Sami Kaikkonen

Fysiikkamallin yhdistäminen simulaattoriin

Insinööri (AMK)

Tieto- ja viestintätekniikka

Kevät 2025

KAMK • University of Applied Sciences

Tiivistelmä

Tekijä: Kaikkonen Sami

Työn nimi: Fysiikkamallin yhdistäminen simulaattoriin

Tutkintonimike: Insinööri (AMK), Tieto- ja viestintätekniikka

Asiasanat: Mallipohjainen suunnittelu, ohjelmistokehitys, ohjelmistotestaus, yhteissimulaatio

Opinnäytetyön aiheena oli rajapinnan luominen Simulink-fysiikkamallin ja Ponssen simulaattorisovelluksen välille. Työn toimeksiantajana toimi Ponsse Oyj. Metsäkoneen kuormaimen fysiikan laskeminen korvattiin simulaattorissa Simulink-mallilta saaduilla arvoilla. Tarkoituksena oli kehittää simulaattoria lähemmäs todenmukaista fysiikkaa ja mahdollistaa uudenlaisia testejä. Työn tavoitteena oli luoda toimiva rajapinta fysiikkamallin ja simulaattorisovelluksen välille. Lisätavoitteena oli saada rajapinta toimimaan itsenäisesti ilman Simulink-sovellusta, millä mahdollistettiin laajempi käyttö testikäytössä.

Aluksi käydään läpi teoriaa tuote- ja ohjelmistokehityksestä yleisesti. Työhön olennaisesti liittyvistä aihealueista, kuten rajapinnoista, ohjelmistotestauksesta, mallipohjaisesta suunnittelusta ja yhteissimulaatiosta kerrotaan myös. Työ alkoi työryhmän perustamisella, tavoitteiden ja toteutusvaihtoehtojen läpikäymisellä sekä projektin suunnittelulla. Projektissa edettiin ketterän ohjelmistokehityksen menetelmiä käyttäen. Toimeksiantajan toiveena oli testata jo olemassa olevaa rajapintaa, joka oli kehitetty Ponssen tarpeisiin yhteistyökumppanin toimesta. Aluksi perehdyttiin olemassa olevan rajapinnan toimintaan, jonka jälkeen Simulink-sovellukseen itsessään ja myöhemmin sillä kehitettyyn fysiikkamalliin. Tämän jälkeen pohdittiin, mitä dataa fysiikkamalli ja simulaattorisovellus vaihtaisivat rajapinnan ylitse. Rajapinta otettiin käyttöön fysiikkamallissa ja sitä kehitettiin eteenpäin sitä mukaa kun fysiikkamalli kehittyi. Rajapinta saatiin toimimaan, mutta sen käytöstä luovuttiin, koska kokonaisuutta ei sen avulla ollut mahdollista saada toimimaan itsenäisesti. Rajapinnassa siirryttiin käyttämään UDP-yhteystapaa, jonka avulla fysiikkamallista sai generoitua jaetun kirjaston. Jaetulle kirjastolle kehitettiin testisovellus, jolla sen sai käyttöön. Lopuksi rajapinnan toimintaa testattiin testikäyttöön suunnitellulla laitteistolla ja dokumentoitiin, miten rajapinnan sai käyttöön.

Työn tuloksena syntyi kaksi tapaa muodostaa yhteys fysiikkamallin ja simulaattorisovelluksen välille. Jo olemassa oleva rajapinta, joka vaati Simulink-sovelluksen toimiakseen, ja rajapinta, joka toimi täysin itsenäisesti. Kaikki työlle asetetut tavoitteet saavutettiin. Yhteys fysiikkamallin ja simulaattorisovelluksen välille saatiin muodostettua onnistuneesti ja myös lisätavoitteeseen päästiin.

Rajapinta mahdollistaa simulaattorissa todenmukaisemman fysiikan laskemisen. Tämän avulla simulaattorilla voidaan suorittaa testejä, jotka eivät aiemmin ole olleet mahdollisia. Testauksen parantuminen taas edistää ohjelmisto- ja tuotekehitystä kokonaisuudessaan, kun toiminta voidaan todeta oikeaksi virtuaalisesti aiemmassa kehitysvaiheessa.

Abstract

Author: Kaikkonen Sami

Title of the Publication: Connecting a physics model to a simulator

Degree Title: Bachelor of Engineering, Information and Communications Technology

Keywords: Model based design, software development, software testing, Co-simulation

The topic of the thesis was to create an interface between a Simulink physics model and Ponsse's simulator application. The assignment was commissioned by Ponsse Oyj. The calculation of the physics of the forest machines loader was replaced in simulator with the values obtained from the Simulink model. The purpose was to develop the simulator towards more realistic physics and to enable new types of tests. The objective of the work was to create a functional interface between the physics model and the simulator application. An additional objective was to get the interface work independently without the Simulink application, which enabled wider use in test use.

First, the theory of product and software development was briefly explained in general. The topics that are essentially related to the work, such as interfaces, software testing, model-based design and co-simulation, were also discussed. The work began with the establishment of a working group, reviewing the objectives and implementation options and planning the project. The project progressed using agile software development methods. The client's wish was to test an existing interface that had been developed for Ponsse's needs by a collaborative partner. At first, the operation of the existing interface was familiarized with, followed by the Simulink application itself and later the physics model developed with it. Next, it was discussed what data the physics model and the simulator application would exchange over the interface. The interface was implemented in the physics model and developed further as the physics model developed. The interface worked, but its use had to be abandoned because it was not possible to make the whole thing work independently using it. The interface was changed to use UDP connection, which allowed to generate a shared library from the physics model. A test application was developed to use the shared library. Finally, the interface was tested on hardware designed for testing and the implementation of the interface was documented.

The work resulted in two ways to establish a connection between the physics model and the simulator application: an already existing interface that required the Simulink application to function and an interface that functioned completely independently. All the objectives set for the work were achieved. The connection between the physics model and the simulator application was successfully established and an additional objective was also reached.

The interface enables more realistic physics calculations in the simulator. This allows performing tests on the simulator that were not possible previously. On the other hand, improved testing promotes software and product development, as the operation can be verified virtually at an earlier stage of development.

Sisällys

Termit ja lyhenteet

API	Ohjelmointirajapinta (engl. Application Programming Interface)
HTTP	Hypertekstin siirtoprotokolla sovellustasolla (engl. Hyper Transfer Protocol)
UDP	Kuljetustason tietoliikenneprotokolla (engl. User Datagram Protocol)
TCP	Kuljetustason tietoliikenneprotokolla (engl. Transmission Control Protocol)
CAN-väylä	Väylätekniikka (engl. Controller Area Network)
COM	Ohjelmistokomponenttimalli (engl. Component Object Model)
V-malli	Kehitysprojekteissa suunnitteluun ja toteutukseen käytetty prosessimalli
S-Funktio	Simulink-sovelluksessa käytetty funktio mukautettujen algoritmien luomiseen
TLC	Simulink-sovelluksessa käytetty kieli, joka muuntaa Simulink-mallin C-koodiksi (engl. Target Language Compiler)

1 Johdanto

Tämän opinnäytetyön aiheena on Simulink-fysiikkamallin yhdistäminen Ponssen simulaattorisovellukseen. Työ keskittyy yhteyden eli rajapinnan luomiseen näiden välille. Työn toimeksiantajana toimi Ponsse Oyj. Kehitysprojektissa edettiin ketterän ohjelmistokehitysmenetelmän toimintatapoja hyödyntäen. Mukana projektissa oli myös yhteistyökumppani työn tavoitteiden saavuttamiseksi.

Opinnäytetyön tavoitteena on luoda toimiva rajapinta metsäkoneen kuormaimesta tehdyn fysiikkamallin ja Ponssen simulaattorisovelluksen välille. Simulaattorista lähetetään signaaleja fysiikkamallille kuormaimen ohjaamiseen ja fysiikkamallista lähetetään asentotietoja simulaattorille kuormaimen fysiikan korvaamiseksi. Lisätavoitteena rajapinta on tarkoitus saada toimimaan täysin itsenäisesti ilman sovellusta, jolla fysiikkamalli on kehitetty. Ponssen simulaattorisovellusta käytetään esimerkiksi ohjelmistotestauksessa ja rajapintaa tehdessä otettiinkin huomioon ohjelmistotestausta edistävät toiveet.

Toimeksiantaja Ponsse Oyj on tavaralajimenetelmän metsäkoneiden myyntiin, tuotantoon, huoltoon ja teknologiaan erikoistunut yritys. Tavaralajimenetelmällä tarkoitetaan metsänhakkuumenetelmää, jossa puut prosessoidaan jo metsässä käyttötarkoituksen mukaan siihen kehitetyllä kalustolla. Ponssen päätuotteita ovat hakkuukoneet ja -päät, ajokoneet ja yhdistelmäkoneet. Kuvassa 1 on ajokone. Yhtiö on perustettu vuonna 1970 Einari Vidgrénin toimesta Vieremällä, missä se sijaitsee edelleen. Sieltä se on laajentunut maanlaajuiseksi ja lopulta maailmanlaajuiseksi. Ponsse Oyj on nykyään yksi maailman suurimmista tavaralajimenetelmään perustuvien metsäkoneiden valmistaja. [1.]

Kuva 1. Ponssen Buffalo-kuormatraktori [2]

2 Tuotekehitys

Tuotekehityksellä tarkoitetaan toimintaa, jonka tavoitteena on kehittää kokonaan uusi tai parannettu tuote. Siinä asetetaan tuotteelle tavoitteet ja ne pyritään täyttämään mahdollisimman hyvin niin teknisesti kuin myös taloudellisesti. Onnistunut tuotekehitystoiminta onkin yrityksen menestymisen keskeisimpiä edellytyksiä. Tuotekehitys jaetaan yleensä neljään toimintavaiheeseen. Prosessin käynnistämiseen, luonnosteluun, kehittämiseen ja viimeistelyyn. Perinteisesti tuotekehityksellä on tarkoitettu valmistettavia esineitä, mutta nykyään sanaa käytetään myös ohjelmistoista ja palveluista. [3, s. 9–14; 4.]

Tuotekehityksen edellytyksenä on, että kehitettävälle tuotteelle on olemassa tarve ja mielikuva sen toteuttamismahdollisuudesta. Ennen lopullista toteuttamispäätöstä on myös selvitettävä kehitettävän tuotteen kehityskustannukset, markkinointinäkymät, saatavat tuotot ja hyödyt sekä myös mahdolliset työterveys- ja ympäristökysymykset. Käynnistämisvaiheessa on otettava huomioon myös yrityksen omat voimavarat, muun muassa henkilökunnan pätevyys, käytettävissä olevat tutkimustilat- ja laitteet ja taloudelliset mahdollisuudet. Jos yrityksen omat voimavarat eivät riitä, on niitä vahvistettava tai vaihtoehtoisesti etsittävä yhteistyökumppaneita. [3, s. 14–20.]

Kun tuoteidea on löydetty, laaditaan kehitysehdotus. Se sisältää kehitettävän tuotteen kuvauksen ja vaatimukset, budjetin, käytettävissä olevat voimavarat sekä aikataulun. Yrityksen johto tekee lopullisen päätöksen kehittämisen aloittamisesta. Riippuen yrityksen organisaatiosta ja kehitysehdotuksen laajuudesta, voidaan päätös tehdä eri tasoilla. Jos kyseessä on jo olemassa olevan tuotteen vähäinen parantaminen, vastaa tuotekehitysosaston johto kehityspäätöksestä. Jos kehitysehdotus on merkittävämpi, kuten tuotteen korvaaminen uudella, kehityspäätöksestä vastaa yrityksen korkein johto. Kun kehityspäätös on tehty, alkaa luonnosteluvaihe. [3, s. 21.]

Luonnosteluvaiheessa etsitään vaihtoehtoisia ratkaisuluonnoksia kehitettävälle tuotteelle. Tässä vaiheessa laaditaan ratkaisuperiaatteita selventäviä piirustuksia. Luonnosteluvaihe aloitetaan tehtävän analysoinnilla, sillä henkilöt, jotka osallistuivat kehityspäätöksen tekoon, eivät välttämättä osallistu varsinaiseen tuotekehitystyöhön. Tämän takia mahdollisesti kaikki henkilöt tässä vaiheessa ovat uusia. Tuotteelle laaditaan vaatimukset ja tavoitteet, mikä voi johtaa sellaisien seikkojen ilmi tulemiseen, joita ei kehityspäätösvaiheessa osattu ottaa huomioon. Tällainen tilanne johtaa keskusteluun kehityspäätöksen tekijöiden kanssa lopullisesta vaatimuslistasta. Luonnosteluvaiheessa kutsutaan mukaan myös teollinen muotoilija, jos tuotteella on ulkonäkövaatimuksia. [3, s. 14–21; 4.]

