Sami Kari-Pekka Äyrämö

REAALIAIKAJÄRJESTELMÄN MALLINTAMINEN UML – KUVAUSMENETELMÄN AVULLA

Tietotekniikan pro gradu -tutkielma

Ohjelmistotekniikan linja

28.11.2002

Jyväskylän yliopisto

Tietotekniikan laitos

Tekijä: Sami Kari-Pekka Äyrämö

Yhteystiedot: sami.ayramo@patria.fi

Työn nimi: Reaaliaikajärjestelmän mallintaminen UML -kuvausmenetelmän avulla

Title in English: Modeling a real-time system with UML

Työ: Pro gradu -tutkielma

Sivumäärä: 106

Linja: Ohjelmistotekniikka.

Teettäjä: Patria Aviation Oy ja Jyväskylän yliopiston tietotekniikan laitos

Avainsanat: UML, reaaliaikajärjestelmä, luokka, olio, prosessi, kommunikointi

Keywords: UML, real-time system, class, object, process, communication

Tiivistelmä: Tässä tutkielmassa selvitetään, kuinka UML-kuvauskieltä voidaan hyödyntää sulautetun reaaliaikajärjestelmän mallintamisessa. Tavoitteena on tutkia, mitkä osat UML-notaatiosta tukevat reaaliaikajärjestelmien mallintamista ja tämän perusteella arvioida, kuinka hyvin notaatio yleensä ottaen soveltuu niiden mallintamiseen. Työ perustuu kirjallisuudesta hankittuun tietoon ja notaation soveltamiseen käytännössä yksinkertaisen esimerkkisovelluksen avulla.

Abstract: The aim of this thesis is to find out the proper ways to utilize UML in modeling an embedded real-time system. The main goal is to find out the parts of the notation that support the modeling a real-time system and, by using this information, to make an evaluation how suitable the notation is for real-time system modeling in general. This work is based on the knowledge acquired from numbers of source books and the experiences obtained by modeling a simple test application.

Esipuhe

Kiitokset opiskeluaikani pitkäaikaiselle työnantajalleni Patrialle mahdollisuudesta toteuttaa tämä opinnäytetyö. Kiitokset erityisesti kaikille teille, jotka olette osallistuneet tämän työn eteenpäin viemiseen ja toisaalta unohtamatta niitäkään, jotka ovat jo aiempien vuosien aikana saaneet minut innostumaan ohjelmistotekniikan kiemuroista. Kiitokset myös ohjelmistotekniikan professori Tommi Kärkkäiselle ohjauksesta ja kannustuksesta niin opintojen kuin tämän työn aikana, sekä sulautettujen järjestelmien professori Jarkko Vuorelle avusta työn viimeistelyvaiheessa. Kaikkein suurimmat kiitokset kuuluvat tietysti vanhemmilleni Seijalle ja Karille, siitä mittaamattoman suuresta tuesta, jota olette opiskeluvuosien aikana antaneet. Ilman sitä ei tämä työ olisi koskaan valmistunut. Lisäksi haluan kiitokseni muistaa myös siskoani Terhiä sekä kaikkia muitakin ystäviä, opiskelu-/työkavereita jne. joita en pysty tähän edes nimeltä luettelemaan. Teistä kaikista on ollut apua ja on mukava huomata nimiä miettiessä, että teitä onkin paljon enemmän kuin tavallisesti ymmärtääkään.

Jyväskylässä, marraskuu 2002 *Sami Äyrämö*

Termiluettelo

ALU	Arithmetic Logic Unit, aritmeettislooginen laskentayksikkö
API	Application Programming Interface, liityntärajapinta käyttöjärjestelmän ja sovelluksen välillä
CPU	Control Processing Unit, mikroprosessori
HW	Hardware, järjestelmän laitteisto-osuus
LAN	Local Area Network
OMG	Object Management Group
RTOS	Real Time Operating System, reaaliaikakäyttöjärjestelmä
SBC	Single Board Computer, yhdellä mikroprosessorilla varustettu sulautettava prosessorikortti
SW	Software, järjestelmän ohjelmisto-osuus
UML	Unified Modelling Language Ohjelmistojen analyysiin ja suunnitteluun tarkoitettu standardoitu kuvauskieli
WAN	Wide Area Network

Sisältö

1 Johdanto

Tämän tutkielman tarkoituksena on selvittää UML-kuvausmenetelmän ja oliokeskeisen lähestymistavan käyttömahdollisuuksia ja soveltuvuutta ohjelmistoprojekteissa, jotka toteutetaan sulautettuihin reaaliaikaympäristöihin. Tietolähteinä käytetään lähinnä alan kirjallisuutta. Saavutettuja tuloksia on tarkoitus hyödyntää tulevaisuudessa Patria Avionics -liiketoiminnan ohjelmistoprojekteissa.

1.1 Tavoite

Koska sulautettujen ohjelmistojen sovellusalue asettaa yleensä ohjelmistoille korkeat laatuvaatimukset, pyritään selvittämään, kuinka oliomallinnusta ja UML-kuvausmenetelmää voitaisiin tarkoituksenmukaisesti hyödyntää projektien eri vaiheissa. Työn tarkoituksena on myös selvittää UML:n ja oliomallinnuksen soveltuvuutta ohjelmistojen toteuttamiseen sulautetussa ympäristössä, joissa alustana voi toimia esimerkiksi VxWorks-reaaliaikakäyttöjärjestelmä [VxW99a]. Tavoitteena on lisäksi löytää UML-kaavioista ne, jotka tukevat reaaliaikajärjestelmien kehittämistä.

1.2 Työn rakenne

Työssä tutkitaan ensiksi yleisellä tasolla sulautettujen reaaliaikajärjestelmien piirteitä ja vaatimuksia. Luvussa 2 tarkastellaan lyhyesti sulautettujen reaaliaikajärjestelmien historiaa ja vertaillaan sulautetun järjestelmän eroja tavanomaisiin tietokonejärjestelmiin. Kolmannessa luvussa tarkastellaan reaaliaikaisuudesta aiheutuvia vaatimuksia ja ongelmia sekä kuinka niitä voidaan ratkaista. Lisäksi tarkastellaan mitä palveluita nykyaikaiset reaaliaikakäyttöjärjestelmät tarjoavat reaaliaikasovellusten kehittäjille. Luvussa 4 keskitytään olioterminologian läpikäymiseen. Luku 5 esittelee UML-kuvauskielen kaaviot. Lisäksi pyritään tarkastelemaan mitä reaaliaikapiirteitä kaavioista löytyy. Luvussa 6 esitellään yksinkertainen reaaliaikajärjestelmien kehittämistä varten suunniteltu prosessimalli. Luvussa 7 on mallinnettu prosessimallia hyödyntäen yksinkertaistettu versio tiedonsiirtojärjestelmästä, joka toimii työssä testisovelluksena olio- ja UML-ideoiden toimivuudelle. Luvun 8

tarkoituksena on tarkastella työn tuloksia ja tehdä johtopäätöksiä olio-/UML-mallinnuksen soveltuvuudesta ko. ympäristössä.

2 Sulautetut järjestelmät

Tässä luvussa tarkastellaan lyhyesti reaaliaikatietokoneiden historiaa sekä sulautettujen reaaliaikajärjestelmien perusominaisuuksia ja eroja tavanomaisiin tietokonejärjestelmiin verrattuna. Luvussa käydään lyhyesti läpi myös sulautettujen järjestelmien arkkitehtuuria ja reaaliaikajärjestelmiin liittyviä vaatimuksia ja ominaisuuksia. Luku perustuu pääasiallisesti lähteisiin [Ced02, Int02, Bal00, Dou99, Wil97].

2.1 Historiaa

Vaikka tietokoneet, mikroprosessorit ja erilaiset tietokoneohjelmat ovat nykyaikana osa jokapäiväistä elämäämme, niiden historia ei ole vielä pitkä. Tietokonealan kehitys- ja kasvu on kuitenkin ollut hyvin nopeaa. Eräs syy alan menestykselle on sen tarjoamat mahdollisuudet saavuttaa merkittäviä kustannushyötyjä, sillä ohjelmoitavalla arkkitehtuurilla laskentayksikön (ALU) kustannukset voidaan jakaa useiden eri ongelmien kesken. Ensimmäinen reaaliaikatoimintoja tukemaan suunniteltu tietokone oli Whirlwind [Wil97]. Sen kehittäminen alkoi 1945 MIT:ssä (Massachusetts Institute of Technology) Yhdysvalloissa ja ensimmäisen kerran se esiteltiin 20. huhtikuuta 1952. Whirlwind oli ensimmäinen digitaalinen tietokone, joka pystyi näyttämään reaaliaikaista tekstiä ja grafiikkaa videoterminaalilla, joka oli siihen aikaan suuri oskilloskooppinäyttö. Tietokone oli hyvin suurikokoinen verrattuna nykyaikaisiin tietokoneisiin, sillä se sisälsi mm. yli 4000 tyhjöputkea.

1970-luvun alussa mikroprosessorit eli CPU-yksiköt korvasivat erillisillä elektroniikkakomponenteilla toteutettavan logiikan. Ensimmäinen mikroprosessori, Intelin valmistama 4004-prosessori, esiteltiin vuonna 1971. Se suunniteltiin laskukonetta varten ja toiminta koostui lähinnä aritmeettisista perusfunktioista. 4004-prosessorin kellotaajuus oli 108 kHz, väyläleveys 4 bittiä ja se koostui 2300 transistorista. Myöhemmin mikroprosessorien kellotaajuudet ja laskentakapasiteetit ovat kehittyneet huomattavaa vauhtia. Uusien 2000-luvun prosessorien kellotaajuudet ovat jo reilusti yli gigahertsin luokkaa, väyläleveydet 32 tai 64 bittiä ja ne sisältävät kymmeniä miljoonia transistoreja yhdellä CPU-sirulla.

Vaikka ohjelmistosuunnittelu mikroprosessorijärjestelmään saattaa olla kallista, on se yleensä vain kertakustannus. Saman toiminnallisuuden toteuttavan laitteistopohjaisen ratkaisun kokonaiskustannukset saattaisivat olla monta kertaa suurempia. Mikroprosessoriratkaisut ovat myös joustavampia. Jos suunniteltavasta järjestelmästä täytyy toteuttaa erilaisia versioita, saattaa niiden toteuttaminen laitteistopohjaisessa ratkaisussa vaatia elektroniikan suunnittelemista lähes kokonaan uudelleen. Mikroprosessoritoteutuksessa saatetaan selvitä muutaman lähdekoodirivin kirjoittamisella. Tästä nähdään, että mikroprosessoripohjaisten toteutusten etuna on erityisesti niiden ohjelmoitavuus. Sama laitteisto voidaan helposti ohjelmoida suorittamaan kahta tai useampaa erilaista tehtävää. Mikroprosessorit ovat myös pienentäneet tietokoneiden kokoa ja ne ovat levinneet osaksi lähes kaikkia nykyajan elektronisia laitteita ja koneita, kuten PC-tietokoneet, lentokoneet, autot, CD-soittimet, digitaaliset televisiot, kamerat jne.

2.2 Sulautetut järjestelmät

Reaaliaikajärjestelmät ovat usein sulautettuja järjestelmiä, joissa tietokone on ”upotettu” osaksi suurempaa laitteistokokonaisuutta. Sulautetun järjestelmän tarkka määritteleminen on vaikeaa, sillä myös tavalliset PC-tietokonekokonaisuudet sisältävät sulautettuja mikroprosessoreja. Myös kirjallisuudesta on vaikea löytää yksiselitteistä määritystä sulautetulle järjestelmälle. Esimerkiksi tulostin tai näppäimistö voivat sisältää erillisen sulautetun mikroprosessorin. Sulautettua järjestelmää voisi kuvata ohjelmiston ja laitteiston yhdistelmäksi, joka on suunniteltu suorittamaan yksi tai useampi ennalta määritelty toiminto. Järjestelmä saa syötteenä parametreja, joiden perusteella se suorittaa jonkin funktion ja palauttaa vasteena funktion arvon. Vasteiden arvoilla saatetaan ohjata esimerkiksi toista järjestelmää tai antaa käyttäjälle informaatiota.

Sulautetut järjestelmät ovat yleensä suljettuja ja ohjelmistot on tallennettu haihtumattomaan muistiin. Tämän vuoksi tavallisen käyttäjän voi olla usein vaikea tietää, että sulautettu CPU-yksikkö suorittaa toimintoja hänen käyttämässään laitteessa. Informaation välitys käyttäjälle voi tapahtua esimerkiksi LCD- eli nestekidenäytön tai ledinäytön kautta. Toisaalta sulautettu järjestelmä ei välttämättä aina edes kommunikoi välittömästi käyttäjän kanssa. Järjestelmä saattaa esimerkiksi ohjata auton jarrujen toimintaa lukkiutumattomassa

ABS-jarrujärjestelmässä tai koodata ja paketoida sanomia, joita radion välityksellä lähetetään lentokoneesta toiseen. Sulautettuja mikroprosessoreja käytetäänkin nykyään moniin eri tarkoituksiin mitä erilaisimmissa ympäristöissä, kuten lentokone- ja avaruuselektroniikassa, autoissa, sykemittareissa, matkapuhelimissa jne. Monet tämän kaltaiset ympäristöt edellyttävät järjestelmältä luotettavaa ja vikasietoista toimintaa.

Edellä mainitut seikat tekevät eroa sulautetun järjestelmän ja tavallisen koti- tai toimistokäytössä olevan PC-tietokoneen välille. Yhteistä sulautetuille järjestelmille ja PC-tietokoneille on, että molempiin kuuluu vähintään mikroprosessori, muistia ja jonkin tasoinen ohjelmisto. Nykyään myös valmiiden käyttöjärjestelmien hyödyntäminen on yleistynyt sulautetuissa järjestelmissä. Tavanomaisia PC-oheislaitteita, kuten monitori, näppäimistö tai hiiri, ei yleensä esiinny sulautetuissa järjestelmissä. Lisäksi olennainen osa PC-tietokoneita on niiden yleiskäyttöisyyttä tukevat massamuistilaitteet, joista ohjelmat voidaan ladata erikseen haihtuvaan muistiin suoritusta varten. Koska sulautetut järjestelmät ovat yleensä sovelluskohtaisia ja toteuttavat vain etukäteen määriteltyjä toimintoja, ei niissä tarvita massamuisteja erilaisten ohjelmistojen tallentamista varten. Useat erilaiset ohjelmistot, laitteet ja avoin rakenne antavat mahdollisuuden laajentaa ja päivittää PC-tietokoneita, sekä käyttää niitä monissa eri tarkoituksissa. Barr käyttää PC-tietokoneille myös hyvin kuvaavaa nimitystä yleiskäyttöinen tietokone (engl. *general-purpose computer*) [Bar99]. Tämä kuvaa erityisesti sitä, että PC-tietokoneen valmistajan on mahdotonta valmistusvaiheessa tarkasti tietää, kuinka käyttäjä haluaa tietokonetta hyödyntää. PC-tietokonetta saatetaan käyttää verkossa tiedostopalvelimena, pelien pelaamiseen, tekstinkäsittelyyn tai sulautettujen järjestelmien ohjelmistokehitykseen.

2.3 Sulautetun järjestelmän arkkitehtuuri

Sulautettu tietokonejärjestelmä koostuu yleensä vähintään seuraavista ohjelmisto- ja laiteelementeistä [Wol00]:

- Mikroprosessori eli CPU on nykyään aina olennainen osa sulautettua tietokonejärjestelmää. Prosessoreihin on tarjolla monia erilaisia arkkitehtuureja ja saman arkkitehtuurinkin prosessoreissa voi olla vaihtoehtoina esimerkiksi eri kellotaajuuksia ja väyläle-

veyksiä. Prosessorin valinta on sulautetun järjestelmän suunnittelussa yksi tärkeimpiä asioita ja sen valinnassa täytyy huomioida myös ohjelmisto, jota sillä aiotaan suorittaa.

- Väylän (engl. *bus*) tehtävä on datan, osoitteiden ja ohjaussignaalien siirtäminen prosessorin, muistin ja oheislaitteiden välillä. Sovelluksissa, joissa tarvitsee siirtää runsaasti dataa väylää pitkin, saattaa väylä olla suorituskyvylle rajoittavampi tekijä kuin prosessori. Suorituskykyä voidaan tosin parantaa käyttämällä useampia väyliä.

- Muistia tarvitaan jokaisessa tietokonejärjestelmässä. Muistityyppejä on hyvin monenlaisia eri tarkoituksia varten. Muistityypin valinta voi vaikuttaa merkittävästi myös kustannuksiin. Muistin määrän tarve on tärkeää analysoida huolella järjestelmää suunniteltaessa. Siihen vaikuttaa tietysti käsiteltävän datan määrä ja ohjelmakäskyjen koko. Muistin nopeus vaikuttaa järjestelmän suorituskykyyn, joka on tärkeää varsinkin aikakriittisissä sovelluksissa.

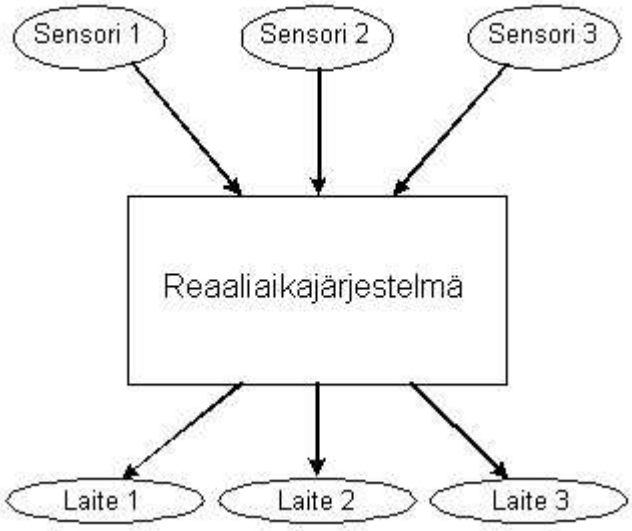
- I/O-laitteiden kautta järjestelmä monitoroi ja kontrolloi sen ulkopuolisia laitteita. I/O-laitteita voivat olla esimerkiksi erilaiset sensorit, näyttölaitteet tai ohjauslaitteet.

Kuva 2.1. Sulautetun reaaliaikajärjestelmän kerrosrakenne.

Edellä esitellyt elementit muodostavat rungon sulautetun järjestelmän laitteistoalustalle. Yhdistävänä kerroksena laitteiston ja sovellusohjelmiston välillä voi toimia käyttöjärjestelmä, joka piilottaa varsinaisen laitteiston ohjelmoijalta (ks. kuva 2.1). Nykyään myös sulautettuja järjestelmiä varten on olemassa käyttöjärjestelmiä, kuten esimerkiksi VRTX [Men02], VxWorks [VxW99a], OS-9 [Rad02] sekä Embedded Linux [Emb02].

3 Reaaliaikajärjestelmät

Edellisessä kappaleessa käytiin läpi yleisesti tietokonejärjestelmiä jakamatta niitä sen tarkemmin reaaliaika- tai ei-reaaliaikajärjestelmiin. Tässä kappaleessa esitellään reaaliaikajärjestelmille tyypillisiä toiminnallisia vaatimuksia, rajoitteita ja käsitteitä. Kappaleessa esitellään lyhyesti myös menetelmiä, joilla voidaan ratkaista vaatimuksia ja rajoitteita, kuten tehtävien ajoittaminen ja resurssien käyttö, koskevia ongelmia. Lopuksi tarkastellaan lyhyesti niitä ominaisuuksia, joita tyypilliset reaaliaikakäyttöjärjestelmät tarjoavat reaaliaikasovelluksen kehittäjälle.



Kuva 3.1. Tyypillinen reaaliaikajärjestelmän rakenne.

