviceIn) OF

       Device1:
          // Read input:
          VariantGet(SRC:=#I_DeviceIn,
                     DST=>#v_Device1);

          // Do something with device 1
          ;

          // Write to output:
               VariantPut(SRC := #v_Device1,
                          DST := #Q_ToDevice);

       Device2:
          VariantGet(SRC:=#I_DeviceIn,
                     DST=>#v_Device2);

          // Do something with device 2...

     END_CASE;

END_IF;
```

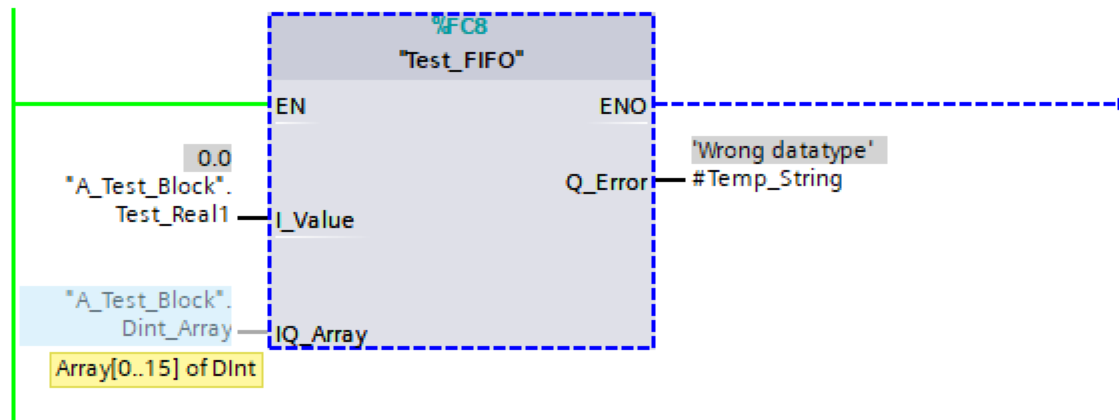
Esimerkkikoodi 6. Datatyypin selvittäminen ja sen käyttö CASE-lausekkeessa.

Esimerkkikoodissa 7 demonstroidaan Variant-datatyypin käyttöä yleiskäyttöisessä ohjelmassa, jonka avulla voidaan lisätä uusia arvoja mihin tahansa taulukkoon. Aluksi ohjelma laskee kohdetaulukon koon, ja siirtää viimeistä muuttujaa lukuun ottamatta kaikkia taulukon arvoja yhden indeksin verran eteenpäin, käytännöllisesti katsoen yli kirjoittaen viimeisen muuttujan. Tämän jälkeen uusi muuttuja lisätään taulukon ensimmäiselle paikalle.

```
// Reset errors at the start
#Q_Error := #No_Errors;

// Check that the target is an array
IF IS_ARRAY(#IQ_Array)= FALSE THEN
     #Q_Error := #Target_Not_Array;
// Check that the target array has space for at least two elements
ELSIF CountOfElements(#IQ_Array) < 1 THEN
     #Q_Error := #Array_Too_Small;
// Check that the input value has same data type as the target array:
ELSIF TypeOf(#I_Value) <> TypeOfElements(#IQ_Array) THEN
     #Q_Error := #Wrong_Datatype;
END_IF;

// If no errors, proceed to move-operations:
IF #Q_Error = #No_Errors THEN
     // Calc array size substracted by one:
    #v_ArraySizeMinus1 := (CountOfElements(#IQ_Array) - 1);

     // Move all values (except the last one) up by one spot in
     // the array, effectively overwriting the last value.
     #v_Errorcode1 := MOVE_BLK_VARIANT(SRC := #IQ_Array,
                                       COUNT := #v_ArraySizeMinus1,
                                       SRC_INDEX := 0,
                                       DEST_INDEX := 1,
                                       DEST => #IQ_Array);

     // Move I_Value into the first spot of the array
     #v_Errorcode2 := MOVE_BLK_VARIANT(SRC := #I_Value,
                                       COUNT := 1,
                                       SRC_INDEX := 0,
                                       DEST_INDEX := 0,
                                       DEST => #IQ_Array);

     // Check for errors in MOVE_BLK_VARIANT -instructions
     IF #v_Errorcode1 <> 0 OR #v_Errorcode2 <> 0 THEN
          Q_Error := #Unknown_Error;
     END_IF;
ELSE
     ENO := FALSE;
END_IF;
```

Esimerkkikoodi 7. Ohjelma, joka lisää taulukon alkuun uuden arvon.

Ohjelmassa on myös virheentunnistus yleisimpien virheiden varalta. Ohjelma mm. tarkistaa, että kohde on tyypiltään taulukko, ja että taulukkoon lisättävän arvon datatyyppi on sama kuin kohdetaulukossa olevien elementtien. Virheilmoitukset on määritelty funktion muuttujarajapinnassa paikallisina vakioina, jotka sopivat yleiskäyttöisten funktioiden virheiden määrittelyyn, koska tällöin niiden sisältö on mahdollista määritellä ilman tarvetta erillisen datalohkon luomiselle. Vakioiden tyypiksi on määritelty ”String”, eli merkkijono. Mahdollisen virhetilanteen tapauksessa virheilmoitus näkyy funktion ulostulossa ”Q_Error”, jolloin käyttäjä näkee selkokielisenä lauseena suoraan mistä virheestä on kysymys. Kuvassa 8 esitellään virheilmoitusten näyttämistä tilanteessa, jossa käyttäjä on yrittänyt syöttää DInt-muuttujista koostuvaan taulukkoon Real-muuttujaa. On myös huomionarvoista, että virhetilanteessa funktion ENO-ulostulo (joka mahdollistaa eri funktioiden ketjuttamisen peräkkäin) laitetaan manuaalisesti tilaan ”epätosi”, jotta käyttäjälle korostuu, että funktiota ei ole käytetty oikein. Tämän toiminnallisuuden huomioiminen on tärkeää, sillä joskus myös logiikkaoperaation epäonnistuminen voi olla haluttu lopputulos, jolloin ENO-ulostulo on manipuloitava palauttamaan arvo ”tosi” tilanteesta huolimatta. TIA Portalin sisäisten käskyjen (kuten MOVE_BLK_VARIANT) palauttamat virhekoodit aiheuttavat automaattisesti käskyä käyttävän funktion ENO-ulostulon asettamisen tilaan ”epätosi”, mutta myös näissä tilanteissa ulostulo on mahdollista manuaalisesti manipuloida tilaan ”tosi". SCL-kielellä manipulointi onnistuu yksinkertaisesti esimerkiksi käskyllä *”ENO := TRUE”*, kun taas FBD- ja LAD-kielillä voidaan käyttää *”Return”* -käskyä. Return-käskyn suhteen on otettava huomioon, että sen on aina oltava jonkin tietyn komentolohkon (Network) viimeinen elementti, muussa tapauksessa TIA Portal antaa projektin kääntämisen yhteydessä vikakoodin, joka johtaa kääntämisen epäonnistumiseen. Mikäli Return-käskyn halutaan kuitenkin johtavan tilansa jonkin tietyn, ei viimeisenä olevan, komentosarjan loogisesta lopputuloksesta (RLO = Return of Logic Operation), on tällöin käytettävä väliaikaisia muuttujia. Tallentamalla lopputulos väliaikaiseen muuttujaan, voidaan sitä käyttää ehtona komentolohkon lopussa, juuri ennen Return-käskyä.



Kuva 8. Esimerkkiohjelma 7 ja virheilmoitusten näyttäminen.

5 Toteutus

5.1 Nosturimalliin ja alkuperäiseen koodiin perehtyminen

Ennen varsinaisten uudistustöiden aloittamista, oli tarpeellista tutustua nosturimalliin ja sen ominaisuuksiin. Konecranesin sisäisistä dokumenteista löytyivät manuaalit sekä itse nosturille, että sen käyttöliittymälle. Näihin tutustumalla saatiin muodostettua kuva siitä, minkälaisiin asioihin nosturia on mahdollista käyttää, joka helpotti myöhemmin varsinaisessa koodin uudelleenkirjoitusvaiheessa. Tämän jälkeen alkoi tutustuminen alkuperäiseen koodiin. Alkuperäisenä suunnitelmana oli tutkia koodia samalla kirjaten Excel-dokumenttiin ylös ne kohdat, jotka vaatisivat muutosta. Pian kuitenkin kävi selväksi, että PLC-ohjelman laajuuden vuoksi tämänkaltaisen kirjanpidon laatiminen olisi vienyt liikaa aikaa ennen kuin varsinainen työ olisi päässy aloittamaan, jonka vuoksi suunnitelma päätettiin hylätä. Sen sijaan päätettiin, että funktioita aletaan systemaattisesti uudistamaan yksi funktio kerrallaan, samalla käyden läpi kyseiseen funktioon liittyvät datalohkot.

5.2 Uuden projektin luominen

Kun alustava tutustuminen nosturimalliin ja alkuperäiseen koodiin oli tehty, siirryttiin varsinaisen ohjelmoinnin pariin. Aivan ensimmäiseksi oli luotava uusi projekti TIA Portalissa. Avautuessaan TIA Portal avautuu joko ”portaalinäkymään” (Portal View) tai ”projektinäkymään” (Project View) ja avausnäkymästä riippuen projektin luominen tapahtuu hieman eri tavalla. Portaalinäkymässä näytölle ilmestyy liuta erilaisia painikkeista, joista yksi on ”Create new project”. Painiketta klikkaamalla ikkunan oikealle puoliskolle ilmestyy tekstilaatikoita, joihin voidaan syöttää mm. projektin nimi ja tallennussijainti. Projektinäkymässä sen sijaan uuden projektin luominen tapahtuu valitsemalla ylävalikosta ”Project” ja sen jälkeen ”New”, jolloin sama ikkuna ilmestyy erillisenä ponnahdusikkunana (Kuva 9).