Vaatimuslistan jälkeen jatketaan luonnosteluvaihetta ratkaisumahdollisuuksien etsimisellä. Tämä alkaa tehtävän yleistämisellä, eli pyrkimällä irtautumaan varsinaisesta tehtävästä ja selvittämään olennaiset ongelmat ja kokonaistoiminto. Seuraavaksi kokonaistoiminto jaetaan osatoimintoihin ja niille etsitään ratkaisumahdollisuuksia käyttäen hyväksi eri ideointimenetelmiä. Osatoimintoja yhdistelemällä etsitään kokonaistoiminnon ratkaisuperiaatteita ja näistä yksi tai useampikin kehitetään konkreettisiksi luonnoksiksi. Tulokseksi saadaan yksi tai useampi ratkaisuluonnos. [3, s. 14–15.]

Tuotteen kehittäminen alkaa kokoonpanoluonnoksen laatimisella valitusta ratkaisusta. Yleensä tässä vaiheessa suunnitelmissa huomataan teknisesti ja taloudellisesti heikkoja kohtia, jotka pyritään poistamaan ja sitä kautta pääsemään parannettuun mittakaavaiseen suunnitelmaan. Jos tuote on merkittävä yrityksen toiminnan kannalta, pyritään sen valmistuskustannuksiin ja teknisiin ominaisuuksiin vaikuttavat osat optimoimaan. Tämän jälkeen päädytään kehitettyyn konstruktioehdotukseen, mutta jos vaatimuksia ei pystytä riittävän hyvin täyttämään, on kehittämisvaihe aloitettava alusta uuden ratkaisuluonnoksen pohjalta. [3, s. 15.]

Viimeinen vaihe tuotekehityksessä on viimeistely. Siinä laaditaan osaluettelo, käyttö- ja huolto-ohjeet sekä mahdolliset työpiirustukset. Jos tuotteesta on tarkoitus aloittaa sarjavalmistus, tehdään siitä tyypillisesti prototyyppi eli koekappale. Prototyypin ominaisuudet tarkastetaan ja varmistetaan, että ne vastaavat tavoitteita. Niin sanotun nollasarjan valmistus prototyypin jälkeen on myös keino testata suunniteltuja valmistusmenetelmiä ja saada lisää tietoa tuotteen ominaisuuksista. Nollasarjan suuruus riippuu valmistettavan tuotteen kustannuksista. Jos kehitettävä laite tai järjestelmä on suuri ja kallis, kuten paperikone, ei prototyyppiä yleensä tehdä. Kun viimeistelyvaihe on saatettu loppuun, voidaan tehdä lopullinen päätös tuotannon aloittamisesta. [3, s. 17.]

3 Ohjelmistotuotanto ja -kehitys

Ohjelmistotuotanto on tietokoneohjelmistojen tuotannosta käytettävä yhteisnimitys koko prosessille ja siinä käytettäville menetelmille. Ohjelmistokehitys on osana ohjelmistotuotantoa. Se on prosessi, jossa itse tietokoneohjelmisto luodaan, ja se sisältää ohjelmiston suunnittelun, lähdekoodin ohjelmoinnin sekä ylläpidon, testauksen, dokumentoinnin ja julkaisemisen. Ohjelmoijien lisäksi prosessissa voi olla mukana eri alojen ammattilaisia, kuten testaukseen, graafisen suunnitteluun ja markkinointiin erikoistuneita henkilöitä. Ohjelmistotuotanto kattaa edellä mainittujen asioiden lisäksi muun muassa projektin- ja organisatorisen hallinnan. [5.]

Ohjelmistoprojektien eteneminen ja hallinta on muuttunut vuosien varrella melko paljon. Ohjelmistokehityksen alkuaikoina ohjelmointi tapahtui kaapeleita yhdistelemällä ja myöhemmin konekielellä. Yleensä ohjelmia käyttävät henkilöt ohjelmoivat sovellukset itse. Kehitys tapahtui niin sanotulla ”code and fix” -periaatteella. Siinä kirjoitettiin koodia ja kokeiltiin, toimiiko ohjelma. Tämä ei ollut kovin hyvä ja tarkka menetelmä ohjelmien kehittämiseen, joten tilalle haluttiin kehittää sellainen menetelmä, joka ottaa huomioon ohjelmiston vaatimukset, suunnittelun ja lopulta kehittämisen sekä testauksen. [6.]

Perinteinen vesiputousmalliksi kutsuttu ohjelmistokehitysmenetelmä on lineaarinen malli, josta tuli suosittu tapa toteuttaa ohjelmistoprojekteja. Siinä selvitetään erillisten tuotantotiimien kesken, mitä ohjelmistolta halutaan ja suunnitellaan se vaatimusten mukaan. Tämän jälkeen ohjelmisto kehitetään ja testataan. Vesiputousmalli ei kuitenkaan osoittautunut kovin hyväksi ohjelmistotuotannossa. Siinä oletetaan, että vaiheet tapahtuvat peräkkäin ja eri ihmisten toimesta, mikä aiheuttaa useita ongelmia. Keskeisin ongelma on vaatimusmäärittelyssä, sillä vesiputousmallissa ei ole valmistauduttu niiden muuttumiseen, mikä on etenkin suurissa projekteissa melkein väistämätöntä. [6.]

Vesiputousmallin jälkeen alkoi yleistymään iteratiivinen malli, jonka oli tarkoitus korjata vesiputousmallissa esiintyneitä ongelmia. Iteratiivisessa mallissa ohjelmistotuotanto jaetaan pienempiin aikaväleihin. Siinä ei pyritä tekemään heti alussa laajaa määrittelyä ja suunnittelua, vaan jokaisen iteraation aikana määritellään, suunnitellaan, toteutetaan sekä testataan ohjelmistoa. Iteratiivisesta mallista on hyötyä myös asiakkaalle, sillä jokaisen iteraation välissä asiakas näkee sen hetkisen version ja pystyy vaikuttamaan seuraavien iteraatioiden kulkuun. Lisäksi ohjelmiston perusversio on mahdollista saada loppukäyttäjien käyttöön jo silloin, kun kehitystyö on vielä käynnissä. [6.]

Ohjelmistokehityksen prosessimallien erinäiset ongelmat johtivat lopulta ketterien ohjelmistokehitysmenetelmien syntyyn. Ketterien menetelmien näkemys ohjelmistokehityksestä käy ilmi ketterästä manifestista. Siinä painotetaan, kuinka oleellisinta ohjelmistokehityksessä on itse kehitettävä ohjelmisto, siihen haluttuihin muutoksiin reagointi, ohjelmiston tekijät, sen tilaaja ja asiakkaat. Tällä tarkoitetaan, että ohjelmiston tekijöiden ja tilaajan sekä loppukäyttäjien interaktion merkitystä korostetaan. Lisäksi ohjelmiston halutaan ennemmin olevan toiminnaltaan hyvä kuin perusteellisesti dokumentoitu. Myös muutoksiin reagointia pidetään tärkeämpänä kuin suunnitelman tiukkaa noudattamista. Ketterä manifesti ei kuitenkaan pidä perinteisissä menetelmissä tärkeinä pidettyjä asioita arvottomana, se vain keskittyy eri seikkojen painottamiseen. [6.]

Ohjelmistokehitysprojektit ovat useimmiten uniikkeja ja niiden vaatimukset ovat erilaiset kuin millään jo tehdyillä ohjelmistoilla. Kaikkien projektien tekijät ovat erilaisia ja niissä käytettävät toteutusteknologiat kehittyvät koko ajan, joten uudet projektit toteutetaan hyvin todennäköisesti tavoilla, jotka eivät ole kaikille mukana oleville ohjelmistokehittäjille ennestään tuttuja. Tämä pohjustaa ketterien menetelmien ajatusta siitä oletuksesta, että kyseessä ei ole kontrolloitu prosessi, joka voidaan suunnitella tarkasti etukäteen. On parempi ajatella ohjelmistokehitystä tuotekehitysprojektina, joka sisältää epävarmuutta ja tuntemattomia seikkoja. Tällaisten projektien hallinnointiin sopii empiirinen prosessi. Sen periaatteina on läpinäkyvyys, tarkkailu ja mukauttaminen. Läpinäkyvyydellä tarkoitetaan, että kaikkiin projektiin liittyvien asioiden suhteen tulee vallita läpinäkyvyys, joka mahdollistaa myös tarkkailun. Tällä tarkoitetaan projektin tilan tarkkailua katsoen, onko tuotteen kehitys menossa oikeaan suuntaan oikeilla tavoilla. Kun projektin edetessä huomataan hyvin todennäköisesti parannuskohteita, mukautetaan tuotteen kehityssuuntaa tai tiimin toimintaa. [6.]

Ketteriä ohjelmistokehitysmenetelmiä on monia, mutta käytetyin niistä on Scrum. Lyhyesti Scrum on iteratiivinen ja inkrementaalinen menetelmäkehys, missä ohjelmistokehitys tapahtuu 1–4 viikon iteraatioissa. Näitä kutsutaan Scrumissa sprinteiksi. Scrumin kehittäjien sanoessa sen olevan menetelmäkehys eikä niinkään menetelmä, millä tarkoitetaan sen antavan kehitykselle suuntaviivat mahdollistaen samalla muidenkin menetelmien ja tekniikoiden käytön. Scrum määrittelee kolme jäsenroolia. Kehittäjä (engl. developer), Scrum master ja tuotteen omistaja (engl. Product owner). Kehittäjän tehtävä on nimensä mukaisesti edistää projektin kehitystä. Scrum master toimii tiimin apuna ohjaten muun muassa prosessin noudattamisessa ja parantamisessa sekä toimien rajapintana muihin sidosryhmiin. Tuotteen omistaja hallinnoi tuotteelle asetettujen vaatimuksien suorittamisjärjestyksestä eli niin sanotusta ”backlogista”. Jokaisessa sprintissä näistä vaatimuksista valitaan osa, jotka on tarkoitus toteuttaa sprintin aikana. [6.]

3.1 Ohjelmointirajapinnat yleisesti

Ohjelmointirajapinta eli API on määritelmä, jonka mukaan ohjelmat voivat vaihtaa dataa ja ominaisuuksia keskenään eli keskustella toistensa kanssa. Esimerkki rajapinnasta on käyttöjärjestelmän rajapinta, jonka avulla siinä suoritettavat ohjelmat voivat käyttää muun muassa laitteen palveluita ja resursseja. Rajapinnan tarkoituksena on siis avata halutulle ohjelmistolle mahdollisuus vuorovaikutukseen ulkopuolisten ohjelmien kanssa. [7.]

Perinteisesti API:lla viitattiin sovellukseen yhteydessä olevaan rajapintaan, joka on tehty jollain matalan tason ohjelmointikielellä, kuten JavaScriptillä. Nykyään ne kuitenkin vaihtelevat arkkitehtuureiltaan ja dataformaattien käytöltään. Tyypillisesti ne on tehty HTTP-protokollaa varten, mikä tekee niistä kehittäjäystävällisiä ja parantaa saavutettavuutta monilla eri ohjelmointikielillä kirjotettujen sovellusten kanssa. [7.]

Rajapinnat jaetaan tyypillisesti käyttötarkoituksen mukaan neljään pääryhmään. Yksi näistä ryhmistä on tietokantarajapinnat (engl. Database API). Niitä käytetään yhdistämään sovelluksia ja tietokantajärjestelmiä keskenään. Toinen on käyttöjärjestelmän rajapinnat (engl. Operating system API). Kuten edellä mainittu, niissä määritellään, kuinka sovellukset käyttävät laitteen palveluita ja resursseja. Kolmas on niin sanottu remote-rajapinta (engl. Remote API). Ne on suunniteltu vuorovaikuttamaan jonkin viestintäverkon kautta. Tämä tarkoittaa, että rajapinnan kautta käsiteltävät resurssit ovat jossain muualla kuin laitteella, josta pyyntö tehdään. Viimeinen ryhmistä on internetin yli toimivat rajapinnat (engl. Web API). Ne ovat remote-rajapintoja, joissa datan ja ominaisuuksien välitys tapahtuu internetin välityksellä HTTP-protokollaa käyttäen. Web-rajapinnat jaetaan myös neljään omaan pääryhmään. Nykyään suurin osa ohjelmointirajapinnoista on web-rajapintoja. [7; 8.]