3.1 Reaaliaikajärjestelmän määritelmä

Tarkkaa rajaa reaaliaikajärjestelmien ja ei-reaaliaikajärjestelmien välille on vaikea asettaa. Kirjallisuudessa reaaliaikajärjestelmät yhdistetään yleensä toiminnan oikea-aikaisuuteen, mikä on tietysti nimenkin perusteella helposti pääteltävissä. Reaaliaikakirjallisuus viittaa monesti myös käsitteisiin luotettavuus, vikatoleranttius, oikeellisuus, turvallisuus, kompleksisuus ja tehokkuus [Dou99], [Bur90], [Sel94], [Gla83]. Lisäksi reaaliaikajärjestelmä on

usein myös sulautettu järjestelmä. Kuvassa 3.1 on esitetty tyypillinen reaaliaikajärjestelmän rakenne.

Reaaliaikaisuus tarkoittaa, että järjestelmän tehtävien suorittamiselle ja palveluiden tarjoamiselle on asetettu aikarajoitteita. Reaaliaikajärjestelmä saa sisäisiä ja ulkoisia herätteitä, jotka edellyttävät määrättyjen toimintojen suorittamista ja vasteiden antamista vasteaikojen asettamissa rajoissa. Toiminnan oikeellisuus ei siis riipu vain saadusta loogisesta tuloksesta, vaan myös ajanhetkestä, jolloin tulos saadaan. James Martinin määritelmä reaaliaikajärjestelmille vuodelta 1967 on edelleen hyvin kuvaava [Mar67]:

”A real-time computer system may be defined as one which controls an environment by receiving data, processing them, and taking action or returning results sufficiently quickly to affect the functioning of the environment at that time.”

Teoksissa [Bur90, You82] esitetään myös kaksi hyvää määritelmää reaaliaikajärjestelmille:

”Any system in which the time at which output is produced is significant. This is usually because the input corresponds to some movement in the physical world, and the output has to relate to that same movement. The lag from input time to output time must be sufficiently small for acceptable timeliness.”

”Any information processing activity or system which has to respond to externally-generated input stimuli within a finite and specified period.”

Reaaliaikajärjestelmissä järjestelmän vasteaika on siis ratkaiseva tekijä. Vasteajat voivat vaihdella huomattavasti eri järjestelmien välillä. Joissakin järjestelmissä vasteajat ovat jopa millisekuntien luokkaa ja joissakin järjestelmissä riittävä vasteaika voi olla useita kymmeniä sekunteja. Esimerkiksi lentokonetutkan digitaalista signaalinkäsittelyä suorittavan järjestelmän toiminnan täytyy olla erittäin nopeaa, jotta oikea kuva ympäristöstä välittyisi ohjaajalle ajoissa [Sti83]. Monille pankki-, tuotanto- tai kaupallisessa käytössä oleville järjestelmille hyväksyttävä vasteaika voi olla kymmeniäkin sekunteja, joskus jopa tunteja [Gla83]. Silti nekin voidaan käsittää reaaliaikajärjestelmiksi.

Esimerkkejä reaaliaikajärjestelmistä ovat mm. digitaaliset ohjausjärjestelmät, signaalinkäsittelyjärjestelmät, reaaliaikatietokannat ja erilaiset multimediasovellukset. Järjestelmiä on esitelty tarkemmin esimerkiksi lähteessä [Liu00].

3.2 Reaaliaikajärjestelmän toiminnalliset vaatimukset

Tässä kappaleessa esitellään muutamia reaaliaikajärjestelmille ominaisia toiminnallisia vaatimuksia. Edellä esitetyt reaaliaikajärjestelmän määritelmät korostavat toiminnan oikea-aikaisuutta. Kappaleessa myös tarkennetaan oikea-aikaisuuteen liittyviä vaatimuksia sekä esitellään muutamia muita tyypillisiä toiminnallisia vaatimuksia.

3.2.1 Oikea-aikaisuus

Monet reaaliaikajärjestelmät toimivat ympäristöissä, joissa niihin kohdistuu ankarampia vaatimuksia kuin tavanomaisiin tietokonejärjestelmiin. Usein on esimerkiksi tärkeää kyky toimia itsenäisesti ja luotettavasti pitkiä aikoja. Reaaliaikajärjestelmien välillä on kuitenkin eroja sen suhteen, kuinka vakavia voivat olla seuraukset väärään aikaan tapahtuneesta toiminnasta. Tämän perusteella reaaliaikajärjestelmät jaetaan yleensä *soft*- ja *hard*-reaaliaikajärjestelmiin [Liu00].

Hard–tyyppisellä reaaliaikajärjestelmällä tarkoitetaan järjestelmää, jossa on ehdottaman tärkeää pystyä noudattamaan kaikkia sen tehtäville asetettuja vasteaikoja. Tällaisessa järjestelmässä vasteajan ylittymisestä voi aiheutua vakavat seuraukset, kuten esimerkiksi lentokoneiden törmääminen tai potilaan menehtyminen.

Soft-tyyppisessä järjestelmässä sallitaan satunnaiset vasteaikojen ylittymiset ja tärkeämpää saattaa olla esimerkiksi nopea keskimääräinen vasteaika tai suuri käskyjen suorituslukumäärä. *Soft*-tyypin vaatimukset voidaan määritellä myös tilastollisina lukuina, jolloin riittävä ehto oikealle toiminannalle voi olla, että määrätty prosenttiosuus vasteajoista saavutetaan ajoissa. Esimerkkinä *soft*-reaaliaikajärjestelmästä voidaan mainita puhelinverkko. Vaikka soittaja yleensä odottaa että puhelu yhdistyy nopeasti, hyväksyy hän satunnaiset viiveet puheluiden yhdistämisessä ja joskus jopa tarpeen uusia soittoyritys useaan kertaan.

Lähteessä [Dou99] esitellään myös kolmas reaaliaikajärjestelmätyyppi *firm*, joka on yhdistelmä *soft*- ja *hard*-tyypin vaatimuksista. Ehdot *firm*–tyypin toiminalle ajan suhteen määritellään seuraavasti: keskimääräisten vasteaikojen on oltava riittävän nopeita ja toiseksi, sen on pystyttävä saavuttavamaan jokainen vasteaika viimeistään jossakin määrätyssä ajassa.

3.2.2 Rinnakkaisuus

Reaaliaikajärjestelmältä vaaditaan yleensä kykyä suorittaa useita tehtäviä rinnakkain. Rinnakkaisuus tarkoittaa useiden toimintosekvenssien suorittamista yhtäaikaisesti. Jos toimintosekvenssit suoritetaan yhdellä prosessorilla, puhutaan ns. pseudorinnakkaisuudesta. Sillä tarkoitetaan, että vaikka käyttäjän näkökulmasta tehtävien suoritus näyttää rinnakkaiselta, suoritetaankin niitä todellisuudessa peräkkäin lyhyissä jaksoissa. Tehtävien suoritusjaksot ovat kuitenkin niin lyhyitä, että käyttäjä ei pysty havaitsemaan tehtävien vaihtumista. Jotta järjestelmä tietäisi mitä tehtävää sen pitäisi suorittaa, on tehtäville määrättävä tärkeysjärjestys. Tehtävän tärkeys kerrotaan prioriteettiarvolla. Todellisissa rinnakkaisissa järjestelmissä tehtäviä suoritetaan useammalla prosessorilla. Rinnakkaisuus tuo monia haasteita reaaliaikajärjestelmien suunnitteluun, kuten rinnakkain suoritettavien tehtävien ajoittaminen ja ulkoisiin herätteisiin reagoiminen oikein, tehtävien välinen kommunikointi ja resurssien yhtäaikainen käytönhallinta. Rinnakkaisuuden hallitsemiseen kehitettyjä menetelmiä esitellään mm. lähteissä [Bur90, Dou99].

3.2.3 Virheettömyys, kestävyys ja vikasietoisuus

Reaaliaikajärjestelmät toimivat usein kriittisissä ympäristöissä, joista esimerkkejä ovat mm. avioniikka-, ydinvoima- sekä lääketieteelliset sovellusympäristöt. Koska näissä ympäristöissä virheellinen toiminta ei ole toivottavaa eikä yleensä edes hyväksyttävää, on reaaliaikajärjestelmää kehitettäessä otettava yleensä huomioon virheettömyys, kestävyys ja vikasietoisuus. Virheettömyydellä tarkoitetaan, että järjestelmä tekee oikeita asioita oikeaan aikaan. Kestävyydellä tarkoitetaan, että järjestelmä toimii oikein jopa odottamattomissa olosuhteissa, ja vielä jonkin järjestelmän osan vioittuessakin pystyy hallitsemaan tilanteen ja mahdollisesti korjaamaan virheen. Tyypillisiä virhetilanteita ovat esimerkiksi *deadlock*–

ja resurssien kilpavaraustilanteet. Näitä tilanteita ja niistä toipumista käsitellään tarkemmin tulevissa kappaleissa. [Dou99]

3.2.4 Rajoitteet ja kompleksisuus

Reaaliaikajärjestelmät ovat usein sekä rakenteeltaan että toiminnaltaan melko kompleksisia. Tämä on seurausta siitä, että useimmiten ne suunnitellaan ympäristöihin, joissa niiden täytyy pystyä käsittelemään useita toisistaan riippumattomia syötevirtoja ja tuottamaan siitä huolimatta vasteita kriittisissäkin aikarajoissa. Vaikka syötteiden saapumistakin voi olla etukäteen vaikea ennustaa, täytyy niihin pystyä vastaamaan vaatimusmäärittelyn antamissa aikarajoissa. Syötekuormituksen huomattavat vaihtelut lisäävät siis järjestelmän kompleksisuutta. [Sel94]

Reaaliaikajärjestelmien suunnittelijoille lisää haasteita luovat myös laitteiden fyysiset rajoitteet. Kuten edellä on mainittu, reaaliaikajärjestelmät ovat usein sulautettuja järjestelmiä, jotka toimivat vähäisellä muisti- ja laitteistotuella toimintaympäristönsä asettamien vaatimusten takia. Hajautetussa reaaliaikajärjestelmässä puolestaan tehtävien suoritus ja resurssit on jaettu useamman prosessorin kesken. Hajautettu järjestelmä tarkoittaa sovellusympäristöä, jossa sovelluksen suoritus on jaettu fyysisesti useaan eri solmuun. Jokainen solmu on erillinen tietokonejärjestelmä, joilla ei ole keskenään jaettua muistia. Solmut yhdistetään keskenään LAN- tai WAN-tietoverkolla. Käyttäjän näkökulmasta katsottuna hajautettu järjestelmä näyttää yhdeltä tietokonejärjestelmältä.[Gom00, Tan92]

3.3 Reaaliaikajärjestelmän tehtävien suoritus

Reaaliaikajärjestelmien tehtävä on tyypillisesti monitoroida erilaisten sensorien kautta ympäristöään ja ohjata erilaisia laitteita vastaanottamiensa syötteiden perusteella. Monesti ulkoinen ympäristö, jonka kanssa reaaliaikajärjestelmä kommunikoi, ei koostu suoraan ihmisistä vaan erilaisista laitteista. Siksi on yleensä tärkeämpää tarkastella reaaliaikajärjestelmän kommunikointia näiden laitteiden kuin ihmisten kanssa, jotka näiden laitteiden kautta hyödyntävät reaaliaikajärjestelmän toimintaa. Sensoreita ja ohjauslaitteita kutsutaan monesti myös I/O-laitteiksi. Ilman näitä laitteita reaaliaikajärjestelmä olisikin kuin PC-tietokone ilman näppäimistöä, hiirtä, näyttöä ja kirjoitinta, koska se ei pystyisi tuottamaan

ulkoiselle ympäristölle mitään itsestään riippumattomia vasteita. Reaaliaikajärjestelmä voi esimerkiksi tarkkailla sensoriensa kautta kemiallista prosessia ilmoittaakseen tarvittaessa virhe- tai vaaratilanteista. Esimerkkejä sensoreista ovat mm. lämpömittari tai vastaanotto-antenni. Reaaliaikajärjestelmät voidaankin jakaa reaktiivisiin ja aikaohjattuihin järjestelmiin sen perusteella, kuinka ne saavat herätteitä. [Dou99]

3.3.1 Reaktiiviset järjestelmät

Sensoreiden kautta saapuvia syötteitä voidaan kutsua herätteiksi tai ulkoisiksi tapahtumiksi, joihin reaaliaikajärjestelmä vastaa. Jotta reaaliaikajärjestelmät pystyvät suorittamaan toimenpiteitä saamiensa herätteiden perusteella niiltä edellytetään reaktiivisuutta. **Reaktiivinen järjestelmä** (engl. *reactive system*) on jatkuvassa vuorovaikutuksessa ympäristönsä kanssa. Samalla, kun ympäristö generoi herätteitä, reagoi järjestelmä niihin vaihtamalla tilaansa tai generoimalla vasteita. Herätteet voivat olla periodisia tai aperiodisia. Vaikka ulkoisten herätteiden ilmaantumista on usein vaikea ennustaa, täytyy järjestelmien yleensä pystyä reagoimaan herätteeseen silloin kun se saapuu. Reaktiiviset järjestelmät käsittelevät yleensä diskreettiä dataa. Diskreetillä datalla tarkoitetaan tietoa, jossa jokainen syöte saapuu tietyllä yksittäisellä ajanhetkellä ja syötteillä on yksilölliset arvot, jotka käsitellään erikseen [Ell94]. Diskreetin datan saapuminen aiheuttaa yleensä keskeytyksen (engl. *interrupt*), jolla kerrotaan reaaliaikaohjelmistolle uuden datan saapumisesta. Muuten järjestelmän täytyy tarkkailla (engl. *poll*) sisääntulokanavaa varmistuakseen, että data ehditään lukea ennen seuraavan datan päällekirjoitusta. [Sel94, Dou99]

3.3.2 Aikaohjatut järjestelmät

Jos järjestelmän toiminnan käynnistää jokin absoluuttisen ajanhetken kohtaaminen tai aikaintervallin täyttyminen, puhutaan **aikaohjatusta järjestelmä** (engl. *time-based system*). Hyvä esimerkki aikaperusteisesta järjestelmästä on järjestelmä, joka ottaa näytteitä analogisesta datasta säännöllisten jaksojen välein. Aikaohjattu järjestelmä käsitteleekin yleensä jatkuvaa datavirtaa. Jatkuva datavirta tarkoittaa syötettä, jossa data on jatkuvasti järjestelmän vastaanotettavissa [Ell94]. Tällöin syötteistä ei aiheudu erillisiä herätteitä, vaan järjestelmän on otettava näytteitä syötevirrasta riittävän suurella taajuudella. Näytteenotossa data

muunnetaan diskreetiksi, jonka jälkeen järjestelmän käsiteltävänä on yksilöllisiä, mitattavia suureita (kuten jännite tai lämpötila) tietyillä ajan hetkillä kuvaavia arvoja. Arvot voidaan käsitellä tämän jälkeen samoin kuin diskreetti syöte. Mitä suurempi on näytteenottotaajuus, sitä paremmin diskreetti data kuvaa jatkuvan datan käyttäytymistä. On kuitenkin hyvä huomata, että analogisia syötteitä vastaanottava järjestelmä ei välttämättä aina ole aikaohjattu, sillä näytteenottohetki voi määräytyä myös jonkin muun satunnaisesti saapuvan herätteen perusteella. [Dou99]

3.3.3 Tehtävien suoritus

Pystyäkseen ohjaamaan ja informoimaan ympäristöään vastaanottamiensa syötteiden perusteella, reaaliaikajärjestelmä suorittaa tehtäviä (engl. *task*), joka käytännössä tarkoittaa suoritettavaa ohjelmakoodisekvenssiä. Tehtävän aikarajoite (engl. *deadline*) voi olla *hard*- tai *soft*–tyyppiä (ks. kpl 3.2.1).

Tehtävät voidaan jakaa periodisiin, aperiodisiin ja sporadisiin tehtäviin niiden suoritukseen saapumisen perusteella. Periodiset tehtävät saapuvat suoritukseen säännöllisin väliajoin. Suoritukseen saapumisvälin vaihtelu on niin pientä, että sitä ei tarvitse yleensä huomioida. Periodinen tehtävä voi käynnistyä esimerkiksi reaaliaikajärjestelmän sisäisen kellon generoimasta herätteestä. Satunnaisesti suoritukseen saapuvia kutsutaan joko aperiodisiksi tai sporadisiksi. Sporadinen tehtävä saapuu suoritukseen satunnaisesti, mutta sillä on *hard*-tyypin aikarajoite. Aperiodisella tehtävällä ei ole aikarajoitteita tai ne ovat *soft*-tyyppisiä. Satunnaisesti käynnistyvien tehtävien saapumista suoritukseen voidaan kuvata esimerkiksi todennäköisyysjakaumalla. Monesti on tarpeellista tietää alaraja satunnaisten tehtävien suoritukseen saapumiselle, jotta voidaan analysoida, ovatko tehtävät skeduloitavissa.

Reaaliaikasovelluksille on tyypillistä, että ne koostuvat useista toimintaa ohjaavista säikeistä (engl. *thread, lightweight process*), jotka käyttävät keskenään yhteistä muistiavaruutta. Yksi säie koostuu useista peräkkäin suoritettavista toiminnoista (engl. *job, action*), jotka muodostavat yhdessä toiminnallisen kokonaisuuden, jolla on yhteinen prioriteetti. Säiettä kutsutaan yleensä tehtäväksi reaaliaikajärjestelmissä. Itse sovellusta kutsutaan

myös prosessiksi (engl. *heavyweight process*). Prosessit eivät käytä keskenään yhteistä muistiavaruutta.[Liu00, Dou99]

3.3.4 Tehtävien suoritusaikaan liittyviä käsitteitä

Koska reaaliaikajärjestelmät ovat toiminnoiltaan aikakriittisiä, esitellään seuraavaksi muutamia niihin liittyviä englannin kielisiä käsitteitä teoksen [Liu00] perusteella.

Release time eli tehtävän suorituksen vapautumishetkellä tarkoitetaan sitä absoluuttisen ajan hetkeä, jolloin reaaliaikajärjestelmän suoritettavaksi tarkoitettu tehtävä on valmiina suoritukseen. Tehtävä voidaan suorittaa milloin tahansa sen suoritukseen vapautumishetken jälkeen, kunhan resurssien varaus ym. edellytykset täyttyvät. *Release time jitter* tarkoittaa vaihteluväliä, jossa tehtävän suorituksen vapautumishetki voi vaihdella. Käytännön syistä voidaan suorituksen vapautumisajaksi approksimoida joko vaihteluvälin pienin tai suurin arvo, jolloin vapautumishetki on kiinteä (engl. *fixed*).

