Juho Feldt

Ohjausjärjestelmien simulaattorin modernisointi

Opinnäytetyö
Kevät 2020
SeAMK Tekniikka
Tietotekniikan tutkinto-ohjelma

SEINÄJOEN AMMATTIKORKEAKOULU

Opinnäytetyön tiivistelmä

Koulutusyksikkö: Tekniikan yksikkö

Tutkinto-ohjelma: Tietotekniikka

Suuntautumisvaihtoehto: Ohjelmistotekniikka

Tekijä: Juho Feldt

Työn nimi: Ohjausjärjestelmien simulaattorin modernisointi

Ohjaajat: Jyri Lehto, Juha Viitanen

Vuosi: 2020 Sivumäärä: 43 Liitteiden lukumäärä: 0

Tämän opinnäytetyön tavoitteena oli aikaisemman ohjausjärjestelmien simulaattorin siirtäminen uuteen kehitysympäristöön ja samalla muokata siitä dynaamisempi, optimoidumpi ja modulaarisempi. Aikaisempi ohjausjärjestelmien simulaattori oli tehty kehitysympäristöllä, joka ei mahdollistanut simulaattoriohjelman lähdekoodin luovuttamista asiakkaiden käyttöön.

Uuden ohjausjärjestelmien simulaattorin kehitysympäristöksi valittiin Microsoftin Visual Studio ja ohjelmointikieleksi C#-ohjelmointikieli. Kehitysympäristön ja ohjelmointikielen vaihdolla päästiin eroon aikaisemman simulaattoriohjelman kehitysympäristön rajoitteista ja mahdollistettiin simulaattorin lähdekoodin luovutus asiakkaille.

Opinnäytetyön tuloksena syntyi uusi ohjausjärjestelmien simulaattori, joka oli aikaisempaan ohjausjärjestelmien simulaattoriin verrattuna dynaamisempi, optimoidumpi ja modulaarisempi. Uudesta ohjausjärjestelmien simulaattorista luotiin pohjarakenne, josta simulaattoreiden kehitystä voidaan jatkaa.

Opinnäytetyön toimeksiantajana toimi Exertus Oy, joka on erilaisten työkoneiden ohjausjärjestelmiin erikoistunut teknologiayritys.

Avainsanat: Ohjausjärjestelmät, Simulaattorit, Exertus, Digital twin, C#, .NET Framework, WPF, CAN

SEINÄJOKI UNIVERSITY OF APPLIED SCIENCES

Thesis abstract

Faculty: School of Technology

Degree programme: Information Technology

Specialisation: Software Engineering

Author: Juho Feldt

Title of thesis: Modernization of a Control Systems Simulator

Supervisor: Jyri Lehto, Juha Viitanen

Year: 2020 Number of pages: 43

The objective of the thesis was to port a previously used control systems simulator to a new development environment and make it more dynamic, optimised and modular. The previously used control systems simulator was made with a development environment that did not allow the source code of the program to be handed over to customers.

Microsoft's Visual Studio was chosen to be the development environment for the new control systems simulator alongside with the C# programming language. With the change of the development environment and programming language, the restrictions of the development environment of the previously used control systems simulator were overcome, and the source code of the simulator program was made available to customers.

As the result of the thesis a more dynamic, optimized and modular control systems simulator was created. The new control systems simulator was created to be used as a base structure, from which the development of simulators can be continued. The thesis was commissioned by Exertus Oy, a technology company specialized in control systems for various machines.

Keywords: control systems, simulators, Exertus, digital twin, C#, .NET Framework, WPF, CAN

SISÄLTÖ

Opinnäytetyön tiivistelmä ... 1
Thesis abstract ... 2
SISÄLTÖ ... 3
Kuvaluettelo ... 5
Käytetyt termit ja lyhenteet ... 7
1 JOHDANTO ... 9
1.1 Työn tausta ... 9
1.2 Työn tavoite ... 9
1.3 Työn rakenne ... 10
1.4 Toimeksiantaja ... 11
2 OHJAUSJÄRJESTELMÄT JA SIMULAATTORIT ... 13
2.1 Yleistä simulaattoreista ... 13
2.2 Yleistä ohjausjärjestelmistä ... 13
2.3 Ohjausjärjestelmien simulaattorit ... 16
3 NYKYINEN SIMULAATTORI ... 17
3.1 Kehitysympäristö ja ohjelmointikieli ... 17
3.2 Toiminta ... 17
3.3 Vahvuudet ja heikkoudet ... 18
4 UUDEN SIMULAATTORIN KARTOITTAMINEN ... 21
4.1 Kehitysympäristön valinta ... 21
4.2 Ohjelmointikielen valinta ... 21
4.3 Ohjelmistokehyksen valinta ... 22
4.4 Ohjelmistoprojektin valinta ... 22
5 UUSI SIMULAATTORI ... 23
5.1 Käyttöliittymä ... 23
5.2 Luokat ... 23
5.2.1 SocketThread-luokka ... 23
5.2.2 Utils-luokka ... 24
5.2.3 LogicalIOSignal-luokka ... 24
5.2.4 FunctionsHandler-luokka ... 25

5.3 Tietueet ... 26
5.4 Metodit ... 27
5.4.1 Signaalien alustaminen ... 27
5.4.2 Yhteyden luominen palvelimeen ... 28
5.4.3 Pakettien vastaanottaminen palvelimelta ... 29
5.4.4 Pakettien muuntaminen tietueiksi ... 29
5.4.5 Pakettien lähettäminen palvelimelle ... 30
5.4.6 Signaalien arvojen kyseleminen ... 31
5.4.7 Signaalien arvojen saaminen palvelimelta ... 33
5.4.8 Signaalien arvojen asettaminen ... 34
5.5 Metodien kutsuminen ... 34
5.6 Simulaatiot ... 35
6 TULOKSET ... 39
7 POHDINTA JA YHTEENVETO ... 40
LÄHTEET ... 42

Kuvaluettelo

Kuva 1. Exertuksen kokonaisarkkitehtuuri. 12

Kuva 2. Avoin säätöjärjestelmä (perustuu Keinänen ym. 2001, 206). 14

Kuva 3. Suljettu säätöjärjestelmä (perustuu Keinänen ym. 2001, 206). 15

Kuva 4. Simulaattorin toiminta. 18

Kuva 5. Esimerkki yhdisteestä tietueessa. 19

Kuva 6. Esimerkki signaalien lisäämisestä 20

Kuva 7. Esimerkki signaalien kyselemisestä. 20

Kuva 8. SocketThread-luokka. 24

Kuva 9. Tavujen nollaus -metodi. 24

Kuva 10. Signaaliluokka. 25

Kuva 11. FunctionsHandler-luokka. 26

Kuva 12. Esimerkki tietueen rakenteesta. 27

Kuva 13. Signaalien alustusmetodi. 28

Kuva 14. Yhteyden luominen palvelimelle 28

Kuva 15. Pakettien vastaanottaminen. 29

Kuva 16. Esimerkki paketin muuntamisesta tietueeksi. 30

Kuva 17. Tietueiden muuntaminen tavuiksi 31

Kuva 18. Signaalien kyseleminen palvelimelta. 32

Kuva 19. Signaalien kyselyiden tarkistus. 33

Kuva 20. Signaalien arvojen päivittäminen. 33

Kuva 21. Signaalien arvojen asettamisen tarkistus. 34

Kuva 22. Metodien lisääminen listaan. 35

Kuva 23. Metodien kutsuminen listalta. 35

Kuva 24. Esimerkki simulaatioluokasta. 36

Kuva 25. Esimerkki simulaatiometodin alustuksesta. 37

Kuva 26. Esimerkki lipputietojen käytöstä simulaatiossa. 37

Kuva 27. Esimerkki viittauksesta simulaatioluokkaan käyttöliittymästä. 38

Käytetyt termit ja lyhenteet

Asynkroninen	Ohjelmointityyli, jossa ohjelmalausekkeet voidaan suorittaa eri järjestyksessä.
Attribuutti	Luokan sisäinen ominaisuus, kuten esimerkiksi muuttuja.
Dynaaminen	Ohjelman ajonaikana muuttuva toiminta.
Header-tiedosto	Tiedosto, jossa esitellään kirjastofunktioiden parametrilistat. Yleisesti tunnetaan .h-päätteisenä tiedostona.
Implementoida	Toteuttaa tai ottaa käyttöön jokin ominaisuus.
JSON	Yksinkertainen avoimen standardin tiedostomuoto tiedonvälitykseen.
Kehitysympäristö	Ohjelma tai joukko ohjelmia, jolla ohjelmoija suunnittelee ja toteuttaa ohjelmistoa.
Kääntäjä	Tietokoneohjelma, joka kääntää ohjelmointikielisen lähdekoodin konekieliseksi binääritiedostoksi.
Lipputieto	Tieto, jonka avulla ohjelmassa voidaan havaita jokin tietyn toiminnon suoritus.
Luokka	Olio-ohjelmoinnissa olion rakenteen määritelmä.
Lähdekoodi	Ohjelman tekstimuotoinen ohjelmointikielen listaus.
Modernisointi	Nykyaikaistaminen. Tuoda jokin ajan tasalle.
Modulaarinen	Itsenäisistä osista koostuva kokonaisuus.
Ohjelmistokehys	Ohjelmistotuote, joka muodostaa ohjelman rungon.
Ohjelmistoprojekti	Kehitysympäristössä valittava projekti, joka määrittelee ohjelman ohjelmointikielen ja rakenteen.