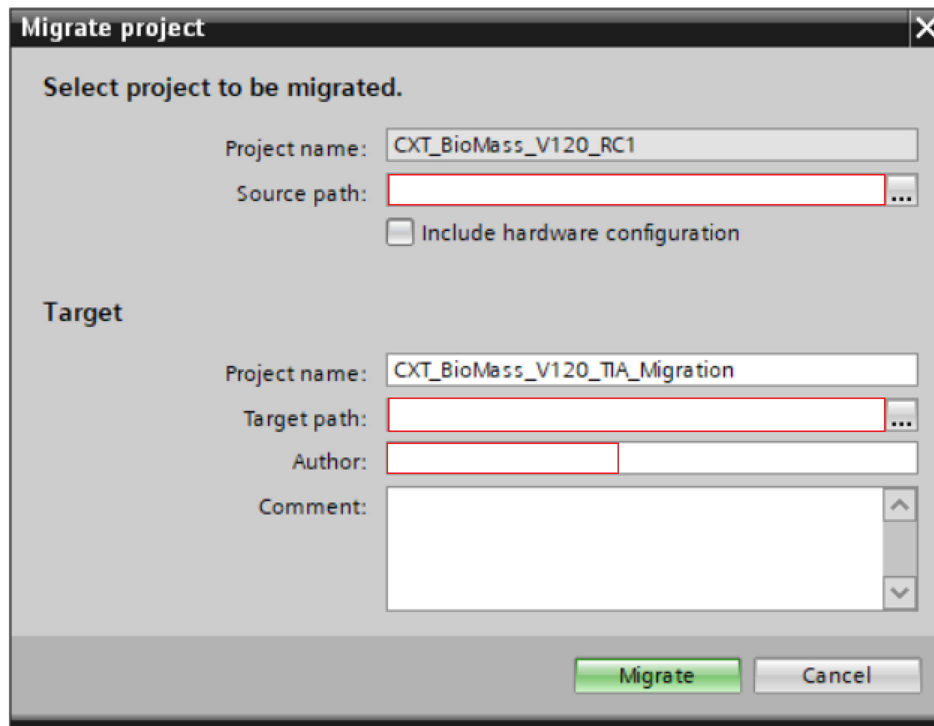
Kuva 9. Uuden projektin luominen TIA Portalin projektinäkymässä (tiedostopolku ja käyttäjänimi peitetty tietoturvasyistä).

Avausnäkymää on mahdollista muuttaa TIA Portalin asetuksista ja itse päädyinkin vaihtamaan oletusnäkymäksi projektinäkymän, sillä se tarjoaa ohjelmoinnin kannalta kokonaisvaltaisemman katsauksen projektin funktioihin ja datalohkoihin. Portaalinäkymä onkin selvästi tarkoitettu enemmän käyttöönottovaiheen työkaluksi, sillä se ohjaa käyttäjää vaihe vaiheelta tekemään tarvittavat toimenpiteet ohjelmoinnin aloittamiseksi, mm. määrittelemään projektissa käytetyn ohjelmoitavan logiikan. Ohjelmoitavan logiikan määrittely on pakollista, ennen kuin

ohjelmointi TIA Portalissa on mahdollista. Tarkempien asetusten määrittely, kuten IP-osoitteiden määrittely, ei kuitenkaan tässä vaiheessa ole pakollista. Tässä projektissa käytettäväksi PLC:ksi valittiin Siemens S7 1500 -logiikan turvaversio ”CPU 1517F-3 PN/DP”.

5.3 Migraatio S7-1500 alustalle

Vaikka projektin toteutus haluttiinkin tehdä käytännöllisesti katsoen puhtaalta pöydältä, pian uudistuksen aloittamisen jälkeen kuitenkin huomattiin, että muuttujien kopiointi STEP7-ohjelmiston ja TIA Portalin välillä on monin tavoin hyvin rajoittunutta. Esimerkiksi kopioidessa muuttujia STEP7:n datalohkosta TIA Portaliin, jäivät kommenttikentän tiedot kokonaan kopioitumatta, kun taas funktion sisään- ja ulostulojen rajapinnasta usean muuttujan kopiointi alustojen välillä ei ollut laisinkaan mahdollista. Tämän vuoksi alkuperäiselle projektille päätettiin tehdä migraatio omaan erilliseen TIA Portal-projektiinsa. Tämän ns. ”rinnakkaisprojektin” päätarkoituksena oli helpottaa muuttujien kopiointia uuteen projektiin. Projektin migraatio TIA Portalissa tehtiin valitsemalla ”Project” -valikosta ”Migrate project”. Tämä avaa käyttäjälle uuden ponnahdusikkunan, johon syötetään alkuperäisen Step7-projektin tiedostosijainti, sekä uuden TIA-projektin nimi ja haluttu tallennussijainti (Kuva 10). Ohjelma tarjoaa myös vaihtoehtoa sisällyttää alkuperäinen laitekonfiguraatio projektiin.



Kuva 10. STEP7-projektin migraatio TIA Portaliin (tiedostopolut ja käyttäjänimi peitetty tietoturvasyistä).

Jotta migraation tekeminen olisi mahdollista, on alkuperäisen projektin oltava ajan tasalla, eli onnistuneesti käännetty STEP7:ssa. Migraatiohetkellä projekti ei myöskään saa olla samanaikaisesti avattuna STEP7-ohjelmistolla. Migraation jälkeen funktioita, datalohkoja ja niiden sisältöjä on mahdollista kopioida suoraan kahden yhtäaikaisesti auki olevan TIA-projektin välillä. Tämä helpotti ja nopeutti datalohkojen läpikäyntiä huomattavasti. Datalohkoja tuotiin rinnakkaisprojektista uuteen projektiin lohko kerrallaan, sitä mukaa kuin kullakin hetkellä uudistettavana oleva funktio on niitä tarvinnut.

5.4 Funktioiden uudelleenkirjoittaminen

Opinnäytetyö rajattiin koskemaan vain nosturin automaatio-osuutta, jonka sisällä voidaan tehdä karkeasti jako kolmeen eri osa-alueeseen:

- Master-tason datanpäivitykset

- Työsekvenssit ja niiden apufunktiot
- Nosturin ajosekvenssit, reititys ja niiden apufunktiot

5.4.1 Master-tason datanpäivitykset

Master-tason datanpäivitykset koostuvat yhteensä kuudesta osa-alueesta:

1) Liikennevalojen hallinta
2) Automaatioalueiden rajojen päivitys
3) Solualueen ja solujen koon määrittely
4) Solujen tilan päivitys
5) Solujen materiaalitasojen päivitys
6) Ulossyöttökohteiden tilojen ja kuormatietojen päivitys

Ohjelmien uudelleenkirjoitus tehtiin edellä mainitun listan mukaisessa järjestyksessä, joskin osalle funktioista tehtiin vielä myöhemmin iterointeja paremman lopputuloksen saavuttamiseksi.

5.4.1.1 Liikennevalojen hallinta

Liikennevalojen hallintaa koskevat funktiot hallitsevat sisääntuontialueiden (infeed) liikennevalojen tilaa. Liikennevalojen hallinta voidaan määritellä toimimaan automaattisesti tai manuaalisesti. Manuaalitilassa liikennevalojen tila määritellään käyttöliittymäpaneelin välityksellä, kun taas automaatiotilassa tila määrittyy automaattisesti nosturin sijainnin ja sisääntuontialueen antureiden tilan perusteella. Uudessa mallissa sisääntuontialueiden määrä muutettiin dynaamiseksi globaalien vakioiden avulla. Myös kaikki sisääntuontialueisiin liittyvät datalohkot määriteltiin käyttämään taulukoissaan samoja vakioita. Vanhassa mallissa jokaiselle sisääntuontialueelle erikseen kopioidut operaatiot korvattiin uudessa mallissa dynaamisilla silmukoilla. Ohjelmaa käytiin muokkaamassa myöhemmin

vielä uudelleen, ja optimoitiin yhdistämällä operaatiot yhden dynaamisen silmukan sisään. Toisella iteraatiolla koodin lisättiin myös paikallisia vakioita (kuten ”lc_Auto”, ”lc_Manual” tai ”lc_Green”) tuomaan merkitystä kokonaisluvuilla tehtäville tilanmuutoksille ja siten selkeyttämään koodin lukemista. Paikallisille vakioille annettiin etuliite ”lc” (lyhenne paikallisen vakion englanninkielisestä merkityksestä ”local constant”) selventämään käyttäjälle kyseessä olevan paikallinen vakio.

5.4.1.2 Automaatioalueiden rajojen päivitys

Automaatioalueiden rajojen päivitysfunktioiden tehtävänä on lukea alueille määritellyt rajat, ja lisätä niihin ennalta määritellyt turvamarginaalit. Myös näiden funktioiden osalta tehtiin kaksi iteraatiokierrosta. Ensimmäisellä iteroinnilla jokaiselle alueelle tehtiin dynaamiset FOR-silmukat, joiden sisällä kutsuttiin rajat laskevaa funktiota. Rajat laskettiin pääautomaatioalueille ja ulossyöttökohteille (outfeed). Toisella iteraatiolla koodiin päätettiin lisätä myös muita alueita, ja samalla päätettiin tehdä silmukointia varten oma funktionsa, joka ottaa jokaista aluetta koskevat taulukot dynaamisina parametreina funktion sisään- ja ulostulojen kautta, ja määrittelee silmukan rajat automaattisesti taulukon koon perusteella. Menetelmää esitelty tarkemmin luvussa ”4.2.2. Taulukoiden kutsuminen”.

5.4.1.3 Solualueen ja solujen koon määrittely

Solualueen määrittely oli kenties tärkein näistä kuudesta osa-alueesta, ja sen kohdalla käytiin läpi useita iteraatioita. Solualue koostuu varsinaisesta varastointialueesta sekä materiaalin sisääntuontialueista. Solualue on kaksiulotteinen ruudukko, jonka rajat on määritelty yhdeksi suorakaiteen muotoiseksi alueeksi. Solualue voi kuitenkin koostua useammasta pienemmästä alueesta, joka mahdollistaa solualueen määrittelyn paremmin ilman kompromisseja. Tätä demonstroidaan kuvassa 11. Kuvassa kaikki solut näkyvät samankokoisina, mutta todellisuudessa solujen koot eri alueiden välillä voivat vaihdella, sillä seinien muodostamat rajat alueilla voivat olla erilaiset, mutta yhden alueen sisällä solujen koot ovat kuitenkin aina identtiset.

							AREA 1											
1,1	1,2	1,3	1,4	1,5	1,6	1,7	1,8	1,9	1,10	1,11	1,12	1,13	1,14	1,15	1,16	1,16	1,16	1,16
2,1	2,2	2,3	2,4	2,5	2,6	2,7	2,8	2,9	2,10	2,11	2,12	2,13	2,14	2,15	2,16	2,16	2,16	2,16
3,1	3,2	3,3	3,4	3,5	3,6	3,7	3,8	3,9	3,10	3,11	3,12	3,13	3,14	3,15	3,16	3,16	3,16	3,16
4,1	4,2	4,3	4,4	4,5	4,6	4,7	4,8	4,9	4,10	4,11	4,12	4,13	4,14	4,15	4,16	4,16	4,16	4,16
5,1	5,2	5,3	5,4	5,5	5,6	5,7	5,8	5,9	5,10	5,11	5,12	5,13	5,14	5,15	5,16	5,16	5,16	5,16
6,1	6,2	6,3	6,4	6,5	6,6	6,7	6,8	6,9	6,10	6,11	6,12	6,13	6,14	6,15	6,16	6,16	6,16	6,16
7,1	7,2	7,3	7,4	7,5	7,6	7,7	7,8	7,9	7,10	7,11	7,12	7,13	7,14	7,15	7,16	7,16	7,16	7,16
8,1	8,2	8,3	8,4	8,5	8,6	8,7	8,8	8,9	8,10	8,11	8,12	8,13	8,14	8,15	8,16	8,16	8,16	8,16
9,1	9,2	9,3	9,4	9,5	9,6	9,7	9,8	9,9	9,10	9,11	9,12	9,13	9,14	9,15	9,16	9,16	9,16	9,16
10,1	10,2	10,3	10,4	10,5	10,6	10,7	10,8	10,9	10,10	10,1	10,12	10,1	10,14	10,15	10,16	10,16	10,16	10,16
11,1	11,2	11,3	11,4	11,5	11,6	11,7	11,8	11,9	11,10	11,1	11,12	11,1	11,14	11,15	11,16	11,16	11,16	11,16
12,1	12,2	12,3	12,4	12,5	12,6	12,7	12,8	12,9	12,10	12,1	12,12	12,1	12,14	12,15	12,16	12,16	12,16	12,16
		AREA 2													AREA 3			

Kuva 11. Solualueen määrittely pienempiin osa-alueisiin.