Interrelease time on aika, joka kuluu tehtävän suoritukseen vapautumisesta siihen hetkeen, kun tehtävä vapautuu suoritukseen seuraavan kerran. Aperiodisten ja sporadisten tehtävien herätteet saapuvat satunnaisin väliajoin, jolloin tehtävän suorituksen vapautumisvälin kestoa voidaan kuvata todennäköisyysjakaumalla. Satunnaisesti saapuvista herätteistä käynnistyvien tehtävien suoritukseen saapumishetkelle käytetään myös nimitystä *arrival time* ja vastaavasti saman tehtävän peräkkäisten suorituksen vapautumishetkien erotukselle nimitystä *interarrival time.*

Deadline on hetki, jolloin tehtävän täytyy olla suoritettuna. Jos *deadline* on suhteellinen, se tarkoittaa maksimiaikaa, joka on sallittua käyttää tehtävän suorituksen vapautumisesta sen suorituksen päättymiseen. *Response time* eli vasteaika kuvaa aikaa tehtävän suorituksen vapautumisesta sen valmistumiseen, joten se ei saa olla suurempi kuin suhteellinen *deadline*. Absoluuttinen *deadline* tarkoittaa absoluuttista ajanhetkeä, jolloin tehtävä täytyy olla suoritettuna. Se voidaan laskea seuraavasti:

$$\textit{Absolute deadline} = \textit{Release time} + \textit{Relative deadline}$$

3.3.5 Tehtävien tilat

Reaaliaikajärjestelmän tehtävät voivat olla kolmessa eri tilassa: *running*, *ready* ja *waiting*. Jos järjestelmään kuuluu vain yksi prosessori, on selvää, että vain yksi tehtävä voi olla suorituksessa eli *running*–tilassa kerrallaan. Tehtävät, jotka ovat valmiina suoritettaviksi, mutta eivät juuri sillä hetkellä saa suoritusaikaa, ovat *ready*–tilassa. Tehtävät, jotka odottavat jotain itseensä nähden ulkoista tapahtumaa, ovat *waiting*–tilassa. Suorituksessa oleva tehtävä voi keskeytyä vain käyttöjärjestelmän määräyksestä tai tehtävän jäädessä odottamaan jonkin resurssin vapautumista. Kun tehtävä ei saa suorituksen aikana tarvitsemaansa resurssia käyttöön, on järkevämpää luovuttaa suoritusvuoro toiselle tehtävälle ja siirtyä *waiting*–tilaan. *Ready*-tilassa olevia tehtäviä voidaan säilyttää suoritukseen valmiina olevien tehtävien listassa, josta käyttöjärjestelmän skeduleri siirtää niitä suoritukseen skedulointisääntöjen mukaan. [Bar99]

3.3.6 Tehtävän kriittisyys

Tehtävän *kriittisyys* (tai *tärkeys*) kuvaa sitä, kuinka tärkeä tehtävän suoritus on muihin tehtäviin verrattuna. Ylikuormitustilanteissa ei aina välttämättä pystytä noudattamaan kaikkien tehtävien aikarajoitteita. Tällöin vähemmän tärkeiden tehtävien suorituksen voidaan antaa myöhästyä. Osa tehtävien skedulointi- ja resurssienvarausalgoritmeista osaa huomioida tehtävän tärkeyden ohjatessaan suoritusvuoroja. Tehtävän kriittisyys kerrotaan skedulerille yleensä tehtävän attribuutilla, jota kutsutaan prioriteetiksi (engl. *priority*) tai painoksi (engl. *weight*). Korkeamman prioriteetin tehtävä on etuoikeutettu suoritukseen ennen matalamman prioriteetin tehtäviä. *Hard*-tyypin tehtävät ovat aina korkean prioriteetin tehtäviä, koska ne eivät saa myöhästyä. Prioriteettia kuvataan yleensä lukuarvolla, esim. 0–255 [VxW01]. Se, kuvaako suurempi vai pienempi luku korkeampaa prioriteettia, riippuu käyttöjärjestelmästä.[Liu00]

3.3.7 Tehtävien pre-emptiivisyys

Pre-emptiivinen tehtävä tarkoittaa vähemmän tärkeä tehtävää, jonka suoritus voidaan keskeyttää, jos kiireisempi tehtävä tulee suoritusjonoon odottamaan. Myöhemmin, kun kiireellisempi tehtävä on suoritettu loppuun, voidaan keskeytetyn tehtävän suorittamista jatkaa.

Tehtävää, jonka suoritusta ei voi keskeyttää, kutsutaan epäpre-emptiiviseksi (engl. *non-pre-emptive*). Tehtävä voi olla myös rajoitettu epäpre-empitiiviseksi vain tietyltä osin. Mm. keskeytysten käsittelyn aikana ei tehtävän suoritusta ole järkevää keskeyttää. Kun tehtävän suoritus keskeytetään toisen tehtävän toimesta, täytyy ennen uuden tehtävän aloitusta tallettaa vanhan tehtävän tila muistiin. Kun keskeytyksen aiheuttanut tehtävä on suoritettu loppuun, voidaan pre-emptiivisen tehtävän suorittamista jälleen jatkaa.[Liu00, Dou99]

3.3.8 Tehtävien toteutus reaaliaikajärjestelmässä

Kuten edellä on todettu, termejä tehtävä, prosessi ja säie käytetään monesti samassa merkityksessä. Usein puhutaankin tietokonejärjestelmän suorittavan säikeitä, mikä tarkoittaa samaa asiaa kuin tehtävien suorittaminen reaaliaikajärjestelmässä. Koska nykyään on olemassa paljon erilaisia reaaliaikakäyttöjärjestelmiä, ei ohjelmoijan tarvitse toteuttaa säikeitä tai tehtäviä yleensä alusta pitäen. Käyttöjärjestelmä suorittaa omien toimintojensa ylläpitoon tarvittavia säikeitä ja ohjelmoija voi pyytää käyttöjärjestelmää luomaan säikeitä sovelluksen edellyttämällä tavalla. Säikeitä käsittelevät aliohjelmakutsut löytyvät yleensä käyttöjärjestelmän API–rajapinnan systeemikutsuista. Kun säie luodaan, käyttöjärjestelmä varaa muistia säikeelle ja koodille, jota säie suorittaa. Samalla säikeelle luodaan oma TCB (*Thread Control Block*) –taulukko, jossa säilytetään kaikkea säikeen hallintaan ja skedulointiin liittyvää tietoa. Lisäksi taulukossa on mm. säikeen ID–numero, jolla säikeet yksilöidään, sekä alkuosoite säikeen suorittamaan koodiin. Taulukon tiedot muuttuvat aina säikeen suorituksen aikana ja sinne tallennetaan mm. ohjelmalaskurin ja tilarekisterin tiedot juuri ennen toisen säikeen tuloa suoritusvuoroon. Käyttöjärjestelmä tuhoaa säikeen poistamalla sen TCB–taulukon ja vapauttamalla sen varaaman muistin.[Liu00, Tan00, Bal00]

3.4 Reaaliaikajärjestelmän tehtävien skedulointi

Jotta reaaliaikajärjestelmän oikea-aikaisuuteen liittyvät vaatimukset toteutuvat, tarvitaan menetelmiä, joilla pyritään ohjaamaan tehtävät suoritukseen oikea-aikaisesti. Tehtävien

suoritusvuoroja jakaa ns. aikatauluttaja (engl. *scheduler*). Englannin kielen sana *schedule* määritellään lähteessä [Jon02] seuraavasti:

”A Schedule is a reservation for spatial (processor, RAM) and temporal (time) resources for a given task set.”

Tässä työssä käytetään englannin kielen sanan *schedule* tilalla sanaa aikataulu, vaikka muuten käytetään vierasperäisiä termejä skedulointi ja skeduleri. Tehtävien skeduloinnilla pyritään siis ratkaisemaan, mikä tehtävä suoritetaan, kun useampi tehtävä on yhtä aikaa odottamassa suoritukseen pääsyä. Skedulointialgoritmi generoi aikataulun annetulle tehtäväjoukolle ja ajonaikaiselle järjestelmälle. Skedulointialgoritmia suorittaa järjestelmän skeduleri, jonka vastuulla on päättää ohjelman suorituksen aikana, mikä tehtävä suoritetaan seuraavaksi. Skeduleri on yleensä osa reaaliaikakäyttöjärjestelmää. Seuraavassa on muutama skedulointiin liittyvä määritelmä [Liu00]:

- Aikataulu on toteuttamiskelpoinen (engl. *feasible*), jos se täyttää annetun tehtäväjoukon osalta kaikki sovellukselle asetetut ehdot.

- Tehtäväjoukkoa kutsutaan skeduloituvaksi (engl. *schedulable*), jos on olemassa vähintään yksi algoritmi, joka pystyy generoimaan toteuttamiskelpoisen aikataulun.

- Skedulointialgoritmin sanotaan olevan optimaalinen skeduloitavuuden suhteen, jos se aina löytää toteuttamiskelpoisen aikataulun tehtäville, kun sellainen on olemassa.

Reaaliaikajärjestelmien skeduloimiseen on olemassa useita menetelmiä, jotka voidaan jakaa staattisiin ja dynaamisiin. Dynaamiset menetelmät voidaan jakaa edelleen staattista ja dynaamista prioriteettia käyttäviin [Vuo01]. Skedulointialgoritmit voidaan jakaa myös pre- ja epäpre-emptiivisiin. Seuraavaksi esitellään lyhyesti kolme yleistä skedulointiperiaatetta, jotka on esitetty tarkemmin lähteessä [Liu00].

3.4.1 Kello-ohjattu ajoittaminen

Kello-ohjattu skedulointi (engl. *clock-driven skeduling*) tarkoittaa menetelmää, jossa päätös siitä, mikä tehtävä suoritetaan, on sidottu jo etukäteen tiettyyn ajanhetkeen. Tätä kutsu-

taan myös staattiseksi tai *offline*-skeduloinniksi. Menetelmä edellyttää, että periodisten tehtävien lukumäärä ja niiden parametrit, millä tahansa ajan hetkellä systeemin ollessa toiminnassa, tiedetään etukäteen. Jaksottainen aikataulu, jossa on informaatio tehtävien suoritusjärjestyksestä, lasketaan etukäteen ja tallennetaan skeduleria varten. Skeduleri ajastaa tehtävät jokaisella skedulointikerralla aikataulun mukaan. Vaihtamalla skedulointiohjelmaa voidaan järjestelmä skeduloida uudestaan eri toimintamoodia varten. Kello-ohjatun skeduloinnin etuja ovat sen yksinkertaisuus, skedulerin tehtävien vaihtamiseen kuluttaman ajan minimoiminen ja rinnakkaisuuden hallinta. Erillisiä synkronointimekanismeja tehtävien välille ei tarvita, koska ne voidaan huomioida jo tehtävien suoritusaikataulua suunniteltaessa. Myös ohjelmakoodin debuggaaminen ja skedulointianalyysin suorittaminen on yksinkertaisempaa. Toisaalta kello-ohjattu skedulointi on tehoton ja joustamaton, koska se ei suoritusaikaa antaessaan huomioi sitä, onko tehtävällä mitään järkevää suoritettavaa. [Liu00]

3.4.2 Weighted round-robin

Weighted round-robin -menetelmää käytetään aikajaetuissa sovelluksissa. Menetelmän ideana on säilyttää suoritukseen valmiina olevia tehtäviä *first-in-first-out*- (FIFO) jonossa. Jonoon ensimmäisenä saapunut tehtävä otetaan ensimmäisenä suoritukseen. Jos tehtävä ei ehdi suorittaa työtään loppuun ennen sille varatun ajan loppumista, siirretään se FIFO–jonon loppuun odottamaan seuraavaa suoritusvuoroa. Round-robin -menetelmässä jokainen jonossa oleva tehtävä saa yhden aikasiivun käyttöönsä kierrosta kohden. Weighted round-robin -menetelmässä prosessoriaikaa ei kuitenkaan jaeta tehtävien kesken tasan, vaan eri tehtävillä on eri painot (engl. *weight*). Painoarvo viittaa siihen, kuinka paljon prosessoriaikaa tehtävä saa kullakin suoritusvuorolla. Painoarvolla *wt* tehtävä saa käyttöönsä *wt* kappaletta aikasiivuja, jolloin yhden kierroksen pituus vastaa painoarvojen summaa. Painoarvoja säätämällä voidaan hidastaa tai nopeuttaa tehtävien valmistumisaikaa. Weighted round-robin -menetelmää sovelletaan lähinnä reaaliaikaisissa tietoverkoissa.[Liu00]

3.4.3 Prioriteetteihin perustuva ajoittaminen

Prioriteetteihin perustuva ajoittaminen ottaa huomioon tehtävien tärkeyden myöntäessään niille suoritusaikaa. Tätä kutsutaan myös dynaamiseksi tai *online*-skeduloinniksi. Tehtävien prioriteetit voivat olla kiinteitä tai ajon aikana dynaamisesti määrättäviä. Prioriteettiin perustuvaa skedulointia kutsutaan myös ahneeksi (engl. *greedy*), koska se pyrkii tekemään lokaalisti optimaalisia päätöksiä. Lokaali optimaalisuus tarkoittaa, että skeduleri tarkastelee jatkuvasti, onko prosessori tai tehtävän pyytämä resurssi vapaana. Jos tehtävän tarvitsema resurssi ja prosessori on vapaana, lokaalisti optimaalinen ratkaisu ohjaa tehtävän aina suoritukseen. Kuten tehtävät voidaan myös prioriteetteihin perustuvat skedulointirajoitteet jakaa pre- ja epäpre-emptiivisiin. Epäpre-emptiivinen skedulointi ei salli suorituksessa olevan tehtävän keskeyttämistä. Sen etuna on esimerkiksi automaattinen *mutual exclusion* -ongelman ratkaiseminen (ks. kpl 3.5.1). Epäpre-emptiivinen skedulointi voi olla kuitenkin tehottomampi, koska se ei pysty hyödyntämään tehtäville annettuja prioriteetteja yhtä hyvin kuin pre-emptiivinen skedulointi ja aikarajoitteiden täyttäminen on hankalampaa. Paremmuutta on tarkasteltava kuitenkin tapauskohtaisesti.

Esimerkkejä dynaamisia prioriteetteja käyttävistä algoritmeista ovat EDF- (*Earliest-Deadline-First)* ja LST- *(Least-Slack-Time-First)* algoritmit. EDF–algoritmi antaa ajonaikana tehtäville prioriteetteja niiden aikarajoitteiden mukaan, jolloin kiireellisin tehtävä suoritetaan ensin. EDF–algoritmin hyödyntäminen edellyttää, että tehtävät ovat pre-emptiivisiä eivätkä kilpaile samoista resursseista. LST–algoritmi antaa korkeimman prioriteetin sille tehtävälle, jolla on vähiten aikaa ”tuhlattavana” ennen *deadline*-aikarajaa. Joutoaika, *slack-time*, voidaan laskea seuraavasti: millä tahansa ajan hetkellä t, työn *slack-time* s, *deadline*–aikarajan d suhteen saadaan laskemalla s = d – t – e, missä e tarkoittaa aikaa, joka tehtävän loppuun suorittamiseen kuluu. Mitä pienempi *slack-time* ajanhetkellä saadaan, sitä korkeampi on tehtävän prioriteetti. LST–algoritmi edellyttää EDF–algoritmin tavoin pre-emptiivisiä tehtäviä. LST–algoritmin heikkous EDF–algoritmiin verrattuna on, että EDF–algoritmi ei vaadi tietoa tehtävän suoritusajasta etukäteen. Suoritusajan arvioiminen on usein hyvin hankalaa etukäteen ja se tekee myös LST–algoritmin käytön vaikeaksi.

Tunnetuimpia staattisiin prioriteetteihin perustuvia algoritmeja ovat RM- (*rate-monotonic*) ja DM- (*deadline-monotonic)* algoritmit. RM–algoritmin prioriteetit jaetaan tehtävien periodien (vrt. *interrelease-time*) mukaan. Mitä lyhyempi on tehtävän periodi, sitä suuremman prioriteetin se saa. Kääntäen voidaan sanoa: mitä suurempi on tehtävän suoritukseen saapumisnopeus, sitä suurempi prioriteetti. RM–algoritmi edellyttää myös pre-emptiivisiä tehtäviä, koska matalamman prioriteetin tehtävät täytyy pystyä keskeyttämään korkeamman prioriteetin tehtävien käynnistyessä. RM–algoritmin etu on, että sille on helppo suorittaa skeduloitavuusanalyysi. DM–algoritmin prioriteetit perustuvat tehtävien suhteellisiin *deadline*–aikoihin. Mitä lyhyempi on tehtävän suorittamiseen sallittu aika, sitä korkeampi on tehtävän prioriteetti. Jos kaikkien skeduloitavien tehtävien *deadline*–ajat ovat suhteutettu tehtävien periodeihin, ovat RM- ja DM–algoritmit identtisiä.[Liu00]

3.5 Tehtävien välinen synkronointi ja kommunikointi

Vaikka reaaliaikajärjestelmissä suoritettavat tehtävät koostuvat periaatteessa itsenäisistä toimintosekvensseistä, ei niitä voida kuitenkaan käsitellä toisistaan riippumattomina kokonaisuuksina. Tämä aiheutuu reaaliaikajärjestelmien rinnakkaisuudesta. Toteuttaakseen järjestelmälle asetetut vaatimukset, täytyy rinnakkaisten tehtävien yleensä myös kommunikoida keskenään ja synkronoida toimintojaan. Kommunikoinnilla viitataan informaation välitykseen tehtävien välillä. Kommunikointi voi olla joko synkronista tai asynkronista. Synkronoidussa kommunikoinnissa tehtävän on odotettava synkronointi-ilmoitusta (monesti käytetään esim. termejä: kuittaus, signaali tai kättely) toiselta tehtävältä ennen kuin suoritus voi jatkua. Muuten puhutaan asynkronisesta kommunikoinnista, jolloin tehtävä voi jatkaa suoritustaan välittämättä siitä, onko toinen tehtävä vastaanottanut viestin. Synkronointi on tarpeellista esimerkiksi käytettäessä useamman tehtävän kesken jaettuja resursseja. Koska resurssienkäyttö täytyy yleensä olla hallittua tiedon eheyden säilyttämiseksi, tarvitaan mekanismeja estämään resurssia siirtymästä toisen tehtävän käyttöön kesken varauksen. Tehtävien väliseen kommunikointiin ja synkronointiin on olemassa useita menetelmiä. Menetelmät voidaan karkeasti jakaa jaettua muistia käyttäviin ja sanomapohjaisiin menetelmiin. Esimerkkejä jaettua muistia käyttävistä menetelmistä ovat *busy waiting* -algoritmi, semaforit, CCR-menetelmä ja monitorit. Sanomapohjaisia menetelmät tarkoitta-

vat viestien lähettämistä jonkin viestinvälitysmekanismin avulla. RPC (*Remote Procedure Call)* -menetelmä on tyypillinen kommunikointimekanismi hajautetuissa järjestelmissä [Tan92].[Bur90, Gal95]

3.5.1 Rinnakkaisten tehtävien synkronointiongelmat

Jos tehtävät käyttävät ns. jaettuja muuttujia tai resursseja (engl. *shared variables / resources)*, syntyy kriittisiä sektioita, joiden aikana on turvattava datan eheys. Jaksoa, joka alkaa tehtävän varatessa jaetun resurssin ja päättyy sen vapauttamiseen, sanotaan kriittiseksi sektioksi (engl. *critical section*) [Liu00]. Kriittisen sektion aikana suoritettavaa komentosekvenssiä eivät toiset tehtävät saa häiritä. Kriittinen sektio voidaan hallita suorittamalla tehtävien synkronointi keskinäisellä poissulkemisella kriittisen sektion aikana (engl. *mutual exclusion -synchronization*). *Mutual exclusion* -ongelman kuvasi ensimmäisenä Dijkstra 1965 [Dij65]. Ongelma voidaan ratkaista esimerkiksi semaforeilla. Jos tehtävät eivät ole keskenään kytkettyjä, ei keskinäistä poissulkemista niiden välillä tarvita. Toinen tehtävien välistä synkronointia edellyttävä tilanne syntyy, kun suoritusvuoroa odottavan tehtävän turvallinen ja hallittu suoritus onnistuu vain, jos jokin toinen tehtävä on suorittanut tietyn toiminnon loppuun tai siirtynyt johonkin vaadittuun tilaan. Tällöin tarvitaan ehdollista synkronointia (engl. *condition synchronization*).

3.5.2 Busy waiting

Yksinkertainen tapa toteuttaa ehdollinen synkronointi on *busy waiting* –silmukka. Tässä menetelmässä tehtävä ilmoittaa tilamuutoksesta vaihtamalla lipun (engl. *flag*) tilaa. Tilamuutosta odottava tehtävä tarkkailee lipun arvoa ja jatkaa eteenpäin vasta havaittuaan oikean lipun arvon. *Busy waiting* on melko resursseja tuhlaava menetelmä, koska odottava tehtävä jää silmukkaan odottamaan lipun vaihtumista oikeaan tilaan samalla kuluttaen suoritusaikaa tekemättä mitään hyödyllistä.