Ohjelmoitava logiikka	Ohjelmoitava mikroprosessoripohjainen laite, jota käytetään automaatioprosessien ohjaamiseen.
Optimoitu	Optimaalinen. Optimoitujen arvojen mukainen.
Prosessi	Suoritettavien toimenpiteiden sarja, joka tuottaa määritellyn lopputuloksen.
Rajapinta	Silta, jonka avulla eri ohjelmat voivat keskustella keskenään.
Socket	Rajapinta tiedon lähettämiseen ja vastaanottamiseen päätepisteiden välillä joko verkossa tai prosessien välillä.
Staattinen	Ohjelman ajonaikana muuttumaton toiminta.
Syntaksi	Ohjelmointikielen kieliopin määritelmä.
Säie	Ohjelman yksittäisessä prosessissa ajettava rinnakkainen ohjelmasuoritus, joka jakaa prosessin kanssa saman muistiavaruuden.
TCP/IP	Internetliikennöinnissä käytettävän tietoliikenneprotokollan yhdistelmä.
Tietue	Tietoalkioiden looginen kokonaisuus.
UTF8	Merkkikoodaus, joka kykenee koodaamaan kaikki Unicode-kelvolliset koodipisteet yhdestä neljään 8-bittisiksi tavuiksi.
Virtuaalinen	Ohjelmallisesti toteutettu ilmentymä jostakin fyysisesti olemassa olevasta.
XAML	Microsoftin kehittämä XML-pohjainen kieli, jota käytetään jäsenneltyjen arvojen ja objektien alustamiseen.
XML	Rakenteellinen kuvauskieli, joka auttaa jäsentämään laajoja tietomassoja selkeämmin.

1 JOHDANTO

1.1 Työn tausta

Ohjausjärjestelmien kehitys ja suunnittelu on insinöörityötä, joka vaatii useiden eri ammattilaisten yhteistyötä. Ohjelmistosuunnittelijat, jotka tekevät ohjausjärjestelmien käyttöliittymiä ja ohjelmistoja eivät välttämättä tiedä juuri mitään koneesta, johon ohjausjärjestelmä asennetaan. Tarvittava tieto saadaan yleensä koneen valmistajalta ja koneen parissa työskenteleviltä toimihenkilöiltä. Ohjausjärjestelmää kehitettäessä ymmärrys koneesta ja sen käyttötarkoituksesta usein kasvaa. Tästä huolimatta ohjelmistosuunnittelijat eivät välttämättä saa samanlaista ymmärrystä koneen käytöstä, kuin esimerkiksi koneen lopulliset käyttäjät. (Viitanen 2020.)

Simulaattoreita käytetään yleensä erilaisten koneiden käytön koulutukseen, mutta niiden on huomattu olevan myös tärkeä osa koneiden kehitystä ja testaamista. Simulaattoreiden avulla voidaan simuloida ja samaan aikaan kehittää koneen ohjausjärjestelmän toimintaa. Samalla se antaa ohjelmistosuunnittelijoille ja muille toimihenkilöille mahdollisuuden ymmärtää koneen toimintaa paremmin. (Chung 2004, 1.3–1.4.)

Koneiden prosessien testaaminen voi olla kallista ja aikaa vievää riippuen prosessista. Joidenkin prosessien testaaminen voi myös viivästyä, jos konetta ei ole sillä hetkellä saatavilla. Simulaattorin avulla koneen prosessia voidaan testata vaikka kyseistä konetta ei olisi fyysisesti vielä edes olemassa. Lisäksi prosesseja voidaan testata ilman, että niistä syntyy suuria kustannuksia. Vaikka koneiden prosessit täytyy lopulta testata käytännössä, simulaattorien käyttäminen nopeuttaa testaamista ja lopullisen koneen läpivientiaikaa. (Viitanen 2020.)

1.2 Työn tavoite

Tämän opinnäytetyön tavoitteena oli tuoda aikaisempi ohjausjärjestelmien simulaattori uuteen kehitysympäristöön ja samalla muokata siitä dynaamisempi, opti-

moidumpi ja modulaarisempi. Aikaisemman simulaattorin kehitysympäristö oli osittain vaikeakäyttöinen, minkä takia simulaattoreiden kehityksestä oli tullut hankalaa. Kehitystä haluttiin helpottaa uudella kehitysympäristöllä, joka mahdollistaisi simulaattorin lähdekoodin luovuttamisen myös asiakkaiden käyttöön. Uudesta simulaattorista haluttiin luoda pohjarakenne, jonka avulla simulaattoreiden kehitystä voidaan jatkaa.

Aikaisempi simulaattori oli tehty RAD Studiolla ja C++-ohjelmointikielellä. Uusi simulaattori päätettiin tehdä Microsoftin Visual Studiolla ja C#-ohjelmointikielellä. Uusi kehitysympäristö ja ohjelmointikieli mahdollistaisivat valmiiden rajapintojen käytön, joita aikaisemman simulaattorin kehitysympäristö ja ohjelmointikieli eivät tukeneet. Ohjelmointikielen ja kehitysympäristön vaihdolla päästäisiin myös eroon header-tiedostoista, jotka olivat ongelma aikaisemman simulaattorin kehitysympäristön käyttöönotossa.

1.3 Työn rakenne

Toisessa luvussa käsitellään ohjausjärjestelmiä ja simulaattoreita. Luvussa kerrotaan lyhyesti simulaattoreista ja niiden käyttökohteista. Tämän jälkeen kerrotaan yleisesti ohjausjärjestelmistä ja niiden toteutuksista. Lisäksi luvussa tutustutaan lyhyesti kenttäväylään, kenttäväylämoduuleihin ja niiden suhteesta ohjausjärjestelmiin. Luvun lopussa kerrotaan lyhyesti ohjausjärjestelmien simulaattoreiden merkityksestä osana ohjausjärjestelmien kehitystä ja tutustutaan myös digitaalisen kaksosen käsitteeseen ja sen rooliin ohjausjärjestelmien simulaattoreissa.

Luvussa kolme tutustutaan nykyiseen ohjausjärjestelmien simulaattoriin ja sen kehitysympäristöön. Luvussa käydään läpi lyhyesti nykyisen ohjausjärjestelmien simulaattorin toiminta. Luvun lopussa tutustutaan nykyisen simulaattorin heikkouksiin ja vahvuuksiin ohjelmistokehittäjän näkökulmasta.

Luvussa neljä kartoitetaan uuden ohjausjärjestelmien simulaattorin rakennetta. Luvussa kerrotaan lyhyesti kehitysympäristön, ohjelmointikielen, ohjelmistokehyksen ja ohjelmistoprojektin valinnasta sekä tekijöistä, jotka vaikuttivat näihin valintoihin.

Luvusta viisi eteenpäin alkaa uuden ohjausjärjestelmien simulaattorin luominen Microsoftin Visual Studiolla. Luvussa käydään läpi uuden ohjausjärjestelmien simulaattorin rakennetta yleisesti. Luvussa esitellään tietueet ja luokat, joita uudessa simulaattorissa käytetään. Samalla esitetään ratkaisuja, jotka tekevät uudesta simulaattorista dynaamisemman, optimoidumman ja modulaarisemman. Luvun lopussa esitellään myös esimerkki simulaatioista, joita simulaattoriin on tarkoitus luoda.

Luvussa kuusi käydään läpi tämän opinnäytetyön tuloksia. Luvussa kerrotaan myös lyhyesti mitä opinnäytetyöstä tuli opittua.

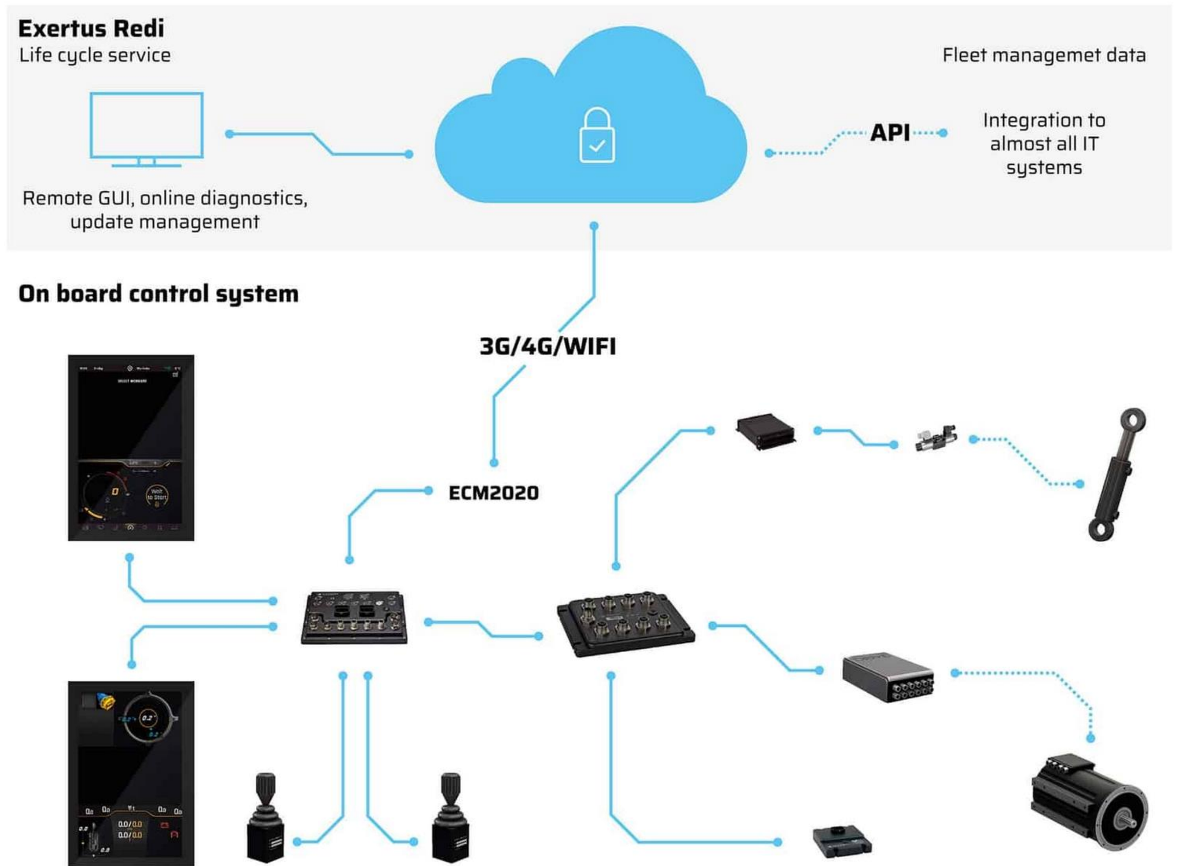
Luvussa seitsemän käydään läpi tämän opinnäytetyön prosessia ja toimeksiantoa. Luvussa tuodaan ilmi suurimpia haasteita toimeksiantoon liittyen ja tutustutaan myös vaihtoehtoihin, joista luovuttiin toimeksiannon toteutuksessa. Luvun lopussa pohditaan myös jatkokehitystä ja sen tuomia haasteita.