Kuvassa näkyvän solualueen kooksi on määritelty 12 x 16 solua, ja se koostuu kolmesta pienemmästä osa-alueesta, jotka on merkitty kuvaan vihreän eri sävyillä. Alue 1 kattaa solut alueelta [1,1] – [6,16], alue 2 kattaa solut [7,1] – [12,7] ja alue 3 kattaa solut [7,13] – [12,16]. Punaisella merkatut solut eivät kuulu mihinkään alueeseen (seinät). Alkuperäisessä ohjelmassa alueiden maksimimääräksi oli määritelty kolme aluetta ja soluruudukon kooksi 12 x 32 ruutua. Uudessa mallissa määrät muutettiin dynaamisiksi globaalien vakioiden avulla ja jokaiselle alueelle erikseen kopioidut ohjelmakutsut korvattiin dynaamisilla silmukoilla.

Myös varsinaisen laskemisen hoitava funktio jouduttiin kirjoittamaan kokonaan uusiksi. Jokaista aluetta koskevat parametrit (mm. eri osa-alueiden rajat sekä mistä solusta kyseinen alue alkaa ja loppuu) data-tyypitettiin ja siirrettiin dynaamisiin taulukoihin. Funktioon lisättiin myös uusia toiminnallisuuksia, kuten mahdollisuus laskea automaattisesti solujen koko ja määrä kahmarin (nosturin köyden/vaijerin päässä roikkuva kuormauselin) koon ja tyypin perusteella. Toisella iteraatiolla lisättiin mahdollisuus määritellä, käytetäänkö referenssinä suljetun

vai avoimen kahmarin mittoja, sekä optimoitiin funktiota lisäämällä sen suoritukseen ehto, jolloin alueet lasketaan vain, mikäli ohjelma huomaa muutoksia alueisiin liittyvissä parametreissa. Tämän lisäksi funktion virheenhallintaa parannettiin lisäämällä funktioon kolme uutta ulostuloa, joista yksi indikoi BOOL-muuttujalla vikatilan aktiivisuuden, toinen näyttää vikatilan kuvauksen tekstimuotoisena viestinä ja vastaavasti kolmas mahdollisen selityksen vian ratkaisemiseksi. Kyseistä menetelmää on esitetty tarkemmin esimerkkikoodissa 7 ja kuvassa 8. Kolmannella iteraatiolla jokaiselle solulle lisättiin käyttöliittymää silmällä pitäen ominaisuus ”Visible” (näkyvä), joka kirjoitetaan kaikkiin käytettyihin soluihin. Muuttuja on myöhemmin mahdollista linkittää käyttöliittymässä solujen näkyvyystietoihin, eli mikäli solu ei kuulu yhteenkään solualueen osa-alueista, on se mahdollista piilottaa HMI:ltä tämän tiedon avulla.

5.4.1.4 Solujen tilan päivitys

Neljäntenä osa-alueena oli solujen tilanpäivitystä koskevat funktiot. Solujen tiloilla tarkoitetaan sitä mitä työtehtäviä nosturin on sallittu tehdä solulle. Alkuperäisessä koodissa soluille oli määritettynä työtehtäviä, joita ei oikeasti käytetty, joten uudessa mallissa nämä käyttämättömät työtehtävät poistettiin solun ominaisuuksista. Uudessa mallissa soluille määriteltiin boolean-muuttujilla kolme eri työtehtävää: syöttöhaku (feedpick), varastointihaku (stackpick) ja talletus (deposit). Syöttöhaku-soluista on sallittua viedä materiaalia ulossyöttökohteeseen, kun taas varastointihaku-soluista on sallittua viedä materiaalia niihin soluihin, joissa talletus on sallittu. Lisäksi jokaiselle työtehtävälle määriteltiin estobitit, joiden aktivointi estää kyseisen työtehtävän käytön. Solulle sallittujen työtehtävien määrittely tapahtuu käyttöliittymän avulla, kun taas estobittejä aktivoidaan myös ajonaikaisesti tilanteen mukaan. Mikäli esimerkiksi solun materiaalinkorkeus ylittää ennalta määritellyn rajan, estetään materiaalin talletus kyseiselle solulle. Osalle soluista estobitit voidaan myös asettaa pysyvästi. Esimerkiksi materiaalin sisääntuontialueella sijaitseville soluille sallituiksi työtehtäviksi määritellään syöttöhaku ja varastointihaku, mutta materiaalin talletus soluun on aina kielletty. Vastaavasti varastointialueella materiaalin talletus ja haku syöttökohteelle on sallittu, mutta haku toisiin varastointialueen soluihin on kielletty.

Monet funktioiden parametreistä saadaan PLC:n ja käyttöliittymän välissä rajapintana toimivan datalohkon kautta. Kuten monien muidenkin datalohkojen, myös kyseisen datalohkon rakenne koostuu ns. kovakoodatuista arvoista, mutta koska käyttöliittymän päivitystä ei vielä tässä vaiheessa haluttu sisällyttää osaksi projektia, oli tämän rajapintana toimivan datalohkon rakenne säilytettävä toistaiseksi ennallaan. Tästä syystä näiden funktioiden osalta jouduttiin keksimään kompromissi ongelman ratkaisemiseksi. Rajapintana toimivasta datalohkosta päädyttiin tekemään kopio, jossa taulukot muutettiin täysin dynaamisiksi, mutta joiden rakenne kuitenkin muistiosoitteiden tasolla vastasi alkuperäisessä datalohkossa olevien taulukoiden rakenteita. Jotta tämä olisi mahdollista, taulukoiden koon määrittelyyn käytettiin erillisiä globaaleita vakioita, joiden määrittelyt vastasivat kyseisten resurssien määrää vanhassa datalohkossa.

Varsinaiset funktiot uudistettiin siten, että tiedot kirjoitettiin tai luettiin uudesta dynaamisesta datalohkosta, ja funktioiden suorittamisen jälkeen (tai tilanteesta riippuen ennen, mikäli tiedon kulkusuunta oli käyttöliittymältä PLC:lle) tiedot kopioitiin suoria muistiviittauksia käyttäen joko alkuperäisestä datalohkosta uuteen tai toisinpäin. Funktioiden sisällä käytetyt silmukat kirjoitettiin siten, että silmukat määritellään kullakin hetkellä pienemmän taulukkokoon mukaisesti. Mikäli käyttöliittymän rajapintalohkon taulukossa on vähemmän tilaa kuin PLC:llä kyseiselle resurssille on määritelty, käytetään silmukan rajana käyttöliittymän rajaa. Vastaavasti PLC:n arvon ollessa pienempi, käytetään sitä silmukan raja-arvona. Näin onnistutaan välttämään muistivirheet kaikissa tilanteissa. Kompromissi ratkaisussa on, että mikäli PLC:llä on määriteltynä enemmän resursseja kuin käyttöliittymän puolella, joudutaan yli menevien resurssien osalta parametrit määrittämään käsin, eivätkä ylimenevien resurssien tiedot myöskään luonnollisesti päivity käyttöliittymän suuntaan. Mikäli tulevaisuudessa käyttöliittymä kuitenkin päätetään päivittää täysin dynaamiseksi, voidaan käyttöliittymän muuttujat linkittää suoraan uuteen datalohkoon. Myös taulukoiden rajat on tarvittaessa helppo päivittää PLC:n varsinaisiin rajoihin. Tämä onnistuu esimerkiksi poistamalla taulukoita varten luodut erilliset globaalit vakiot kokonaan ja väliaikaisesti uudelleennimeämällä PLC:n varsinaiset globaalit vakiot samannimisiksi, jolloin taulukoiden rajat päivittyvät osoittamaan oikeisiin muuttujiin.

Näiden funktioiden osalta myöhemmille iteraatioille ei juurikaan ollut tarvetta, mutta jälkikäteen yksi aikaisemmin työsekvenssien alaisuudessa ollut funktio siirrettiin tähän osuuteen funktioiden suoritusjärjestyksen optimoimiseksi. Lisäksi alueelle luotiin yksi täysin uusi funktio, jonka tehtävänä on pitää kirjaa siitä, missä solussa nosturin sijainti kullakin hetkellä on.

5.4.1.5 Solujen materiaalitasojen päivitys

Kuten nimestä voinee päätellä, solujen materiaalitasojen päivitystä koskevien funktioiden tehtävänä on pitää kirjaa jokaisen solun materiaalin pinnankorkeuden tasosta, ja päivittää niitä tarpeen mukaan. Tapoja pinnankorkeuden mittaamiseen/arviointiin on useita. Esimerkiksi rekan poistuessa materiaalin sisääntuontialueelta, kasvatetaan kyseisellä alueella olevien solujen pinnankorkeutta tietyn verran. Materiaalin haun yhteydessä korkeus voidaan mitata vaijerin sen hetkisen pituuden perusteella ja materiaalia talletettaessa pinnankorkeuden nousu voidaan arvioida sen hetkisen lastin painon perusteella. Joissain projekteissa saatetaan myös käyttää lasermittauslaitetta reaaliaikaiseen mittaamiseen nosturin liikkuessa. Varsinaisen materiaalin pinnankorkeuden tason lisäksi jokaiselle solulle on määritelty erikseen myös absoluuttinen korkeus suhteessa nollapisteeseen, sillä lattian korkeus eri solujen välillä saattaa vaihdella.

Kuten solujen tilan päivitystä koskevien funktioiden tapauksessa, myös materiaalitasojen päivitystä koskevissa funktioissa jouduttiin tekemään vastaavia kompromisseja käyttöliittymään liittyvien funktioiden suhteen. Näiden osalta muutoksen teko kävi kuitenkin huomattavasti nopeammin, sillä aiemmin keksittyä ratkaisua voitiin suoraan soveltaa myös näiden funktioiden päivittämiseen.