3.2 Kirjastot ohjelmoinnissa

Yhteys kirjastoihin (engl. Library) ovat myös hyvä esimerkki rajapinnasta. Kirjastot ovat aliohjelmia, luokkia (engl. Class) ja/tai ohjelmia, joita käytetään tietokoneohjelmien modulaarisessa kehittämisessä. Kirjastoissa olevia ohjelmia ja palveluita käytetään itsenäisesti suoritettavien ohjelmien apuna, eikä niitä yleensä suoriteta itsenäisesti. Kirjastoihin on mahdollista tehdä lähes minkälainen toiminto vaan, mikä on ohjelmistoteknisesti mahdollista. Kirjastot jakautuvat pääasiassa

kahteen tyyppiin. Staattisiin kirjastoihin (engl. Static Library) ja jaettuihin kirjastoihin (engl. Shared Library). [9.]

Staattinen kirjasto on osana ohjelmaa ja linkittyy siihen käännöksen (engl. Compiling) yhteydessä. Kääntämisellä tarkoitetaan vaihetta, jossa ohjelma muutetaan ihmisten kirjoittamasta muodosta koneen suorittimen ymmärtämään muotoon. Tässä vaiheessa tapahtuvaa linkitystä kutsutaan staattiseksi linkitykseksi. Tämän jälkeen kirjasto on ohjelman sisällä. Linkitys täytyy tehdä uudestaan aina, kun ohjelmamoduuli käännetään uudelleen. Alun perin oli olemassa vain staattisia kirjastoja. [9.]

Jaettujen kirjastojen avulla on mahdollista jakaa ohjelmakoodia monien eri ohjelmien kesken. Microsoftin Widows-käyttöjärjestelmissä käytetään DLL-kirjastoja (lyhenne sanoista Dynamic-link Library). Useat Unix-tyyppiset käyttöjärjestelmät, kuten Linux, käyttävät SO-kirjastoja (lyhenne sanoista Shaded Object). Jaetun kirjaston käytöllä on hyötyjä staattiseen kirjastoon verrattuna. Se säästää levytilaa, kun jokaisen ohjelman, joka kirjastoa haluaa käyttää, ei tarvitse linkittää kirjastoa erikseen, vaan kaikki niistä voivat käyttää yhtä kirjastoa. Jos käyttöjärjestelmä osaa jakaa kirjaston myös keskusmuistissa, jaettuun muistialueeseen ladataan vain yksi kopio kirjastosta, jota ohjelmat käyttävät. Tämä vähentää keskusmuistin käyttöä. [9.]

Jaetun kirjaston ongelmat liittyvät lähinnä eri versioiden hallintaan. Jos kirjaston rajapinta muuttuu, on kirjastosta säilytettävä sekä vanha että uusi versio, jotta eri ohjelmaversiot voivat edelleen käyttää samaa kirjastoa. Edellä mainittu levytilan säästäminen on yksi syy käyttää jaettuja kirjastoja, mutta useiden eri versioiden varastoiminen mitätöi tätä hyötyä. Joissain tapauksissa voi olla parempi käyttää staattisia kirjastoja jaettujen kirjastojen sijaan, sillä ne ovat ohjelman sisällä ja välttävät vastaavanlaisen versio-ongelman. [9.]

3.3 Ohjelmistotestaus

Ohjelmistotestaus on tärkeä prosessi ohjelmistokehityksessä, sillä se varmistaa tuotteen laadun. Ohjelmistotestauksessa verifioidaan ohjelman oikea toiminta ja vaatimusten täyttyminen, parannetaan sen suorituskykyä sekä etsitään siitä mahdollisia virheitä. Virheiden etsiminen onkin tärkeää, sillä pahimmassa tapauksessa ne voivat aiheuttaa ohjelmassa vakavia toimintahäiriöitä. Esimerkiksi jos kyseessä on ohjelma, joka on jollain tavalla kytköksissä ihmisten turvallisuuteen, on

ohjelman oikea toiminta ensisijaisen tärkeää. Vaikka ohjelma ei liittyisikään ihmisten turvallisuuteen tai muihin kriittisiin toimintoihin, antaa se silti hyvän kuvan yrityksestä, jos heidän ohjelmansa toimii hyvin ja vaatimusten mukaisesti. [10.]

Ohjelmiston kehityksessä testaus pyritään aloittamaan mahdollisimman aikaisessa vaiheessa, koska on yleistä, että viivästykset kehityksessä vievät aikaa testaukselta. Tämän avulla virheet löydetään mahdollisimman aikaisessa vaiheessa, mikä säästää yritykseltä rahaa, kun kehityksessä ei jouduta palaamaan takaisin virheiden takia. Ohjelmistotestaus aloitetaan yleensä luomalla testaussuunnitelma. Siihen kirjataan testattavan ohjelmiston tunnistetut riskit sekä kartoitetaan, mitä, miten, missä ja milloin testataan. Suunnitelman valmistumisen myötä testaajat pääsevät yleensä tutustumaan ohjelmistoon ja suunnittelemaan testien suorittamista testitapausten mukaan. Testitapaukset tulevat määrittelyistä, joita ohjelmistolle on annettu etukäteen ja mahdollisesti myös luonnin aikana. Testaus luokitellaan valmiiksi, kun testaus on suoritettu hyväksytysti luodun testaussuunnitelman mukaan. [11.]

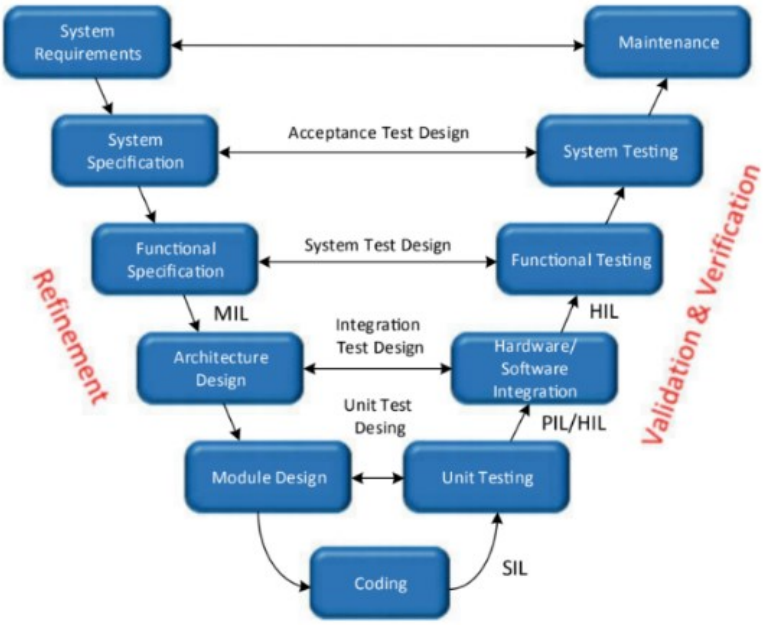
Ohjelmistoja voidaan testata eri tavoilla, joista jokaisella on omat tavoitteensa ja menetelmänsä. Testejä suoritetaan esimerkiksi tietoturvan, suorituskyvyn, käytettävyyden, ohjelman yksittäisten osa-alueiden sekä myös kokonaisuuden toiminnan varmistamiseksi. Tietoturvatesteillä pyritään varmistamaan, että ohjelmassa ei ole tietoturva-aukkoja, joita hakkerit tai haittaohjelmat voisivat hyödyntää ja näin aiheuttaa ohjelmiston vääränlaisen toiminnan. Pahimmassa tapauksessa tämä voisi johtaa käyttäjän arkaluontoisien tietojen vuotamiseen. Suorituskykytestien avulla ohjelman suorittamista kokeillaan eri työkuormilla ja nähdään, kuinka se reagoi niihin. Käytettävyystestauksessa loppukäyttäjiä pyydetään suorittamaan tiettyjä tehtäviä ja näin saadaan palautetta mahdollisista kehityskohteista. [10.]

Ohjelmiston eri osa-alueita testataan yksikkötesteillä (engl. Unit Testing). Yksikkötesteissä ohjelmistosta testataan yksittäisiä osioita ja varmistetaan niiden oikea toiminta erillään muista toiminnallisuuksista. Integraatiotestauksella taas varmistetaan kokonaisuuden toiminta, jotta ohjelmiston eri komponentit tai funktiot toimivat yhdessä. Iteratiivisessa ja ketterässä ohjelmistokehityksessä jokaisen iteraation aikana ohjelmistoon tulee uusia ominaisuuksia. Näitä muutoksia testataan regressiotestauksella ja varmistetaan, että ohjelman vanhat ominaisuudet toimivat samalla tavalla muutoksien jälkeen. [10; 12.]

Koko kehitettävän systeemin oikeaa toimintaa testataan järjestelmätestauksella ja hyväksymistestauksella (engl. Acceptance Testing). Järjestelmätestauksen tarkoitus on varmistaa, että ohjel-

misto toimii kuten vaatimusmäärittelyssä on kirjattu. Sovellusta testataan ilman tietoa järjestelmän sisäisestä rakenteesta eli niin sanotulla ”black-box” -testausmenetelmällä. Aiemmin mainitut tietoturvan, suorituskyvyn ja käytettävyyden testaaminen ovat järjestelmätestauksen muotoja. Tietoturvan ja suorituskyvyn testaaminen jätetään joissain tapauksissa niihin erikoistuneille tahoille. Hyväksymistestaus tapahtuu usein ohjelmiston tilanneen organisaation tai sen loppukäyttäjien toimesta. Tätä kutsutaan käyttäjän hyväksymistestaukseksi (engl. User Acceptance Testing). Hyväksymistestaus keskittyy testaamiseen käyttäjän tarpeiden näkökulmasta, eikä tarkastele tulosta täsmällisesti vaatimusmäärittelyn pohjalta. [10; 12.] Testauksen eri tasoja havainnollistettu kuvassa 2. Kuvassa on V-malli, joka havainnollistaa kunkin kehitysvaiheen ja siihen liittyvän testausvaiheen.

Ohjelmistotestausta voidaan tehdä manuaalisesti tai automatisoidusti. Manuaalitestauksessa testaaja itse suorittaa testit tehden sen eri vaiheet ja lopuksi vertaillen, onko saatu lopputulos se, johon pyritään. Manuaalistestausta tehdään usein enemmän ohjelmiston kehityksen alkuvaiheessa ja myöhemmässä vaiheessa siirrytään automaatiotestaukseen tehokkuuden parantamiseksi. Automaatiotestauksessa testit suoritetaan automatisoidusti testiskriptien mukaisesti. Testiautomaatio on manuaalitestausta parempi vaihtoehto etenkin silloin, jos testattavaa on paljon ja testitapaukset ovat pitkiä. Esimerkiksi aiemmin mainitussa regressiotestauksessa testiautomaatio on yleistä, kun samaa toiminnallisuutta testataan useaan kertaan ohjelmistoon tehtyjen muutoksien jälkeen. [11.]



Kuva 2. V-malli ohjelmistokehityksen ja -testauksen tasoista [13]

3.4 Mallipohjainen suunnittelu

Mallipohjainen suunnittelu on hyvä työkalu parantamaan tuote- ja ohjelmistokehitystä. Se mahdollistaa sulautettujen järjestelmien ja eri tilanteiden simuloinnin sekä kehitettävän sovelluksen jatkuvan testauksen. Mallipohjaisen suunnittelun tavoitteena on parantaa ymmärrystä joko täysin uuden tai jo olemassa olevan järjestelmän toiminnasta. Nämä simulointimallit ovat ohjelmallisia kuvauksia mistä vain fyysisen järjestelmän komponentista. Ne voivat liittyä sähköön, mekaniikkaan, lämpöön, hydrauliikkaan, pneumatiikkaan ja optiikkaan tai mihin tahansa näiden yhdistelmään. Mitä vaativampi kehitystyö on kyseessä, sitä enemmän mallipohjaisesta suunnittelusta on hyötyä. [14; 15.]