1.4 Toimeksiantaja

Exertus Oy on vuonna 2003 perustettu teknologiayritys, jonka toimipiste sijaitsee Seinäjoella Teknologiakeskus Framin C-rakennuksen viidennessä kerroksessa. Yritys työllistää n. 35 henkilöä. Yrityksen toimiala on ohjelmistojen suunnittelu ja valmistus sekä elektroniikkasuunnittelu. Yritys on erikoistunut erilaisten työkoneiden ohjausjärjestelmiin, jotka on toteutettu ohjelmoitavilla loogisilla moduuleilla. Työkoneita käytetään muun muassa kaivos-, rakennus- ja metsäteollisuudessa. (Viitanen 2020.)

Yritys tekee tiivistä yhteistyötä asiakkaidensa kanssa varmistaakseen, että kaikki tuotteet ja palvelut ovat parhaimpia ratkaisuja niihin tarkoituksiin, joihin ne on rakennettu. Yritys haluaa auttaa asiakkaitaan olemaan edelläkävijöitä tuottavuudessa, turvallisuudessa ja laadussa. (Tukeva 2020.)

Yrityksen arvot ovat: kuuntele, kunnioita, kommunikoi, kehitä, kehity, käyttäydy. Yrityksen visiona onkin tulla tunnetuksi innovatiivisista ratkaisuista ja olla asiakkaan ensimmäinen valinta ohjausjärjestelmien tuotteiden ja palveluiden alalla. (Tukeva 2020.)



Kuva 1. Exertuksen kokonaisarkkitehtuuri.

2 OHJAUSJÄRJESTELMÄT JA SIMULAATTORIT

2.1 Yleistä simulaattoreista

Simulaattorit ovat ohjelmia, jotka simuloivat eli jäljittelevät jonkin teknisen tai biologisen järjestelmän toimintaa täydellisesti tai osittain (Salakari 2010, 96). Yleisimmät simulaattoreiden käyttökohteet ovat koulutuksessa, kehityksessä ja testaamisessa, mutta niitä voidaan käyttää myös ymmärtämään jonkin järjestelmän toimintaa paremmin (Chung 2004, 1.3).

Koulutuksessa simulaattoreita voidaan käyttää parantamaan tai opettamaan päätöksentekokykyä, työtehtävien suorittamista ja työtaitoja. Koulutussimulaattoreilla voidaan varmistaa, että koulutus tapahtuu turvallisessa ympäristössä. Koulutussimulaattoreiden tehokkuus perustuu niiden ajankäyttöön ja alhaisiin kustannuksiin. Jonkin järjestelmän tai koneen käytännön koulutus voi viedä huomattavasti aikaa, jos järjestelmiä tai koneita on vain rajallinen määrä. Yleensä jonkin järjestelmän koulutuksessa käytännön harjoittelua edeltää teorian opetus. Simulaattoreiden avulla teoriaa voidaan opettaa käytännön harjoittelun yhteydessä, mikä parantaa koulutettavan asian ymmärtämistä ja nopeuttaa oppimisprosessia. Samalla myös säästetään kuluissa, joita käytännön harjoittelu oikealla koneella voisi aiheuttaa. (Salakari 2010, 13–15.)

Järjestelmien kehittämisessä simulaattoreita voidaan käyttää uusien konseptien kehityksessä ja testauksessa sekä nykyisten prosessien kehittämisessä. Tällä tavalla ei sotketa jo olemassa olevien järjestelmien prosesseja tai toimintoja, ja vältytään myös tahattomasti syntyviltä vahingoilta. Simulaattoreiden avulla voidaan myös avata järjestelmän toimintaa ja sielunelämää kehittäjille ja käyttäjille, mikä itsessään nopeuttaa ja parantaa järjestelmän kehitystä. (Chung 2004, 1.4.)

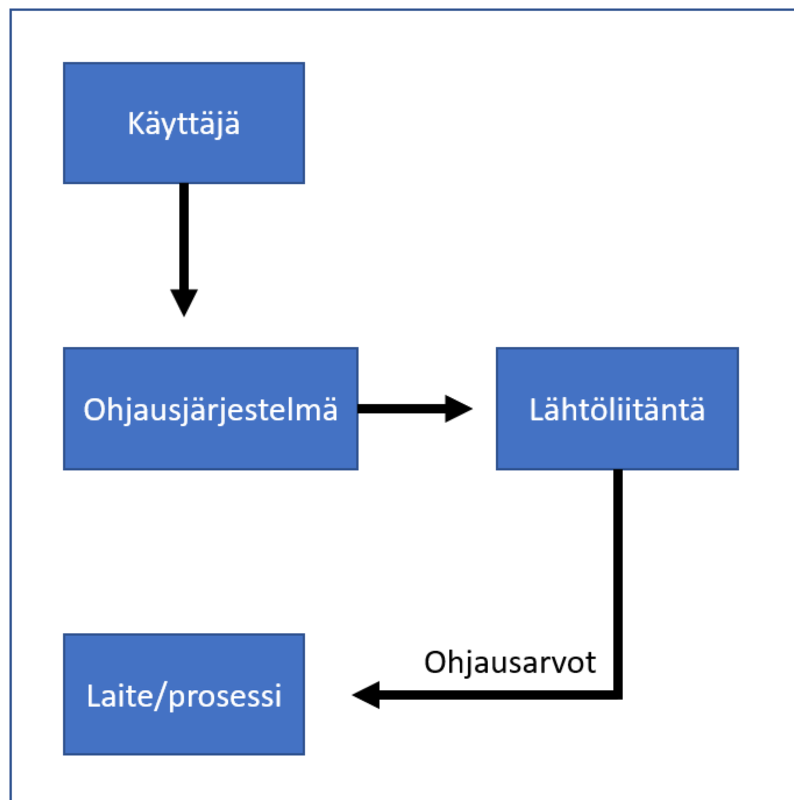
2.2 Yleistä ohjausjärjestelmistä

Ohjausjärjestelmistä puhuttaessa tarkoitetaan automaattisesti toimivien koneiden ja laitteiden ohjausmenetelmiä. Ohjausjärjestelmän tehtävä on ohjata järjestelmän eri

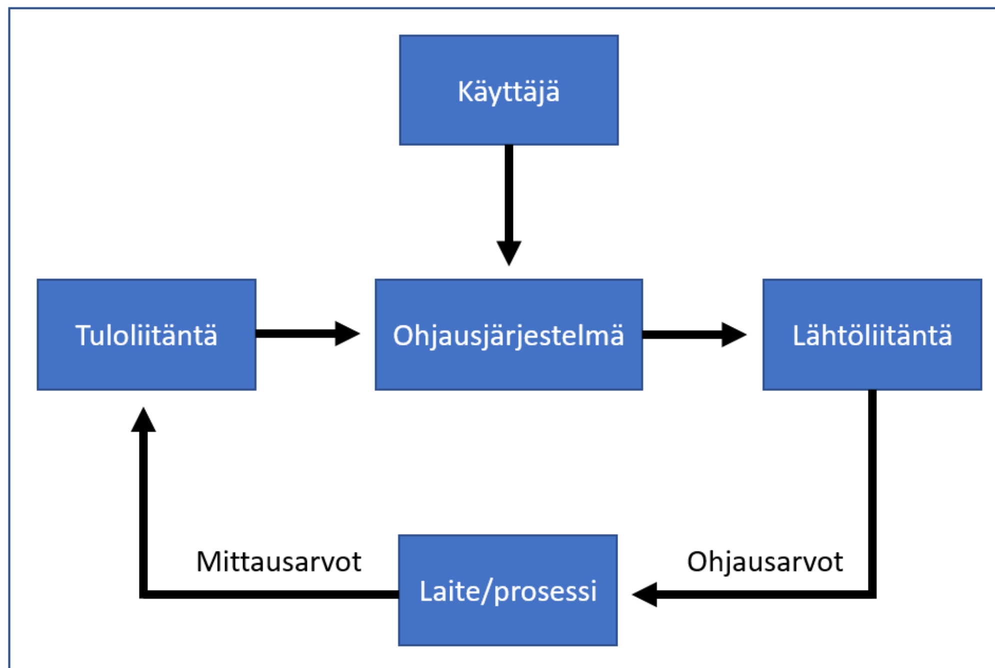
osia käyttäjän komentojen ja saatujen tietojen perusteella. (Keinänen, Kärkkäinen, Lähetkangas & Sumujärvi 2007, 209.) Käytännössä ohjausjärjestelmä toimii siltana käyttäjän ja koneen välillä (Keinänen, Kärkkäinen, Metso & Putkonen 2001, 206).

Yksinkertainen ohjausjärjestelmän toteutus voi koostua ainoastaan kytkimistä, joista käyttäjä voi aktivoida jonkin tietyn toiminnon. Monimutkaisemmat ohjausjärjestelmät koostuvat usein ohjelmoitavista logiikoista, jotka ovat uudelleen ohjelmoitavissa, jos ohjausjärjestelmän toimintaa halutaan muuttaa. Ohjelmoitavat logiikat myös vähentävät koneen johdotusten tarvetta. (Keinänen ym. 2001, 206 - 207.)