Osalle funktioista tehtiin kuitenkin toinen iterointikierros, sillä niiden toiminnallisuutta haluttiin parantaa verrattuna alkuperäiseen. Esimerkiksi talletuksen yhteydessä pinnankorkeuden nousua arvioivan funktion laskentakaavat uudistettiin kokonaan. Vanhassa mallissa arvio perustui kahmarin painoon sekä mielivaltaisesti valittuun arvioon täydellä kahmarilla tapahtuvasta pinnankorkeuden noususta. Vanha malli ei myöskään ottanut huomioon mahdollisia eroja solujen

koossa eri solualueiden välillä. Uudessa mallissa referenssinä käytettiin täyden kahmarin mitattua painoa ja tilavuutta, joita nykyiseen kuormatietoon vertaamalla voitiin laskea solun pinta-alaan perustuva arvio pinnankorkeuden noususta.

Lisäksi yksi aiemmin työsekvenssien alaisuuteen kuuluneista apufunktioista päätettiin siirtää tämän osuuden alle, sillä sen toiminnallisuus liittyi oleellisemmin materiaalitasojen päivitykseen kuin työsekvensseihin. Osuuteen luotiin myös yksi täysin uusi funktio, jonka tehtävänä on etsiä automaatioalueelta matalimman ja korkeimman materiaalitason omaavat solut.

5.4.1.6 Ulossyöttökohteiden tilojen ja kuormatietojen päivitys

Ulossyöttökohteiden tilojen päivityksellä tarkoitetaan mm. anturidatan, käyttöliittymän asetusten tai asiakkaan ohjausjärjestelmän perusteella päivitettäviä tilatietoja, joita ovat esimerkiksi syötön salliminen tai keskeyttäminen. Ulossyöttökohteella tarkoitetaan laitetta, joka hoitaa materiaalin kuljettamisen lopulliseen polttopaikkaan. Ulossyöttökohde voi olla esimerkiksi hopperi (ylhäältä täytettävä varastointisäiliö, joka tyhjennetään pohjasta) tai kuljetinhihna. Ulossyöttökohteille on myös mahdollista määritellä useita sisäisiä maalipisteitä (subtarget), joihin materiaali pudotetaan, ja niitä voidaan vaihdella syöttökertojen välillä. Alkuperäisestä mallista löytyi tuki ainoastaan hopperi-tyyppisille ulossyöttökohteille ja maalipisteiden määrä oli rajoitettu kolmeen.

Funktioiden kanssa käytiin läpi kaksi iteraatiota. Ensimmäisellä funktioista tehtiin dynaamisia, päivittämällä ulossyöttökohteita koskevien datalohkojen tiedot globaaleilla vakioilla määriteltyihin taulukoihin ja lisäämällä funktioihin uusia datarakenteita tukevat silmukat. Myös maalipisteiden maksimimäärä muutettiin dynaamiseksi yhteisellä globaalilla vakiolla. Käytössä olevien maalipisteiden määrää voidaan ulossyöttökohtaisesti vaihdella boolean-muotoisen ”inUse”-bitin avulla. Toisella iteraatiolla funktioista tehtiin modulaarisia, eli niihin lisättiin mahdollisuus erityyppisten ulossyöttökohteiden käyttöön. Tämä toteutettiin poistamalla funktioista kaikki suorat viittaukset hoppereihin, ja siirtämällä ne funktion

sisään- ja ulostulojen rajapinnan kautta vastaanotettaviksi parametreiksi. Myös silmukoiden määrittelyitä muokattiin siten, että niiden rajat määriteltiin globaaleiden vakioiden sijaan lukemalla rajat suoraan parametrina toimivan taulukon raja-arvoista. Menetelmää on esitelty tarkemmin kappaleessa ”4.2.2 Taulukoiden kutsuminen”.

Samat muutokset tehtiin myös kuormatietojen päivityksistä vastaavalle funktiolle. Sen tehtävänä on kerätä ylös jokaiselle ulossyöttöpaikalle kerättyjen lastien määrää ja painoa, joita voidaan hyödyntää nosturin kunnonvalvonnassa. Funktioon lisättiin myös ns. ”nollattava matkamittari”, jonka avulla käyttöliittymässä näytettäviä tietoja voidaan vaihdella viimeisimmän nollauksen jälkeen kerättyjen tietojen sekä pysyväisdatan välillä.

5.4.2 Työsekvenssit ja niiden apufunktiot

Nosturin työsekvensseillä tarkoitetaan niitä sekvenssejä, jotka määrittelevät mitä nosturin halutaan tekevän. CXT Biomassasta löytyy useita erilaisia työsekvenssejä, kuten mm. nosto- ja talletussekvenssi. Kaikkia työsekvenssejä ohjataan niin sanotun ohjaussekvenssin kautta, joka aktivoi tietyn sekvenssin nosturin sisääntulojen ja HMI-signaalien perusteella. Työsekvenssejä käsittelevään osuuteen kuului myös useita muita funktioita, ja alkuperäisessä mallissa osuuden voitiin katsoa koostuvan yhteensä seitsemästä eri osa-alueesta. Datankulun yksinkertaistamisen myötä yksi osuus voitiin kuitenkin poistaa kokonaan ja monia voitiin supistaa huomattavasti. Uudessa mallissa osuus kostuu seuraavista kuudesta osa-alueesta:

1) Hälytykset ja tapahtumat
2) Sisääntulot
3) Kohteen tarkistus
4) Aloitusehtojen ja prioriteettien tarkistus
5) Sekvenssit

6) Ulostulot

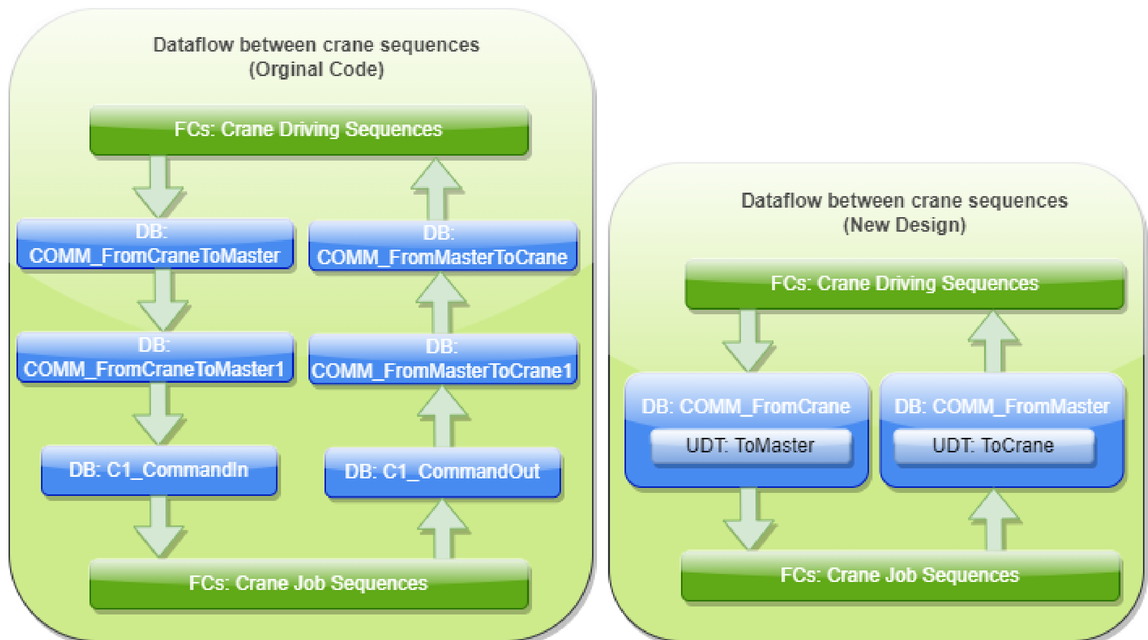
5.4.2.1 Hälytykset ja tapahtumat

Hälytysten ja tapahtumien osalta tarpeet uudistuksille olivat hyvin pienet, ja funktiot voitiinkin kopioida uuteen malliin lähestulkoon sellaisinaan. Joidenkin muuttujien kutsupolut jouduttiin päivittämään datalohkoihin tehtyjen muutoksien vuoksi, ja joistain hälytyksistä poistettiin päällekkäisiä ehtoja (eli ehtoja, jotka löytyivät valmiiksi jonkun toisen ehtona olevan muuttujan omista ehdoista), mutta tämän suuremmille toimenpiteille ei ollut tarvetta.

5.4.2.2 Sisääntulot

Siinä missä hälytyksien suhteen päästiin vähällä vaivalla, jouduttiin sisääntuloja käsittelevässä osuudessa tekemään huomattavasti suurempia muutoksia. Osuuden nimellä viitataan koodin muiden osa-alueiden lähettämien tietojen lukemiseen, eli ns. ohjelmallisiin sisääntuloihin varsinaisten rautatason sisääntulojen sijasta.

Yksi suurimmista ongelmista alkuperäisessä koodissa oli se, miten dataa kommunikoitiin ajosekvenssien ja työsekvenssien välillä. Osuudesta löytyi paljon funktioita, joiden ainoa tarkoitus oli kopioida dataa paikasta toiseen. Uudessa mallissa datan kulkua yksinkertaistettiin poistamalla ylimääräiset datalohkot ja päivittämällä ohjelmakutsut osoittamaan oikeaan sijaintiin. Tämän ansioista myös datankopiointifunktiot voitiin poistaa koodista kokonaan. Kuvassa 12 näemme vertailun alkuperäisen ja uudistetun koodin välillä. Alkuperäisessä koodissa dataa siirrettiin ensin useiden datalohkojen välillä, ennen kuin data päätyi lopulliseen osoitteeseensa. Uudistetussa koodissa ylimääräiset datalohkot on poistettu, ja jäljelle on jätetty vain yksi kommunikointilohko kumpaankin suuntaan. Uudessa mallissa datalohkojen sisällöt on myös datatyypitetty, joka mahdollistaa niiden käytön parametrinä funktioiden sisään- ja ulostulojen rajapinnassa.



Kuva 12. Datankulku työsekvenssien ja ajosekvenssien välillä (vasemmalla alkuperäinen koodi vs oikealla uudistettu koodi).

Myös monia käyttämättömiä muuttujia poistettiin, ja funktion pariin jouduttiinkin palaamaan useita kertoja, sillä monessa tapauksessa muuttujat saattoivat paljastua käyttämättömiksi vasta myöhemmin tarkemman tutkinnan alaisuudessa. Esimerkiksi muuttujaa saatettiin mm. käyttää ehtona toiselle muuttujalle, jota todellisuudessa ei lopulta käytetty mihinkään. Ulossyöttökohteisiin liittyvät funktiot muokattiin jälleen tukemaan erityyppisiä ulossyöttökohteita. Tämän lisäksi tiettyjä ulossyöttökohteisiin liittyviä ominaisuuksia järkeistettiin, jonka myötä osa niihin liittyvistä funktioista voitiin poistaa.