Ohjelmistokehityksen näkökulmasta mallipohjainen suunnittelu eroaa perinteisestä ohjelmistokehityksestä muun muassa roolien työnjakojen muuttumisena. Tavallisesti esimerkiksi sovellussuunnittelija vastaa kehitettävän ohjelmiston kokonaisuudesta ja kehittäjät tekevät heille määrättyjä toiminnallisuuksia. Mallipohjaisessa suunnittelussa kehittäjä mallintaa toiminnallisuuksia kokonaisuuksina koodin kirjoittamisen sijaan ja usein mallipohjaisen suunnittelun työkalut generoivatkin koodin automaattisesti mallin pohjalta. Tämä tarkoittaa, että sovellussuunnittelijan vastuulle ei tule niin suurta vaatimusta luoda toimivaa kokonaisuutta kerralla oikein, vaan kehittäjän on myös mahdollisuus ottaa vastuuta kokonaisuuden toimivuudesta. Kehittäjä voikin löytää suunnitteluvaiheen mahdolliset virheet heti kehitysprosessin alussa simuloinnin avulla. Mallipohjaisessa suunnittelussa kehittäjä siis vastaa osittain myös suunnittelusta. [14.]

Mallipohjaisen suunnittelun hyödyt tuote- ja ohjelmistokehityksessä näkyvät nopeampana, edullisempana ja myös laadukkaampana kehitystyönä. Mallipohjainen suunnittelu mahdollistaa kehitettävän järjestelmän automaattisen validoinnin, vaatimusten jäljitettävyyden ja monipuolisen testauksen. Näiden avulla tuotekehityksen laatu varmistuu aikaisemmassa vaiheessa. Esimerkiksi kalliiden fyysisten prototyyppien valmistukselta voidaan välttyä, kun suunnittelijat voivat varmistaa järjestelmän toiminnan ennen sen rakentamista virtuaalisella prototyypillä mallipohjaista simulointia hyödyntäen. Näillä malleilla voidaan simuloida erilaisia skenaarioita sen sijaan, että testaus suoritettaisiin lopullisella laitteistolla. Tämä nopeuttaa lopullisen tuotteen valmistumista. Mallipohjainen suunnittelu parantaa myös näiden järjestelmien testauksen turvallisuutta, kun fyysisen järjestelmän sijaan toiminta voidaan testata virtuaalisesti. Ja kaikkia, kuten ihmisten turvallisuuteen liittyviä järjestelmiä, ei voida aina edes testata käytännössä. [14; 15.]

Mallipohjaisen suunnittelun työkalut eivät kuitenkaan ole kovin edullisia, joten kaikissa projekteissa niiden käyttö ei ole välttämättä taloudellisesti kannattavaa. Sen edut tulevat parhaiten esiin

projekteissa, joissa vaaditaan nopeaa etenemistä tai järjestelmän simulointi ja monipuolinen testaus on tärkeää. [14.]

3.5 MiL-, SiL-, PiL- ja HiL-simulointi

Kuvassa 2 olevasta V-mallista käy ilmi ohjelmistokehityksen eri vaiheet ja niihin liittyvät testivaiheet. Mallipohjaisessa suunnittelussa näitä eri testivaiheita kutsutaan sanoilla Mil, SiL, PiL ja HiL. Jokaisella näistä vaiheista on oma tehtävänsä ja niillä varmistetaan, että kehitettävä simulointimalli täyttää sille asetetut vaatimukset. [16, s. 3–5.]

MiL-simulointi (lyhenne sanoista Model-in-the-Loop) on testivaihe, jossa ensin havainnollistetaan kehitettävän järjestelmän tai alijärjestelmän käyttäytyminen mallintamalla se. Myöhemmin mallia testataan, simuloidaan ja lopuksi verifioidaan sen toiminnallisuus. Tässä vaiheessa testataan siis pelkkää simulointimallia. Kun mallin toiminta on varmistettu MiL-simuloinnissa, testataan mallista generoidun koodin toimintaa SiL-simuloinnissa (lyhenne sanoista Software-in-the-Loop). Tämä testivaihe on MiL-simuloinnin tapaan riippumaton fyysisestä kohdelaitteistosta, sillä koodia testataan samalla laitteistolla, jolla simulointimallia suoritetaan. Ohjelmistolle asetetut vaatimukset ja spesifikaatiot analysoidaan, varmistetaan ja mahdolliset muutokset niihin tehdään tässä vaiheessa. [16, s. 5.]

PiL-simuloinnissa (lyhenne sanoista Processor-in-the-Loop) otetaan kohdelaitteisto mukaan testaukseen. Siinä generoidun koodin testaus suoritetaan kohdelaitteen mikroprosessorilla. Tämän avulla huomataan mahdollinen laitteisto-ominaisuuksien riittämättömyys. Tätä ei huomata SiL-testauksessa, koska siinä käytetyn laitteiston laskentateho on usein suurempi kuin kohdelaitteessa. PiL-simuloinnin avulla voidaan muun muassa tarkkailla laitteiston ja ohjelmiston keskeytyksiä, havaita järjestelmän pullonkaulat sekä lämpötilan ja elektromagneettisten häiriöiden aiheuttamat vaikutukset. [16, s. 5–6.]

HiL-simuloinnissa (lyhenne sanoista Hardware-in-the-Loop) yhdistetään oikea ohjainlaitteisto virtuaaliympäristöön, jossa fyysinen järjestelmä simuloidaan. Yhteydet niiden välillä ovat oikeita analogisia tai digitaalisia sisään- ja ulostulosignaaleja. Niissä käytetään usein eri tietoliikenneprotokollia, joita ovat muun muassa UDP, TCP ja CAN. Yhdistämällä fyysisen ohjainlaitteiston simuloituun ympäristöön mahdollistetaan realistinen testausympäristö eri olosuhteille ja skenaarioille, joita ei voi toteuttaa aiemmin mainituissa MiL- ja SiL-simuloinneissa. Etuna HiL-simuloin-

nissa on mahdollisuus testata ohjainalgoritmeja laitteistokomponenteilla ennen kuin koko fyysinen laitteisto on saatavilla. Se mahdollistaa myös näiden algoritmien ja järjestelmän vuorovaikutusten toiminnan vahvistamisen ennen järjestelmän käyttöönottoa. Toiminnan vahvistamisella löydetään mahdolliset suunnitteluvirheet ja vältytään kuluilta myöhemmässä kehitysvaiheessa. HiL-simulaattorit voivat kuitenkin olla itsessään kalliita ja siksi niitä ei kaikissa kehitysprosesseissa käytetä. [16, s. 6–7; 17.]

3.6 Yhteissimulaatio

Simuloinnilla pyritään jäljittelemään todellisuutta, esimerkiksi jonkin monimutkaisen järjestelmän toimintaa, kuten tässä tapauksessa metsäkoneen. Tällaisien monimutkaisien järjestelmien simulointia voidaan helpottaa yhteissimulaatiolla. Siinä simuloitava järjestelmä jaetaan alijärjestelmiin, jotka simuloidaan erillään toisistaan. Niiden simulointiin käytetään eri työkaluja, mutta niitä suoritetaan samanaikaisesti. Yhteissimulaation hyödyt tulevat esiin esimerkiksi silloin, kun halutaan yhdistää jonkin järjestelmän eri osa-alueiden fysiikkojen simuloinnit keskenään. Simuloinnit vaihtavat dataa keskenään, kuten painetietoja, nopeuksia ja virta-arvoja. Kommunikointi simulointien välillä voi tapahtua esimerkiksi jaetun muistin avulla, jos kyseessä on yksi tietokone tai vaihtoehtoisesti eri tietoliikenneprotokollia käyttäen. Yhteissimulaation ansiosta kunkin alijärjestelmän simulointiin voidaan valita niihin parhaiten sopivat työkalut. [18.]

4 Rajapinnan toteutus

Tässä luvussa käsitellään käytännön rajapintatyön tekoa. Tavoitteena oli saada Ponssen K111-kuormaimesta tehty fysiikkamalli yhdistettyä simulaattorisovellukseen. Tarkoituksena oli saada korvattua kuormaimen vanha kinematiikka simulaattorissa, joka ei sisältänyt fysiikan laskentaa lainkaan. Tavoitteena oli myös saada rajapinta toimimaan täysin itsenäisesti ilman sovellusta, jolla fysiikkamallia suoritettiin. Projektissa hyödynnettiin ketterän ohjelmistokehitysmenetelmän Scrumin toimintatapoja ja edettiin 2 viikon iteraatioissa.

4.1 Käytetyt ohjelmistot ja laitteet

Työssä käytetyt ulkoiset ohjelmistot olivat MATLAB, Simulink ja Visual Studio. Fysiikkamalli tehtiin MATLABin ja Simulinkin avulla. Visual studiolla luotiin testiohjelma, jolla lopullista toteutusta pystyi käyttämään. Ponssen omia sovelluksia olivat Optisimulator ja Virnex, jotka yhdessä simuloivat oikean metsäkoneen toimintaa. Ponssella on myös testikäyttöön tehtyjä niin sanottuja ”testiseiniä”, joilla rajapinnan toimintaa lopuksi testattiin.

4.1.1 MATLAB ja Simulink

The MathWorks -yhtiön ylläpitämä MATLAB (lyhenne sanoista Matrix Laboratory) on numeeriseen laskentaan tarkoitettu tietokoneohjelmisto sekä siinä käytettävä ohjelmointikieli. Se sisältää työkalut matriisien käsittelyyn, funktioiden ja datan visualisointiin, algoritmien toteuttamiseen, käyttöliittymien luomiseen ja vuorovaikutukseen muilla kielillä luotujen ohjelmien kanssa. Lisäksi ohjelmistoon on saatavilla monia lisätyökaluja erilaisiin käyttötarkoituksiin. Alun perin MATLAB kehitettiin opiskelijoiden käyttöön, mutta nykyään se on käytössä myös yritysmaailmassa. [19.]

Simulink on MATLABin graafinen ympäristö, jolla voidaan mallintaa, analysoida ja simuloida dynaamisia järjestelmiä eri teknisen laskennan osa-alueilla. Järjestelmät voivat olla lineaarisia tai epälineaarisia ja niissä voi olla aikariippuvuutta. Aikariippuvuus voi olla diskreettiä, jatkuvaa tai näiden sekoitusta. Simulinkissä malli voidaan luoda käyttämällä lohkokaavioita, jotka koostuvat lohkoista ja signaaliviivoista. Lohkokaaviot voivat sisältää matemaattisia operaatioita, MATLAB-koodia sekä sisään- ja ulostuloja. Lohkokaaviot ovat hierarkkisia, mikä mahdollistaa sekä ylhäältä-

alas- että alhaalta-ylös-lähestymistavan mallia luotaessa. Simulinkiin on myös saatavilla useita lisätyökaluja. [19.]

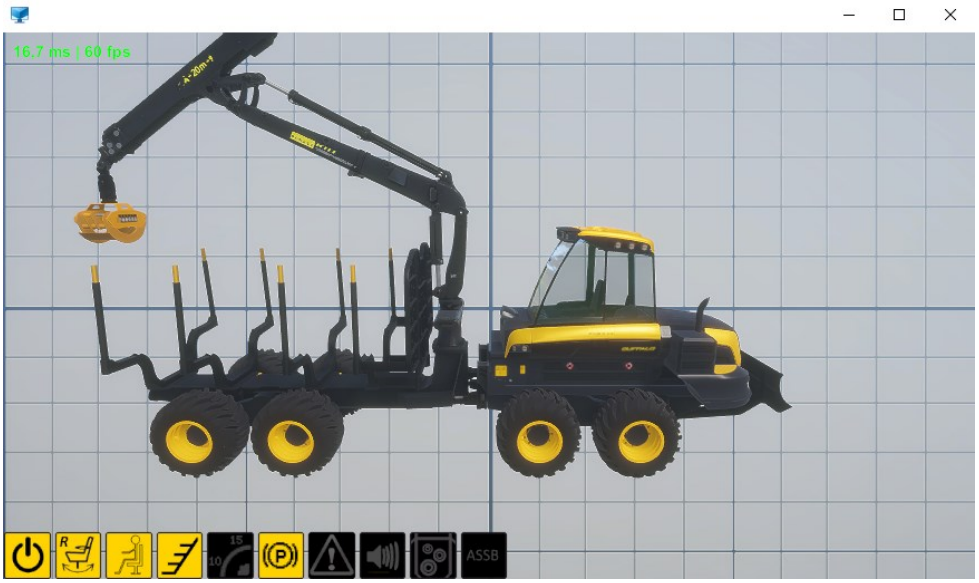
Ylhäältä-alas-lähestymistavalla tarkoitetaan järjestelmän hajottamista pienempiin osiin, jotta kokonaisuus on helpompi hahmottaa. Ensin järjestelmästä laaditaan yleiskatsaus, jossa määritellään pääpiirteittäin ensimmäisen tason alijärjestelmät. Kutakin alijärjestelmää jalostetaan myöhemmin yksityiskohtaisimmiksi, kunnes koko määrittely on jaettu peruselementteihin. Alhaalta-ylös on päinvastainen ylhäältä-alas-lähestymistavasta. Siinä yksittäiset elementit määritellään ensin hyvin yksityiskohtaisesti ja ne sitten linkitetään yhteen muodostaen suurempia alijärjestelmiä. Näitä taas linkitetään, kunnes järjestelmän ylin taso on muodostunut. [20.]