Ohjausjärjestelmät voidaan toteutuksensa puolesta jakaa avoimiin ja suljettuihin säätöjärjestelmiin (Keinänen ym. 2001, 206). Avoimissa säätöjärjestelmissä toimilaitteilta ei saada tilatietoa, jonka perusteella ohjausta voitaisiin korjata tai muuttaa. Järjestelmä ei siis tarkista toimilaitteen ohjaamiseen käytettyä toimintoa. Suljetussa säätöjärjestelmässä prosessiin liitetään takaisinkytkentä, jolloin ohjausarvoa voidaan muuttaa saatujen tietojen perusteella. (Keinänen ym. 2007, 210.)



Kuva 2. Avoin säätöjärjestelmä (perustuu Keinänen ym. 2001, 206).



Kuva 3. Suljettu säätöjärjestelmä (perustuu Keinänen ym. 2001, 206).

Ohjausjärjestelmät, joissa tulot ja lähdöt on hajautettu, toteutetaan yleensä kenttäväylämoduulien avulla. Yleensä yksi kenttäväylämoduuleista toimii keskusyksikkönä eli niin sanottuna ”Master”-moduulina, joka ohjaa muiden ”Slave”-moduulien laitteita. (Keinänen ym. 2007, 214.)

Kenttäväylä eli CAN-väylä kehitettiin alun perin ajoneuvojen sisäiseksi verkoksi. Nykyään se on käytössä mm. kuljetusjärjestelmissä, teollisuuden koneohjausjärjestelmissä, työkoneissa, laboratorioautomaatiossa ja lääketieteellisissä laitteissa. Kenttäväylä on erittäin luotettava ja monikäyttöinen sarjaväylämenetelmä. (CAN Knowledge, [viitattu: 10.1.2020.) Fyysisesti kenttäväylä on parikaapelilinja, jonka kautta tieto siirtyy järjestelmässä yksiköltä toiselle (Hietikko, Alanen & Tiusanen 1996, 8). Huomioitavaa on myös se, että jokainen kenttäväylän viesti välittyy kaikille väylällä oleville yksiköille (CAN Knowledge, [viitattu 10.1.2020). Tällä tavalla voidaan yhdistää esimerkiksi moottorin, jarrujen, voimansiirron ja käyttöliittymän ohjausyksiköt (Hietikko ym. 1996, 8).

2.3 Ohjausjärjestelmien simulaattorit

Teknologiayritykset, jotka rakentavat ja kehittävät erilaisia työkoneita ammattilaiskäyttöön, ovat yleensä kiinnostuneita työkoneidensa ja niiden ohjausjärjestelmien simuloinnista. Kyseiset yritykset yleensä kehittävät työkoneidensa simulaattoreita tai ovat vähintään keskeisessä roolissa simulaattoreiden kehitysprosessissa. Simulaattoreiden avulla yritykset voivat kouluttaa henkilöstöä järjestelmän käytössä ilman, että siitä koituu suuria kustannuksia. Samalla simulaattoreita voidaan käyttää uusien tai olemassa olevien toimintojen ja prosessien testaamisessa ja kehittämisessä. (Viitanen 2020.)

Ohjausjärjestelmien simulaattoreissa ohjausjärjestelmä voi olla joko täysin tai osittain virtuaalinen. Ohjausjärjestelmä tai sen osa voidaan toteuttaa niin sanottuna digitaalisena kaksosena eli virtuaalisena kopiona tai ilmentymänä (Digital Twin, [viitattu 15.1.2020]). Virtuaaliset kaksoset helpottavat simulaattoriympäristön käyttöönotossa, koska fyysisiä moduuleja ei välttämättä ole aina saatavilla. Simulaattoriympäristö on myös helposti muokattavissa, kun suurin osa moduuleista toteutetaan virtuaalisina. (Viitanen 2020.)

3 NYKYINEN SIMULAATTORI

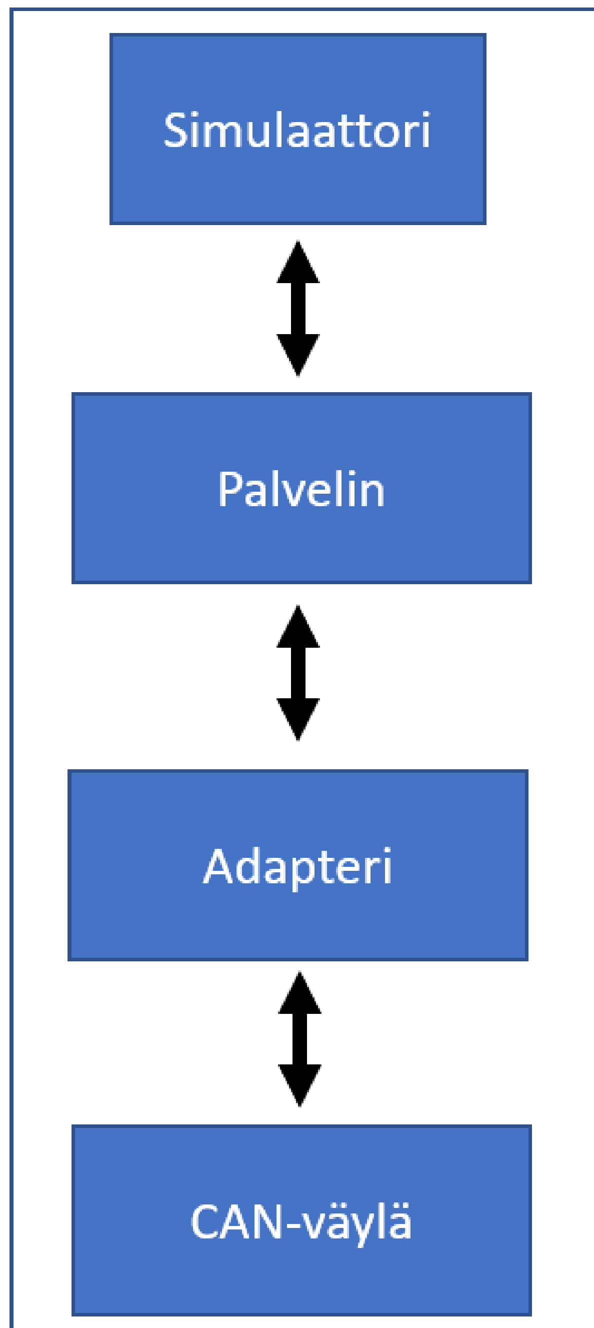
3.1 Kehitysympäristö ja ohjelmointikieli

Nykyinen ohjausjärjestelmien simulaattori on tehty RAD Studiolla ja C++-ohjelmointikielellä. RAD Studio on Embarcadero Technologies yhtiön tuottama kehitysympäristö Win32-sovellusten kehittämistä varten. Kehitysympäristöstä löytyy perinteinen tekstieditori ja työkalut käyttöliittymän tekemistä varten. Kehitysympäristö on maksullinen, mutta siitä on saatavilla ilmainen kokeiluversio. (RAD Studio, [viitattu 15.1.2020].)

3.2 Toiminta

Ohjelman alussa simulaattori luo TCP/IP-yhteyden palvelimeen, joka on yhteydessä adapteriin CAN-väylälle. Tämän jälkeen simulaattorissa alustetaan signaalit, joita halutaan kysellä palvelimelta. Alustamisessa signaalit lisätään ohjelman ajonaikaiseen listaan, josta niihin voidaan viitata muualta ohjelmassa. Alustuksen jälkeen aloitetaan ajonaikainen silmukka, jossa jokaista signaalia kysellään palvelimelta. Jossain vaiheessa palvelin vastaa kyselyihin paketeilla, joista saadaan kysyttyjen signaalien senhetkinen arvo.

Simulaattorilla voidaan myös asettaa signaalien arvoja lähettämällä palvelimelle tietynlainen paketti, joka sisältää signaalin nimen ja asetettavaksi halutun arvon. Tällä tavalla voidaan simuloida eri prosesseja palvelimelta saatujen signaalien arvojen perusteella.



Kuva 4. Simulaattorin toiminta.

3.3 Vahvuudet ja heikkoudet

Nykyisen simulaattorin vahvoja puolia ovat tietueiden ja yhdisteiden käyttö. Palvelimelta vastaanotetut paketit voidaan kopioida suoraan niitä vastaaviin tietueisiin,

minkä takia erillistä funktiota pakettien muuntamiselle tavuista niitä vastaaviksi tietueiksi ei ole tarvinnut luoda. Sama pätee myös palvelimelle lähetettyihin paketteihin, koska tietueita ei tarvitse erikseen muuntaa tavuiksi.

Yhdisteiden avulla tietueessa voi olla erityyppisiä muuttujia samalla muistialueella. Jaetulla muistialueella voi olla esimerkiksi yksi tietue ja taulukko. Yhdisteessä vain toinen näistä muuttujista voi sisältää arvon, kun tietueesta luodaan ilmentymä. (Stroustrup 2000, 876.) Tällä tavalla on mahdollista luoda tietueita, jotka voivat sisältää erityyppistä tietoa eri aikoina. Yhdisteitä on käytetty tietueiden kohdalla, jotka vastaavat osittain tosiaan. Yhdisteiden takia nykyisessä simulaattorissa ei ole tarvinnut luoda montaa erillistä tietuetta, jotka sisältävät samankaltaisia muuttujia samassa järjestyksessä muistialueillansa.

```
struct UnionTest {
    int ID;
    int PacketLength;
    union {
        struct ServerPacket serverpacket;
        char data[500];
    };
};
```

Kuva 5. Esimerkki yhdisteestä tietueessa.