5.4.2.3 Kohteen tarkistus

Kohteen tarkistus on yksi vaihe useille sekvensseille, joka suoritetaan silloin, kun nosturin kohteeksi on asetettu jokin solualueen soluista. Tarkistusfunktio varmistaa, että nosturin maalipisteeksi määritelty sijaintikoordinaatti löytyy jostain solualueen soluista, ja että solun työtyyppi täsmää sen hetkisen sekvenssin kanssa. Sekvenssi pääsee jatkumaan vain, mikäli tarkistusfunktion kriteerit täyt-

tyvät. Alueen alaisuuteen kuului myös kaksi muuta operaatiota, mutta näistä toinen voitiin poistaa käyttämättömänä ja toinen päädyttiin siirtämään solualueiden tilan päivityksiin liittyvien funktioiden yhteyteen, sillä funktion tehtävänä oli pitää kirjaa kullakin hetkellä kullekin työtyypille sallittujen solujen määrästä. Varsinainen tarkistusfunktio kirjoitettiin uudelleen SCL-kielellä, ja suoriin muistiviittauksiin perustuvat silmukat uudistettiin käyttämään FOR-rakennetta globaaleilla vakiolla määriteltyjen taulukoiden kanssa. Tulevaisuudessa kyseisen funktion varsinainen toiminnallisuus saattaa vielä joutua yksityiskohtaisemman tarkastelun kohteeksi, mutta toistaiseksi toiminnallisuus jätettiin kuitenkin ennalleen.

5.4.2.4 Aloitusehtojen ja prioriteettien tarkistus

Aloitusehtojen ja prioriteettien tarkistus oli varsinaisten sekvenssien ohella kenties tärkein työsekvensseihin liittyvä osa-alue. Sen vastuulla on tarkistaa nosturin täysautomaatioajoa ja työsekvenssejä koskevat aloitusehdot, työsekvenssien keskinäiset prioriteetit, sekä syöttöjonon päivittäminen.

Täysautomaation aloitusehtoja voivat olla mm. erinäiset anturitiedot sekä sen aktivointi käyttöliittymästä. Työsekvenssien suorittaminen on mahdollista vain nosturin täysautomaatiotilan ollessa aktivoituna. Lisäksi kullekin työsekvenssille on määritelty omat aloitusehtonsa, jotka myös samalla asettavat sekvenssien prioriteetit tiettyyn järjestykseen. Esimerkiksi syöttösekvenssin aloitusehdot määritellään syöttöhaku-solujen sekä ulossyöttökohteiden tilojen perusteella. Mikäli kaikki solualueen syöttöhakusolut ovat tyhjiä tai yksikään ulossyöttökohteista ei tarvitse lisää materiaalia, syöttösekvenssin aloitus estetään. Syöttösekvenssiin liittyy myös syöttöjono, joka määrittelee missä järjestyksessä ulossyöttökohteita halutaan syöttää. Ensimmäisenä lisää materiaalia pyytänyt syöttökohde laitetaan jonossa ensimmäiseksi, toinen toiseksi jne.

Automaation aloitusehtojen sekä työsekvenssien keskinäisten prioriteettien suhteen tarpeet uudistuksille olivat melko vähäisiä ja useimpien operaatioiden osalta vaadittiin ainoastaan kutsupolkujen päivittäminen ajan tasalle. Syöttösekvenssin aloitusehtojen tarkistuksesta sekä syöttöjonon päivityksestä vastaavat

funktiot täytyi kuitenkin kirjoittaa kokonaan uudelleen. Kuten muissakin ulossyöttökohteisiin liittyneissä funktiossa, päivitettiin kaikki silmukat ensimmäisellä iteraatiolla dynaamisiksi, ja toisella iteraatiolla modulaarisiksi, eli tukemaan erityyppisiä ulossyöttökohteita. Myös syöttöjonon koko määriteltiin uudessa mallissa globaalilla vakiolla ja kaikille syöttöjonossa oleville kohteille määriteltiin syöttökohteen numeron lisäksi myös tyyppi, jonka perusteella vain kyseiselle tyypille määritelty instanssi funktiosta suorittaa halutun operaation.

5.4.2.5 Sekvenssit

Viidennestä, ja kaikkein laajimmasta osuudesta löytyivät varsinaiset sekvenssit. CXT Biomassan työsekvensseistä kaksi tärkeintä ovat syöttösekvenssi (feeding) ja varastointisekvenssi (stacking). Työsekvenssien valinta tehdään edellisessä osuudessa määriteltyjen prioriteettien perusteella. Korkeimmalla prioriteetillä on syöttösekvenssi, jonka tehtävänä on hakea tavaraa varastoalueelta ja viedä se ulossyöttökohteeseen. Mikäli syöttösekvenssille ei ole tarvetta, aktivoituu varastointisekvenssi, jonka tehtävänä on hakea tavaraa materiaalin sisääntuontialueelta ja viedä se varastointialueelle. Tavara tuodaan sisääntuontialueelle yleensä esimerkiksi varastoalueen reunalla olevalta rekkaportilta tai kuljetinhihnan välityksellä. Mikäli kumpaakaan kahdesta pääsekvenssistä ei tarvita, aktivoidaan ennalta määritellyn viiveen jälkeen odotustilaan siirtymissekvenssi (idle move). Tällöin nosturi siirretään ennalta määritettyyn sijaintiin odottamaan seuraavaa työpyyntöä.

Sekvenssien uudelleenkirjoitus aloitettiin suunnittelemalla yhteinen mallipohja (Liite 1), jota voitaisiin käyttää kaikkien sekvenssien kanssa. Mallipohjan kieleksi valittiin SCL, koska kaikki työsekvenssit olivat pohjimmiltaan tilakoneita ja SCL-kielen tarjoama CASE-lauseke sopii erittäin hyvin tilakoneiden rakentamiseen. Toisena syynä oli mahdollisuus alueiden (REGION) käyttöön. Kuva 13 esittelee varastointisekvenssin sisällysluetteloa, joka on luotu alueita käyttämällä. Funktiolle on luotu omat alueet funktion kuvaukselle sekä kaikille eri tilakoneen tiloille. Lisää tietoa alueiden käytöstä kappaleessa 3.2.3 Structured Control Language (SCL).

Description
BYPASS
State 0: Stay at state 0
State 1: Activate Pick
State 2: Picking Material
Feed request active with higher priority?
Deposit
No DepositAreas available?
State 3: Depositing Material
State 4: Save sequence stats
State 5: Idle State, do nothing

Kuva 13. Sisällysluettelon käyttö eräässä työsekvenssissä.

Mallipohjasta tehtiin kopiot jokaiselle sekvenssille ja nimettiin asianmukaisesti uudelleen. Tämän jälkeen vanhat STL-kielellä kirjoitetut sekvenssit käytiin huolellisesti läpi yksi funktio kerrallaan, ja kirjoitettiin uusiin mallipohjiinsa SCL-kielisinä. Käyttämättömiä muuttujia poistettiin ja kutsupolut päivitettiin tukemaan uutta yksinkertaistettua kommunikointirakennetta. Myös parametrien suhteen alkuperäisessä koodissa tapahtui paljon tarpeetonta väliaikaissijainteihin kopiointia, joten myös näiden ohjelmakutsuja päivitettiin osoittamaan suoraan parametri-datalohkoihin ja muualla (useimmiten sisääntulojen osuudessa) tapahtuvat datan kopiointiosuudet voitiin jälkikäteen käydä poistamassa.

Lähes kaikista sekvensseistä löytyi myös manuaalisesti (tiettyjen muuttujien tilaa pakottamalla) ohitettuja ylimääräisiä tiloja, jotka liittyivät useamman nosturin väliseen kommunikointiin. Koska CXT Biomassa ei todellisuudessa tue useamman nosturin toimintaa, poistettiin nämä tilat uudesta mallista kokonaan. Toinen useasta sekvenssistä löytynyt käyttämätön tila oli ”dynaaminen ajokorkeuden säätö”. Alkuperäisestä mallista löytyivät käytännössä kaikki tarvittavat ominaisuuteen liittyvät parametrit, mutta itse ominaisuuden suorittavaa funktiota ei kuitenkaan ollut olemassa. Huolellisen harkinnan jälkeen päätettiin, että poistamisen sijaan kyseinen toiminnallisuus lisättäisiin uuteen malliin. Toiminnallisuudelle suunniteltiin uusi funktio, joka asettaa ajokorkeuden perustuen solualueen korkeimpaan kohtaan ja ennalta määriteltyyn turvamarginaaliin (offset).

Käyttämättömien tilojen lisäksi alkuperäisestä koodista löytyi useissa sekvensseissä tapahtuvia identtisiä toimenpiteitä, esimerkkeinä sekvenssien statistiikkatietojen tallentaminen tai virhetilanteen myötä tehtävä kahmarin avaaminen, jotka päätettiin siirtää omiin funktioihinsa. Statistiikkatietoja keräävä funktio toteutettiin modulaarisella rakenteella, jolloin jokaiselle sitä käyttävälle sekvenssille voitiin määritellä kyseistä työtehtävää vastaavat sisään- ja ulostulot. Kahmarin avaamisesta taas tehtiin kokonaan oma sekvenssinsä, joka voitiin virhetilanteissa aktivoida useista eri sekvensseistä käsin. Myös muita virhetilanteisiin liittyviä epäloogisuuksia korjattiin paremmiksi. Alun perin esimerkiksi virhetilanteen syntyminen aktivoi *”työ valmis”* -muuttujan, jonka perusteella ohjaussekvenssi osasi lopettaa kyseisen sekvenssin, mutta samaan aikaan kyseinen statustieto myös aktivoi onnistuneista työkierroista kerättyjen statistiikkatietojen keräämisen. Uuteen malliin lisättiin virhetilanteita varten kokonaan uusi, virhetilanteista indikoiva muuttuja, joka lisättiin ohjaussekvenssissä jokaisen sekvenssin keskeyttämisehdoksi. Statistiikkatietojen keräämiselle voitiin nyt myös määritellä omat ehtonsa, jolloin tiettyjä tietoja kerätään vain onnistuneiden työkiertojen yhteydessä.

Sekvenssien yhteyteen kuului myös nosturin maalipisteen etsimiseen käytetty funktio, joka vaati osaltaan suuria uudistuksia. Maalipisteen etsiminen on yksi käytännössä kaikkien työsekvensseiden tiloista, joka aktivoidaan sekvenssin tietyssä vaiheessa. Maalipiste määritellään eri tavoin riippuen minkä sekvenssin aikana se aktivoidaan. Syöttöhaku-sekvenssin aikana maalipisteeksi määritellään sillä hetkellä korkeimman materiaalitason omaava syöttöhaku-solu, varastointisekvenssin aikana etsitään vastaavasti korkeimman materiaalitason omaavaa varastointihaku-solua ja talletussekvenssin aikana etsitään matalimman materiaalitason omaavaa talletussolua. Ulossyöttösekvenssin aikana taas maalipisteeksi valitaan sillä hetkellä ensimmäisenä jonossa olevan ulossyöttökohteen koordinaatit.