4.1.2 Optisimulator, Virnex ja testiseinät

Simulaattorisovellus Optisimulator, johon fysiikkamalli yhdistettiin, on Ponssen metsäkoneita simuloiva sovellus. Metsäkoneessa olevat ohjainmoduulit simuloidaan Virnex-nimisellä sovelluksella. Virnex on yhteydessä Optisimulaattoriin lähettäen sille dataa virtuaalisilta ohjainmoduuleilta, kuten virta-arvoja ja painetietoja. Ponssen yhteistyökumppani on tehnyt molemmat ohjelmat.

Optisimulatorissa on oikeiden ohjainkahvojen pohjalta tehdyt virtuaaliset ohjainkahvat, joilla metsäkonetta ohjataan. Konetta ohjattaessa liikkeet näkyvät Optisimulatorin omassa visualisointi-ikkunassa, joka näkyy kuvassa 3. Alun perin kuormaimen liikkeissä ei ole fysiikan laskentaa ollenkaan mukana, vaan liike päivitetään pelkästään asentotietojen perusteella.

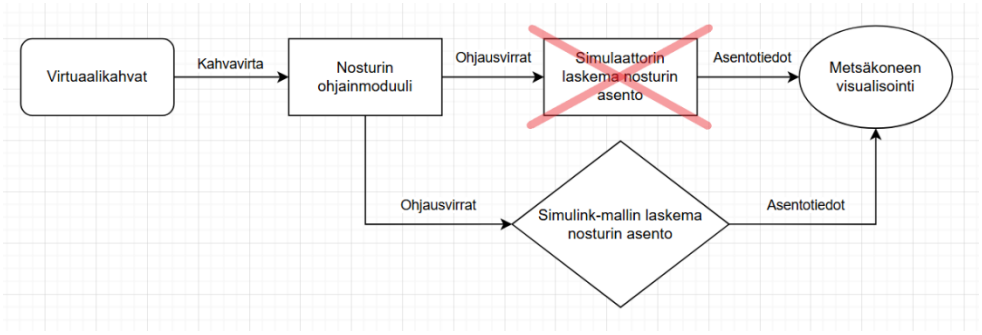
Ponssen testaussimulaattorit eli niin sanotut testiseinät ovat HiL-simulaattoreita. Niissä on oikean metsäkoneen ohjainmoduulit, jotka vaihtavat dataa Optisimulator-sovelluksen kanssa. Virnex-sovellusta ei siis näissä simulaattoreissa tarvita. Testiseiniä käytetään muun muassa ohjelmisto- ja automaatiotestaukseen ja niillä testattiin lopullista rajapinta toteutusta.



Kuva 3. Simulaattorin visualisointi-ikkuna

4.2 Suunnittelu ja tavoitteet

Projekti alkoi perustamalla työryhmä. Olin työryhmän vetäjä ja sain suhteellisen vapaat kädet toteutukseen. Työn tavoitteena oli saada yhteys Simulink-fysiikkamallin ja Optisimulator-sovelluksen välille niin, että simulaattorin laskemat kuormaimen asentotiedot saataisiin korvattua mallin laskemilla asentotiedoilla kuvan 4 mukaisesti.



Kuva 4. Havainnollistava kuva projektin suunnitelmasta

Työryhmään kuului kolme avainhenkilöä. Yksi jäsenistä teki Simulink-fysiikkamallin ja kehitti sitä eteenpäin. Toisen tehtävänä oli tehdä esiselvitystyötä rajapintamahdollisuuksista antaen osviittaa mahdollisista toteutusvaihtoehdoista. Minun tehtävänäni oli itse rajapintatyön toteuttaminen, jolla fysiikkamalli saataisiin yhdistettyä simulaattoriin. Lisäksi simulaattorin kehittänyt yhteistyökumppani oli mukana projektissa.

Alussa pohdittiin mahdollisia rajapintamahdollisuuksia ja ensimmäisenä nousi esiin jo olemassa oleva rajapinta, joka oli suunniteltu muodostamaan yhteys Simulinkin ja Optisimulatorin välille. Rajapinta on tehty edellä mainitun yhteistyökumppanin toimesta, joka on myös kehittänyt kyseessä olevan simulaattorisovelluksen. Toisena rajapintamahdollisuutena nousi esiin UDP-yhteyden muodostaminen mallin ja simulaattorisovelluksen välille. UDP:n käyttöä Simulinkissä oli tutkittu Ponssella aiemmin, mutta ei tähän käyttötarkoitukseen. Lopulta päädyimme aloittamaan tutkimalla jo olemassa olevan rajapinnan käyttöä, sillä toteutus sen avulla vaikutti mahdolliselta. Lisäksi sen testaaminen käytännössä oli myös yksi toimeksiantajan toiveista, koska sitä ei ollut aiemmin hyödynnetty ollenkaan. Lisätavoitteena, mutta ei vaatimuksena, oli saada toteutus toimimaan täysin itsenäisesti siten, että se toimisi myös ilman Simulink-sovellusta. Tämä mahdollistaisi laajemman käytön testikäytössä, kun Simulink-lisenssejä ei tarvittaisi.

4.3 Rajapintaan perehtyminen

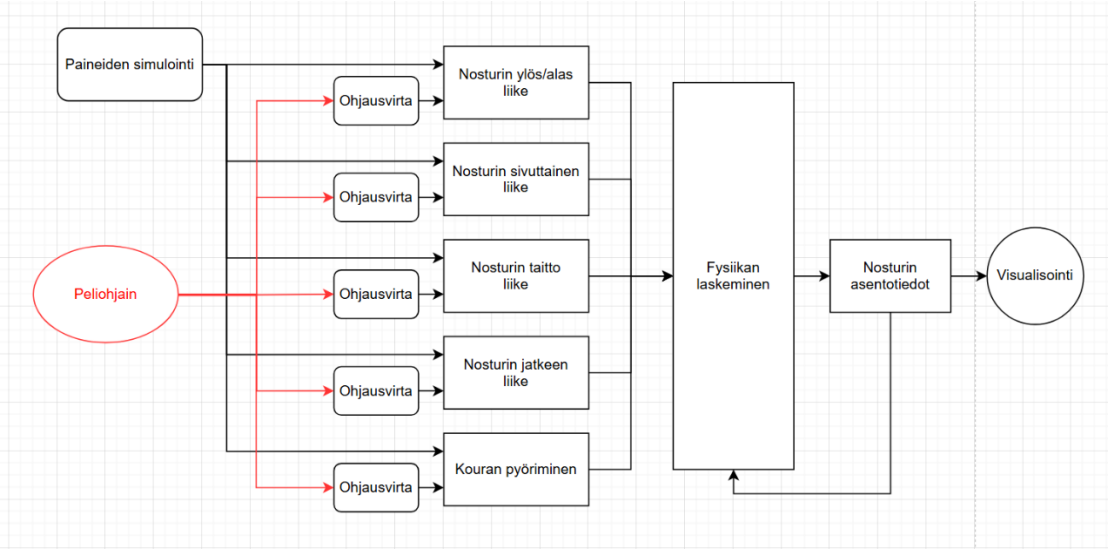
Tutkiminen alkoi rajapintaan perehtymisellä. Rajapinnan asennusohjelma asentaa COM-komponentin ja rekisteröi sen Windows-rekisteriin. Käytännössä se on ”In-process server” -tyyppinen komponentti, mikä tarkoittaa, että jaettu kirjasto eli DLL, suoritetaan samassa prosessissa isäntäsovelluksen kanssa eli tässä tapauksessa Simulinkin kanssa. Rajapinta saadaan käyttöön Simulinkissä S-funktion avulla. S-funktio pyytää mallin käynnistyessä käyttöjärjestelmää, tässä tapauksessa Windowsia, lataamaan komponentin ja antamaan sieltä kahvan rajapintaan. Näin tiedonsiirto Simulinkin ja simulaattorisovelluksen välillä on mahdollista. Haluttua dataa kutsutaan skriptissä ”invoke”-käskyllä kuvan 5 mukaisesti viittaamalla niihin signaalien nimillä, joilla ne on määritelty simulaattorissa. Esimerkiksi ”KY-LSFET15” on simulaattorissa puomin ylös nostamisen virta-arvo.

```
%%%%%%%%%%
% Inputs %
%%%%%%%%%%
sys(1) = iSimuIO.invoke('GetInput','KY-LSFET15', 0); %BoomUp
sys(2) = iSimuIO.invoke('GetInput','KY-LSFET16', 0); %BoomDown
```

Kuva 5. Esimerkki S-funktion sisääntulojen määrittelemisestä

Aluksi luotiin muutamia testimalleja Simulinkillä, joihin lisättiin kyseessä olevan rajapinnan S-funktio. Näillä malleilla lähinnä harjoiteltiin rajapinnan käyttöä ja testattiin yhteyden toimivuutta. Mallille tulevat signaalit ovat vektorina uint8-muodossa ja ne puretaan yksittäisiin signaaleihin. Mallilta lähtevät signaalit taas muutetaan yksittäisistä signaaleista vektoriin.

Kun oikeasta fysiikkamallista oli valmis ensimmäinen käyttökelpoinen versio, aloitettiin tutkimaan sitä ja rajapinnan lisäämistä siihen. Pohdittiin sisään- ja ulostuloja, joita malli ja simulaattori tarvitsisivat. Mallille tulisi ohjausvirrat ohjauskahvojen liikkeiden mukaan, joilla nosturi liikkuisi haluttuun suuntaan halutulla nopeudella. Malli taas lähettäisi simulaattorille asentotiedot, joilla se päivittäisi nosturin asennon metsäkoneen visualisoinnissa. Alun perin malli oli rakennettu siten, että sitä voi ohjata peliohjaimella. Tämä korvattaisiin virtuaalikahvoilta tulevilla ohjausvirroilla. Lisäksi malliin oli lisätty oma visualisointi, josta sen toimintaa voi seurata. Fysiikkamallin alkuperäinen rakenne havainnollistettu kuvassa 6.



Kuva 6. Yksinkertaistettu kuva fysiikkamallin alkuperäisestä rakenteesta

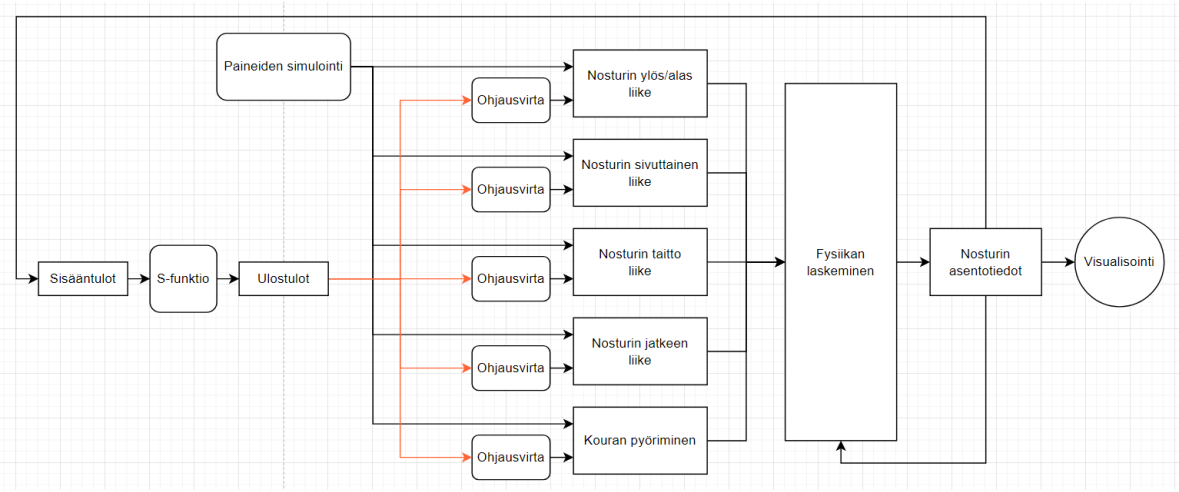
Mallia tutkiessa huomattiin, että siinä on käytössä vaihteleva aika-askeleen ratkaisija (engl. Variable step solver) kiinteän (engl. Fixed step solver) sijaan, jollaista esimerkkimalleissa oli käytetty. Siinä aika-askel vaihtelee ja sille määritellään maksimiaika-askel, jonka ratkaisija saa ottaa simuloinnin aikana. Kyseisessä mallissa asetettu maksimiaika-askel oli millisekunti.