Heikkoja puolia nykyisessä simulaattorissa on sen osittainen staattisuus, heikko optimointi ja vaikea käyttöönotto. Kaikki alustettavat signaalit on jouduttu alustamaan käsin eli jokainen mahdollinen signaali on täytynyt kirjoittaa käsin funktioon, joka lisää signaalin ohjelman ajonaikaiseen listaan. Jos signaaleja tulee lisää tai niiden nimet muuttuvat, ne joudutaan muuttamaan käsin ohjelman lähdekoodiin.

```
void AddSignal(std::string signalname) {
    SignalsList.push_back(signalname);
}
void AddSignals() {
    AddSignal("TestSignal_DI_01");
    AddSignal("TestSignal_DI_02");
    AddSignal("TestSignal_DI_03");
    AddSignal("TestSignal_DI_04");
    AddSignal("TestSignal_DI_05");
}
```

Kuva 6. Esimerkki signaalien lisäämisestä.

Signaalien kysyminen väylältä tapahtuu funktiolla, jolle annetaan parametrinä signaalin nimi. Ohjelman ajonaikana suoritetaan funktiota, joka koostuu signaalien kyselyistä. Jokainen signaali, jota halutaan kysyä, on lisätty käsin kyseiseen funktioon. Tämän lisäksi simulaattori kyselee jatkuvasti kaikkia signaaleja väylältä, vaikka niiden arvoa ei käytettäisi mihinkään. Jos signaaleja on useita niin jatkuva signaalien kyseleminen voi kuormittaa väylää.

```
void AskSignals(std::string signalname) {
    AskSignal("TestSignal_DI_01");
    AskSignal("TestSignal_DI_02");
    AskSignal("TestSignal_DI_03");
    AskSignal("TestSignal_DI_04");
    AskSignal("TestSignal_DI_05");
}
```

Kuva 7. Esimerkki signaalien kyselemisestä.

Simulaattorin kehitysympäristö on maksullinen ja tästä syystä sen lähdekoodia ei ole ollut mahdollista suoraan luovuttaa asiakkaiden käyttöön. Myös header-tiedostojen polkuja ei ole pystytty muokkaamaan, mikä on vaikeuttanut kehitysympäristön käyttöönottoa.

4 UUDEN SIMULAATTORIN KARTOITTAMINEN

4.1 Kehitysympäristön valinta

Uuden simulaattorin kehitysympäristöksi valittiin Microsoftin Visual Studio, jonka tarkoituksena on helpottaa ohjelmistokehityksen prosessia tarjoamalla kehitystä helpottavia työkaluja ja ominaisuuksia. Kehitysympäristöstä löytyy lukuisia eri kääntäjiä ja komponentteja. Kehitysympäristöllä on mahdollista suunnitella ja toteuttaa verkkopohjaisia ohjelmia ja käyttöliittymiä sekä useita eri rajapintoja hyödyntäviä ratkaisuja. Kehitysympäristöstä on olemassa ilmainen Community-versio ja maksullinen Enterprise-versio. (Microsoft 19.3.2019.)

4.2 Ohjelmointikielen valinta

Ohjelmointi kieleksi valittiin C#-ohjelmointikieli, koska se on moderni ja yksinkertainen ohjelmointikieli, joka tukee useita eri komponentteja sekä rajapintoja (Microsoft 5.4.2019).

Ohjelmointikielestä tekee modernin sen olio- ja komponenttipohjaiset ratkaisut. Yksi tärkeimmistä ratkaisuista C#-ohjelmointikielessä on yhtenäinen perintä, jonka takia kaikki primitiivisetkin tietotyypit, kuten reaali- ja kokonaisluvut, periytyvät yhtenäisestä luokasta. Tämä tarkoittaa sitä, että kaikentyyppiset muuttujat jakavat samat perusoperaatiot ja niiden arvoja voidaan varastoida, siirtää ja käyttää jatkuvasti. (Hejlsberg, Torgersen, Wiltamuth & Golde 2010, 1.)

Yksinkertaisen C#-ohjelmointikielestä tekee sen automaattinen muistinhallinta, jonka takia kehittäjän ei tarvitse erikseen varata ja vapauttaa muistia (Hejlsberg ym. 2010, 132).

4.3 Ohjelmistokehyksen valinta

Ohjelmistokehykseksi valittiin .NET Framework, joka tarjoaa useita valmiiksi integroituja palveluita, kuten versioiden yhteensopivuuden, jonka ansiosta kaikki .NET Framework -ohjelmat toimivat aina uusimmilla versioilla. Tämän lisäksi .NET Frameworkista löytyy myös kattava luokkakirjasto, jonka avulla ohjelmien kehittäminen on nopeampaa. (Microsoft 30.3.2017.)

Toisena ohjelmistokehyksen vaihtoehtona olisi ollut .NET Core, joka tarjoaa myös tuen Linuxille (Microsoft 17.9.2019). Tämä ei kuitenkaan ollut tarpeellista, koska simulaattoria on tarkoitus käyttää Windows-koneilla, ja simulaattorin ei tarvitse olla suoraan yhteydessä ohjausjärjestelmän kanssa.

4.4 Ohjelmistoprojektin valinta

Projektiksi valittiin Windows Foundation Presentation (WPF), josta löytyy useita valmiiksi määriteltyjä käyttöliittymäkomponentteja, kuten painikkeita, liukusäätimiä, palkkeja, kuvia ja tekstikenttiä. Samalla se tarjoaa tuen XAML-merkintäkielelle, joka pohjautuu XML-kieleen. Vaikka käyttöliittymä voidaan edelleen tehdä vedä ja pudota -tyylisesti, niin XAML-merkintäkieli mahdollistaa käyttöliittymän tekemisen myös pelkästään XML-kielellä. (Microsoft 4.11.2016.)

5 UUSI SIMULAATTORI

5.1 Käyttöliittymä

Koska opinnäytetyön tavoitteena ei ollut luoda erillistä simulaattoria jostain tietystä ohjausjärjestelmästä, niin käyttöliittymä jätettiin vielä toistaiseksi tyhjäksi. Tarkoituksena olisi kuitenkin jossain vaiheessa luoda käyttöliittymäkomponenteilla kytkimiä ja liukureita, joilla voidaan asettaa eri signaalien arvoja.

5.2 Luokat

Uudessa simulaattorissa on useita eri luokkia, jotka on jaoteltu niiden käyttötarkoitusten perusteella. Tarkoituksena oli tehdä mahdollisimman modulaarinen ohjelmarakenne, jota ei tarvitsisi jatkuvasti muuttaa kehityksen aikana. Tämän lisäksi kaikista tämänhetkisistä simulaatioista tehtiin omat luokkansa, jotka koostuvat simulaatiokohtaisista muuttujista kuten parametreistä ja lipputiedoista.

5.2.1 SocketThread-luokka

SocketThread-luokka koostuu Socket-luokan ilmentymästä, jolla otetaan TCP/IP-yhteys palvelimeen. Palvelimen yhteyttä varten luokkaan määriteltiin myös muuttujat, joihin asetetaan palvelimen portti ja nimi. Näiden lisäksi luokasta löytyy Utils- ja FunctionsHandler-luokan ilmentymät.

Luokalla on metodeja, joilla voidaan ottaa yhteys palvelimeen ja vastaanottaa paketteja palvelimelta. Lisäksi luokkaan on määritelty erillinen metodi, jolla voidaan tarkistaa palvelimen senhetkisen yhteyden tila.

```
public class SocketThread
{
    public Utils Utils;
    private Socket soc;
    private int Port;
    private string Host;

    public int clientID;
    public string Name;
    public FunctionsHandler FunctionHandle;
```

Kuva 8. SocketThread-luokka.

5.2.2 Utils-luokka

Utils-luokalla ei ole lainkaan omia muuttujia, vaan sen on tarkoitus toimia niin sanottuna työkaluluokkana. Luokka koostuu ainoastaan metodeista, joita simulaattorissa tarvitaan. Metodeja ovat esimerkiksi signaalin nimen muuttaminen UTF8-koodatuiksi tavuiksi, tietueen muuttaminen tavuiksi ja tavujen nollaus puskurista.

```
public void EmptyBuffer(byte[] buf, int start, int length)
{
    int i;
    for (i = start; i < length; i++)
    {
        buf[i] = 0;
    }
}
```

Kuva 9. Tavujen nollaus -metodi.

5.2.3 LogicalIOSignal-luokka

LogicalIOSignal-luokka toimii ilmentymänä loogisista signaaleista. Luokka sisältää muuttujia, kuten signaalin nimen, arvon ja signaalikohtaiset lipputiedot. Luokan il-

mentymälle on perinteinen alustus, jossa jokainen luokan muuttuja saa arvon. Tämän lisäksi siihen on myös määritelty tyhjä alustus, jos signaalin nimeä ei jostain syystä vielä tiedetä tai signaalia ei haluta alustaa.

```
public class LogicalIOSignal
{
    public string SignalName;
    public int Value;
    public int ValueOld;
    public bool Ask;
    public bool AskEnabled;

    public LogicalIOSignal(string signalname)
    {
        this.SignalName = signalname;
        this.Value = 0;
        this.ValueOld = 0;
        this.Ask = false;
        this.AskEnabled = false;
    }
    public LogicalIOSignal()
    {

    }
}
```

Kuva 10. Signaaliluokka.

5.2.4 FunctionsHandler-luokka

FunctionsHandler-luokan on tarkoitus toimia siltana simulaatioiden ja käyttöliittymän välillä. Luokkaan voidaan määritellä jokaisen simulaatioluokan ilmentymä, jonka tiedot voidaan välittää käyttöliittymälle tai toiselle simulaatiolle. Luokan muuttujista löytyy SocketThread-luokan ilmentymä, johon haetaan palvelimelle yhteydessä olevan luokan ilmentymä.

Luokasta löytyy edellä mainittujen muuttujien lisäksi myös Utils-luokan ilmentymä, joka mahdollistaa tarvittavien metodien kutsun. Tämän lisäksi siitä löytyy myös lista signaaleista ja erillinen lista ohjelmassa ajettavista metodeista ja simulaatioista.

```
public class FunctionsHandler
{
    private SocketThread Handle;
    private Socket Sock;
    private Utils Utils = new Utils();
    private List<Action> Functions = new List<Action>();
    private List<LogicalIOSignal> LogicalSignals = new List<LogicalIOSignal>();
```

Kuva 11. FunctionsHandler-luokka.