Aikaisemmin maalipisteen etsintään liittyvät funktiot oli jaoteltu kolmeen osa-alueeseen, ”syöttökohde”, ”varastointikohde” ja ”odottamiskohde”. Funktiot uudistettiin dynaamisia taulukoita käyttäen ja jaoteltiin uudelleen, paremmin uuden

mallin rakenteeseen sopivaksi. Ulossyöttökohteiden valinta eriytettiin ”syöttökohteet” -osuudesta omaan funktioonsa, ja toteutettiin jälleen modulaarisuuden periaatteita noudattaen, jotta erityyppisten ulossyöttökohteiden käyttö olisi mahdollista. Syöttöhaku-, varastointihaku- ja talletuskohteiden, eli kaikkien solualueen sisällä olevien kohteiden, valintaa varten suunniteltiin yksi yleiskäyttöinen funktio, josta luotiin omat instanssit jokaista työtehtävää kohden. Jokaiselle instanssille määriteltiin omat aktivoitumisehdot (jotta vain yksi instansseista voi olla kerrallaan aktiivisena, riippuen mistä sekvenssistä maalipisteen haku on aktivoitu) ja omat parametrit. Tärkeimpänä parametrina käytettiin työtehtäväkohtaista tietoa matalimman ja korkeimman materiaalitason omaavista soluista. Odottamiskohteen (eli sijainti johon nosturi siirtyy, kun mitään varsinaisista työtehtävistä ei ole saataville) valintaan liittyvä funktio ei vaatinut suuria muutoksia. Muuttuneet kutsupolut päivitettiin ajan tasalle, ja joitain operaatioita voitiin yksinkertaistaa, mutta funktion toiminnallisuus säilyi tismalleen samana kuin aikaisemminkin.

5.4.2.6 Ulostulot

Ulostuloja koskeva osuus ei alkuperäisessäkään koodissa sisältänyt kovin montaa funktiota, mutta lähestulkoon kaikki operaatiot liittyivät datan kopiointiin väliaikaisdatalohkoista toiseen. Koska uudessa mallissa datan kulkua yksinkertaistettiin huomattavasti, kaikki nämä operaatiot voitiin poistaa ja lopulta funktion alaisuuteen jäikin ainoastaan yksi operaatio.

5.4.3 Nosturin ajosekvenssit, reititys ja niiden apufunktiot

Automaatio-osuuksista viimeiseen kuuluivat nosturin ajosekvenssit, reititys ja näihin molempiin liittyvät apufunktiot. Alkuperäisessä mallissa alue koostui yhteensä neljästätoista eri osa-alueesta, mutta kutsurakenteen selkeyttämiseksi uudessa mallissa funktiot ryhmiteltiin seuraaviin kuuteen osa-alueeseen:

1) Rajojen ja reititysalueiden määrittely

2) Nosturin reititys
3) Aloitus- ja lopetusehdot
4) Monitorointi
5) Ajosekvenssit
6) Hälytykset

Osa näistä osa-alueista haluttiin jättää asiakkaan puolesta ennalleen. Aivan kaikkia alun perin suunnitelmaan kuuluneita funktioita ei myöskään ehditty opinnäytetyön puitteissa tekemään valmiiksi, joten tässä kappaleessa osa-alueita käsitellään sen mukaan, mitä toimenpiteitä niille ehdittiin tekemään ja mitä toimenpiteitä osa-alueelle on tarkoitus vielä tehdä.

5.4.3.1 Rajojen ja reititysalueiden määrittely

CXT Biomassalle on määritelty useita erilaisia rajoja, jotka ovat tärkeitä nosturin ajosekvenssien kannalta. Rajoja voivat olla esimerkiksi kahmarin minimikorkeus tietyllä alueella ajamiseksi tai vaunulle/sillalle määritellyt päätyrajat. Rajat voivat olla toteutettu fyysisillä antureilla tai ne voivat olla laskentaan perustuvia ohjelmistopohjaisia rajoja, joita tässä osuudessa käsitellään. Useiden rajoihin liittyvien datalohkojen rakennetta uudistettiin itse määritellyillä datatyypeillä ja globaalivakioilla määritellyillä taulukoilla. Rajoihin liittyviä operaatioita siirrettiin dynaamisten silmukoiden sisään ja niihin liittyvien muuttujien kutsupolkuja päivitettiin tukemaan datalohkojen uusia rakenteita. Kaikkia rajojen määrittelyyn liittyviä datalohkoja tai operaatioita ei opinnäytetyön aikana ehditty uudistamaan, ja niiden osalta työ jatkuukin projektin seuraavassa vaiheessa.

Rajojen lisäksi nosturille on määritelty niin sanottuja reititysalueita, joiden perusteella nosturia ohjataan paikasta toiseen. Tässä osuudessa haettiin monia erilaisia reititykseen liittyviä tietoja, kuten esimerkiksi onko nosturin sijainti sillan ja/tai vaunun suuntaisesti linjassa jonkun alueen vyöhykkeellä. Siinä missä al-

kuperäisessä mallissa näitä tietoja haettiin vain muutamia tiettyjä alueita koskien, päätettiin uudessa mallissa tiedot hakea kaikkia reititysaluetta koskien. Toiminnallisuutta varten luotiin kokonaan uusi funktio, joka ottaa sisääntulonaan tietyn alueen rajatiedot sisältävän taulukon, ja määrittelee automaattisesti taulukon koon perusteella FOR-silmukan tehtäville operaatioille. Erityyppisille alueille määriteltiin omat instanssit. Toiminnallisuutta parannettiin aikaisempaan verrattuna lisäämällä tarkistus myös nosturin maalipisteen sen hetkistä aluetta koskien. Myöhemmin toiminnallisuutta laajennettiin entisestään ja muualla tapahtuneita operaatioita (esimerkiksi onko nosturin sillan/vaunun sijainti linjassa sen hetkisen kohdealueen kanssa) keskitettiin tämän funktion alaisuuteen. Funktion keräämiä tietoja varten luotiin myös oma datatyyppi, ja jokaista erityyppistä aluetta kohtaan luotiin kyseisestä datatyypistä koostuva oma taulukkonsa. Lisäksi luotiin yksi suurempi taulukko, johon kaikkien erityyppisien alueiden tiedot yhdistettiin, jotta alueista kerättyjä tietoja voitaisiin sujuvasti käyttää osana uutta reitityssysteemiä. Taulukoiden yhdistämistä varten luotiin oma funktionsa, joka suunniteltiin siten, että sitä voidaan käyttää minkä tahansa taulukoiden yhdistämiseen, kunhan lähde- ja kohdetaulukoiden datatyypit ovat samat. Funktio toimii siten, että jokaista kopioitavaa taulukkoa kohden luodaan oma instanssinsa, jotka ketjutetaan TIA Portalissa peräkkäin. Funktio lukee kunkin taulukon koon, ja lisää sen kohdetaulukossa edellisen taulukon perään. Funktion parametreissä määritellään lähdetaulukko, kohdetaulukko, indeksimuuttuja ja ketjun ensimmäinen instanssi. Näistä kohdetaulukon ja indeksimuuttujan on oltava kaikissa instansseissa samat. Funktion toimintaa demonstroidaan liitteessä 2.

5.4.3.2 Nosturin reititys

Reititys on sekvenssien ohella kenties tämän osuuden tärkein osa-alue ja siihen jouduttiin tekemään paljon radikaaleja muutoksia. CXT Biomassan reititys perustuu reititysalueisiin, jotka määritellään projektikohtaisesti erillisen excel-asiakirjan avulla. Kun reititykseen liittyvät tiedot on määritelty asiakirjaan, niistä luodaan excel-skriptien avulla AWL-tiedostomuodossa olevat tiedostot, jotka voidaan tuoda TIA Portaliin datalohkoina. Uudessa mallissa kyseisten datalohkojen rakennetta parannettiin omien datatyyppien ja dynaamisten taulukoiden

avulla, mutta niiden luomisesta vastaaviin excel-skripteihin ei kuitenkaan vielä opinnäytetyön aikana ehditty tekemään muutoksia. Niiden päivittäminen projektin seuraavassa vaiheessa onkin siis vielä tarpeellista, jotta luotujen datalohkojen rakenne vastaisi uuteen malliin suunniteltua rakennetta.

Reititysfunktioiden uudistaminen aloitettiin maalipisteiden valinnan tekevästä funktiosta. Alkuperäinen reititys pohjautui hyvin vahvasti suorien muistiviittausten perusteella tehtyihin toimenpiteisiin ja esimerkiksi välivaiheissa käytettyjen maalipisteiden valinta tehtiin viittaamalla muuttujiin suoraan niiden muistiosoitteen perusteella. Suoria viittauksia muistiosoitteisiin käytettiin jopa reittejä excel-asiakirjaan määriteltäessä. Maalipisteen valinnan tekevä funktio kirjoitettiin kokonaan uudelleen siten, että muistiosoitteiden sijaan valintaan käytetään numeroindeksiä. Funktiosta luotiin omat instanssit jokaista nosturin pääkoneistoa (nosto, silta ja vaunu) kohden.

Seuraavaksi kirjoitettiin uudelleen reitin valintaan liittyvät funktiot. Reittien valinta tehdään excel-asiakirjaan tehtyjen määrittelyiden perusteella, riippuen siitä millä reititysalueella nosturin sen hetkinen sijainti ja maalipiste ovat. Alkuperäisessä mallissa tämä tieto etsittiin suoraan reititysfunktioiden yhteydessä, vertaamalla nykyistä sijaintia kunkin alueen rajoihin, kun taas uudessa mallissa tiedot etsitään jo edellisellä alueella (5.4.3.1. Rajojen ja reititysalueiden määrittely) ja reititys käyttää samoja tietoja hyväkseen.

Varsinainen reitin valinta suoritetaan reitityksen hallintasekvenssin yhteydessä, joka kirjoitettiin kokonaan uudelleen SCL-kielellä. Alkuperäisessä mallissa valinta tehtiin kahden erillisen funktion voimin, mutta uudessa mallissa nämä poistettiin ja niiden toiminnallisuus sijoitettiin suoraan hallintasekvenssin alaisuuteen. Tiedot haetaan excel-tiedostojen perusteella luoduista datalohkoista ja valintaan käytetyt silmukat kirjoitettiinkin uudelleen siten, että ne ottavat huomioon datalohkojen uuden rakenteen ja käyttävät rajoinaan globaaleita vakioita. Myös diagnostiikkaa parannettiin lisäämällä varmistus sille, että valittu reitti on määritelty oikein.