Ensimmäisenä ongelmana esiin nousi simulointimallin hidastuminen, kun rajapinta lisättiin mukaan. Yhteys mallin ja simulaattorin välille muodostui, mutta simulointi hidastui merkittävästi. Simuloinnista ja datan lähetyksestä sekä vastaanottamisesta samanaikaisesti tuli liian raskasta, mikä johti siihen, että simuloinnin yhdessä aika-askeleessa kesti haluttua millisekuntia kauemmin.

Tämä saatiin ratkaistua vähentämällä S-funktion kommunikointia simulaattorin kanssa. Rajapinta oli määritelty perivän näytteenottonopeus mallilta, johon se liitetään. Ja koska malli oli vaihtelevalla aika-askeleella tehty, lähetti ja vastaanotti se dataa aina, kun yksi aika-askel oli suoritettu. S-funktion näytteenottonopeus vaihdettiin kiinteään 10 millisekuntiin. Tällä näytteenottonopeudella mallin ja simulaattorin välille ei tullut vielä huomattavaa viivettä ja mallin suorituskyky parani merkittävästi. Simulointi hidastui käynnistyessä edelleen noin viiden sekunnin ajaksi, mutta sen jälkeen simulointi pysyi taas alle määritellyn aika-askeleen.

4.4 Testausta

Yhteysongelman ratkettua aloitettiin lisäämään s-funktioon sisääntuloja mallille. Kuten aiemmin mainittu, malli oli tehty siten, että sitä pystyi ohjaamaan tavallisella peliohjaimella. Malli oli kuitenkin suunniteltu vastaanottamaan virta-arvoja milliampeereina, mikä mahdollisti venttiilien ohjausvirtojen tuomisen sellaisenaan simulaattorisovelluksen virtuaalikahvoilta. Ohjausvirrat tuotiin tässä vaiheessa nosturin kaikille muille liikkeille, paitsi kouralle, sillä sen lohkot mallissa eivät olleet vielä valmiit. Kuvassa 7 näkyy mallin rakenne rajapinnan kanssa.



Kuva 7. Yksinkertaistettu kuva fysiikkamallin rakenteesta rajapinnan kanssa

Kun ohjausvirrat oli tuotu mallille, testattiin liikkeiden toimivuus. Testatessa huomattiin, että mallissa ei ollut minkäänlaisia päätyrajoituksia liikkeille. Kuormainta pystyi liikuttamaan loputtomiin, mikä lopulta johti mallin kaatumiseen. Myöhemmin malliin lisättiin päätyrajoitukset liikkeille peliohjainta käytettäessä. Nämä sovellettiin käyttöön myös simulaattorin ohjausvirroille. Niin sanottu päätyvaimennus lähtee vaimentamaan liikettä vähentämällä ohjausvirtaa. Mitä lähempänä

sylinteri on sen ääriasentoja, sitä enemmän virtaa vaimennetaan. Ohjausvirtojen tuomisen jälkeen alettiin pohtimaan, miten simulaattorin oma kuormaimen kinematiikka saataisiin korvattua.

4.5 Kinematiikan korvaaminen

Lähetettävät signaalit saa määriteltyä samalla tavalla S-funktiossa kuin tulevatkin, signaalien nimillä. Simulaattorissa ei kuitenkaan ollut olemassa signaaleja, joista kuormaimen asentotiedot luetaan ja visualisointi-ikkunan liikkeet päivitetään. Asentotiedot määriteltiin siis jollain muulla tavalla. Koska simulaattori ei ole Ponssen kehittämä, ei sen selvittäminen ja muokkaaminen onnistunut sisäisesti. Projektin tavoitteista keskusteltiin simulaattorin kehittäjien kanssa ja he tekivät siihen halutut muutokset. Simulaattoriin luotiin uudet signaalit, joihin halutut asentotiedot voisi tuoda ja josta simulaattori lukisi ne visualisoinnin päivittämiseksi. Siihen lisättiin myös valinta sille, haluaako simulaattorissa käyttää alkuperäistä kinematiikan laskua vai ulkoisesti laskettuja arvoja. Lisätyt signaalit näkyvät kuvassa 8. Mallilta tuotiin kuormaimen asentotiedot uusiin signaaleihin, lukuun ottamatta kouran asentotietoja, ja todettiin sen toiminta hyväksi.

0	IN_EXT_ControlEnabled [0/1]
0.00	IN_EXT_SlewingJointValue [°]
25.00	IN_EXT_MainBoomJointValue [°]
-60.00	IN_EXT_LuffingJointValue [°]
0.000	IN_EXT_ExtensionJointValue [m]
35.00	IN_EXT_HangerTiltJointValue [°]
0.00	IN_EXT_HangerRollJointValue [°]
0.00	IN_EXT_RotatorJointValue [°]
-62.00	IN_EXT_GrappleArmLeftJointValue [°]
50.00	IN_EXT_GrappleArmRightJointValue [°]

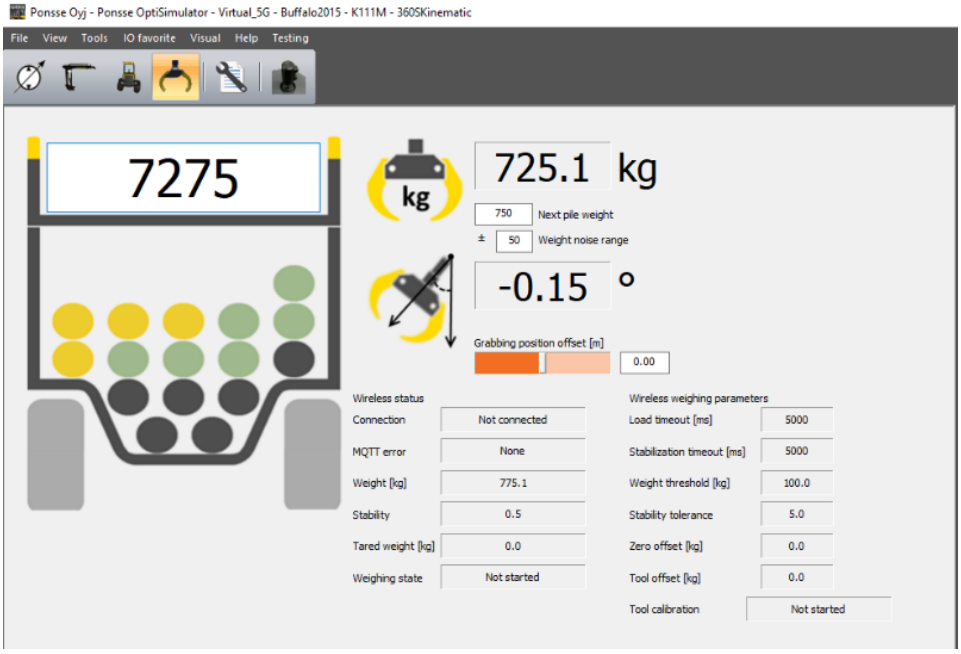
1	IN_EXT_ControlEnabled [0/1]
0.00	IN_EXT_SlewingJointValue [°]
19.84	IN_EXT_MainBoomJointValue [°]
-45.12	IN_EXT_LuffingJointValue [°]
-0.012	IN_EXT_ExtensionJointValue [m]
35.30	IN_EXT_HangerTiltJointValue [°]
-0.10	IN_EXT_HangerRollJointValue [°]
0.00	IN_EXT_RotatorJointValue [°]
-62.00	IN_EXT_GrappleArmLeftJointValue [°]
50.00	IN_EXT_GrappleArmRightJointValue [°]

Kuva 8. Lisätyt signaalit simulaattorissa. Signaalien ulkoinen päivitys päällä ja pois

4.6 Puun massan muuttaminen ja kouran asentotiedot

Simulaattorin kehittäjien tehdessä muutoksia vastaanotettaviin kuormaimen asentotietosignaaleihin saatiin idea lisätä Simulink-malliin mahdollisuus kourassa olevien puiden massan muuttamiseen. Simulaattorissa on kuvan 9 mukaisesti mahdollisuus asettaa ajokoneen kourassa olevien puiden massa ”Next pile weight” -arvoa muuttamalla. ”Weight noise range” -arvo määrittelee, kuinka paljon lopullinen massa saa poiketa asetetusta massasta. Simulink-mallissa oli valmiiksi määritelty kiinteä muuttujan arvo kourassa olevalle massalle. Simulaattorin massa-arvo lisättiin mallille tuleviin signaaleihin ja mallin kiinteän arvo korvattiin tällä.

Toimintaa testattiin simuloimalla mallia samalla muuttaen arvoa simulaattorista. Nosturin fysiikan laskeminen muuttui odotetusti massaa lisättäessä tai vähennettäessä. Mallille tuleva massa-arvo kuitenkin erosi simulaattorissa asetetusta arvosta 50 kilogrammalla. Kävi ilmi, että simulaattorista lähetettävään signaaliin oli lisätty toisen signaalin arvo, joka määrittelee kouran massan simulaattorissa. Koska kouran massa määriteltiin mallissa erikseen, vähennettiin tämä mallille tulevasta massa-arvosta, jotta simulaattorissa asetettu arvo täsmää mallille tulevaan arvoon.

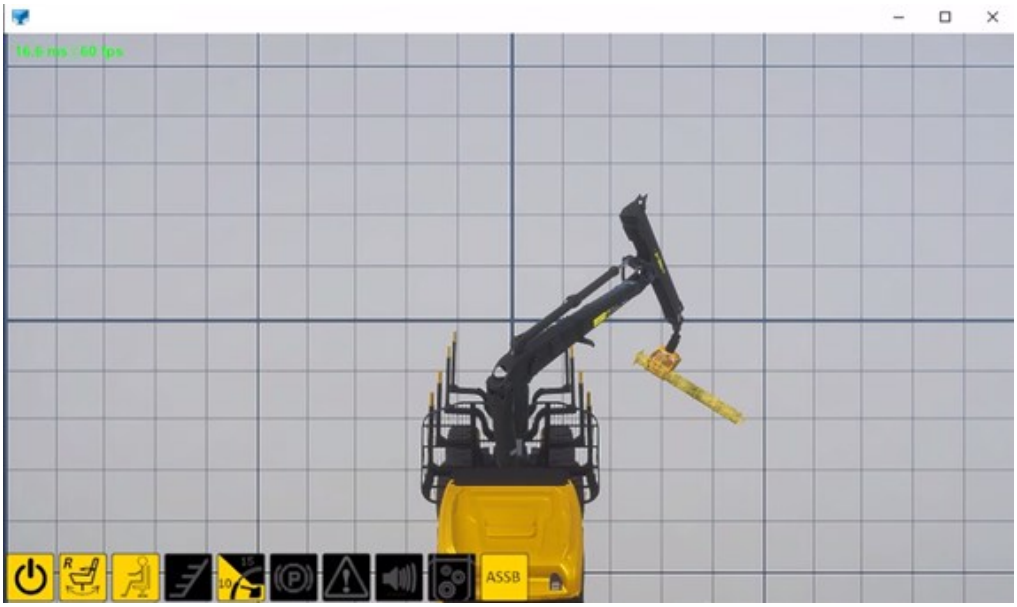


Kuva 9. Puun massan ja kiinniottokohdan asettaminen simulaattorissa

Massan päivittäminen asetettuun arvoon oli toteutettu simulaattorissa avaamalla ja sulkemalla koura. Tässä vaiheessa mallille tuotiin ohjausvirrat kouran avaamiselle, sulkemiselle ja pyörittämiselle. Lähetettäviin signaaleihin lisättiin kouran kynsien asentotiedot ja rotaattorin eli kourankääntäjän asentotieto. Kouran kynsien asentotiedot oli mallissa määritelty sylinterin pituudella. Simulaattoria varten ne jouduttiin kuitenkin konvertoimaan asteiksi, sillä visualisointi lukee arvot asteina.