5.3 Tietueet

Simulaattorissa suurin osa tietueista toimii paketteina, joita lähetetään ja otetaan vastaan palvelimelta. Jokaisessa tällaisessa tietueessa on metodi, jonka avulla palvelimelta saatu paketti voidaan muuttaa sitä vastaavaksi tietueeksi. Alkuperäisessä simulaattorissa tämä voitiin toteuttaa suoraan muistin kopioimisella, mutta uudessa simulaattorissa haluttiin vielä välttää muistin suoraa kopiointia palvelimelta vastaanotettujen pakettien suhteen.

Jokaisesta palvelimelta saaduista paketeista löytyy tunniste, paketin pituus ja pakettikohtainen tieto. Tunnisteen avulla paketista voidaan tehdä sitä vastaava ilmentymä tietueesta.

```
public struct ServerPacket
{
    public ushort PacketLength;
    public ushort PacketID;
    public int Port;
    public bool Connected;
    public byte[] Data;
}
```

Kuva 12. Esimerkki tietueen rakenteesta.

5.4 Metodit

Suurin osa aikaisemman simulaattorin metodeista käännettiin uudelle simulaattorille, jotta yhteensopivuus rajapintaohjelmien kanssa säilyisi. Joidenkin käännettävien metodien kohdalla jouduttiin kuitenkin tekemään pieniä muutoksia ohjelmointikielten välisen syntaksin takia.

5.4.1 Signaalien alustaminen

Aikaisemmassa simulaattorissa signaalit alustettiin funktiolla, johon signaalin nimi kirjoitettiin käsin. Uudesta simulaattorista haluttiin tehdä paljon dynaamisempi signaalien alustamisen suhteen.

Signaalit on määritelty erillisessä JSON-tiedostossa, josta ne voidaan helposti lukea. Signaalien alustusmetodi lukee JSON-tiedoston ja luo jokaisesta määritetystä signaalista signaaliluokan ilmentymän ja lisää sen ohjelman ajonaikaiseen listaan signaaleista. Tällä tavalla simulaattorista tulee myös modulaarisempi, kun tehdään konekohtaisia simulaatioita, joissa signaalit ja niiden nimet voivat vaihtua JSON-tiedostossa.

```
public void ReadSignals()
{
    string filepath = "logicalsignals.json";
    try
    {
        using (StreamReader r = new StreamReader(filepath))
        {
            var json = r.ReadToEnd();
            var jobj = JObject.Parse(json);
            var iomux = jobj["signals"];
            foreach (var item in iomux)
            {
                LogicalIOSignal signal = new LogicalIOSignal(item["name"].ToString());
                this.LogicalSignals.Add(signal);
            }
        }
    }
    catch (Exception ex)
    {
        throw new Exception(ex.Message);
    }
}
```

Kuva 13. Signaalien alustusmetodi.

5.4.2 Yhteyden luominen palvelimeen

Uudessakin simulaattorissa palvelimelle luodaan yhteys TCP/IP Socketin avulla erillisessä säikeessä. Socket-luokan ilmentymä löytyy palvelimen yhteydestä vastaavasta luokasta. Ennen yhteyden luomista Socketin ilmentymä alustetaan ja sille asetetaan tarvittavat asetukset. Tämän jälkeen kutsutaan Socket-luokan metodia, jolla yhteys palvelimeen luodaan.

```
public void ConnectTo()
{
    this.soc = new Socket(AddressFamily.InterNetwork, SocketType.Stream, ProtocolType.Tcp);
    this.soc.SetSocketOption(SocketOptionLevel.Socket, SocketOptionName.ReuseAddress, 1);
    this.soc.SetSocketOption(SocketOptionLevel.Socket, SocketOptionName.NoDelay, 1);
    this.soc.SetSocketOption(SocketOptionLevel.Socket, SocketOptionName.DontLinger, 1);
    LingerOption lingerOption = new LingerOption(true, 0);
    this.soc.SetSocketOption(SocketOptionLevel.Socket, SocketOptionName.Linger, lingerOption);
    this.soc.Connect(this.Host, this.Port);
}
```

Kuva 14. Yhteyden luominen palvelimelle.

5.4.3 Pakettien vastaanottaminen palvelimelta

Pakettien vastaanottamisessa käytetään tietuetta, joka toimii puskurina. Tietueen muuttujia ovat paketin koko tavuina, paketin koko desimaaleina ja paketin tunniste.

Aivan aluksi tarkistetaan, ettei yhteys palvelimeen ole katkennut. Tämän jälkeen luodaan uusi ilmentymä puskuri-tietueesta ja alustetaan sen muuttuja, johon palvelimelta saadun paketin koko tallennetaan tavuina. Seuraavaksi kutsutaan palvelimeen yhteydessä olevan Socketin metodia, jolla vastaanotetaan ensimmäiset kaksi tavua palvelimen lähettämästä paketista. Tavut muutetaan kokonaisluvuiksi ja asetetaan puskuri-tietueen muuttujaan, joka vastaa vastaanotetun paketin kokoa kokonaislukuina.

Jos paketin koko ei ole nolla, luodaan uusi tavuista koostuva muuttuja, joka on kooltaan vastaanotettavan paketin kokoinen. Tämän jälkeen palvelimelta lähetetty paketti vastaanotetaan kokonaisuudessaan ja paketin mukana tullut tunniste muunnetaan tavuista kokonaisluvuiksi.

```
public void ReceiveData()
{
    if (this.soc.Connected)
    {
        try
        {
            ReceiveBuffer rcvBuf = new ReceiveBuffer();
            rcvBuf.PacketLengthInBytes = new byte[2];
            this.soc.Receive(rcvBuf.PacketLengthInBytes, 2, SocketFlags.Peek);
            rcvBuf.PacketLength = BitConverter.ToUInt16(rcvBuf.PacketLengthInBytes, 0);
            if (rcvBuf.PacketLength != 0)
            {

                byte[] recvData = new byte[rcvBuf.PacketLength];
                this.soc.Receive(recvData, rcvBuf.PacketLength, SocketFlags.None);
                rcvBuf.PacketID = BitConverter.ToUInt16(recvData, 2);
```

Kuva 15. Pakettien vastaanottaminen.

5.4.4 Pakettien muuntaminen tietueiksi

Paketin tunnisteen avulla paketti voidaan muuntaa sitä vastaavaksi tietueeksi. Aikaisemmassa simulaattorissa palvelimelta vastaanotetut paketit muunnettiin niitä

vastaaviksi tietueiksi kopioimalla vastaanotetun paketin muistialue suoraan sitä vastaavan tietueen ilmentymän muistialueelle. Uudessa simulaattorissa päädyttiin vielä toistaiseksi erillisiin tietuekohtaisiin muunnosmetodeihin.

Palvelimelta vastaanotetaan paketit tavuina. Jokainen paketti sisältää vähintään pakettikohtaisen tunnisteen ja paketin koon. Pakettien tunnisteiden avulla voidaan määritellä, minkälaiseksi tietueeksi se muunnetaan. Paketin muuntaminen sitä vastaavaksi tietueeksi tapahtuu tavukohtaisesti hyödyntäen Buffer- ja BitConverter-luokkien metodeja.

```
public ServerPacket SerializeToServerPacket(byte[] buf, int size)
{
    this.PacketLength = BitConverter.ToUInt16(buf, 0);
    this.PacketID = BitConverter.ToUInt16(buf, 2);
    this.Port = BitConverter.ToInt32(buf, 4);
    this.Connected = BitConverter.ToBoolean(buf, 8);
    this.Data = new byte[size];
    Buffer.BlockCopy(buf, 9, this.Data, 0, size);
    return this;
}
```

Kuva 16. Esimerkki paketin muuntamisesta tietueeksi.

BitConverter-luokan metodien avulla tavuista voidaan muuntaa niitä vastaava muuttuja, oli sitten kyseessä 16- tai 32-bittinen kokonaisluku. Buffer-luokan BlockCopy-metodin avulla tavuja voidaan kopioida taulukosta toiseen.

5.4.5 Pakettien lähettäminen palvelimelle

Pakettien lähettäminen tapahtuu käänteisesti siihen nähden, miten palvelimelta vastaanotetaan paketteja. Jokaisesta lähetettävästä paketista tehdään ensiksi tietue, joka koostuu paketin tunnisteesta ja koosta. Tämän lisäksi paketista löytyy yleensä myös pakettikohtainen tieto, joka halutaan välittää palvelimelle.

Paketit pitää lähettää tavuina palvelimelle. Sen sijaan, että jokaiseen tietueeseen olisi tehty erillinen metodi, jolla tietue muutetaan tavuiksi, päädyttiin luomaan lähes kaikkien tietueiden kanssa yhteensopiva metodi, joka muuttaa tietueen tavuiksi. Metodissa on hyödynnetty Marshal-luokan metodeja. Näiden metodien avulla voidaan osittain tehdä samantyylistä muistinkopioimista kuin C++-ohjelmointikielellä. Ainoa ero on, että muisti kopioidaan erilliseen tavuista koostuvaan taulukkoon, jonka metodi palauttaa.

```
public byte[] GetBytes(object str)
{
    int size = Marshal.SizeOf(str);
    byte[] arr = new byte[size];

    IntPtr ptr = Marshal.AllocHGlobal(size);
    Marshal.StructureToPtr(str, ptr, false);
    Marshal.Copy(ptr, arr, 0, size);
    Marshal.FreeHGlobal(ptr);
    return arr;
}
```

Kuva 17. Tietueiden muuntaminen tavuiksi.