Viimeisenä uudistettiin varsinainen reititназsekvenssi, joka kirjoitettiin kokonaan uudelleen SCL-kielellä. Funktion varsinainen toiminnallisuus pidettiin pääpiirteittäin ennallaan, mutta siitä saatiin tehtyä dynaamisempi ja luettavuudeltaan huomattavasti selkeämpi. Myös funktion sisään- ja ulostulojen määrää onnistuttiin pienentämään huomattavasti; siinä missä alkuperäisestä funktiosta löytyi yhteensä 46 sisään/ulosulostuloa, tarvitsi näitä uudessa mallissa olla vain 17. Tämä saavutettiin lisäämällä omien datatyyppien käyttöä funktioon liittyvissä datalohkoissa, jolloin monia samasta datalohkosta käytettyjä muuttujia saatiin tuotua funktioon yhdellä kertaa.

Reititysosuuteen kuului myös muita funktioita, jotka liittyivät lähinnä nosturin paikoitukseen ja nopeudensäätöön. Osa näistä funktioista oli tyypiltään funktiolohkoja, joiden tapauksessa itsenäiset instanssi-datalohkot poistettiin ja luotiin uudelleen multi-instansseina reitityksen pääfunktion alaisuuteen, jonka tyyppi muutettiin myös funktiolohkoksi. Tällöin kaikki reititykseen liittyvät instanssidatalohkot saatiin kätevästi keskitettyä yhteen paikkaan. Myös funktioihin liittyvien muuttujien kutsupolut päivitettiin ajan tasalle, mutta funktiot itsessään jätettiin kuitenkin ennalleen, sillä ne liittyivät enemmänkin nosturin varsinaiseen pohjaohjelmistoon kuin automaatiomoodiin.

5.4.3.3 Aloitus- ja lopetusehdot

Aloitus- ja lopetusehdot ryhmitettiin uudessa mallissa yhdeksi osioksi. Funktiot tuotiin projektiin sellaisenaan ja ainoastaan muuttujien kutsupolut täytyi päivittää ajan tasalle.

5.4.3.4 Monitorointi

Monitorointi-osioon ryhmitettiin nostosekvenssien monitorointiin liittyvät sekvenssit, joita oli kaksi kappaletta. Toisen sekvenssin tehtävänä oli monitoroida noston alarajoja, kun taas toinen sekvenssi monitoroi mahdollisia ylikuormatilanteita. Asiakkaan toiveesta sekvenssit pidettiin ennallaan ja tuotiin uuteen

malliin sellaisenaan. Muuttujien kutsupolut päivitettiin kummastakin sekvenssistä ajan tasalle, mutta muutoin kaikki jätettiin ennalleen.

5.4.3.5 Ajosekvenssit

Nosturin ajosekvenssit ovat sekvenssejä, jotka aktivoidaan työsekvenssien hakemien tietojen perusteella. Asiakkaan toiveesta ajosekvenssit haluttiin säilyttää STL-kielisenä, mahdollisimman vähäisillä muutoksilla. Niinpä sekvenssit tuotiin uuteen malliin sellaisenaan. Osassa sekvensseissä päivitettiin kuitenkin suoraan tietoja nosturin reititystä varten, ja näiden osalta muutoksilta ei pystytty välttymään. Lisäksi nostoon ja talletukseen liittyvät sekvenssit muutettiin tyypiltään funktiolohkoiksi, jotta niiden alaisuudessa olleiden muiden funktiolohkojen instanssidatalohkot voitiin poistaa ja luoda uudelleen multi-instansseina. Reititystä ohjaavat osiot päivitettiin tukemaan uutta reitityssysteemiä, jossa suorien muistiviittausten sijaan käytettiin kohdenumeroita. Myös monien sekvensseissä käytettyjen muuttujien kutsupolut olivat uudistuksen aikana muuttuneet, joten ne täytyi päivittää ajan tasalle ja koska osaa käytetyistä datalohkoista ei ollut tuotu vielä lainkaan uuteen malliin, ne täytyi tuoda projektiin tässä vaiheessa. Uusista datalohkoista poistettiin käyttämättömiä muuttujia, ja lisäksi niille tehtiin jaottelua parametrimuuttujien ja väliaikaismuuttujien välillä.

5.4.3.6 Hälytykset

Myös hälytyksiä koskevat funktiot tuotiin uuteen malliin sellaisenaan, mutta jotkut funktiot vaativat yhteensopivuusongelmien vuoksi pieniä muokkauksia, sillä uusi prosessori ei tukenut aikaisemmin käytettyjä käskyjä. Myös muutamia ulossyöttökohteisiin liittyviä hälytyksiä jouduttiin hieman muokkaamaan, jotta ne saatiin tukemaan uuden nosturikoodin dynaamista rakennetta. Lisäksi muuttujien kutsupolut täytyi päivittää ajan tasalle.

6 Pohdintaa

6.1 Tavoitteiden toteutuminen

Tämän opinnäytetyön aiheena oli modernisoida biomassan käsittelyyn tarkoitetun teollisuusnosturin PLC-koodista ne osa-alueet, jotka liittyivät nosturin automaatioon. Työn aikatauluksi oli määritelty puoli vuotta, ja lähes kaikki osa-alueet ehdittiin saada valmiiksi annetun aikataulun puitteissa. Automaatio-osuus oli jaoteltu kolmeen eri osa-alueeseen, jotka sisälsivät yhteensä yli 100 funktiota, sekä useita kymmeniä datalohkoja. Päätavoitteina olivat dynaamisuuden parantaminen ja datan kulun selkeyttäminen, jotka on käsitelty erikseen omissa alaluvuissaan. Datankulun selkeyttämisen osalta tavoitteet oli lisäksi jaettu neljään pienempään osa-alueeseen. Lisäksi opinnäytetyön osana tutustuttiin logiikkaohjelmoinnin teoriaan, hyviin käytäntöihin sekä eroihin alkuperäisen ja uuden alustan, sekä niiden ohjelmointiympäristöjen välillä. Opittuja tietoja käytettiin hyödyksi asetettujen tavoitteiden saavuttamisessa.

6.1.1 Dynaamisuuden parantaminen

Funktiot ja datalohkot uudelleenkirjoitettiin hyödyntäen TIA Portalin tarjoamia uusia toiminnallisuuksia. Erinäisiin resursseihin liittyvien datalohkojen sisällöt siirrettiin dynaamisin taulukoihin, joiden koko määriteltiin globaaleiden vakioiden perusteella. Funktiot kirjoitettiin uudelleen niin, että suoriin muistiviittauksiin perustuvat silmukat poistettiin ja korvattiin dynaamisilla silmukoilla. Funktiosta ja siihen liittyvästä resurssista riippuen silmukoiden rajat määriteltiin joko suoraan globaalilla vakiolla tai vaihtoehtoisesti lukemalla lähde/kohdetaulukon rajat funktion suoritushetkellä. Jälkimmäistä metodia käytettiin sellaisten funktioiden kanssa, joiden haluttiin olevan toiminnallisuudeltaan modulaarisia, eli funktiota haluttiin käyttää useammassa eri instanssissa erikokoisten lähde/kohdetaulukoiden kanssa. Joidenkin käyttöliittymään liittyvien datalohkojen/funktioiden suhteen jouduttiin tekemään kompromisseja, joiden pariin joudutaan mahdollisesti

palaamaan myöhemmin. Kokonaisuutena uudesta mallista saatiin kuitenkin paljon dynaamisempi verrattuna alkuperäiseen ratkaisuun ja lopputuloksen voidaan todeta olleen melko onnistunut.

6.1.2 Datan kulun selkeyttäminen

6.1.2.1 Turhan datansiirtelyn vähentäminen

Turhaa datan siirtelyä vähennettiin etenkin työsekvenssien ja ajosekvenssien välillä, mutta myös monia parametrien tarpeettomia kopiointeja poistettiin koodista. Monien funktioiden tapauksessa muuttujia ainoastaan luettiin väliaikaisista sijainneista, jolloin väliaikaismuuttujan käyttäminen oli tarpeetonta. Tällaisissa tilanteissa väliaikaismuuttujat poistettiin ja funktiot päivitettiin kutsumaan suoraan alkuperäisiä parametrejä. Lopputuloksena datan kulkua saatiin selkeytettyä huomattavasti, kun yksittäisen tiedon ristiviittauksia ei tarvitse seurata lukuisten datalohkojen välillä.

6.1.2.2 PLC datatyyppien käytön lisääminen

PLC datatyyppien, eli itse määriteltyjen datatyyppien käyttöä lisättiin huomattavasti. Kenties parhaana esimerkkinä PLC datatyyppien onnistuneesta käytöstä voidaan pitää solujen määrittelyyn tarkoitettua taulukkoa, joka oli kenties käytetyin taulukko koko automaatio-osuuteen liittyvässä koodissa. Kyseinen taulukko määriteltiin käyttämään omaa PLC datatyyppiään, joka puolestaan koostui useista muista itse määritellyistä datatyypeistä. Projektin edetessä datatyypin rakenteeseen tehtiin paljon muutoksia ja lisättiin uusia ala-datatyyppejä. Lopputuloksena hyvin suuri osa soluihin liittyvistä tiedoista saatiin päivitettyä keskitetysti yhden, omasta PLC datatyypistä koostuvan, taulukon alaisuuteen. PLC datatyyppien käytön ansiosta datalohkon rakennetta ei tarvinnut kerralla lyödä lukkoon ja näin suunnitteluun saatiin joustavuutta ja säästettiin aikaa, sillä muutokset päivittyivät automaattisesti joka paikkaan. Jälkikäteen ajateltuna huomattiin, että samaa periaatetta olisi kannattanut hyödyntää myös muihin resursseihin (esim. ulossyöttökohteet) liittyvien tietojen kanssa. Suunnitelmissa onkin, että

ennen seuraavaan vaiheeseen siirtymistä, myös muihin resursseihin liittyvät tiedot keskitettäisiin yhdestä datatyypistä koostuvan taulukon alaisuuteen.

Omien datatyyppien käyttöä lisättiin myös monissa muissa paikoissa. Esimerkiksi nosturin parametreja tai statustietoja yhdistettiin yhden datatyypin alaisuuteen, jolloin useita samaan aiheeseen liittyviä tietoja voitiin tuoda kerralla funktioille niiden sisään/ulostulojen rajapinnan kautta.