Kuvassa 9 näkyy, että simulaattorissa on myös mahdollisuus säätää kourassa olevien puiden kiinniottokohtaa ”grabbing position offset” -arvoa muuttamalla. Simulink-mallissa ei kuitenkaan ollut mahdollista muuttaa tätä. Keskustelimme mallin kehittäjän kanssa tästä ja hän lisäsi ominaisuuden seuraavassa versiossa. Mallille tuleviin signaaleihin lisättiin kiinniottokohdan säätöarvo. Simulaattorissa säätöarvo oli määritelty olevan maksimissaan metrin keskikohdasta molempiin suuntiin. Mallilta lähteviin signaaleihin lisättiin kouran kallistuskulman arvo asteina ja lähetettiin

simulaattorin kuvassa 8 näkyvään ”HangerTiltJointValue”-arvoon. Säätämällä kiinniottokohtaa simulaattorista, puun asento visualisoinnissa muuttui kuvan 10 mukaisesti ja koura kallistui, jos puu ei ollut tasapainossa.



Kuva 10. Puun kiinniottokohdan muutos visualisoinnissa

4.7 DLL:n generointi mallista

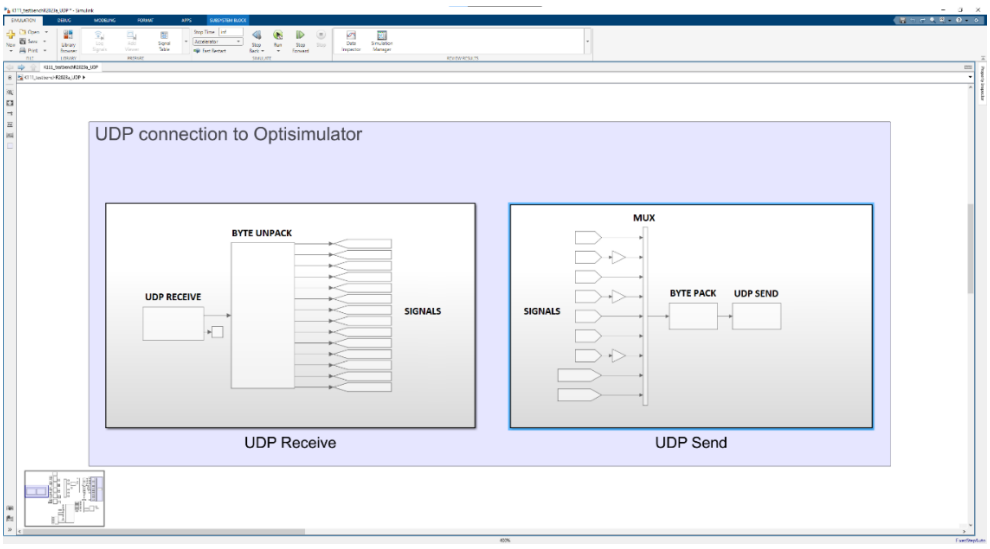
Kun Simulink-malli ja rajapinta simulaattoriin oli saatu toimimaan kokonaisuudessaan, aloitettiin pohtimaan, kuinka rajapinnan saisi käyttöön ilman Simulink-sovellusta. Tällä vältyttäisiin lisenssien ostamiselta testikäytössä. Lopulta päädyttiin koittamaan DLL:n eli jaetun kirjaston generointia mallista. DLL:n generointi vaatisi sen, että mallista generoisi ensin C-koodia. Siinä kuitenkin tuli heti vastaan ongelmia, sillä mallista ei ollut mahdollista generoida koodia rajapinnan S-funktion kanssa. S-funktio vaati TLC-tiedoston, joka korvaa simuloinnissa käytetyn koodin ja muuntaa sen DLL:ää varten. Koska S-funktio on käsin kirjoitettu, vaatii se TLC-tiedoston kirjoittamisen myös käsin. Kirjoittamista tutkittiin työryhmän kanssa, mutta koodin generointia ei saatu toimimaan. Lopulta olimme yhteydessä Mathworksin tukeen ja kävi ilmi, että TLC-tiedoston tekeminen ei ollut mahdollista S-funktion sisältämän COM-serverin takia. Tämä tarkoitti sitä, että nykyisen rajapinnan käytöstä oli luovuttava, jos kokonaisuus haluttiin saada toimimaan täysin itsenäisesti ilman Simulink-sovellusta.

4.8 UDP-yhteystapaan vaihtaminen

Kun DLL:n generoiminen nykyisen rajapinnan kanssa todettiin mahdottomaksi, palattiin alussa vaihtoehtona olleen UDP-yhteyden muodostamiseen. Työryhmän kanssa selvitettiin, mitä muutoksia vaihdos vaatisi Simulink-malliin ja simulaattoriin. Mallissa ei muuttuisi mikään muu kuin yhteystapa. Lähtevät ja tulevat signaalit pysyisivät samana. Simulaattori taas vaatisi sen, että siihen voisi muodostaa yhteyden UDP-protokollaa käyttäen. Simulaattorin kehittäjille esitettiin kysymyksiä uudesta yhteystavasta. He totesivat sen mahdolliseksi ja simulaattoriin lisättiin mahdollisuus UDP-yhteyden muodostamiseen. Lähtevät ja tulevat signaalit määriteltiin samalla tavalla kuin aiemmassa rajapinnassa, ainoastaan yhteystapa muuttui.

Kun Simulaattoriin oli saatu mahdollisuus muodostaa UDP-yhteys, aloitettiin mallin muokkaaminen UDP-yhteyttä varten. Aiemman rajapinnan sisältäneestä mallista tehtiin kopio ja poistettiin siitä kaikki rajapintaan liittyvä toiminnallisuus. Koska signaalit ja muu toiminta pysyi samana, tarvittiin lisätä vain yhteyden muodostamiseen vaadittavat elementit. Simulinkin lisäosan ”DSP System Toolbox” avulla käyttöön saa ”UDP Send” - ja ”UDP Receive” -lohkot, joilla voi määritellä lähetettävän ja vastaanotettavan datan. Lohkoissa täytyy määritellä IP-osoitteet, portit, datatyypit ja puskureiden koko.

Simulaattorisovellus lähettää ja vastaanottaa dataa uint8-muodossa. Vastaanotettavat signaalit muutetaan ”Byte Unpack” -lohkoa käyttäen vektorista yksittäisiin signaaleihin double-muotoon mallia varten. Lähetettävät signaalit taas muutetaan double-muodosta ”Byte Pack” -lohkoa käyttäen vektoriin uint8-muotoon. Vastaanoton ja lähetyksen lohkot mallissa näkyvät kuvassa 11.



Kuva 11. UDP-lohkot mallissa

Kun yhteystapa oli muutettu, testattiin sen toimivuus. UDP-yhteyden todettiin odotetusti olevan toiminnaltaan samankaltainen aiempaan rajapintaan verrattuna. Etuna siinä oli kuitenkin sen parempi suorituskyky. Kuten aiemmin mainittu, edellisessä rajapinnassa suorituskyky oli simuloinnin alussa hetken huonompi, kun yhteys muodostui. UDP-yhteyden kanssa ei ilmennyt vastaavanlaisia ongelmia simuloinnin hidastumisen kanssa ja kommunikointi mallin ja simulaattorin välillä UDP:tä käytettäessä voitiin asettaa yhteen millisekuntiin.

4.9 DLL:n generointi ja testisovelluksen teko

C-Koodin generointi mallista UDP-yhteystapaa käytettäessä ei aiheuttanut ongelmia. Siinä käytettiin Simulinkin ”Embedded Coder” -lisäosaa, jonka avulla generoidusta koodista saa suoraan generoitua jaetun kirjastotiedoston eli DLL:n. DLL-tiedoston käyttämiseksi sitä tarvitsee kutsua jotenkin. Koska simulaattorisovellusta ei ollut mahdollista muokata sisäisesti ja siten saada DLL:ää suoraan käyttöön simulaattorista, alettiin sille kehittämään testiohjelmaa, jolla toimintaa voi testata ja toteutuksen saa käyttöön.

Koodissa käytettiin Mathworksin sivustolta löytyviä esimerkkejä [21]. Testiohjelma tehtiin C-kielellä ja käytettäväksi Windowsin 64-bittisessä versiossa. Testiohjelman koodista kuvat liitteessä 1. Mallista generoitu DLL-kirjastotiedosto ladataan muistiavaruuteen LoadLibraryA -funktiota käyttäen. ”Mdl_initialize” on koodin generoinnissa luotu ”entry point” -funktio, jota kutsumalla Simulink-malli alustetaan. ”Mdl_step” -funktio sisältää koodin mallin aika-askeleiden suorittamiseen. Se laskee sen hetkiset Simulink-mallin lohkojen arvot sekä päivittää ulostulot ja koodin lohkoille. ”Mdl_terminate” -funktio sisältää koodin mallin lopettamisen.

Jotta Simulink-mallin ja testiohjelman suoritus saatiin synkronoitua, tarvittiin ohjelmaan ajastinfunktio. ”TimeSetEvent” -funktion avulla ohjelman askellus on samassa tahdissa mallin aika-askeleen kanssa. Se kutsuu ”Mdl_step” -funktiota asetetun yhden millisekunnin välein, joka on myös mallille asetetun aika-askeleen kiinteä arvo.

Ohjelman toiminta ja käyttö on hyvin yksinkertaista. Käynnistyessä se avaa komentoikkunan, joka kertoo Simulink-mallin käynnistymisestä tai mahdollisista virheilmoituksista. Ohjelma lopetetaan painamalla Enter-näppäintä, jonka jälkeen ohjelma kutsuu ”Mdl-terminate” -funktiota mallin suorittamisen lopettamiseksi ja ”timeKillEvent” -funktiota ajastimen lopettamiseksi. Lopuksi DLL-tiedosto vapautetaan muistista ”FreeLibrary”-funktiota käyttäen.

4.10 Lopullinen testaus, dokumentointi ja kehityskohteet

Lopuksi testiohjelman ja mallin toimintaa testattiin testiseinillä, jotka ovat HiL-simulaattoreita. Aiemmin rajapintaa oli testattu vain paikallisesti tietokoneella, jossa metsäkoneen ohjainmoduulien simulointi tapahtui täysin virtuaalisesti. Testiseinillä oli käytössä oikeat ohjainmoduulit. Ohjelmaa testattiin muutamalla eri seinällä, jossa oli käytössä kyseinen kuormainmalli, josta fysiikkamalli oli tehty. Joillain testiseinillä testiohjelma ei kuitenkaan käynnistynyt aluksi ollenkaan, sillä niiltä puuttui jaettuja kirjastotiedostoja, jotka kuuluvat Microsoft Visual C:n ja C++:n ajonaikaisiin kirjastoihin. Nämä jaetut kirjastotiedostot lisäämällä testiohjelma toimi ongelmitta ja yhteys simulaattoriin muodostui. Rajapinnan todettiin toimivan halutulla tavalla.

Testaamisen jälkeen kirjoitin dokumentin, kuinka rajapinnan saa käyttöön joko Simulink-sovelluksen vaativaa yhteystapaa käyttäen tai myöhempää UDP-yhteystapaa käyttäen. Molempien toiminta oli muuten samanlainen, mutta UDP-rajapintaa käyttämällä suorituskyky oli parempi ja lisäksi mahdollistettiin rajapinnan käyttö ilman Simulink-sovellusta.

Simulink-malli ei vielä rajapinnan valmistuessa ollut täysin valmis. Kaikki tarvittava toiminnallisuus oli jo tässä vaiheessa mukana, mutta itse fysiikkamallin toiminnan lopullinen verifiointi jäi tämän opinnäytetyön ulkopuolelle. Mallista generoitu DLL-tiedosto oli myös suunnitelmissa integroida suoraan osaksi simulaattorisovellusta käyttöönoton helpottamiseksi, mutta simulaattorin muokkaaminen sisäisesti ei ollut mahdollista ja täten jäi simulaattorin kehittäjien tehtäväksi.