3.5.3 Semafori

Semaforit ovat yksinkertainen mekanismi *mutual exclusion* –ongelman ratkaisemiseen ja ehdolliseen synkronointiin tehtävien välillä. Yksinkertaisimmillaan semafori voidaan mää-

ritellä ei-negatiiviseksi kokonaisluvuksi, jonka arvoa voidaan muuttaa alustusta lukuunottamatta vain kahdella operaatiolla [Bur90]. Semaforin alkuarvon määrää sillä varattavien resurssien lukumäärä. Operaatioille käytetään useita erilaisia nimiä lähteestä riippuen, mutta tässä käytetään lähteessä [Tan92] esiintyviä nimiä *UP* ja *DOWN*.

Kun tehtävä haluaa varata semaforilla valvotun resurssin, se kutsuu *DOWN*-operaatiota, joka tarkistaa onko semaforin arvo suurempi kuin nolla. Jos arvo suurempi kuin nolla, vähennetään semaforin arvoa yhdellä jolloin tehtävä saa resurssin käyttöön ja voi jatkaa suoritustaan. Jos semaforin arvo on nolla, niin semaforia pyytävä tehtävä siirtyy *waiting*-tilaan, koska kaikki semaforilla varattavat resurssit ovat käytössä. Semaforin arvon tarkastaminen, muuttaminen ja tehtävän siirtyminen *waiting*-tilaan täytyy olla ns. jakamaton eli atominen toimenpide (engl. *atomic action*). Kun tehtävä ei enää tarvitse varaamaansa resurssia, se kutsuu semaforin UP-operaatiota vapauttaakseen resurssin muiden tehtävien käyttöön. UP-operaatio kasvattaa semaforin arvoa yhdellä. Tämän jälkeen vähintään yksi resurssi on seuraavien tehtävien käytettävissä. Jos yksi tai useampi tehtävä on samanaikaisesti odottamassa semaforilla varattua resurssia, valitsee järjestelmä odottavista tehtävistä tärkeimmän (eli prioriteetiltaan korkeimman) suorittamaan seuraavan *DOWN*-operaation. Tällöin semaforin arvo on heti *UP*-operaation jälkeen edelleen nolla, mutta semaforia odottavien tehtävien määrä on vähentynyt yhdellä. Semafori, joka voi saada vain arvot nolla tai yksi (eli *true* tai *false*) on binääri- eli mutex-semafori, jolla voidaan toteuttaa yhden resurssin *mutual exclusion* –synkronointi. [Tan92]

3.5.4 Deadlock

Kahdella tehtävällä sanotaan olevan resurssikonflikti, jos ne yrittävät varata samaa resurssia käyttöönsä yhtä aikaa. Jos tehtävän pyytämä resurssi ei ole vapaana, skeduleri estää automaattisesti tehtävän suorittamisen ja tehtävä siirretään pois suoritukseen valmiina olevien tehtävien listalta odottamaan resurssin vapautumista. Tällöin resurssin sanotaan olevan varattuna (engl. *blocked*). Tehtävä pysyy *waiting*–tilassa kunnes resurssi vapautuu. Tämän jälkeen tehtävä siirtyy takaisin *ready* –tilassa olevien tehtävien jonoon ja se suoritetaan aikataulun mukaan [Dou99], [Liu00].

Kun resurssit varataan epäpre-emptiivisesti, voi korkeamman prioriteetin tehtävä tulla estetyksi matalamman prioriteetin tehtävän toimesta. Tällöin on vaarana, että kiireellisen tehtävän suoritus myöhästyy vähemmän tärkeän tehtävän kustannuksella. Tilannetta kutsutaan prioriteetin inversioksi. Huono resurssien hallinta voi aiheuttaa *deadlock*-tilanteen, jossa ohjelman suoritus estyy kokonaan. *Deadlock*-tilanteessa ovat seuraavat neljä ehtoa voimassa [Bur90]:

1. **mutual exclusion**: vain yhden tehtävän sallitaan käyttävän resurssia kerrallaan (resurssi on jakamaton).

2. **hold and wait**: on olemassa tehtäviä, jotka pitävät resursseja varattuna samaan aikaan, kun odottavat toisia resursseja vapautuvan.

3. **no pre-emption**: tehtävää ei voi pakottaa vapauttamaan resurssia.

4. **circular wait**: on olemassa tehtävistä muodostuva suljettu rengas, jossa jokainen tehtävä pitää sellaista resurssia varattuna, jota seuraava tehtävä pyytää.

Esimerkkinä tästä on tilanne, jossa kaksi tehtävää pyytävät resursseja X ja Y käyttöönsä. Järjestelmä on *deadlock*-tilanteessa, jos toinen tehtävä on saanut X:n käyttöönsä ja samanaikaisesti pyytää Y:tä, kun taas toinen tehtävä on jo varannut Y:n, mutta pyytää X:ää käyttöönsä.

3.5.5 Deadlock-tilanteiden välttäminen

Resurssikonfliktien ja *deadlock*–tilanteiden hallitsemiseksi on olemassa menetelmiä, joilla ne voidaan hallita. Yksinkertaisimmin *deadlock*–tilanteet voidaan välttää non-pre-emptiivisien kriittisten sektioiden käytöllä. Tämä tarkoittaa, että resurssin saatuaan tehtävän prioriteetti nousee kaikkien muiden tehtävien prioriteettien yläpuolelle. Koska resurssin saanutta tehtävää ei voida koskaan keskeyttää, ei *deadlock*-tilannettakaan voi syntyä. Menettelytapaa kutsutaan nimellä *NonPre-emptive Critical Section* (NPCS).

Muita resurssiensaantiprotokollia ovat mm. prioriteetinperintäprotokolla (*Priority-Inheritance Protocol*) ja prioriteetinrajoitusprotokolla (*Priority Ceiling Protocol*). Priori-

teetinperintä ei poissulje *deadlock*–tilanteita, vaan se pitää hoitaa jollain muulla menetelmällä. Se kuitenkin estää tehtävien joutumista blokatuksi hallitsemattoman pitkäksi ajaksi. Prioriteetinrajoitusprotokolla on laajennus prioriteetinperintäprotokollasta ja sillä voidaan estää *deadlock*-tilanteen syntyminen. Tämä edellyttää, että tehtävien prioriteetit ovat kiinteitä ja tehtävien tarvitsemat resurssit tiedetään etukäteen. Resurssienhallintaprotokollien toiminta on kuvattu tarkemmin lähteissä[Bur90, Liu00].

3.6 Reaaliaikakäyttöjärjestelmät

Monet reaaliaikaohjelmistot toteutetaan nykyään reaaliaikakäyttöjärjestelmien (RTOS) päälle. Reaaliaikakäyttöjärjestelmiä on markkinoilla runsas ja monipuolinen valikoima (mm. [Pro01]). Hyvin pienet ja erityisesti aikakriittiset sulautetut reaaliaikajärjestelmät saatetaan toteuttaa myös ilman reaaliaikakäyttöjärjestelmää, jos halutaan tarkasti kontrolloida sitä kuinka ohjelmiston resurssit käytetään. Keskikokoisten ja suurten reaaliaikajärjestelmien toteuttamista RTOS:n käyttö kuitenkin yksinkertaistaa tarjoamalla laitteistorajapinnan, menetelmiä rinnakkaisuuden hallintaan sekä resurssien- ja muistinhallintaan. Reaaliaikakäyttöjärjestelmät sisältävät suurelta osin samoja toimintoja kuin tavallisetkin käyttöjärjestelmät [Dou99]:

- sovellusten ja laitteiston välisen rajapinnan hallinta
- tehtävien skedulointi
- muistinhallinta
- yleisten palvelujen tarjoaminen, sisältäen I/O-palvelut standardilaitteisiin, kuten näppäimistö, video tai nestekidenäyttö, sekä osoitinlaitteisiin ja tulostimiin.

Olennaisimpia eroja tavanomaisiin käyttöjärjestelmiin ovat [Dou99]:

- skaalattavuus
- skedulointimenetelmät
- resurssienhallintaprotokollat
- tuki sulautetulle käyttöympäristölle (ei tarvita massamuisteja)

3.6.1 Skaalattavuus

Skaalattavuudella tarkoitetaan sitä, että käyttöjärjestelmään voidaan sisällyttää täsmälleen sovelluksen tarvitsemat ominaisuudet. Käyttöjärjestelmän ydin (engl. *kernel*) sisältää käyttöjärjestelmän oleellisimmat ominaisuudet. Ytimen tehtäviä on hoitaa mm. tehtävien skedulointi, muistinhallinta, resurssienhallinta, keskeytysten käsittely sekä tehtävien välinen kommunikointi. Muita ominaisuuksia voidaan lisätä tarpeen mukaan. Tyypillisiä valinnaisia palveluja ovat mm. tiedostopalvelut, laiteiden I/O-palvelut, verkkopalvelut, ylätason kommunikointiprotokollat sekä ohjelmistojen debuggaustoiminnot. Käyttöjärjestelmään voidaan lisätä esimerkiksi kehitystyön ajaksi tietoverkko-ominaisuuksia, jolla voidaan nopeuttaa kehitystyötä ja poistaa ne resurssien säästämiseksi ohjelman lopullisesta versiosta [VxW99a]. Palvelujen vapaavalintaisuus helpottaa reaaliaikajärjestelmien käyttöä yhtä hyvin pienissä yksiprosessorijärjestelmissä kuin suuremmissa hajautetuissa järjestelmissä. [Dou99, Liu00]

3.6.2 Skedulointimenetelmät

Tavanomaiset käyttöjärjestelmät kohtelevat yleensä kaikkia tehtäviä tasa-arvoisesti, jolloin kaikki tehtävät saavat varmasti suoritusaikaa. Tällöin skeduleri ei pysty keskeyttämään suorituksessa olevaa tehtävää ennen kuin se saadaan valmiiksi. Ongelmaksi voi kuitenkin muodostua se, että yksi virheellisesti toimiva tehtävä voi estää muiden tehtävien suorituksen kokonaan pitämällä suoritusvuoron itsellään. Reaaliaikakäyttöjärjestelmät sisältävät yleensä prioriteettipohjaisen pre-emptiivisen skedulerin. Tällöin korkeamman prioriteetin tehtävät ohittavat aina alemman prioriteetin tehtävät niiden siirtyessä valmiiden tehtävien

jonoon. Tällä varmistetaan, että aikarajoitteita pystytään noudattamaan mahdollisimman hyvin.[Dou99]

3.6.3 Tehtävien väliset synkronointi- ja kommunikointimenetelmät

Reaaliaikakäyttöjärjestelmät sisältävät myös valmiita menetelmiä tehtävien väliseen kommunikointiin. Esimerkkinä voidaan mainita mm. VxWorks-käyttöjärjestelmässä olevia menetelmiä [VxW00]:

- jaetut tietorakenteet
- semaforit
- sanomajonot ja –putket
- soketit (engl. *socket*)
- RPC-kutsut
- signaalit

Lisäksi hajautettuja järjestelmiä varten VxWorks-käyttöjärjestelmää voi laajentaa sisältämään eri prosessorikortteilla toimivien tehtävien väliseen kommunikointiin tarkoitettuja jaettuja semaforeja, sanomajonoja, muistialueita ja nimitietokannan.

3.6.4 Laitteistoriippumattomuus

Yksi suurimpia hyötyjä reaaliaikajärjestelmien käytöstä on niiden tarjoama laitteistorajapinta. Sen avulla voidaan piilottaa laitteistoyksityiskohdat sulautetulta ohjelmistolta. Tärkeimpiä ovat tuki I/O-laitteille, muistinhallintayksiköille, reaaliaikakelloille, verkkopalveluille sekä kommunikointiväylille. Tämä helpottaa sovellusten siirrettävyyttä alustalta toiselle ja toisaalta käyttöjärjestelmän tarjoamat peruskirjastot antavat rungon kehittää helpommin useita erilaisia sovelluksia samalle alustalle.

3.7 Reaaliaikasovellus

Kuten edellisistä kappaleista voi huomata, on suuri osa reaaliaikajärjestelmän komponenteista valmiiksi toteutettuna reaaliaikakäyttöjärjestelmissä. Reaaliaikajärjestelmän palveluita ohjaamaan tarvitaan kuitenkin reaaliaikasovellus, joka määrittelee mitä asioita tehtävien tulee suorittaa, mitkä ovat ehdot tehtävien suorittamiselle, mikä on tehtävien tärkeysjärjestys ja mitä resursseja tehtävät käyttävät.

Edellisen toteuttamiseksi reaaliaikajärjestelmään tulee toteuttaa prosessit, jotka sisältävät tarvittavat funktiot, joilla tuotetaan vasteita saapuneisiin herätteisiin. Sovelluksen tarvitsee alustaa käyttöjärjestelmä vaadittavaan toimintatilaan käynnistyksen yhteydessä määrittelemällä mm. priorisointimenetelmät sekä luomalla prosessit ja niiden tarvitsemat resurssit. Prosesseille määritellään prioriteetit, joiden avulla käyttöjärjestelmä tietää, mikä tehtävä on kiireellisempi kuin toinen. Sovellus määrittelee myös kommunikointi- ja synkronointipisteet sekä menetelmät, joilla kommunikointi suoritetaan (esim. synkronisesti tai asynkronisesti). Herätteitä sovellus saa käyttöjärjestelmältä. Satunnaiset herätteet saapuvat esim. I/O-laitteilta ja periodiset sovelluksen varaamilta käyttöjärjestelmän ajastimilta. Sovellus määrittelee myös kuinka virhetilanteissa toimitaan.

4 Oliokeskeisen ohjelmistokehittämisen perusteet

Ohjelmistojen kehitystä voidaan lähestyä monilla eri tavoin. Nykyaikana yleistynyt tapa on oliokeskeinen ohjelmistojen kehittäminen, jonka ideat ovat syntyneet 1960-luvun lopulla. Oliokeskeisen ajattelutavan yleistyminen kesti kuitenkin melko kauan ja vasta 1990-luvulla siitä tuli varteenotettava kilpailija ns. perinteisille menetelmille. Tässä luvussa esitellään oliokeskeisen ohjelmistokehittämiseen liittyviä käsitteitä ja periaatteita. Luku perustuu pääasiassa lähteisiin [Pre00, Hie01, Jac92, Dou99, Lee97].

4.1 Oliokeskeisyyden tavoitteet

Oliokeskeinen ajattelutapa perustuu ajatukseen siitä, että elämme olioiden maailmassa. Oliot ovat erilaisia asioita luonnossa tai ihmisen rakentamassa maailmassa, jossa päivittäin toimimme. Sen vuoksi luonnollinen tapa lähestyä ohjelmiston kehitysongelmaa on ajatella ohjelmisto joukoksi olioita, jotka toimivat keskenään vuorovaikutuksessa tuottaakseen palveluita käyttäjilleen.

Oliokeskeisellä lähestymistavalla on useita etuja ns. perinteisiin lähestymistapoihin verrattuna. Perinteinen proseduraalinen ohjelmointitapa keskittyy tiedon prosessointiin ja laskennan suorittamiseen vaadittaviin algoritmeihin. Alkuperäinen ns. proseduraalisen ohjelmoinnin periaate on kuvattu lähteessä [Str95] seuraavasti:

”Decide which procedures you want, use the best algorithms you can find”

Oliokeskeinen ajattelutapa keskittyy enemmän tiedon kapselointiin ja yleistämiseen. Periaate kuvataan seuraavalla tavalla [Str95]:

”Decide which classes you want;
provide a full set of operations for each class;
make commonality explicit by using inheritance.

Oliokeskeisesti kehitetty malli on usein helppo ymmärtää, koska ihminen yleensä hahmottaa ympäristönsä joukkona olioita, jolloin oliokeskeisyys pienentää semanttista eroa todel-

lisuuden ja ohjelmistomallin välillä. Oliokeskeisyys helpottaa myös ohjelmistokomponenttien uudelleenkäyttöä. Uudelleenkäytettävyys on usein hyödyllinen piirre, koska se mahdollistaa nopeamman ohjelmistokehityksen ja korkeamman laadun. Hyväksi testattujen ja luotettavien komponenttien uudelleen tekeminen ei ole yleensä tarpeellista tai hyödyllistä. Myös ohjelmistojen ylläpidettävyys paranee, koska lokaalien muutoksien tekeminen ja virheiden jäljittäminen on helpompaa, sillä ne keskittyvät oliokeskeisissä ratkaisuissa yksittäisiin kokonaisuuksiin (eli olioihin). Oliot helpottavat myös skaalattavuutta ja mukautettavuutta, koska oliot ovat itsenäisiä ohjelmistokomponentteja, joilla on helppo laajentaa ja karsia järjestelmiä.

Ohjelmointikielissä oliokäsite esiintyi ensimmäisen kerran Simula-67:ssä [Rin00]. Nykypäivänä olio-ohjelmointikieliä on useita kuten mm. Ada95, Java, C++, Eiffel, Smalltalk, Object-Pascal [Ell94, Bec97, Pre00].

4.2 Oliot ja luokat

Olio on todellinen tai käsitteellinen kokonaisuus, jolla on identiteetti. Sen tehtävä on kapseloida sekä data että funktiot, jotka käsittelevät dataa. Olio voi esittää mitä tahansa reaalimaailman asiaa kuten eläintä, autoa, lentokonetta, sensoria tai moottoria. Toisaalta olio voi olla myös jokin käsitteellinen kokonaisuus, kuten pankkitili, avioliitto tai lista. Jokaisella oliolla on määrätty identiteetti, attribuutteja (data), tiloja (muisti), käytösmalli (operaatiot ja metodit) sekä vastuut. Taulukossa 4.1 on esimerkki reaalimaailman oliosta ja taulukossa 4.2 oliosta, joka on käsitteellinen kokonaisuus.

Attributes	**Behavior**	**State**	**Identity**	**Responsibilty**
• Linear value • Rate of change (RoC)	• Acquire • Report • Reset • Zero • Enable	• Last value • Last RoC	Instance for robot arm joint	Provide information for the precise location of the end of the robot arm in absolute space coordination

Taulukko 4.1. Sensoriolion ominaisuudet [Dou99].

Attributes	Behavior	State	Identity	Responsibilty
• Retry count • Max retries • Time to retry	• Send • Receive • Notify sender	• Idle • Sending • Waiting	Transaction for msg 0x1199876	• Implements reliable transmission for the sender • Resends message after fixed period of time has elapsed until some max retry count is exceeded • Notifies the sender if unable to complete transaction

Taulukko 4.2. Sanomanvälitysolion ominaisuudet [Dou99].

Luokka tarkoittaa määritystä tai mallia, josta voidaan luoda uusia olioita. Se kuvaa joukon yhteisiä piirteitä usealle oliolle. Ohjelmointikielissä luokka määrittelee yleisen rakenteen, tilat ja rajapinnat [Rin00]. Lähteessä [Jac92] luokka määritellään seuraavasti:

*”A **class** represents a template for several objects and how these objects are structured internally. Objects of the same class have the same definition both for their operations and for their information structure.”*

Oliokeskeisissä järjestelmissä jokainen olio kuuluu johonkin luokkaan. Kun luokasta luodaan uusi olio puhutaan instantioinnista [Rin00]. Monesti olioita kutsutaankin luokkien ilmentymiksi tai instansseiksi.

4.2.1 Attribuutit

Olioterminologiassa attribuutit tarkoittavat dataa, joka on kapseloituna olion sisään. Attribuutit kuvaavat luokan rakennetta ja niihin sijoitetut arvot tekevät olioista yksilöllisiä. Luokka kapseloi attribuutit sisäänsä ja tarjoaa aliohjelmarajapinnan attribuuttien käsittelemiseen. Tällöin tieto on piilotettuna luokan sisään, jota luokan proseduurit ”valvovat”. Jos attribuutteja tarkastellaan teknisemmin, niin niiden voidaan huomata tarkoittavan ohjelmointikielissä tavallista tietotyyppiä tai luokkaa. Joissakin ohjelmointikielissä kaikki tietotyypit ovat olioita.