Metodi ei ole yhteensopiva sellaisten tietueiden kanssa, jotka sisältävät alustamattomia taulukoita tai muuttujia, joiden maksimikokoa ei ole ennalta määritelty. Suurin osa tietueista ei kuitenkaan sisällä edellä mainittujen tyyppisiä muuttujia, ja ne voidaan yksinkertaisesti ja nopeasti muuntaa palvelimelle lähetettäviksi tavuiksi. Tietueet, jotka sisältävät alustamattomia muuttujia, taulukoita tai toisia tietueita joudutaan muuntamaan tavuiksi hyödyntämällä Buffer- ja BitConverter-luokkien metodeja.

5.4.6 Signaalien arvojen kyseleminen

Aikaisemmassa simulaattorissa jokaista alustettua signaalia kyseltiin jatkuvasti palvelimelta. Uudessa simulaattorissa haluttiin optimoida CAN-väylän liikennettä signaalien kyselyiden suhteen. Tätä varten jokaiselle signaalille luotiin lipputieto, joka mahdollistaa vain haluttujen signaalien kyselemisen. Lipputiedon avulla voidaan erikseen määritellä, mitä signaaleja palvelimelta voidaan ylipäätänsä kysyä. Kuten

aikaisemmassakin simulaattorissa jokaiselle signaalille luotiin myös lipputieto siitä, onko sen arvoa kertaalleen kysytty palvelimelta. Tällä varmistetaan, ettei samoja signaaleja kysytä jatkuvasti palvelimelta ennen kuin palvelin on ehtinyt vastata kyselyihin.

Haluttujen signaalien arvoja kysellään palvelimelta jokaisella ohjelmakierroksella. Metodi, jolla signaalien arvoja kysellään, tarkistaa ensimmäisenä lipputiedon, näin varmistetaan, että kyseistä signaalia halutaan kysyä palvelimelta. Seuraavaksi tarkistetaan lipputieto, joka kertoo, onko signaalin arvoa jo kysytty palvelimelta. Jos arvo on kysytty, sitä ei kysytä uudestaan.

```
public void AskSignalValues()
{
    for(int i = 0; i < this.LogicalSignals.Count; i++)
    {
        if (this.LogicalSignals[i].AskEnabled)
        {
            AskSignalValue(this.LogicalSignals[i].SignalName);
        }
    }
}
```

Kuva 18. Signaalien kyseleminen palvelimelta.

```
public void AskSignalValue(string signalName)
{
    int listlen = this.LogicalSignals.Count;
    bool NotAsked = false;
    for(int i = 0; i < listlen; i++)
    {
        if (this.LogicalSignals[i].SignalName == signalName)
        {
            if (this.LogicalSignals[i].Ask == false)
            {
                this.LogicalSignals[i].Ask = true;
                NotAsked = true;
            }
        }
    }
```

Kuva 19. Signaalien kyselyiden tarkistus.

5.4.7 Signaalien arvojen saaminen palvelimelta

Signaalien arvot saadaan palvelimelta vastaanotetulla paketilla, jonka palvelin rakentaa signaalia kysyttäessä. Paketti koostuu yksilöivästä tunnisteesta, paketin koosta, signaalin nimestä ja signaalin sen hetkisestä arvosta. Vastaanotetussa paketissa olevaa signaalin nimeä etsitään ohjelman signaalien listasta. Jos signaali löytyy listalta ja sen lipputieto signaalin kyselystä on päällä, sen arvo päivitetään ja lipputieto signaalin kyselystä nollataan.

```
for (int i = 0; i < this.LogicalSignals.Count; i++)
{
    if (this.LogicalSignals[i].SignalName == ReceivedPacketSignalName && this.LogicalSignals[i].Ask == true)
    {
        this.LogicalSignals[i].Value = ReceivedPacketSignalValue;
        this.LogicalSignals[i].Ask = false;
    }
}
```

Kuva 20. Signaalien arvojen päivittäminen.

5.4.8 Signaalien arvojen asettaminen

Signaalien arvojen asettamista varten ohjelmassa rakennetaan paketti, joka sisältää signaalin nimen ja halutun arvon. Ennen varsinaisen paketin rakentamista ohjelmassa tarkistetaan kyseisen signaalin aikaisempi arvo. Jos signaalin aikaisempi arvo on sama, mitä sille halutaan asettaa, pakettia ei rakenneta ja lähetetä palvelimelle. Tällä tavalla vähennetään myös väylän kuormitusta.

```
for (int i = 0; i < this.LogicalSignals.Count; i++)
{
    if (this.LogicalSignals[i].SignalName == signalname && (this.LogicalSignals[i].ValueOld != value))
    {
        this.LogicalSignals[i].ValueOld = value;
        found = true;
    }
}
```

Kuva 21. Signaalien arvojen asettamisen tarkistus.

5.5 Metodien kutsuminen

Uudessa simulaattorissa päädyttiin erilaiseen ratkaisuun metodien kutsumisessa ohjelmakierron aikana. Aikaisemmassa simulaattorissa kaikki metodit, jotka simuloivat eri toimintoja tai prosesseja, lisättiin käsin metodiin, jota kutsuttiin jokaisen ohjelmakierron aikana. Jos simulointeja olisi useampia, kasvaisi metodista hyvin nopeasti suuri, ja se veisi suuren osan pääsäikeellä suoritettavasta ohjelmakoodista.

Kaikki metodit, joita ohjelmassa halutaan kutsua ohjelmakierron aikana, lisätään erilliseen listaan metodeista. Lista metodeista löytyy FunctionsHandler-luokasta, jonka tiedot voidaan jakaa myös käyttöliittymän puolelle. Tällä tavalla lähdekoodista tulee jaotellumpi, kun kaikki kutsuttavat metodit löytyvät yhdestä luokasta eivätkä sotke muiden luokkien lähdekoodia.

```
public void InitFunctions()
{
    AddFunction(Handle.ReceiveData);
    AddFunction(AskSignalValues);
    AddFunction(EmcyCircuitLogic);

}
```

Kuva 22. Metodien lisääminen listaan.

Aikaisemmassa simulaattorissa jokaisen metodin kutsu lisättiin käsin pääsäikeen lähdekoodiin. Uudessa simulaattorissa metodit lisätään erilliseen listaan, jolloin metodien kutsuminen käy helpommin. Jokaista metodia, jota halutaan kutsua ohjelmakierron aikana, voidaan kutsua käymällä läpi listaa, johon metodit on lisätty. Tällöin jokaista metodin kutsua ei tarvitse kirjoittaa käsin, vaan ohjelma tekee sen automaattisesti.

```
public void RunFunctions()
{
    foreach (Action function in this.Functions)
    {
        function();
    }
}
```

Kuva 23. Metodien kutsuminen listalta.

5.6 Simulaatiot

Jokaisesta simulaatiosta on tarkoitus tehdä oma erillinen luokka ohjelman lähdekoodiin. Tällöin ohjelman lähdekoodi pysyy hallittavampana. Tämän lisäksi jokaisesta simulaatiosta lisätään ilmentymä FunctionsHandler-luokan attribuutteihin. Tällä tavalla simulaatioon ja sen mahdollisiin tietoihin voidaan viitata muista simulaatiosta ja tarvittaessa myös käyttöliittymästä.

Simulaatioissa tarvitaan yleensä lipputietoja, kun jokin toiminto on suoritettu. Tämä johtuu siitä, että kaikkia toimintoja ei välttämättä haluta suorittaa kuin vain kerran simulaation aikana. Simulaatioissa joudutaan yleensä myös laskemaan eri arvoja, minkä vuoksi on hyödyllistä, että erilaiset laskennassa käytetyt muuttujat ja lipputiedot löytyvät simulaatiosta tehdystä luokasta eivätkä esimerkiksi jostain toisesta luokasta, jossa voi olla useita eri muuttujia.

```
public class TestSimulationClass
{
    public bool Init, Flag1, Flag2;

    public TestSimulationClass()
    {
        this.Init = false;
        this.Flag1 = false;
        this.Flag2 = false;

    }

}
```

Kuva 24. Esimerkki simulaatioluokasta.

Simulointi itsessään toteutetaan metodeilla, joita kutsutaan ohjelmakierron aikana. Lipputietojen avulla voidaan määritellä erilaisia ehtoja simulaatioihin. Lipputietojen avulla voidaan määritellä, mitä toimintoja halutaan toteuttaa, koska ne halutaan toteuttaa ja kuinka useasti ne halutaan toteuttaa. Tästä toimii hyvänä esimerkkinä simulaatiometodin alustus, jossa tiettyjen signaalien kysely mahdollistetaan. Alustusta varten on luotu erillinen lipputieto, jonka avulla alustusta ei tarvitse tehdä joka kerta metodia kutsuttaessa.

```
if(TestSimulationClass.Init == false)
{
    SetAskEnabled("test_signal1_DO", true);
    SetAskEnabled("test_signal2_DO", true);
    SetAskEnabled("test_signal1_DI", true);
    SetAskEnabled("test_signal2_DI", true);
}
```

Kuva 25. Esimerkki simulaatiometodin alustuksesta.

Suurin osa simulaatioista on riippuvaisia väylältä saaduista signaalien arvoista. Jotkin simulaatiot voivat kuitenkin vaatia myös tietyn järjestyksen toiminnoilleen. Jos simulaatiossa halutaan asettaa tiettyjen signaalien arvoja väylältä saatujen signaalien arvojen perusteella tietyssä järjestyksessä, tarvitaan lipputietoja järjestyksen määrittelemiseksi. Muussa tapauksessa signaalien arvot voitaisiin asettaa väärässä järjestyksessä ja simulaatio ei toimisi halutulla tavalla.

```
if(TestSimulationClass.Flag1 == false && GetSignalValue("test_signal1_DO") != 0)
{
    TestSimulationClass.Flag1 = true;
    SetSignalValue("test_signal1_DI", 1);
}
else if (TestSimulationClass.Flag1 == true && TestSimulationClass.Flag2 == false && GetSignalValue("test_signal2_DO") != 0)
{
    TestSimulationClass.Flag2 = true;
    SetSignalValue("test_signal2_DI", 1);
}
```

Kuva 26. Esimerkki lipputietojen käytöstä simulaatiossa.