5 Yhteenveto ja pohdinta

Tämän opinnäytetyön aiheena oli luoda rajapinta Simulink-fysiikkamallin ja Ponssen simulaattorisovelluksen välille. Tarkoituksena oli parantaa simulaattorin realistisuutta, kun kuormaimen liikkeissä on otettu fysiikka mukaan. Fysiikan mukaan ottaminen mahdollistaa muun muassa uudenlaisia testejä, joita simulaattorin avulla voi suorittaa. Tällaisia testejä voisi olla kuormaimen käyttäytymisen testaaminen eri olosuhteissa. Esimerkiksi miten se käyttäytyy eri kuormilla ja miten se vaikuttaa vaikkapa painearvoihin ja muihin komponentteihin? Myös erilaisten törmäystilanteiden testaaminen voisi olla mahdollista, ja näin selvittää, kuinka kuormain niissä reagoi. Tällaisilla uusilla testeillä voidaan löytää mahdollisia kehityskohteita. Tämä hyödyttää tuotekehitystä, sillä kun toiminta voidaan todentaa realistisesti virtuaalisesti, vähentää se muutoksia myöhemmässä tuotekehitysvaiheessa. Vaikka luotu rajapinta tehtiin tässä työssä toimimaan vain tietystä kuormainmallista luodun mallin kanssa, voi sen toiminnallisuutta hyödyntää tulevaisuudessa muissakin vastaavanlaisissa sovellutuksissa.

Projekti alkoi työryhmän perustamisella, suunnittelulla sekä fysiikkamalliin ja simulaattoriin perehtymisellä. Yhteyttä fysiikkamallin ja simulaattorin välille lähdettiin toteuttamaan jo olemassa olevalla rajapinnalla. Toteutus sen avulla vaikutti aluksi mahdolliselta ja sen testaamien oli toimeksiantajan toive. Rajapinnan toiminnallisuutta päivitettiin sitä mukaa, kun fysiikkamalli ja simulaattori kehittyivät eteenpäin.

Vaikka kyseessä olikin lisätavoite eikä vaatimus, rajapinta haluttiin saada toimimaan itsenäisesti ilman Simulink-sovellusta. Tämän ansiosta rajapinnan käyttö testikäytössä olisi edullisempaa, kun Simulink-lisenssejä ei tarvittaisi. Tämä ei ollut mahdollista alkuperäisellä rajapintatoteutuksella, joten rajapinnan yhteystapa jouduttiin muuttamaan UDP-yhteystapaan. Näin fysiikkamallista saatiin luotua jaettu kirjastotiedosto eli DLL. DLL saatiin käyttöön luomalla sille testiohjelma, joka kutsuu sitä ja käynnistää mallin. Lopuksi rajapinnan käyttö dokumentoitiin ja yhteyttä testattiin HiL-simulaattorilla. Olisin kuitenkin halunnut testata rajapintaa enemmän käytännössä, sillä testaaminen jäi melko suppeaksi aikataulusyistä. Olisi myös ollut mielenkiintoista olla mukana mallin toiminnan verifioinnissa ja viimeistelyssä ja nähdä, miten rajapinta toimisi silloin.

Aluksi opinnäytetyön aihe vaikutti haastavalta, mutta mielenkiintoiselta. Projektissa alkuun pääseminen tuntui vaikealta, koska minulle annettiin toteutukseen paljon päätäntävaltaa eikä selkeää suunnitelmaa heti aluksi ollut. Myös Simulink-mallin toiminnan ymmärtäminen itsessään vei jonkin verran aikaa, sillä Simulinkin käyttö ei ollut entuudestaan kovin tuttua. Hyvän työryhmän

ja muiden mukana olleiden osapuolien ansiosta apua kuitenkin sai aina tarvittaessa. Haasteena oli myös projektin hidas eteneminen, sillä siinä oli useita osapuolia mukana. Uusia toiminnallisuuksia joutui odottamaan niin Simulink-malliin kuin välillä simulaattorisovellukseen, ennen kuin rajapinnan kehittämisessä päästiin eteenpäin.

Oli hienoa olla mukana projektissa, jossa omalla tekemisellä voi pitkällä tähtäimellä olla suuriakin hyötyjä tuotekehitykseen. Työssä opin monia uusia asioita, muun muassa simuloinnista, mallipohjaisesta suunnittelusta ja projektityöskentelystä yleisesti. Työssä myös kävi ilmi se, kuinka ohjelmistokehitysprojekti ei aina etene niin kuin on alun perin ajateltu. Kaiken kaikkiaan olen erittäin tyytyväinen lopputulokseen, johon päästiin, sillä kaikki toimeksiantajan tavoitteet täyttyivät ja rajapinnan toiminta oli hyvä. Simulaattorin oma kuormaimen kinematiikka saatiin korvattua onnistuneesti fysiikkamallissa lasketuilla arvoilla. Lisäksi toteutus saatiin halutusti toimimaan täysin itsenäisesti ja näin mahdollistettiin laajempi käyttö testikäytössä.

Lähteet

1. Vuosikertomus. Ponsse. [Internet]. [viitattu 9.10.2024]. Saatavilla: https://www.ponsse.com/documents/1812961/1820735/Vuosikertomus_2023.pdf/514f7b58-754c-e231-a16f-41750dfa5ffe?t=1710493658148

2. Ponsse. Buffalo. [Valokuva]. [viitattu 4.11.2024] Saatavilla: https://cdn-ponsse.contenthub.fi/cdn-cgi/image/format=auto/api/v1/cdn/17964572

3. Jokinen T. 6. painos. Tuotekehitys. Helsinki: Aalto-yliopisto, Teknillinen korkeakoulu; 2001. Saatavilla: http://lib.tkk.fi/Reports/2010/isbn9789526033204.pdf

4. Tuotekehitys. Wikipedia. [Internet]. [viitattu 11.11.2024] Saatavilla: https://fi.wikipedia.org/wiki/Tuotekehitys

5. Software engineering. Wikipedia. [Internet]. [viitattu 20.10.2024]. Saatavilla: https://en.wikipedia.org/wiki/Software_development

6. Ohjelmistotuotanto osa 1. Github. [Internet]. [viitattu 18.11.2024]. Saatavilla: https://ohjelmistotuotanto-hy.github.io/osa1/

7. Api. IBM. [Internet]. [viitattu 21.10.2024]. Saatavilla: https://www.ibm.com/topics/api

8. What is an API? Redhat. [Internet]. [viitattu 5.11.2024]. Saatavilla: https://www.redhat.com/en/topics/api/what-are-application-programming-interfaces

9. Kirjasto (tietotekniikka). Wikipedia. [Internet]. [viitattu 21.10.2024]. Saatavilla: https://fi.wikipedia.org/wiki/Kirjasto_(tietotekniikka)

10. What is software testing? IBM. [Internet]. [viitattu 7.2.2025]. Saatavilla: https://www.ibm.com/think/topics/software-testing

11. Mitä on testaus? Gofore. [Internet]. [viitattu 7.2.2025]. Saatavilla: https://gofore.com/mita-on-testaus-ohjelmistotestaus-laadunvarmistajana/

12. Ohjelmistotuotanto osa 3. Github. [Internet]. [viitattu 8.2.2025]. Saatavilla: https://ohjelmistotuotanto-hy.github.io/osa3/

13. Springer. V-model. [Valokuva]. [viitattu 8.2.2025] Saatavilla: https://books.google.fi/books?id=3C3YDwAAQBAJ&pg=PA4#v=onepage&q&f=false

14. Miksi mallipohjainen suunnittelu? Gofore. [Internet]. [viitattu 9.1.2025]. Saatavilla: https://gofore.com/miksi-mallipohjainen-suunnittelu-simulointi-ja-testaus-osaksi-vaativaa-ohjelmistokehitysta/

15. What is Model-Based Design? Synopsys. [Internet]. [viitattu 9.1.2025]. Saatavilla: https://www.synopsys.com/glossary/what-is-model-based-design.html

16. Llorente, R. M. Practical Control of Electric Machines – Model-Based Design and Simulation. Advances in Industrial Control. Springer; 2020. Saatavilla: https://books.google.fi/books?id=3C3YDwAAQBAJ&printsec=frontcover&redir_esc=y#v=onepage&q&f=true

17. What is Hardware-in-the-Loop (HIL)? MathWorks. [Internet]. [viitattu 10.3.2025]. Saatavilla: https://www.mathworks.com/discovery/hardware-in-the-loop-hil.html

18. Co-simulation introduction. Altair. [Intertnet]. [viitattu 16.4.2025]. Saatavilla: https://2022.help.altair.com/2022/hwsolvers/ms/topics/solvers/ms/an_introduction_to_co-simulation.htm

19. Matlab. Wikipedia. [Intertnet]. [viitattu 10.12.2024]. Saatavilla: https://en.wikipedia.org/wiki/MATLAB

20. Bottom-up and top-down design. Wikipedia. [Intertnet]. [viitattu 11.12.2024]. Saatavilla: https://en.wikipedia.org/wiki/Bottom%E2%80%93up_and_top%E2%80%93down_design

21. Generate shared library for export to external code base. MathWorks. [Intertnet]. [viitattu 8.1.2025]. Saatavilla: https://www.mathworks.com/help//releases/R2021a/rtw/ug/export-generated-shared-libraries.html

Liitteet

K111_Testbench_UDP (Global Scope)

```c
#include <windows.h>
#include <stdio.h>
#include <stddef.h>
#pragma comment(lib, "winmm.lib")

//WINDOWS
#define GETSYMBOLADDR GetProcAddress
#define LOADLIB LoadLibrary
#define CLOSELIB FreeLibrary

//Initialize model entry point functions
void (*mdl_initialize)(void);
void (*mdl_step)(void);
void (*mdl_terminate)(void);

int timerID;

//Timer callback function
void CALLBACK TimerCallback() {
    //user application step function
    mdl_step();
}

//Stop the timer if exit signal is read
BOOL WINAPI CtrlHandler(DWORD signal) {
    if (signal == CTRL_C_EVENT) {
        timeKillEvent(timerID);  // Stop the timer
    }
    return TRUE;
}

int main() {
    void* handleLib;
    //Specify the dll path
    const char* dllPath = "C:\\OptiSimulator\\UDPConnection\\K111_testbenchR2023a_UDP_win64.dll";

    //LoadLibraryA loads the specified module into the address space of the calling process.
    handleLib = LoadLibraryA(dllPath);

    //Print message if the specified library is not found
    if (!handleLib) {
        printf("Cannot open the specified shared library from C:\\OptiSimulator\\UDPConnection."
            "\nPlease check if the *K111_testbenchR2023a_UDP_win64.dll* file is in the folder."
            "\n\nPress Enter to exit.");
        getchar();
        return(-1);
    }
    //Model entry point functions
    mdl_initialize = (void(*)(void))GETSYMBOLADDR(handleLib, "K111_testbenchR2023a_UDP_initialize");
    mdl_step = (void(*)(void))GETSYMBOLADDR(handleLib, "K111_testbenchR2023a_UDP_step");
    mdl_terminate = (void(*)(void))GETSYMBOLADDR(handleLib, "K111_testbenchR2023a_UDP_terminate");

```

```
52
53      //Check if model entry point functions are found
54      if (mdl_initialize && mdl_step && mdl_terminate) {
55          //User application initialization function
56          mdl_initialize();
57          printf("Simulink model is running.");
58
59          //Print message if control handler cant be set
60          if (!SetConsoleCtrlHandler(CtrlHandler, TRUE)) {
61              fprintf(stderr, "ERROR: Could not set control handler\n");
62              return 1;
63          }
64          //Variable for event delay in ms
65          const UINT rate_in_milliseconds = 1;
66
67          //Start specified timer event
68          timerID = timeSetEvent(
69              rate_in_milliseconds,   //uDelay. Event delay,
70              1,                      //uResolution.Resolution of the timer event in ms,
71              TimerCallback,          //lpTimeProc. Pointer to a callback funtion,
72              0,                      //dwUser. User-supplied callback data,
73              TIME_PERIODIC           //fuEvent. Event occurs every uDelay
74          );
75          //Print message if timer cannot be created
76          if (timerID == 0) {
77              fprintf(stderr, "\nFailed to create multimedia timer\n");
78              return 1;
79          }
80          printf("\nPress Enter to quit...");
81          getchar();
82
83          printf("\nStopping simulation...");
84          //User application terminate function
85          mdl_terminate();
86
87          //Cancel timer event
88          timeKillEvent(timerID);
89
90          printf("\nExiting cleanly...\n");
91          Sleep(500);
92
93          while (1) {
94              printf("\nPress Enter again to close this window.");
95              getchar();
96              return 0;
97          }
98      }
99      //Print message if entry point functions are not found
100     else {
101         printf("Cannot locate the specified reference(s) in the shared library. Press Enter to exit.");
102         getchar();
103         return(-1);
104     }
105     return(CLOSELIB(handleLib));
106 }
107 /* End of file */
```