4.2.2 Olion käyttäytyminen

Kuten edellä on todettu, olioiden tehtävä on kapseloida dataa. Kapseloinnin lisäksi olioiden on tarjottava algoritmit datan käsittelemiseen. Näitä algoritmeja kutsutaan joko metodeiksi, operaatioiksi tai palveluiksi. Jokainen olion kapseloima operaatio kuvaa omalla tavallaan olion käyttäytymistä. Operaatiot voivat olla primitiivisiä (kuten *create, add, delete*) tai monimutkaisempia, kuten raportin kokoaminen useamman olion ylläpitämistä tiedoista. Jos operaatiot ovat liian monimutkaisia, voi olla järkevää pyrkiä muodostamaan uusia olioita, sillä liian monimutkaisia olioita on pyrittävä välttämään [Jac92]. Olion julkisen rajapinnan muodostaa joukko operaatioita, joka kertoo käyttäjälle mitä palveluita olio tarjoaa.

4.2.3 Olioiden väliset sanomat

Olio käynnistää operaatioita vastaanottamalla herätteitä (engl. *stimulus*) ja ne kommunikoivat keskenään välittämällä sanomia (engl. *message*). Sanomien avulla oliot kertovat toisilleen, mitä palveluita ne haluavat käyttää. Olioiden väliset sanomat voivat olla esimerkiksi tavallisia aliohjelmakutsuja tai käyttöjärjestelmäviestejä. Vastaanotettuaan sanoman olio toteuttaa palvelupyynnön suorittamalla tarvittavat operaatiot. Palvelupyyntö on suoritettu, kun operaatiot on suoritettu ja kontrolli palautettu palvelunpyytäjälle. Olio voi saada sanoman mukana myös parametreja, joita se tarvitsee palveluiden suorittamiseksi. Sanomat sitovat oliokeskeisen järjestelmän yhteen ja antavat kuvan järjestelmän toiminnasta kokonaisuutena.

4.3 Kapselointi

Aiemmin on todettu olion sisältävän dataa ja operaatioita, joilla käsitellään dataa. Tätä sanotaan kapseloinniksi (engl. *encapsulation*). Se on yksi oleellinen piirre oliokeskeiselle järjestelmälle. Kapseloinnin avulla pyritään saavuttamaan merkittävää hyötyä, kuten lähteessä [Pre00] luetellaan:

- Datan ja aliohjelmien sisäinen toteutus ja yksityiskohdat on piilotettu ulkopuoliselta maailmalta (engl. *information hiding*). Tämä ehkäisee sivuvaikutuksia, kun ohjelmaan tehdään muutoksia.

- Tietorakenteet ja operaatiot, joilla tietorakenteita käsitellään, on sulautettu yhteen nimettyyn kokonaisuuteen eli luokkaan. Tämä mahdollistaa komponenttien uudelleenkäytön.

- Rajapinnat kapseloitujen olioiden välillä yksinkertaistuvat. Sanoman lähettävän olion ei tarvitse olla tietoinen vastaanottavan olion sisäisestä rakenteesta. Tällöin rajapinnat yksinkertaistuvat ja järjestelmän sisäiset kytkennät vähenevät.

4.4 Periyttäminen

Periyttäminen (engl. *inheritance*) on merkittävimpiä eroja tavanomaisten ja oliokeskeisten ohjelmistojen välillä. Periytymisessä on kyse suhteesta, jossa yksi luokka pohjautuu toiseen luokkaan ja perii sen ominaisuudet. Lähde [Rin00] tarjoaa hyvän yksinkertaisen selityksen periytymiselle:

”Uuden luokan muodostuminen olemassa olevan luokan pohjalta niin, että uusi luokka sisältää kaikki toisen luokan ominaisuudet.”

Yliluokkia (tai kantaluokkia) (engl. *superclass, base class, ancestor*) voidaan yleistää samoja piirteitä omaavista luokista, joita kutsutaan aliluokiksi (engl. *subclass, descendant, derived class*). Yhteisten piirteiden lisäksi aliluokat voivat sisältää uusia ominaisuuksia. Kun yliluokassa oleviin yhteisiin piirteisiin tehdään muutoksia tai lisäyksiä, periytyvät ne välittömästi myös aliluokkien ominaisuuksiksi. Luokkahierarkia toimii tällöin mekanismina, jossa ylätasolla tehdyt muutokset etenevät automaattisesti koko järjestelmän läpi. Tällöin vältetään paljon työtä suunnittelun, testaamisen sekä vianetsinnän osalta ja myös uudelleenkäytettävyysperiaate toteutuu.

Jos periyttäminen ei onnistu suoraan ja periytymishierarkian rakentaminen uudelleen on liian monimutkaista, voidaan käyttää apuna ylimäärittelyä (engl. *overriding*). Ylimäärittely tarkoittaa sitä, että attribuutit ja operaatiot peritään normaaliin tapaan, mutta sen jälkeen niitä modifioidaan uuden luokan tarpeiden mukaan. Ylimäärittely on helppo ja joustava tapa muokata olemassa olevia luokkia, mutta se vaikeuttaa luokkahierarkian ymmärtämistä, koska saman nimisillä operaatioilla voi olla erilainen semanttinen merkitys eri luokissa.

Lähteessä [Jac92] muistutetaankin, että ylimäärittelyä käytettäessä periytyminen ei ole transitiivista.

Uuden luokan muodostamista periyttämällä ominaisuuksia useammasta kuin yhdestä yliluokasta sanotaan moniperinnäksi (engl. *multiple inheritance*). Tällöin aliluokalla on useita yliluokkia. Kuten ylimäärittely, myös moniperintä on kyseenalaistettu mm. lähteissä [Pre00, Jac92], koska se monimutkaistaa luokkahierarkiaa. Moniperintä voidaan usein välttää käyttämällä luokan jäseninä muita luokkia [Lap96] eli koostaa luokkia.

4.5 Monimuotoisuus

Monimuotoisuudella (engl. *polymorphism*) tarkoitetaan tilannetta, jossa olio voi lähettää herätteen toiselle oliolle, tietämättä tarkalleen mihin luokkaan vastaanottava olio kuuluu. Heräte voidaan tällöin tulkita eri tavoin riippuen olion luokasta. Tästä seuraa, että vastaanottava olio tekee päätöksen siitä, miten heräte tulkitaan. Käytännössä se tarkoittaa, että yksi operaatio on toteutettu usealla eri tavalla eri luokkiin. Esimerkkinä monimuotoisuudesta voidaan esittää Kaavio-luokka [Pre00]. Yliluokkana toimivalla Kaavio-luokalla voi olla aliluokkana esimerkiksi pylväs-, viiva-, ympyrä ja pistekaavioluokat. Käyttämällä operaatioiden ylikuormittamista (engl. *overloading*), jokainen aliluokka voi määritellä oman operaation draw(). Tällöin olio voi lähettää draw()-funktiokutsun mille tahansa aliluokista luodulle oliolle. Kutsun vastaanottava luokka suorittaa draw()-kutsun ja piirtää luokkaan kuuluvan kaavion. Jos sovellukseen tarvitsee vaihtaa uusi kaaviotyyppi, voidaan se lisätä ilman, että kaavion piirtoaliohjelmaa kutsuvia luokkia tarvitsee muuttaa, koska draw()-funktion rajapinta pysyy muuttumattomana.

5 Unified Modelling Language, UML

Tässä luvussa esitellään yleispiirteet UML-notaatiosta ja –kaavioista. UML on graafinen kolmannen sukupolven kuvausnotaatio ohjelmistojen oliopohjaiseen visualisointiin, määrittelyyn, rakentamiseen ja dokumentoimiseen. UML ei ole itsessään menetelmä eikä ohjelmointi- tai määrittelykieli. Lähteessä [Rin00] annettu määritelmä on varsin hyvä: UML on **kuvaustapa**, joka sijoittuu ohjelmointikielten ja suunnittelumenetelmien välimaastoon. UML:hän ei esimerkiksi määrittele, mitä kuvaustapoja missäkin ohjelmistonkehitysvaiheessa tulisi käyttää. UML:ssä on pyritty yhdistämään kolmen oliomallinnusmenetelmän, Booch [Boo94], OMT [Rum91] ja OOSE [Jac92], parhaat ominaisuudet yhdeksi mahdollisimman yleiskäyttöiseksi ja kattavaksi notaatioksi. UML:n kehittäminen alkoi lokakuussa 1994 ja ensimmäinen ns. draft-versio 0.8 julkaistiin lokakuussa 1995. Menetelmää kutsuttiin silloin vielä nimellä *Unified Method*. Draft-versio oli lähinnä yhdistelmä Booch- ja OMT-menetelmistä. Syksyllä 1995 aloitettiin OOSE-menetelmän sulauttaminen UML:ään ja kesäkuussa 1996 julkaistiin ensimmäinen varsinainen UML-versio 0.9. 1997 OMG hyväksyi UML:n version 1.1 standardiksi. Tässä luvussa esitettävä notaatio perustuu syyskuussa 2001 julkaistuun versioon 1.4. Luku perustuu pääasiassa lähteisiin [OMG01, Pre00, Dou99].

UML koostuu joukosta kaavioita, jotka on lueteltu seuraavassa (suluissa UML version 1.4 spesifikaatiossa esitetyt englannin kielen nimet) [OMG01]:

- Käyttötapauskaavio (*use case diagram*)
- Luokkakaavio (*class diagram*)
- Käyttäytymiskaaviot (*behavioral diagrams*):
  - Tilakaavio (*state diagram*)
  - Aktiviteettikaavio (*activity diagram*)
  - Vuorovaikutuskaaviot (*interaction diagrams*):
    - Sekvenssikaavio (*sequence diagram)*
    - Yhteistoimintakaavio (*collaboration diagram*)
- Toteutuskaaviot (*implementation diagrams)*:
  - Komponenttikaavio (*component diagram*)
  - Sijoittelukaavio (*deployment diagram*)

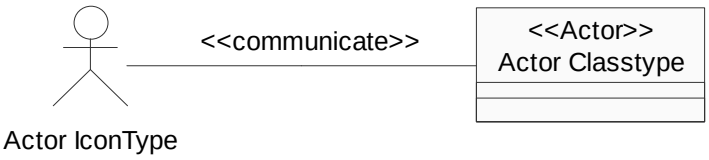
5.1 Käyttötapauskaavio

Käyttötapaus kuvaa funktiota, joka palauttaa havaittavan vasteen käyttäjälle eli aktorille (engl. *actor*) paljastamatta funktion toteutusrakennetta [Dou99]. UML:ssä käyttötapaukset kuvataan aktoreiden ja käyttötapausten avulla käyttötapauskaavioissa.

Aktori on järjestelmän ulkopuolinen olio, joka toimii vuorovaikutuksessa järjestelmän kanssa. Aktori käynnistää käyttötapauksen, johon dokumentoidaan aktorin ja järjestelmän välinen vuorovaikutussekvenssi. Sekvenssiä voidaan kuvata erilaisilla menetelmillä kuten skenaariokaaviolla, aktiviteettikaaviolla tai tilakaaviolla. Aktori voi olla esimerkiksi lentokoneen lentäjä, asiakas kirjastossa tai työnjohtaja yrityksessä. Sulautetuissa järjestelmissä aktori voi olla myös ulkopuolinen laite [Awa96] tai reaaliaikajärjestelmissä

aktori voi olla myös ulkopuolinen laite [Awa96] tai reaaliaikajärjestelmissä ajastin (engl. *timer*) [Gom00], joka lähettää periodisesti herätteitä järjestelmälle. Vaikka ajastimet ovat käytännössä järjestelmän sisäisiä osia, on reaaliaikajärjestelmissä joskus järkevämpää käsitellä niitä loogisesti järjestelmän ulkopuolisina aktoreina, jotka käynnistävät toimintoja. Tosin ajastimien käytöstä aktoreina käyttötapauskaavioissa esiintyy myös vastustavia mielipiteitä kuten lähteessä [Rat02].

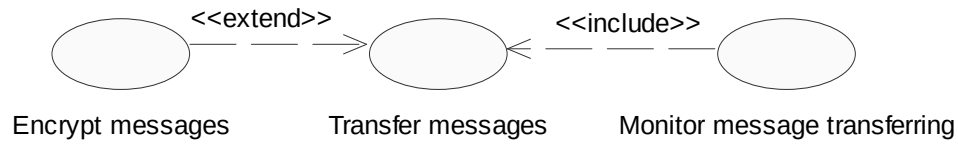
Aktori kuvataan UML:ssä *actor*-stereotyyppinä. *Actor*-stereotyyppi voidaan esittää joko luokkana, johon liitetään stereotyyppi tai standardin mukaisella tikku-ukko-ikonilla (vrt. kuva 5.1). Käyttötapausta kuvataan ellipsillä, johon lisätään käyttötapauksen nimi. Kun käyttötapaus ja aktori kytketään assosiaatiolla toisiinsa, tarkoitetaan tällä sitä, että aktorilla on jokin rooli käyttötapauksen suorituksessa. Jos halutaan selkeyttää käyttötapauskaaviossa rajapintaa ympäristön ja järjestelmän välillä, voidaan käyttötapaukset "paketoida" järjestelmän sisään pakettikomponenteilla.



Kuva 5.1. Käyttötapauskaavion aktorikomponentteja.

Käyttötapauksien välillä voi olla myös *Extend*-, *Include*- ja *Generalization*-tyyppisiä relaatioita. *Include* tarkoittaa, että käyttötapauksen suorittaminen edellyttää myös toisen käyttötapauksen suorittamista. *Extend* tarkoittaa, että määrätyissä tilanteissa toisen käyttötapauksen toiminnallisuudella voidaan laajentaa toista käyttötapausta. *Generalization* riippuvuus käyttötapausten välillä tarkoittaa, että ns. lapsikäyttötapaus on erikoistettu versio vanhemmastaan. Lapsi perii vanhemmaltaan kaiken toiminnallisuuden, assosiaatiot ja voi sisältää uusia ominaisuuksia. Käyttötapauksen kuvaustapa ja esimerkkejä riippuvuussuhteista on kuvassa 5.2. Kuvan *Extend*-riippuvuussuhteella esitetään tilannetta, jossa viestien välittäminen on mahdollista ilman salausta, mutta kuvan mallissa esim. käyttäjä voi valita viestin

salaamisen välitystoiminnon lisäksi, jolloin siitä tulee osa viestinvälitys-käyttötapauksen toimintaa. Kuvan *Include*-riippuvuussuhteella taas ilmaistaan, että monitorointitoiminnon tulee sisältää myös viestinvälitys. Jos viestinvälitys ei ole käytössä, ei välitystä silloin ole mahdollista monitoroidakaan. [OMG01, Pre00]



Kuva 5.2. Esimerkkejä käyttötapauksista ja niiden välisistä riippuvuuksista.

5.2 Luokkakaavio

Luokkakaaviolla esitetään järjestelmän staattista rakennetta. Se kuvaa olemassa olevia asioita (kuten luokat ja tyypit), niiden sisäistä rakennetta ja suhteita muihin asioihin. Luokkakaavio ei kuvaa ajallista informaatiota [OMG01]. Luokkakaaviossa luokka kuvataan laatikkona, jossa luetellaan luokan ominaisuudet, kuten luokan nimi, attribuutit ja julkiset palvelut eli luokan ulkopuolelle näkyvät operaatiot. Kuvassa 5.3 on esimerkki UML-luokkakaaviosta. Luokkakaaviossa kuvataan sensoriluokkaa, joka monitoroi esimerkiksi lämpötilaa ja välittää informaation viestinvälitysolion kautta näyttöluokalle.

Luokkien välisiä yhteyksiä kuvataan niiden välille piirretyillä kaarilla. **Riippuvuussuhde** (engl. *dependency*) tarkoittaa sitä, että luokka tarvitsee toisen luokan olioita suorittaakseen omia toimintojaan. Riippuvuus ilmaistaan katkoviivalla varustetulla nuolella. Jos nuoli osoittaa luokasta A luokkaan B, sanotaan luokan A olevan riippuvainen luokasta B. **Assosiaatio** (engl. *association*) on kahden luokan välille piirretty viiva, jolla kerrotaan luokkien liittyvän ohjelmiston rakenteen kannalta toisiinsa. Assosiaatio on riippuvuutta vahvempi ominaisuus, koska luokat käyttävät toistensa rajapintapalveluita osana oman vastuualueensa toteuttamista [Rin00]. Assosiaatioita voidaan myös tarkentaa lisäämällä niihin esim. assosiaation nimi, luokkien roolinimet ja lukumääräsuhteet (engl. *multiplicity*) (ks. taulukko 5.1). Assosiaatio voi olla yksi- tai kaksisuuntainen. Yksisuuntainen assosiaatio tarkoittaa, että vain palveluita pyytävä luokka tietää palveluita tarjoavan luokan olemassaolosta.

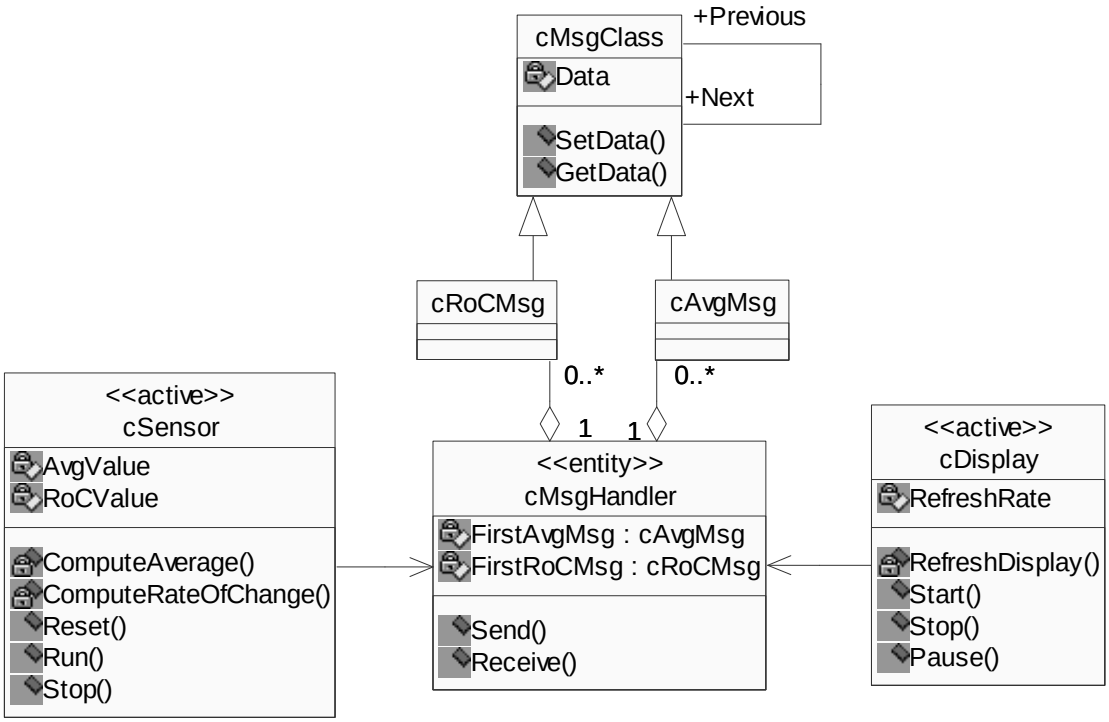
Tämä voidaan ilmaista piirtämällä nuoli sitä luokkaa kohti, jonka ei tarvitse tietää assosiaatiosta. Jos nuolia ei piirretä, on assosiaatio oletuksena kaksisuuntainen.

Multiplicity	Description
1	Exactly one
0..1	Optional
*	Zero or more
1..*	One or more
m..n	Numerically specified, where n and m have numeric values

Taulukko 5.1. Esimerkkejä assosiaatioiden lukumääräsuhteista.

Muodoste ja **kooste** ovat assosiaatioiden erikoistapauksia, joissa määritellään luokkien välisen yhteyden lisäksi niiden välille omistussuhde. Muodoste (engl. *composition*) on vahvempi muoto omistussuhteesta kuin kooste (engl. *aggregation*). Muodoste kuvataan piirtämällä täytetty salmiakkikuvio assosiaatioviivan päähän kiinni siihen luokkaan, jonka osana toinen olio on. Kooste kuvataan samalla tavoin, mutta salmiakkikuvio ei ole täytetty. **Periytyminen** (engl. *inheritance*) kuvataan UML:ssä nuoliviivalla, joka osoittaa aliluokasta yliluokkaan.