Jos simulaatiossa halutaan aloittaa jokin toimenpide käyttäjän toimesta, ohjelman käyttöliittymän puolella voidaan viitata suoraan simulaatioluokkien muuttujiin FunctionsHandler-luokan ilmentymän avulla. Tällä tavalla voidaan toteuttaa esimerkiksi hätäseis-painike käyttöliittymän puolelle. Samalla voidaan luoda erillisiä kytkimiä, joiden avulla voidaan asettaa signaalien ja parametrien arvoja tai aloittaa tiettyjä simulaatioita.

```
public void TestButton_Click(object sender, RoutedEventArgs e)
{
    if(SocThread.FunctionHandle.TestSimulationClass.Flag1 == true)
    {
        SocThread.FunctionHandle.TestSimulationClass.Flag2 = false;
    }
}
```

Kuva 27. Esimerkki viittauksesta simulaatioluokkaan käyttöliittymästä.

6 TULOKSET

Opinnäytetyön tuloksena ei syntynyt ainoastaan uusi ohjausjärjestelmien simulaattori, vaan myös modernimpi työskentelytapa, jossa korostuvat dynaamisuus, optimointi ja modulaarisuus. Toimeksiannon tavoitteet täyttyivät vaikka joitakin aikaisemman simulaattorin toiminnoista ei vielä ehditty implementoida uuteen simulaattoriin.

Uutta simulaattoria ei ole vielä otettu virallisesti käyttöön, mutta joitakin simulaatioita sillä on jo testattu. Kaikki testatut simulaatiot ovat toimineet vähintäänkin yhtä hyvin kuin aikaisemmassa simulaattorissa. Uuden simulaattorin kehitysympäristön ansiosta käyttöliittymän rakentaminenkin on huomattavasti helpompaa.

Työssä opittiin todella paljon ohjausjärjestelmien toiminnasta, simulaattoriohjelmasta ja sen rajapintaohjelmista. Uuden simulaattorin avulla voidaan myös oppia enemmän koneista, joihin ohjausjärjestelmiä asennetaan. Simulaatioiden tekeminen vaatii oman aikansa, mutta uuden simulaattorin myötä kehitys helpottuu, kun asiakkaidenkin on mahdollista jatkaa simulaattoreiden kehitystä.

7 POHDINTA JA YHTEENVETO

Opinnäytetyön tavoitteena oli ohjausjärjestelmien simulaattorin modernisointi. Tavoitteena ei ollut luoda erillistä simulaattoria jonkin tietyn työkoneen ohjausjärjestelmää varten, vaan hyvä pohjarakenne, joka on helposti muokattavissa ja jonka avulla voidaan parantaa ohjausjärjestelmien simulaattoreiden kehitystä.

Työ aloitettiin selvittämällä aikaisemman simulaattorin toiminta. Selvitystyö oli opinnäytetyön haastavin osuus, koska rajapintaohjelmien lähdekoodeja ei ollut saatavilla. Tästä huolimatta aikaisemman simulaattorin toiminta saatiin selvitettyä käänteisellä suunnittelulla ja testaamisella.

Aikaisemman simulaattorin toiminnan selvittämisen jälkeen aloitettiin uuden simulaattorin kartoittaminen. Vaihtoehtoja oli useita, mutta lopulta päädyttiin mahdollisimman luotettavaan kehitysympäristöön ja ohjelmointikieleen, jotka tarjoavat myös mahdollisuuden valmiiden rajapintojen käyttöönottoon tulevaisuudessa.

Uuden simulaattorin luominen Microsoftin Visual Studiolla oli helppoa kehitysympäristön yksinkertaisuuden takia. Pohjarakenteen määritteleminen vei suurimman osan ajasta, koska se merkitsi kaikista eniten toimeksiannossa. Uuden simulaattorin pohjarakenteen luomisessa jouduttiin myös ottamaan huomioon se, että simulaattorin pohjarakennekin voi tulevaisuudessa muuttua, joten lähdekoodi pyrittiin pitämään mahdollisimman yksinkertaisena.

Suurimmat haasteet uuden simulaattoriohjelman tekemisessä olivat pakettien lähettäminen ja vastaanottaminen palvelimelta. Pakettien muuntaminen tavuiksi ja palvelimelta saatujen tavujen muokkaaminen niitä vastaaviksi tietueiksi vaati huomattavan määrän aikaa ja perehtymistä C++- ja C#-ohjelmointikielten välisiin eroihin.

Uuden simulaattorin pohjarakenteessa olisi voitu hyödyntää asynkronisia operaatioita. Tästä ideasta kuitenkin luovuttiin toistaiseksi, koska palvelimen yhteyttä pidetään yllä erillisellä säikeellä, ja simulaattorista ei ainakaan toistaiseksi luoda yhteyksiä tietokantoihin tai tehdä erillisiä pyyntöjä ulkoisille palvelimille.

Työssä päädyttiin luomaan uudenlainen ohjausjärjestelmien simulaattorin pohjarakenne, josta kehitystä on hyvä jatkaa. Uuden simulaattorin perustoiminta ei eroa

aikaisemmasta simulaattorista huomattavasti, mutta siitä tuli rakenteeltaan aikaisempaa simulaattoria dynaamisempi, optimoidumpi ja modulaarisempi.

Dynaamisuus saatiin aikaiseksi alustamalla signaalit suoraan tiedostosta ja muuttamalla ohjelman ajonaikaista signaalien kyselyä.

Uudesta simulaattorista tuli myös optimoidumpi, kun signaaleille lisättiin erilliset lipputiedot, joiden avulla ohjelman ajonaikana ei kysellä kaikkia signaaleja jatkuvasti.

Modulaarisuutta lisäsi simulaatioiden jako omiin luokkiinsa ja lähdekoodin hajauttaminen eri osioihin.

Jatkokehitysmahdollisuuksia uuden ohjausjärjestelmien simulaattorin suhteen on useita ja se tuo mukanaan omat haasteensa. Yksi esimerkki jatkokehityskohteesta on MATLAB Simulinkin mallipohjaisen simulaation integroiminen osaksi simulaattoriohjelmaa. Jatkokehitys on syytä suunnitella ja kartoittaa tarkasti, jotta kehitys pysyy ajan tasalla ja vastaa senhetkisiin tarpeisiin. Tulevaisuudessa tuleekin miettiä, mihin suuntaan ohjausjärjestelmien simulaattoreiden kehitystä halutaan viedä, ja kenelle suurin osa vastuusta ohjausjärjestelmien simulaattoreiden ja simulaatioiden kehityksestä annetaan.

LÄHTEET

CAN Knowledge. Ei päiväystä. CAN in Automation. [Verkkosivu]. [Viitattu 10.1.2020]. Saatavilla: https://www.can-cia.org/can-knowledge/

Chung, C. 2004. Simulation Modeling Handbook: A Practical Approach. Yhdysvallat: CRC Press.

Digital Twin. Ei päiväystä. [Verkkosivu]. Siemens. [Viitattu 13.1.2020]. Saatavilla: https://www.plm.automation.siemens.com/global/en/our-story/glossary/digital-twin/24465

Hejlsberg, A., Torgersen, M., Wiltamuth, S. & Golde, P. 2010. The C# Programming Language. Yhdysvallat: Addison-Wesley.

Hietikko, M., Alanen, J. & Tiusanen, R. 1996. Työkoneiden ja automaation CAN-väyläsovellusten turvallisuus. [Verkkojulkaisu]. Espoo: Valtion teknillinen tutkimuskeskus (VTT). [Viitattu: 27.1.2020]. Saatavilla: https://www.vtt.fi/inf/pdf/tiedotteet/1996/T1745.pdf

Keinänen, T., Kärkkäinen, P., Lähetkangas, M. & Sumujärvi, M. 2007. Automaatiojärjestelmien logiikat ja ohjaustekniikat. Helsinki: WSOY.

Keinänen, T., Kärkkäinen, P., Metso, T. & Putkonen, K. 2001. Koneautomaatio 2: Logiikat ja ohjausjärjestelmät. Helsinki: WSOY.

Microsoft. 17.9.2019. About .NET Core. [Verkkosivu]. Microsoft corp. [Viitattu 18.12.2019]. Saatavilla: https://docs.microsoft.com/en-us/dotnet/core/about

Microsoft. 19.3.2019. Welcome to the Visual Studio IDE. [Verkkosivu]. Microsoft corp. [Viitattu 19.12.2019]. Saatavilla: https://docs.microsoft.com/en-us/visualstudio/get-started/visual-studio-ide?view=vs-2019

Microsoft. 30.3.2017. Overview of the .NET Framework. [Verkkosivu]. Microsoft corp. [Viitattu 5.1.2020]. Saatavilla: https://docs.microsoft.com/en-us/dotnet/framework/get-started/overview

Microsoft. 4.11.2016. WPF overview. [Verkkosivu]. Microsoft corp. [Viitattu 19.12.2019]. Saatavilla: https://docs.microsoft.com/en-us/dotnet/framework/wpf/introduction-to-wpf.

Microsoft. 5.4.2019. A tour of the C# language. [Verkkosivu]. Microsoft corp. [Viitattu 18.12.2019]. Saatavilla: https://docs.microsoft.com/en-us/dotnet/csharp/tour-of-csharp/

RAD Studio. Ei päiväystä. [Verkkosivu]. Embarcadero Technologies. [Viitattu 15.1.2020]. Saatavilla: https://www.embarcadero.com/products/rad-studio

Salakari, H. 2010. Simulaattorikouluttajan käsikirja. Ylöjärvi: Eduskills Consulting.

Stroustrup, B. 2000. C++ -ohjelmointi. Helsinki: Teknolit Oy.

Tukeva, P. 2020. Chief Executive Officer. Exertus Oy. Haastattelu 17.2.2020.

Viitanen, J. 2020. Senior Software Design Engineer. Exertus Oy. Haastattelu 10.1.2020.