6.1.2.3 Parametrimuuttujien erittely ja absoluuttiarvojen poisto

Parametrimuuttujien erittely väliaikaismuuttujista oli kenties yksi haastavimmista osa-alueista koko projektissa. Muuttujien käyttöstatusten (luku/kirjoitus) perusteella on mahdollista määrittää, onko kyseessä parametrimuuttuja (jota ainoastaan luetaan) vai väliaikaismuuttuja (jonka päälle myös kirjoitetaan). Käyttöstatukset täytyi usein tarkistaa jokaiselta datalohkon muuttujalta erikseen ja tämä saattoi viedä hyvin paljon aikaa, varsinkin mikäli dataa kopioitiin paikasta toiseen. Jotkut parametreistä esimerkiksi datatyypitettiin ja siirrettiin keskitetysti nosturin parametrejä sisältävään datalohkoon. Absoluuttiarvot taas korvattiin uusilla parametrimuuttujilla, jotka sijoitettiin sopivan asiayhteyden omaaviin datalohkoihin.

Joissain tapauksissa datalohkon rakenne säilytettiin ennallaan, esimerkiksi jos datalohko liittyi selkeästi enemmän nosturin varsinaiseen pohjaohjelmistoon, mutta muutamia sen muuttujista käytettiin myös automaatio-osuudessa. Kuitenkin myös näiden datalohkojen osalta saatettiin tehdä pintapuolinen tarkistus, jossa tarkistettiin, onko datalohko linjassa sen sisältämien muuttujien kanssa. Mikäli esimerkiksi datalohkon tulisi sen nimen perusteella sisältää vain väliaikaismuuttujia, mutta sen huomattiin sisältävän myös parametrejä, lisättiin nimeen tarkenne ”_PV”, joka kertoo datalohkon sisältävän niin parametrejä kuin väliaikaismuuttujia.

Lopputuloksena voidaankin todeta, että yksinomaan automaatio-osuuteen liittyvien datalohkojen suhteen parametrit saatiin eriteltyä suhteellisen onnistuneesta, mutta myös pohjaohjelmistoon liittyvien datalohkojen osalta muutosten tekoa täytyy jatkaa projektin seuraavassa vaiheessa.

6.1.2.4 Koodin luettavuuden parantaminen

Koodin luettavuuden parantamisen voidaan katsoa olevan muiden osa-alueiden onnistumisen summa. Esimerkiksi silmukoiden osalta jo ohjelmointikielen vaihto (STL → SCL) itsessään toi koodiin huomattavasti lisää selkeyttä. Tämän lisäksi kaikkien uudistettujen funktioiden luettavuutta parannettiin lisäämällä koodiin enemmän kommentteja, sekä uudelleennimeämällä epäselvästi nimettyjä muuttujia siten, että nimestä itsestään kävisi paremmin ilmi muuttujan käyttötarkoitus. Myös turhien väliaikaismuuttujien poistamisella voidaan katsoa olevan positiivinen vaikutus koodin luettavuuteen, sillä datan jäljitys saadaan tehtyä aikaisempaa nopeammin.

6.2 Yhteenveto

Tehtyjen muutosten ansioista nosturin projektikohtainen konfigurointi voidaan tehdä tulevaisuudessa nopeammin ja nosturin markkinointi asiakkaille voi olla helpompaa, sillä koodi taipuu joustavammin erilaisiin tarpeisiin ja projektikohtaisten muutosten teko on helpompaa. Määritellyt tavoitteet saavutettiin suurimmilta osin onnistuneesti ja aikataulussa onnistuttiin pysymään. Koodin modernisointiin kului aikaa noin 6 kuukautta, eli työtunteja kertyi arviolta 6kk * 20 päivää * 7,5 tuntia = 900 tuntia, josta opinnäytetyön kirjoittamiseen ja teorian opiskeluun kului arviolta kollektiivisesti noin yksi kuukausi (= 150 tuntia). Paranneltavaakin kuitenkin jäi ja työ nosturikoodin modernisoinnin parissa jatkuu edelleen. Projektin siirtyessä seuraavaan vaiheeseen on tarpeen tehdä päätös siitä, päivitetäänkö nosturin varsinainen pohjaohjelmisto uuden sukupolven versioon, joka tarjoaa monia etuja mm. turva- ja kunnossapitosovellusten suhteen, vai muokataanko alkuperäinen pohjaohjelmisto tukemaan uutta mallia. Tämän lisäksi on tehtävä päätös käyttöliittymän modernisoinnin suhteen.

Lähteet

Cleverism. Verkkoaineisto. Ladder Logic. <https://www.cleverism.com/skills-and-tools/ladder-logic/>. Luettu 15.2.2022.

Cross, Don. 2020. Is Goto Always Evil? Verkkoaineisto. Gitconnected. <https://levelup.gitconnected.com/is-goto-always-evil-d31c6c945398>. 19.5.2020. Luettu 10.11.2021.

Denilson Pegaia (nimimerkki). 2015. TIP: STEP7 V13 SP1 and the use of global constants as array limits. Siemens keskustelufoorumi. <https://support.industry.siemens.com/tf/ww/en/posts/tip-step7-v13-sp1-and-the-use-of-global-constants-as-array-limits/120164>. 1.10.2015. Luettu 27.10.2021.

Reed, Jessica. 2022. What is Logical Programming? Verkkoaineisto. EasyTechJunkie. <https://www.easytechjunkie.com/what-is-logical-programming.htm>. 30.1.2022. Luettu 15.2.2022.

Konecranes nosturiesite. 2016. Nosturiesite. Biomass – waste to energy brochure.

Konecranes. -a. Verkkoaineisto. Laitteet. <https://www.konecranes.com/fi/laitteet>. Luettu 25.10.2021.

Konecranes. -b. Verkkoaineisto. Huolto. <https://www.konecranes.com/fi/huolto>. Luettu 26.10.2021.

Konecranes. -c. Verkkoaineisto. Biomassanosturit. <https://www.konecranes.com/fi/teollisuudenalat/energiajatteen-kasittely/jatteenkasittely-ja-biomassanosturit/biomassanosturit>. Luettu 25.10.2021.

Konecranes PLC programming guide. 2020. Yrityksen sisäinen dokumentti. Konecranes.

Collins, Danielle. 2019. Why is the instruction list (IL) language for PLCs falling out of favor? Verkkoaineisto. Motioncontroltips. <https://www.motioncontroltips.com/why-is-the-instruction-list-il-language-for-plcs-falling-out-of-favor/>. 16.7.2019. Luettu 16.2.2022.

OEM. Simatic S7-300 tuotesivu. <https://www.oem.fi/tuotteet/logiikat-ja-kaytot/logiikat/simatic-s7-300-_-730023>. Luettu 15.11.2021.

Peter (artikkelin kirjoittajan nimimerkki sivustolla). 2017. PLC Ladder Logic Programming Tutorial (Basics). Verkkoaineisto. PLC Academy

<https://www.plcacademy.com/ladder-logic-tutorial/>. 4.9.2017. Luettu 15.2.2022.

Peter (artikkelin kirjoittajan nimimerkki sivustolla). 2018. Function Block Diagram (FBD) PLC Programming Tutorial for beginners. Verkkoaineisto. PLC Academy <https://www.plcacademy.com/function-block-diagram-programming/>. 13.3.2018. Luettu 15.2.2022.

PLCopen. -a. What is PLCopen. Verkkoaineisto. PLCopen. <https://plcopen.org/what-plcopen>. Luettu 16.2.2022.

PLCopen. -b. Certification. Verkkoaineisto. PLCopen. <https://plcopen.org/technical-activities/certification>. Luettu 16.2.2022.

Siemens. 1998. Structured Control Language (SCL) for S7-300/S7-400 Programming. Manual.

Siemens. 2013 -a. Verkkoaineisto. How do you program the PLC with STEP 7 (TIA Portal) in compliance with the IEC 61131-3 standard? <https://support.industry.siemens.com/cs/document/50204938/how-do-you-program-the-plc-with-step-7-(tia-portal)-in-compliance-with-the-iec-61131-3-standard-?dti=0&lc=en-WW>. 13.11.2013. Luettu 16.2.2022.

Siemens. 2013 -b. Verkkoaineisto. Delivery Release for the new SIMATIC S7-1500 Controllers. <https://support.industry.siemens.com/cs/document/67856446/delivery-release-for-the-new-simatic-s7-1500-controllers?dti=0&lc=en-US>. 27.2.2013. Luettu 16.2.2022.

Siemens. -a. Verkkoaineisto. Simatic S7-300 tuotesivu. <https://new.siemens.com/global/en/products/automation/systems/industrial/plc/simatic-s7-300.html>. Luettu 15.2.2022.

Siemens. -b. Verkkoaineisto. Software for SIMATIC Controllers – The STEP 7 family. <https://new.siemens.com/global/en/products/automation/systems/industrial/controller-sw.html>. Luettu 16.2.2022.

Siemens. -c. Verkkoaineisto. TIA Portal tuotesivu. < https://new.siemens.com/global/en/products/automation/industry-software/automation-software/tia-portal.html>. Luettu 16.2.2022.

Siemens. 2018. Programming Guideline for S7-1200/1500 (Entry ID: 81318674, V1.6, 12/2018).

Realpars. Verkkoaineisto. What are the differences between simatic S7-300 and S7-1500 PLCs? <https://realpars.com/s7-300-versus-s7-1500/>. Luettu 15.11

Liite 1. Mallipohja SCL-kielellä toteutetulle tilakoneelle.

SCL_StateMachine

	Name	Data type	Default value	Comment
4	<Add new>			
5	InOut			
6	IQ_State	Byte		
7	Temp			
8	<Add new>			
9	Constant			
10	BYPASS	Byte	-1	Bypass this state machine
11	St0_	Byte	0	
12	St1_	Byte	1	
13	St2_	Byte	2	

IF... CASE... OF... FOR... TO DO... WHILE... DO... (*...*) REGION

- Description
- BYPASS
- State 0:
- State 1:
- State 2:
- State 3:
- State 4:
- State 5:

```
1  REGION Description
2  (*---------------------------------------------------------------------------
3
4  FUNCTION:           Function Title
5
6  SYMBOLIC NAME:      Symbolic name
7
8  DESCRIPTION:
9                      0: State 0 description
10                     1: etc etc
11                     2: etc etc
12
13 PROJECT/PRODUCT:    CXT Biomass
14
15 AUTHOR:             Konecranes/APK/MLI
16
17 VERSION:            V1.0.0
18
19 REVISION HISTORY:
20                     V1.0.0 author-name 10.11.2021
21                     -
22
23 REMARKS:          -
24
25 (C) Konecranes. All rights reserved.
26
27 ---------------------------------------------------------------------------*)
28 END_REGION
29
30 // Go to the selected state:
31 CASE #IQ_State OF
32 #BYPASS:
33 REGION BYPASS
34     ;
35 END_REGION
36
37
38 #"St0_":
39 REGION State 0:
40     ;
41 END_REGION
42
43
```

Liite 2. Taulukoiden kopioimiseen suunniteltu funktio.