Luokkakaavioiden avulla voidaan kuvata myös ns. oliokaavioita (engl. *object diagram*). Oliokaavio kuvaa ilmentymiä, eli olioita ja data-arvoja. Staattinen oliokaavio on luokkakaavion ilmentymä, joka kuvaa luokkakaavion tilaa jollain tietyllä ajanhetkellä. [OMG01]



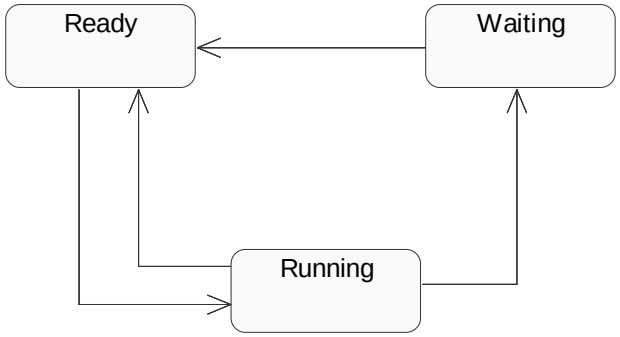
Kuva 5.3. Esimerkki UML-luokkakaaviosta.

5.3 Tilakaavio

UML-tilakaavio tarkoittaa perinteistä tilakonetta esittävää kaaviota. Kuvassa 5.4 on esimerkki UML-tilakaaviosta, jossa kuvataan kappaleessa 3.3.5 esitettyjä reaaliaikajärjestelmän tehtävien tiloja ja tilasiirtymiä. Tilakaavioita käytetäänkin kuvaamaan olioiden dynaamista käyttäytymistä. UML-notaatiossa tilaa kuvataan kulmista pyöristetyillä laatikoilla. Tiloille annetaan yksilölliset nimet. Tilakoneen suoritus alkaa alkutilasta ja päättyy lopputilaan. Alkutilaa kuvataan UML:ssä mustalla ympyrällä ja lopputilaa mustalla ympyrällä, jossa on valkoinen reunus. Tilakaavio voi sisältää myös sisäkkäisiä tiloja, jolloin alitilat (engl. *substate*) piirretään toisen tilalaatikon eli ylitilan (engl. *superstate*) sisään. Tilojen välillä on tilasiirtymiä, joita kuvataan nuolilla. Nuoli osoittaa aina siihen tilaan, johon siirrytään. Tilasiirtymiä voidaan kuvata tekstuaalisesti seuraavassa muodossa:

event-name ’(’ comma-separated-parameter-list ’)’ ’[’ guard-condition ’]’ ’/’ action-expression,

missä event-name tarkoittaa tilasiirtymän aiheuttavaa tapahtumaa, johon voi olla liitetty parametreja. *guard-condition* on ehtolause, joka täytyy olla voimassa, jotta tapahtumasta aiheutuva tilasiirtymä voi laueta. *action-expression* on operaatio, joka suoritetaan tilasiirtymän yhteydessä. Tilaan voidaan liittää myös operaatioita, joita suoritetaan tilan ollessa voimassa. *Entry*-operaatio suoritetaan saavuttaessa tilaan. *Do*-operaatio on aktiviteetti, jota suoritetaan niin kauan kuin tila on voimassa tai kunnes operaatio on suoritettu loppuun. Operaation suorituksen valmistuminen voi generoida myös herätteen, joka aiheuttaa seuraavan tilasiirtymän. *Exit*-operaatio suoritetaan tilasta poistuttaessa. [OMG01]

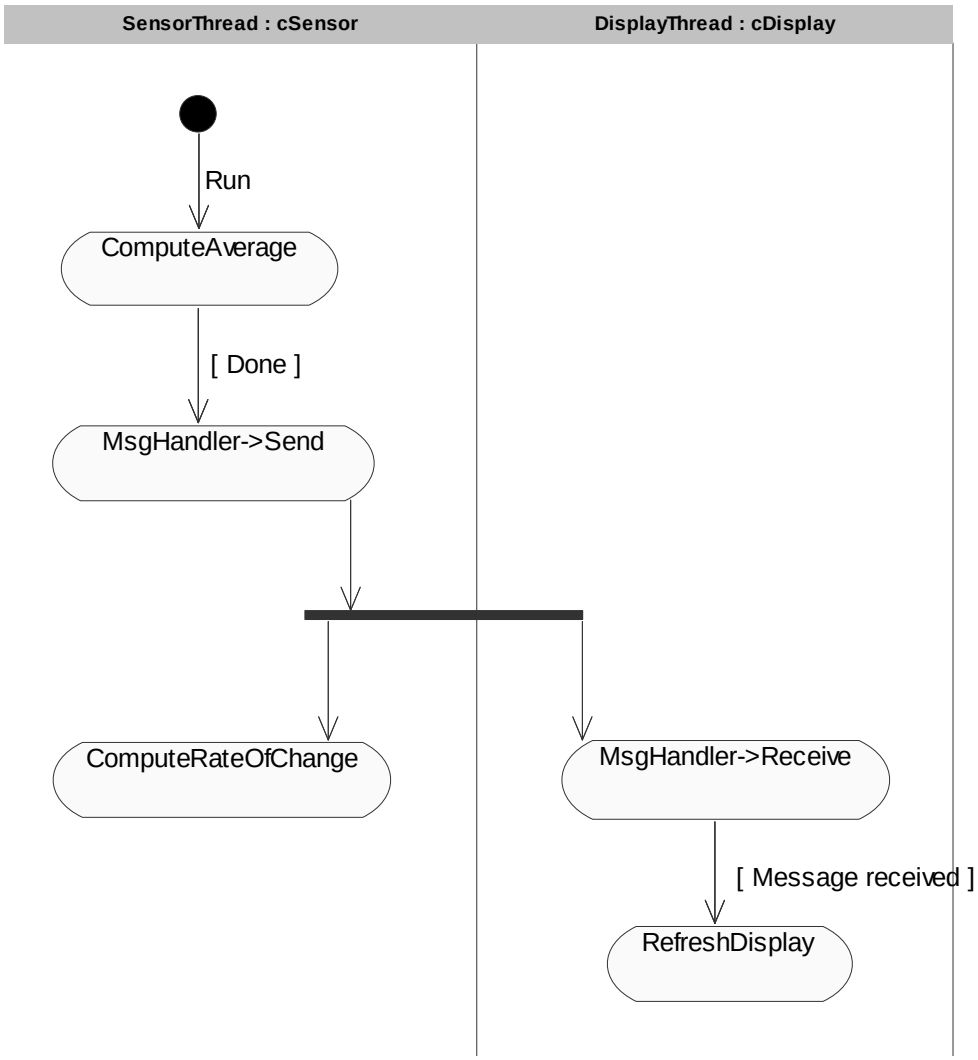


Kuva 5.4. Tehtäväsäikeen tilat UML-kaaviolla kuvattuna.

5.4 Aktiviteettikaavio

Aktiviteettikaavio on muunnelma tilakaaviosta. Sen tarkoituksena on kuvata järjestelmän sisäisten prosessien ja säikeiden proseduraalista etenemistä (ks. kuva 5.5). Aktiviteettikaavio kuvaa toimintojen (engl. *action*) ja subaktiviteettien (engl. *subactivity*) suorittamista. Tilasiirtymät laukeavat toimintojen ja subaktiviteettien suorituksen valmistumisesta, eikä asynkronisia ulkoisia herätteitä odoteta kuten tilakaavioissa. Kaavio koostuu pääasiassa toiminto- ja subaktiviteettitiloista sekä tilasiirtymistä. Toimintoja ja subaktiviteetteja kuvataan pyöristetyillä neliöillä ja siirtymiä nuolilla. Kaavio voi sisältää myös päätöspisteitä (engl. *decision point),* synkronointitiloja (engl. *synchstate*) ja kaistoja (engl. *swimlane*). Päätöspisteisiin liitetään ehtoja, joilla määrätään mihin ”suuntaan” haaraudutaan. Päätöspisteitä kuvataan vinoneliöillä, joihin liittyvät ehtolauseet esitetään hakasulkeissa. Haarau-

tuminen voi tapahtua kuinka moneen suuntaan tahansa, mutta vähintään yksi haara on valittava tai järjestelmä on *deadlock*-tilassa. Kaistoja voidaan käyttää rinnakkaisten olioiden tai säikeiden kuvaamiseen. Niihin ei liity mitään erityistä semantiikkaa eivätkä ne mitenkään rajoita siirtymäpolkujen etenemistä, sillä siirtymät voivat kulkea vapaasti kaistalta toiselle. Synkronointitiloja kuvataan lyhyellä suoralla viivalla. Niillä voidaan kuvata rinnakkain suoritettavien toimintojen synkronisointia. Aktiviteettikaaviolla voidaankin luonnollisemmin kuvata rinnakkaisuutta kuin tilakaavioilla, mikä on tärkeää esimerkiksi reaaliaikajärjestelmiä mallinnettaessa.[OMG01, Dou99]

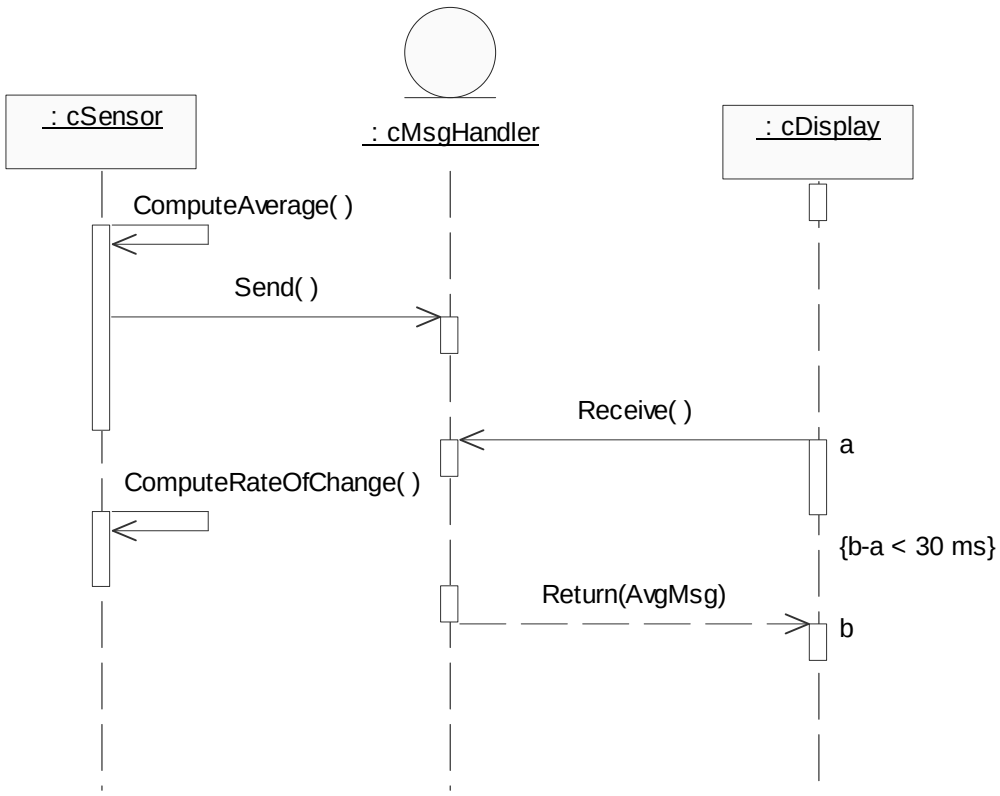


Kuva 5.5. Esimerkki aktiviteettikaavion käytöstä rinnakkaisten tehtävien kuvaamisessa.

5.5 Sekvenssikaavio

UML tarjoaa olioiden välisten vuorovaikutussekvenssien kuvaamiseen kaksi erilaista kaaviota: sekvenssi- ja yhteistoimintakaaviot. Sekvenssikaaviolla voidaan kuvata olioiden välistä vuorovaikutusta ajallisessa järjestyksessä (ks. kuva 5.7). Sekvenssikaaviolla on kaksi dimensiota, joista pystysuunta kuvaa yleensä aikaa (UML-standardin mukaan dimensiot ovat sinänsä vaihdannaisia keskenään, mikäli kuvaustyökalu sen vain mahdollistaa). Aika etenee alaspäin. Yleensä vain tapahtumien ajallisella järjestyksellä on merkitystä, mutta reaaliaikajärjestelmiä kuvattaessa voidaan aika-akseliin liittää myös mittayksiköt. Horisontaalinen suunta kuvaa normaalisti vuorovaikutuksessa toimivia olioita. Sekvenssikaavioissa kuvataan nimenomaan olioita eikä luokkia. Olioita ei tarvitse yksilöidä, vaan ne instantioidaan luokista lisäämällä kaksoispiste luokan nimen eteen, jolloin tiedetään, että kyseessä on luokan ilmentymä. Olioiden horisontaalisella järjestyksellä ei ole erityistä merkitystä muuten kuin kaavion selkeyden kannalta.

Olioiden elinkaarta kuvataan pystysuuntaisella katkoviivalla. Katkoviiva muutetaan paksummaksi ”palkiksi”, kun olio on aktiivinen. Samalla palkki kuvaa myös toiminnon suorittamiseen kuluvaa aikaa. Aktori kuvataan yleensä kaavion vasempaan reunaan. Oliot kommunikoivat keskenään lähettämällä sanomia toisilleen. Sanomat kuvataan nuolilla, jotka alkavat viestin lähettäjästä ja loppuvat vastaanottajaan. Sanomiin voi liittää myös reaaliaikajärjestelmien aikarajoitteita käyttämällä esim. matemaattisia ehtolausekkeita. UML mahdollistaa erityyppisten sanomien kuvaamisen stereotyyppien avulla (ks. kpl 5.8.4). Aliohjelmakutsua kuvataan täytetyllä nuolen kärjellä. Se tarkoittaa, että kutsuvan olion suoritus ei jatku ennen kuin palvelupyyntö on suoritettu loppuun. Tavallinen nuolen kärki kuvaa asynkronista kommunikointia, jolloin olio lähettää toiselle oliolle viestin tai herätteen, mutta ei jää odottamaan toiminnon loppuun suorittamista. Katkoviivalla varustettu tavallinen nuoli kuvaa aliohjelmasta paluuta. [OMG01, Dou99, Qua98]



Kuva 5.7. Malli sekvenssikaavion käytöstä säikeiden ja aikarajoitteiden kuvaamiseen.

5.6 Yhteistoimintakaavio

Yhteistoimintakaavio on vaihtoehtoinen tapa kuvata skenaarioita. Se kuvaa samaa asiaa eri perspektiivistä kuin sekvenssikaavio. Sekvenssikaavio kuvaa selkeästi sanomien välistä järjestystä olioiden välillä, kun taas yhteistoimintakaaviosta on helpompi hahmottaa se, miten oliot ovat sidoksissa keskenään. Myöskään yhteistoimintakaavio ei kuvaa luokkien vaan niiden ilmentymien eli olioiden toimintaa. Oliot kuvataan laatikoina ja linkit esitetään piirtämällä viiva vuorovaikutuksessa olevien olioiden välille. Olioiden välillä kulkevien sanomien nimet kirjoitetaan tekstimuodossa ja niihin lisätään nuoli kuvamaan suuntaa, johon sanoma kulkee. Koska sanomien järjestystä on yhteistoimintakaaviossa vaikeampi hahmottaa kuin sekvenssikaaviossa, voidaan sanomat numeroida niiden ajallisen järjestyksen mukaan. [OMG01, Gom00, Qua98]

5.7 Toteutuskaaviot

UML sisältää kaksi ohjelmiston toteutusaikaisen rakenteen kuvaamiseen tarkoitettua kaaviotyyppiä. Niillä kuvataan sekä lähdekoodin että ajonaikaisen toteutuksen rakennetta. Komponenttikaavio kuvaa koodinrakennetta ja sijoittelukaavio ajonaikaisen järjestelmän rakennetta. Toteutuskaavioiden avulla voidaan kuvata esimerkiksi reaaliaikajärjestelmän toteutusympäristöä ja fyysisiä yhteyksiä eri järjestelmäosien välillä. Kahdessa seuraavassa kappaleessa esitellään lyhyesti komponentti- ja sijoittelukaavioita lähteen [Rat97] perusteella.

5.7.1 Komponenttikaavio

Komponenttikaavio kuvaa ohjelmistokomponenttien väliset riippuvuudet. Ohjelmistokomponentteja ovat esimerkiksi lähde- ja binäärikooditiedostot sekä ajonaikaiset tiedostot. Eri komponentit esiintyvät ohjelmiston käännös-, linkitys- sekä ajonaikana. Komponenttikaaviot kuvaavat vain tyyppejä. Jos halutaan kuvata komponenttien instansseja, täytyy käyttää sijoittelukaaviota.

Komponenttikaavion notaatio koostuu komponenteista ja niiden välisistä riippuvuuksista. Riippuvuuksia kuvataan katkoviivalla piirretyillä nuolilla. Komponenttikaavio voi sisältää lisäksi koostesuhteita sekä rajapintoja. Koostesuhteet kuvaavat sitä, kuinka komponentit koostuvat fyysisesti toisista komponenteista, ja rajapinnat palveluita, joita komponenttien osat tarjoavat.

5.7.2 Sijoittelukaavio

Sijoittelukaavio kuvaa järjestelmän ajonaikaista arkkitehtuuria, jolla ohjelmistoa suoritetaan. Sijoittelukaavion keskeisimpiä komponentteja ovat solmut eli järjestelmän fyysiset osat sekä linkit niiden välillä. Solmut sisältävät ohjelmistokomponentteja, prosesseja ja olioita. Ei-ajonaikaisia komponentteja (kuten lähdekooditiedostot) ei sijoittelukaavioissa esiinny, koska ne on jo ”käännetty pois” suoritettavaksi ohjelmaksi. Niitä kuvataan komponenttikaaviolla. Kukin komponentti sijaitsee jossain solmussa (engl. *node*), joka on suoritusaikainen laiteresurssi. Solmut ovat laitteita, joilla on yleensä muistia ja jonkinlainen

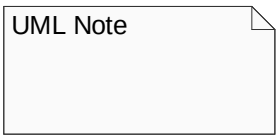
prosessori. Solmut voivat koostua komponenteista ja komponentit olioista. Komponentteja voidaan yhdistää riippuvuussuhteilla kuvaamaan sitä, että jokin komponentti käyttää toisen komponentin palveluita. Luvussa 7 (kuva 7.1) on esimerkki sijoittelukaavion käytöstä.

5.8 Kaavioille yhteisiä mallinnuselementtejä

UML käsittää monia elementtejä, jotka ovat yhteisiä kaikille tai lähes kaikille kaavioille. Seuraavassa esitetään muutamia tällaisia elementtejä, jotka eivät ole tulleet aiemmin esille.

5.8.1 Muistilaput

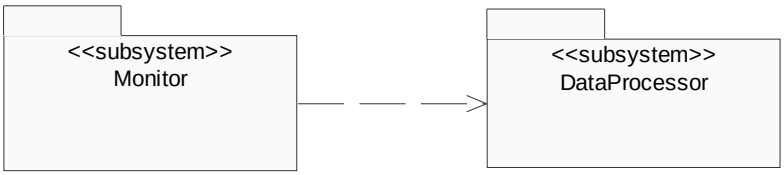
Muistilappu (engl. *note*) on graafinen symboli, jota voidaan käyttää esittämään sanallista informaatiota UML-kaavioista sekä niiden elementeistä (ks. kuva 5.8). Muistilappu voi sisältää esimerkiksi rajoitteita, kommentteja, metodien runkoja tai muuttujien arvoja. Muistilaput voidaan yhdistää yhteen tai useampaan UML-kaavioelementtiin katkoviivoilla.



Kuva 5.8. UML-muistilappu.

5.8.2 Paketit

Paketeilla (engl. *package*) voidaan ryhmitellä mallinnuselementtejä (ks. kuva 5.9). Paketeilla voidaan esimerkiksi reaaliaikajärjestelmä jakaa rinnakkain suunniteltaviin osiin eli alijärjestelmiin. Paketti voi sisältää uusia sisäkkäisiä paketteja tai muita mallinnuselementtejä. UML-standardin mukaan paketteihin voi ryhmitellä mitä tahansa UML-mallinnuselementtejä. Tyypillisesti paketteihin ryhmitellään luokkia ja olioita. Usein pakettikuvausta käytetään alijärjestelmien mallintamiseen [Gom00] tai *domain*-mallinnukseen [Dou99]. *Domain* tarkoittaa riippumatonta asiakokonaisuutta, jolla on oma nimiavaruus ja oliomalli, kuten esimerkiksi käyttöliittymä tai laiterajapinta. Pakettien välille voidaan piirtää relaatioita kuvaamaan pakettien välisiä suhteita.



Kuva 5.9. Esimerkki pakettinotaation käytöstä.

5.8.3 Rajoitteet

Rajoite (engl. *constraint*) on lisäehto tai väittämä, joka täytyy olla voimassa systeemin toimiessa oikein. Rajoitetta kuvataan merkkijonoilla tavallisissa aaltosulkeissa. Merkkijonot voidaan sisällyttää myös muistilappujen sisään. Kuten muistilaput voidaan myös rajoitteita kuvaavat merkkijonot liittää mallinnuselementteihin katkoviivoilla. Rajoitekieli on käyttäjälle vapaavalintainen. Se voi olla epäformaali sanallinen ilmaisu tai jokin määritelty rajoitekieli, kuten esimerkiksi OCL (*Object Constraint Language*) [OMG01]. Rajoitteita voidaan liittää esimerkiksi sekvenssikaavioihin esittämään reaaliaikajärjestelmän aikarajoitteita sanomien välillä (ks. kuva 5.7).

5.8.4 Stereotyypit

Stereotyypit (engl. *stereotype*) ovat UML:n laajennusmekanismi, joka mahdollistaa mallinnuskielen laajentamisen vastaamaan paremmin käyttäjän tarpeita. Jokainen mallinnuselementti UML:ssä on metaluokka (engl. *metaclass*) ja stereotyypitetty metaluokka on johdettu UML-metaluokasta. Esimerkiksi UML:n luokkamallinnuselementistä voidaan luoda uusi metaluokka stereotyyppien avulla, joka on kuin tavallinen luokkaelementti, mutta sitä on jollain tavoin laajennettu tai erikoistettu. Stereotyyppejä voi käyttää minkä tahansa UML-elementin merkityksen laajentamiseen tai erikoistamiseen. UML-notaatioon kuuluu useita valmiiksi määriteltyjä stereotyyppejä ja käyttäjä voi vapaasti luoda myös uusia.

Reaaliaikajärjestelmissä voidaan stereotyyppejä hyödyntää monipuolisesti määriteltäessä uusia metaluokkia. Stereotyypeillä voidaan erikoistaa esimerkiksi UML:n *class*-metaluokasta uusi metaluokka *active class*, jonka stereotyyppi on `<<active>>`. Sillä tarkoitetaan säiettä tai prosessia. (Aktiivisen ja passiivisen olion kuvaustapa ja merkitys

UML:ssä on määritelty spesifikaatiossa UML-standard 1.4 [OMG01, s. 3-129]). Olioiden välisiä sanomaluokkia voidaan erikoistaa sanomien ominaisuuksien mukaan (esimerkiksi `<<synchronous>>`, `<<asynchronous>>`, `<<periodic>>`, `<<aperiodic>>`). Stereotyyppejä voidaan esittää myös graafisilla ikoneilla (ks. kuvat 5.1 ja 5.7) . Tyypillisimmät UML-stereotyypit ovat `<<boundary>>`, `<<entity>>` ja `<<control>>`, jotka kuvaavat sovellusten pääluokkatyyppejä. Reaaliaikajärjestelmiin sopivia stereotyyppejä on esitelty tarkemmin lähteissä [Dou98, Dou99, Gom00]. Stereotyyppien käyttöä on havainnollistettu esimerkiksi kuvissa 5.3 ja 5.9.

6 Prosessimalli reaaliaikajärjestelmien kehittämiseen UML-notaation avulla

Tässä luvussa esitellään yksinkertainen kolmivaiheinen prosessimalli reaaliaikajärjestelmien kehittämiseen, joka perustuu lähteessä [Pre00] kuvattuihin viiteen näkymään. Käytetyt prosessivaiheet ovat vaatimusmäärittely, yleissuunnittelu ja yksityiskohtainen suunnittelu. Vaihejako muistuttaa OMT++ -menetelmän vaiheita pois lukien OMT++:n *feasibilty study* -vaihe [Lep99]. Näkymien kuvaamiseen käytetään edellisessä luvussa esitettyjä UML-kaavioita. Esitettävää prosessimallia sovelletaan luvussa 7 esitettävän DTS Proton mallintamisessa, jolloin voimme käytännössä havainnollistaa prosessimallin toimivuutta sekä UML-notaation sopivuutta reaaliaikasovelluksen tuottamiseen. Tämä luku esittelee sen, mitä näkymiä pyritään mallintamaan kussakin vaiheessa sekä mitä kaavioita voidaan käyttää kunkin näkymän kuvaamiseen. Luku ei ota kantaa muihin ohjelmistoprosessiin liittyviin asioihin kuten rooleihin, työnjakoon, metriikoihin, aikatauluihin, resurssien käyttöön jne.

6.1 Reaaliaikaohjelmistojen oliokeskeisiä kehitysmenetelmiä

Olio-ohjelmoinnin suosion kasvaminen on siirtänyt huomion pelkästä oliokeskeisestä toteutustavasta myös oliokeskeiseen suunnitteluun. Oliokeskeisiä suunnittelumenetelmiä on olemassa tavanomaisten oliosovellusten sekä erikseen reaaliaikasovellusten suunnittelemiseen. 1990-luvun aikana Grady Booch, James Rumbaugh ja Ivar Jacobson alkoivat yhdistämään kehittämiään oliomenetelmiä yhtenäiseksi standardiksi. Booch oli kehittänyt 1980- ja 1990-luvuilla Booch-menetelmää [Boo94], Rumbaugh kollegoineen suosittua OMT (Object Modelling Technique)-menetelmää [Rum91] ja Jacobson OOSE (Object-Oriented Software Engineering)-menetelmää [Jac92]. Näistä menetelmistä parhaita ominaisuuksia yhdistelemällä syntyi UM (Unified Method)-menetelmä ja lopulta UML-notaatio [OMG01, Pre00].

Erityisesti reaaliaikajärjestelmien kehittämistä varten on myös kehitetty useita oliokeskeisiä menetelmiä. Gomaa on kehittänyt UML-notaatiota käyttävän COMET (Concurrent Object Modeling and architechtural design Method)-menetelmän, joka perustuu pääasiassa luokka- ja yhteistoimintakaavioiden hyödyntämiseen [Gom00]. Douglass esittelee lähteessä [Dou99] ROPES (Rapid Object-Oriented Process for Embedded Systems)-prosessia, jossa on myös pohjana UML-notaatio. OCTOPUS-menetelmä on yhdistelmä OMT- ja Fusion-menetelmistä [Awa96]. Ellis esittelee RTSA (Real-Time Structured Analysis)-menetelmän pohjalta kehitettyä RTOOSA (Real-Time Object-Oriented Structured Analysis)-menetelmää lähteessä [Ell94] ja Selic ROOM (Real-Time Object-Oriented Modelling)-menetelmää lähteessä [Sel94]. Edelliset prosessimallit hyödyntävät monin eri tavoin toisistaan poikkeavia oliomallinnusnotaatioita. Menetelmien lisäksi myös oliomallinnustyökalujen valikoima on nykyään runsas ja erilaisia mallinnusohjelmia löytyy myös Internetistä (esim. [ARG02]) .

6.2 Tyypillinen oliomallinnusprosessi

Tyypillinen oliopohjainen mallinnusprosessi jakaantuu yleensä analyysi- ja suunnitteluvaiheiseen. Analyysivaihe voidaan lisäksi jakaa vaatimus- ja olioanalyysivaiheisiin. Vaatimusanalyysin tavoitteena on luoda ymmärtämys siitä, mitä järjestelmän tulee tehdä, minkälaisessa ympäristössä kehitettävä järjestelmä toimii ja mitkä ovat järjestelmälle asetettavat ei-toiminnalliset vaatimukset. Vaatimusmäärittelyvaiheen tuotokset kuvaavat järjestelmää käyttäjän näkökulmasta. Rakenteellisen olioanalyysin tavoite on jakaa järjestelmä pienempiin osakokonaisuuksiin: alijärjestelmiin, luokkiin ja olioihin. Kun järjestelmä on ”purettu” pienempiin osiin, pyritään näiden osien keskinäistä toimintaa ja riippuvuutta mallintamaan toiminnallisen olioanalyysin avulla. Rakenteellisen ja toiminnallisen olioanalyysin välillä voidaan joutua suorittamaan useampia iterointikierroksia ennen kuin malli on valmis, sillä toiminnallisen analyysin tehtävä on myös tuoda esiin rakenteessa olevia puutteita. Ylätason mallissa havaittavien puutteiden ja virheiden korjaaminen on analyysivaiheessa helpompaa kuin suunnitteluvaiheessa. Suunnitteluvaiheen tavoite on tuottaa analyysivaiheen kanssa yhdenmukainen ratkaisumalli sille, kuinka sovellus voidaan implementoida. Suunnitteluvaiheessa tarkennetaan analyysivaiheen mallia suunnittelemalla yksittäisten

luokkien ja olioiden toteutusta tarkemmin ja sitä, miten luokkien ja olioiden välinen kommunikointi toteutetaan. Myös tämä vaihe voi sisältää iterointia rakenteellisen ja toiminnallisen mallin välillä. Olioanalyysivaihetta ja suunnitteluvaihetta yhdessä voi kutsua ohjelmistosuunnittelijan näkökulmaksi kehitettävästä sovelluksesta. Suunnitteluvaiheen tuotosten pitäisi olla mahdollisimman yksiselitteisiä ja antaa selkeä ratkaisumalli koodin implementoimiselle.[Pre00]

6.3 ”Viisi näkymää”

Lähteessä [Pre00] esitetään, kuinka oliokeskeisesti kehitettävää järjestelmää kuvataan viidestä eri ”näkökulmasta”. UML:ään pohjautuva analyysivaihe keskittyy yleensä kuvaamaan järjestelmää sekä rakenteellisesta että käyttäjän näkökulmasta. Suunnitteluvaiheessa keskitytään yleensä toiminnallisuuden, toteutuksen sekä toteutusympäristön mallintamiseen. Kunkin näkymän kuvaamiseen hyödynnetään yhtä tai useampaa UML-kaaviota. Seuraavassa on lyhyet selitykset eri näkymistä [Pre00]:

User model view (UMV): Mallissa kuvataan järjestelmää käyttäjän (UML:ssä actor) näkökulmasta. Käyttäjän näkökulmaa mallinnetaan yleensä käyttötapausten avulla. Käyttötapaukset kuvaavat skenaarioita järjestelmän vuorovaikutuksesta ympäristönsä kanssa. Huomioitavaa on, että reaaliaikajärjestelmiä mallinnettaessa loppukäyttäjät eivät aina ole välittömästi ihmisiä vaan esimerkiksi laitteita tai ajastimia, jotka voivat myös pyytää järjestelmää suorittamaan palveluita.

Environmental model view (EMV): Näkymä kuvaa järjestelmän toteutusympäristön rakenteellisesta ja toiminnallisesta näkökulmasta.

Structural model view (SMV): Tämä näkymä kuvaa datan sekä erilaisten rakenteellisten elementtien välisiä riippuvuuksia järjestelmän sisällä. Se on staattinen kuvaus luokista, olioista sekä niiden välisistä assosiaatioista. SMV-näkymän tuotoksia käytetään apuna kuvattaessa järjestelmää toiminnallisesta BMV-perspektiivistä.

Behavioral model view (BMV): Näkymä kuvaa järjestelmän dynaamista toimintaa ja käyttäytymistä eli sillä pyritään esittämään UMV- ja SMV-näkymissä kuvattujen erilaisten

rakenteellisten elementtien välistä vuorovaikutusta ja yhteistoimintaa. BMV-näkymää voidaan käyttää apuna rakenteellisen SMV-näkymän tuottamisessa. Jos BMV-näkymää kuvattaessa todetaan aiemmin suunnitellussa rakenteessa sellaisia puutteita, että toiminnallisuuden toteuttaminen ei onnistu järkevästi, palataan SMV-näkymään ja pyritään muuttamaan sitä niin, että toiminnalliset vaatimukset pystytään toteuttamaan. Näkymien välillä voi joutua suorittamaan useitakin iterointikierroksia.

Implementation model view (IMV): Näkymä kuvaa, kuinka rakenteelliset ja toiminnalliset mallit toteutetaan.

Näkymien kuvaamisessa käytettävät UML-kaaviot on esitelty luvussa 5. Seuraavissa kappaleissa on tarkoitus sovittaa näkymät kolmivaiheiseen kehitysprosessiin. Lisäksi esitellään mitä UML-kaavioita käytetään apuna näkymien kuvaamisessa.

6.4 Kolmivaiheinen prosessimalli reaaliaikajärjestelmän suunnittelemiseen

6.4.1 Vaatimusmäärittely

Vaatimusmäärittelyvaiheen tavoitteet vastaavat hyvin pitkälle tyypillisen oliokeskeisen ohjelmistoprosessin vaatimusanalyysia. Reaaliaikajärjestelmän vaatimusmäärittely lähtee liikkeelle toimintaympäristön kuvaamisesta, koska reaaliaikajärjestelmä tai –sovellus ei välttämättä toimi välittömästi ihmisten ohjaamassa käyttöympäristössä. Toimintaympäristö kuvaa järjestelmää EMV-näkökulmasta. Kuvauskaaviona voidaan käyttää joko **luokka-** tai **kontekstikaaviota** [Dou98, Awa96]. Kontekstikaavio ei ole varsinainen UML-kaavio, mutta se voidaan esittää soveltamalla sijoittelukaaviota. Kontekstikaavio kuvaa järjestelmän ympäristön rakennetta sekä laitteita, joiden kanssa järjestelmä toimii vuorovaikutuksessa. Sulautettuja reaaliaikajärjestelmiä kuvattaessa laitteet ovat usein myös aktoreita.

Vaatimusmäärittelyvaiheen ensisijainen tehtävä on tuottaa mahdollisimman selkeä kuvaus siitä, mitä järjestelmän pitäisi tehdä. Järjestelmän ulkoista toiminnallisuutta kuvattaessa mallinnetaan UMV-näkymää. UMV-näkymän kuvaamisessa käytetään apuna **käyttötapauskaavioita**, joita voidaan tarkentaa sanallisin kuvauksin ja havainnollistaa skenaariokaavioiden avulla. Skenaariokaaviot kuvaavat tavallaan järjestelmän ulkoista BMV-

näkymää, joissa itse järjestelmä kuvataan vielä ns. mustana laatikkona, jonka sisäistä toteutusta ei tunneta. Käyttötapauksia havainnollistavia skenaarioita kuvataan **sekvenssi-** tai **yhteistoimintakaavioiden** avulla. Näissä kaavioissa järjestelmä ja aktorit ovat olioita, jotka kommunikoivat keskenään käyttötapauksissa kuvatulla tavalla. Käyttötapauksiin ja skenaarioihin tulee kuvata myös toiminnallisiin vaatimuksiin liittyvät aikarajoitteet eli tieto siitä, missä ajassa järjestelmän tulee tuottaa vaste tietylle syötteelle. Koska sekvenssikaavion yhtenä dimensiona on aika, käytetään niitä käyttötapausskenaarioiden mallintamiseen. Kuten edellä on mainittu, käyttötapauksissa esiintyviä aktoreita voidaan irrottaa kontekstikaavioista ja tämän lisäksi on analysoitava, onko järjestelmän käyttäjinä ihmisiä, jotka ovat järjestelmän kannalta aktoreita. Reaaliaikajärjestelmien erityispiirre on, että myös ajastimia voidaan kuvata aktoreina [Awa96, Gom00]. Reaaliaikajärjestelmän toiminnallisten vaatimusten kuvaamiseen esitellään kirjallisuudessa myös ns. EERL-listoja (*External Events and Response List*) [Dou99, Ell94]. Niillä pyritään kuvaamaan, mitä toimintoja järjestelmän tulee suorittaa tietyn syötteen saapuessa. Tässä työssä ei listoja kuitenkaan käytetä, koska pyritään UML-keskeiseen mallinnukseen, eikä notaatio ei tue listojen tekemistä.

6.4.2 Yleissuunnitteluvaihe

Yleissuunnittelun tavoitteena on kuvata ensimmäisen kerran järjestelmän arkkitehtuuri eli SMV-näkymä yleisellä tasolla. Yleissuunnittelussa pyritään löytämään järjestelmälle ensimmäinen luokkarakenne, joka esitetään **luokkakaavioilla**. Tämän vaiheen syötteenä tarvitaan vähintään vaatimusmäärittelyvaiheen käyttötapaukset. Luokkien ja olioiden tunnistamiseen on esitetty useita menetelmiä mm. lähteessä [Dou99]. Tyypilliset analyysitason luokkatyypit ovat *boundary, entity* ja *control*. Reaaliaikajärjestelmiä mallinnettaessa käytetään myös tyyppiä *active*, mikä tarkoittaa, että olio suorittaa toimintaansa omassa säikeessään. Tällainen olio vastaa luvussa 3 esitettyä 'tehtävä'-käsitettä (engl. *task*).

Yleissuunnitteluvaiheen luokille määritellään vastuut, mitä niiden tulee osata tehdä ja kuinka ne kommunikoivat keskenään. Määrittely tapahtuu kuitenkin täysin toteutusriippumattomasti. Jos olioita on paljon, voidaan ne ryhmitellä **pakettinotaatiolla** alijärjestelmiin. Keskenään tiukasti kytketyt (engl. *tightly coupled*) oliot kuuluvat samaan alijärjestelmään, jolloin alijärjestelmien välille pitäisi jäädä heikompia kytkentöjä (engl. *loosely coupled*).

Alijärjestelmäjako voidaan toteuttaa myös ennen luokkien ja olioiden tunnistamista, jolloin sitä voidaan käyttää apuna myös esim. suunnittelijoiden välisessä työnjaossa. Tällöin kukin suunnittelija voi itsenäisesti suunnitella alijärjestelmän sisäistä luokkarakennetta sillä edellytyksellä, että alijärjestelmien väliset rajapinnat on määritelty. Suuremmat järjestelmät voivat edellyttää koodin suorittamista useammalla prosessorilla, jolloin rakenteen kuvaamiseen voidaan käyttää myös **sijoittelukaavioita**.

Sen jälkeen kun karkean tason luokkarakenne on suunniteltu, testataan sen toimivuutta ja luokkien välisiä kommunikointitapoja siirtymällä BMV-näkymän tarkasteluun **sekvenssikaavioiden** avulla. Lähteessä [Gom00] suositellaan yhteistoimintakaavioiden käyttämistä reaaliaikajärjestelmän sisäisten luokkien välisen toiminnallisuuden mallintamiseen. Perusteena on esitetty, että ne kuvaavat paremmin olioiden välisiä riippuvuuksia. Tässä työssä otetaan kuitenkin lähtökohdaksi se, että olioiden ja luokkien väliset riippuvuudet näkyvät staattisesta luokkakaaviosta ja käyttäytyminen kuvataan sekvenssikaavioilla, koska ne havainnollistavat paremmin toimintojen ajallista järjestystä. Sekvenssikaavioiden tavoitteena on varmistaa, että suunniteltu alijärjestelmä- ja/tai luokkarakenne pystyy toteuttamaan käyttötapauksissa esitetyt vaatimukset. Tällöin palataan vaatimusmäärittelyssä esitettyyn UMV-näkymään ja sitä kuvaavaan käyttötapauskaavioon. Jos käyttötapauksista aiheutuvien skenaarioiden läpikäyminen ei onnistu järkevästi yleissuunnitteluvaiheen luokkakaavioiden avulla, täytyy luokkakaavion rakennetta miettiä uudelleen ja tehdä uusi iterointikierros yleissuunnittelussa SMV-näkymästä BMV-näkymään.

Kun luokkakaavio ja mahdollinen alijärjestelmä rakenne on todettu riittävän kattavaksi, voidaan luokkakaavio antaa syötteeksi seuraavalle vaiheelle eli yksityiskohtaiselle suunnittelulle, jota kuvataan seuraavassa kappaleessa.

6.4.3 Yksityiskohtainen suunnittelu

Yksityiskohtaisen suunnittelun lähtökohtana on yleissuunnitteluvaiheessa tuotettu luokkakaavio. Yleissuunnitteluvaiheessa on suunniteltu mitä luokkien ja olioiden tulee osata sekä kuinka ne kommunikoivat keskenään, jotta järjestelmän toiminnalliset vaatimukset toteutuisivat. Yksityiskohtaisen suunnittelun tavoitteena on tuottaa yleissuunnitteluvaiheen

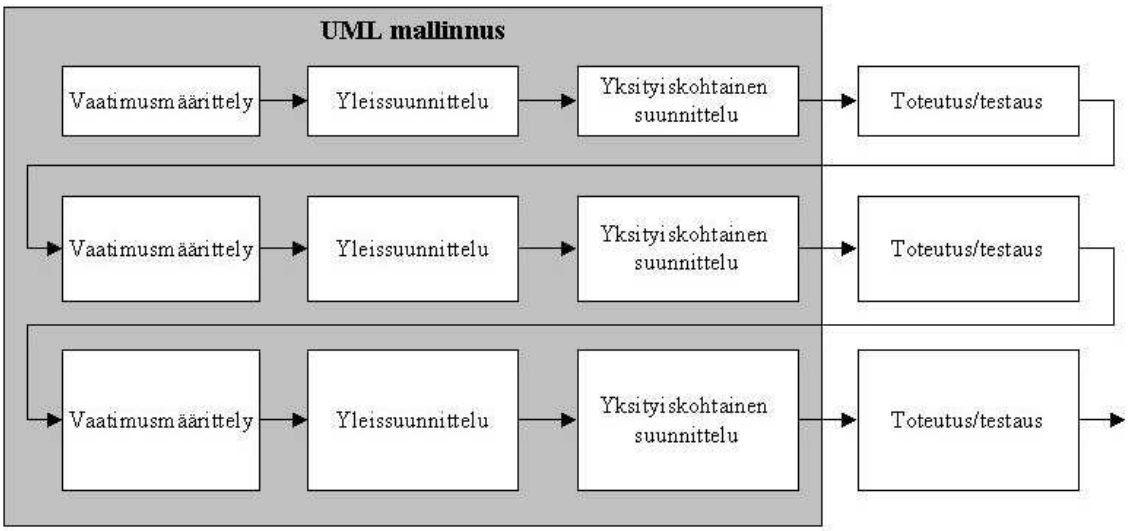
luokkakaaviosta tarkempi toteutuskelpoinen **luokkakaavio** eli yksityiskohtainen kuvaus sovelluksen SMV-näkymästä. Samalla voidaan tarkastella onko tarjolla uudelleenkäytettäviä luokkia tai voidaanko jotkin luokat toteuttaa periyttämällä ne jostain samantyyppisestä luokasta. Luokkiin suunniteltavia julkisia rajapintametodeja voidaan etsiä yleissuunnitteluvaiheessa mallinnetuista luokkien välistä yhteistoimintaa kuvaavista skenaarioista. Jos järjestelmä toimii vuorovaikutuksessa muiden ulkoisten laitteiden kanssa, voidaan vaatimusmäärittelyvaiheen EMV-näkymää käyttää avuksi ja mahdollisesti tarkentaa rajapintojen suunnittelun yhteydessä.

Jos luokkarakenne sisältää aktiivisia olioita (esim. reaaliaikajärjestelmät sisältävät yleensä useita), tarvitaan myös kommunikointimekanismeja. Niiden suunnittelussa voi tarkastella, mitä palveluita sovelluksen käytössä oleva käyttöjärjestelmä tarjoaa. Lisäksi aktiivisille olioille määritellään prioriteetit. Prioriteetit voidaan mallintaa luokkien attribuutteina. Olioiden sisäistä käyttäytymistä (BMV-näkymä) voidaan kuvata **aktiviteettikaavioiden** avulla. Ne ovat hyödyllisiä lähinnä kuvattaessa aktiivisten olioiden toimintosekvenssejä. Niillä voidaan kuvata tarvittaessa myös olioiden välistä rinnakkaisuutta. Jos sovellus käyttää esimerkiksi tietokantakomponentteja, voidaan rakenteen toteuttavien komponenttien suhteita kuvata IMV-näkökulmasta **komponenttikaavioiden** avulla. Kun yksityiskohtainen suunnittelu on toteutettu, tulisi implementointivaiheeseen olla syötteenä ohjelmistorakenne, josta selviää mitä luokkia, aliohjelmia, säikeitä ja tietorakenteita ohjelmakoodin tulee sisältää.

6.5 Yhteenveto näkymien ja vaiheiden suhteesta

Käyttötapaukset ohjaavat alusta lähtien suunnittelua ja mallintamista. Toteuttamalla käyttötapauksista pienemmän osajoukon kerrallaan saadaan prosessista inkrementaalinen. Koska suunnitteluvaiheessa ei tarkastella, ovatko esim. sovelluksen suorittamat tehtävät skeduloitavissa, vaan aikavaatimusten toteutuminen todennetaan vasta testausvaiheessa, voidaan mallinnusprosessia joutua iteroimaan. Tällöin prosessia voidaan kutsua myös iteratiiviseksi, koska aikavaatimusten toteutumattomuus aiheuttaa sovelluksen määrittelylle ja suunnittelulle uuden kierroksen, jossa tulee huomioida testauksessa kohdatut ongelmat. Ratkaisu voi olla esimerkiksi vaatimuksista tinkiminen tai ohjelman suorituksen optimoiminen. Ku-

vassa 6.1 on havainnollistettu inkrementaalisuutta ja iteratiivisuutta. Prosessilaatikoiden koon kasvaminen kuvaa ohjelmiston koon kasvua. Ensimmäinen kierros voi olla esimerkiksi protoiluversio, jossa toteutetaan vain oleellisimmat käyttötapaukset ja testataan niiden toteutettavuutta ja toimivuutta. Samalla voidaan tuottaa testausvaiheesta tietoa seuraavalle mallinnuskierrokselle, jolloin voidaan samalla pyrkiä ratkaisemaan edellisen version ongelmat (kuten aikarajoiteongelmat). Jos edellinen versio todetaan toimivaksi, voidaan seuraavassa vaiheessa toteuttaa lisää käyttötapauksia, jolloin prosessi toimii myös inkrementaalisesti. Samalla sovelluksen koko kasvaa.



Kuva 6.1. Kolmivaiheinen mallinnusprosessi.

Taulukossa 6.1 esitetään, mitä näkymiä mallinnetaan prosessin eri vaiheissa. Kuten taulukosta 6.1 näkyy, vaatimusmäärittelyssä keskitytään lähinnä käyttäjän näkökulman ja toimintaympäristön mallintamiseen. Käyttötapauksia havainnollistavat skenaariot kuvaavat käytännössä samaa asiaa kuin mitä kirjoitetaan käyttötapausten sanallisiin kuvauksiin, joten tässä yhteydessä nekin mallintavat järjestelmää UMV-näkymän kannalta. Toisaalta ne kuvaavat järjestelmän ulkoista käyttäytymistä eli BMV-näkymää, kuten taulukossa 6.1 on tulkittu.

Yleissuunnitteluvaiheessa mallinnetaan sisäistä rakennetta (SMV-näkymä) ja käyttäytymistä (BMV-näkymä). Yksityiskohtainen suunnittelu keskittyy luokkakaavion tarkentami-

seen sekä metodien ja attribuuttien määrittelyyn. Lisäksi voidaan mallintaa järjestelmää toteutusnäkökulmasta (IMV-näkymä). Oleellista on, että vaatimusmäärittelystä saadaan yleissuunnitteluvaiheelle syötteeksi yksiselitteinen kuvaus siitä, mitä järjestelmän pitäisi tehdä, ja edelleen, yleissuunnitteluvaiheesta yksityiskohtaiselle suunnittelulle syötteeksi yksikäsitteinen luokkarakenne, jota voidaan alkaa tarkentamaan kohti toteutuskelpoista mallia.

	Vaatimusmäärittely	**Yleissuunnittelu**	**Yksityiskohtainen suunnittelu**
UMV:	X		
EMV:	X		
SMV:		X	X
BMV:	X	X	X
IMV:			X

Taulukko 6.1. Näkymien sijoittuminen ohjelmistoprosessin eri vaiheisiin.

Eri näkymien kuvaaminen onnistuu usein monilla eri kaavioilla. Taulukkoon 6.2 on kerätty luvun 5 perusteella ne UML-kaaviot, joita voidaan käyttää eri näkymien mallintamiseen. Luvun 6.4 perusteella käytettävät kaaviot on tummennettu.

	UMV	**EMV**	**SMV**	**BMV**	**IMV**
Käyttötapauskaavio	**X**	**X**			
Luokkakaavio			**X**		
Tilakaavio				x	x
Aktiviteettikaavio				**X**	**X**
Sekvenssikaavio	**X**			**X**	
Yhteistoimintakaavio	x			x	
Komponenttikaavio					x
Sijoittelukaavio		**X**			

Taulukko 6.2. Eri näkymien kuvaamiseen käytettävät kaaviot.

7 DTS Proto –esimerkkisovellus

DTS Proto –sovellus toimii tiedonsiirtosovelluksena lentokoneympäristössä. Se on osa suurempaa järjestelmäkokonaisuutta, johon kuuluu päätietokone (engl. *Main Computer, MC*), modeemi (engl. *modem*) ja radio. Päätietokone tarjoaa käyttäjälle tiedonsiirtojärjestelmän käyttöliittymän. Lisäksi se ohjaa MIL-STD-1553–standardin mukaista väyläliikennettä itsensä ja DTS Proto –sovelluksen välillä. Tämän lisäksi se vastaa myös järjestelmäkontekstin muiden osien, kuten DTS Proton ja modeemin, tilan valvonnasta. Modeemin tehtävä on luoda yhteys DTS Proton ja radion välille. Radion avulla datasanomat (engl. *data message*) lähetetään ilmateitse muihin vastaaviin järjestelmiin. Tässä luvussa mallinnetaan osa DTS Proto –tiedonsiirtosovelluksen kokonaisuutta, soveltaen luvussa 6 esitettyä prosessimallia ja UML-kuvausnotaatiota. Varsinaiset UML-kaaviot on pääosin esitetty erillisessä liitteessä luvussa 10.

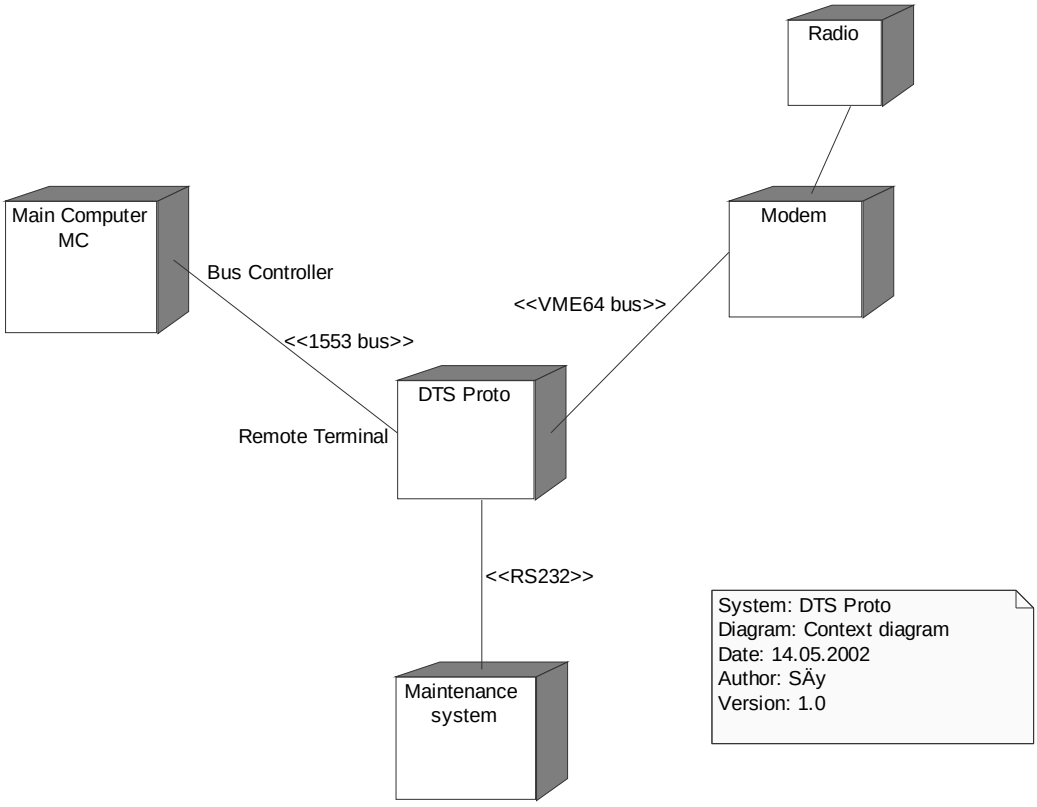
7.1 DTS Proto –sovelluksen vaatimusmäärittely

DTS Proton vaatimusmäärittelyssä kuvataan sovelluksen toimintaympäristö, tehtävät sekä sovelluksen toimintaan ja suunnitteluun liittyvät rajoitteet. Ensimmäiseksi lähdetään liikkeelle toimintaympäristön mallintamisesta kontekstikaavion avulla.

7.1.1 DTS Proton järjestelmäkonteksti

DTS Proton järjestelmäkonteksti kuvataan kappaleessa 6.4.1 esitellyn kontekstikaavion avulla. Kontekstikaavion piirtämiseen käytetään tässä yhteydessä sijoittelukaaviota (kuva 7.1). Siinä kuvataan DTS Proton toimintaympäristön pääelementit sekä niiden väliset fyysiset liitynnät. DTS Proto –prosessorikortin sekä muiden solmujen väliset liityntäratkaisut esitetään solmujen välisiä assosiaatioita tarkentavilla stereotyypeillä. DTS Proton ja MC:n välisenä liityntäväylänä toimii MIL-STD-1553-standardin mukainen 1553-väylä. 1553-väylän väyläroolit voidaan liittää helposti assosiaatioihin lisäämällä roolimääritykset MC:n ja DTS Proton väliseen assosiaatioon. *Bus Controller* ohjaa väylän fyysistä liikennettä ja

Remote Terminal vastaa väylälle *Bus Controllerin* ohjeiden mukaan. Väyläroolit tulee huomioida alustettaessa DTS Proton 1553-väylärajapintaa.



Kuva 7.1. DTS Proton järjestelmäkonteksti.

7.1.2 DTS Proton ensisijaiset tehtävät

DTS Proton ensisijainen tehtävä on salata MC:n lähettämiä datasanomia ja välittää niitä modeemille (tosin vain modeemin niitä pyytäessä), joka puolestaan muodostaa niistä sanomia radiolle. Vastaavasti modeemin tehtävä on välittää radion kautta vastaanottamansa datasanomat DTS Protolle, joka purkaa sanomien koodaukset ja muodostaa niistä MIL-STD-1553-standardin mukaisia väyläsanomia MC:lle. Näistä vaatimuksista voidaan muodostaa DTS Proton kolme peruskäyttötapausta, jotka toimivat perustana sovelluksen toiminnalle. Edellisten lisäksi sovelluksen tulee suorittaa käynnistysvaiheessa joukko toimenpiteitä, jotka luovat valmiuden varsinaiselle operatiiviselle toiminnalle. Myös ne kuvataan

omana käyttötapauksena. Käyttötapaukset esitellään käyttötapauskaaviossa (kuva L.1) ja niihin liittyvä toiminta sanallisesti kappaleessa 7.1.4.

7.1.3 Muut tehtävät

DTS Proton tehtävä on myös valvoa omaa toimintakuntoaan ja välittää siitä tietoa, kun MC sitä pyytää. Lisäksi DTS Proto valvoo modeemin toimintakuntoa lähettämällä tilakyselyjä, joihin modeemi vastaa lähettämällä tilatietonsa. Jos modeemin vaste tilakyselyyn kertoo ongelmista, on tieto välitettävä myös MC:lle.

DTS Protoon kuuluu myös liityntä huoltojärjestelmää varten. Huoltojärjestelmän avulla simuloidaan modeemia silloin, kun varsinaista modeemia ei ole käytössä. Simulointi suoritetaan vain sanomanvälityksen osalta. Esim. tilakyselyihin ei huoltopäätteen kautta saada vasteita. Lisäksi huoltoliitynnän kautta on pystyttävä monitoroimaan 1553-väylälle lähetettyjä ja vastaanotettuja sanomia. Nämä toiminnot edellyttävät, että tiedonsiirtoon liittyvät ensisijaiset käyttötapaukset ovat myös toteutettu. Tässä kappaleessa kuvatuista tehtävistä koostuvat käyttötapaukset esitellään DTS Proton käyttötapauskaaviossa (kuva L.1), mutta niitä ei oteta varsinaiseen mallintamiseen mukaan. Ne toteutetaan vasta seuraavaan versioon, sen jälkeen, kun tiedonsiirrosta vastaava osuus on toimintavalmis.

7.1.4 Käyttötapauskuvaukset

Seuraavassa kuvataan kappaleessa 7.1.2 esitettyihin ensisijaisiin tehtäviin liittyvät käyttötapaukset:

1. Transfer DataMsg to MC

Yhteenveto: Modeemi välittää radion vastaanottaman datasanoman DTS Protolle. DTS Proto purkaa sanoman koodauksen ja välittää sen MC:lle.

Aktorit: Ensisijainen aktori modeemi, toissijainen aktori MC.

Esiehdot: DTS Proto sekä rajapinnat MC:lle ja modeemille toimivat normaalisti. Lisäksi MC:n tulee olla kuitannut edellinen sanoma vastaanotetuksi.

Kuvaus: Modeemi lähettää radiolta vastaanottamansa datasanoman DTS Protolle [Poikkeus 1]. DTS Proto purkaa sanoman koodauksen, muodostaa uuden sanoman ja lähettää sen MC:lle. Sanoman sisällöstä tulee

selvitä myös sanoman pituus. Lisäksi DTS Proton tulee erikseen ilmoittaa väylälle kirjoitetusta datasta MC:lle [Poikkeus 2].

Poikkeukset: [1] Jos DTS Proto ei ole toimintakunnossa, modeemin tulee hylätä sanoma. MC yrittää DTS Proton uudelleen käynnistämistä havaittuaan ongelman. MC havaitsee ongelman vain, jos DTS Proto lopettaa 1553-väylälle vastaamisen (Jatkossa ongelma voidaan havaita myös tilakyselyiden avulla). Modeemi ei suorita toimenpiteitä DTS Proton käynnistämiseksi. [2] Jos MC ei ole kuitannut edellistä sanomaa vastaanotetuksi, niin DTS Proto ei lähetä sanomaa eikä ilmoitusta uudesta sanomasta, vaan jää odottamaan kuittausta.

Jälkiehdot: Kun MC on vastaanottanut ja kuitannut datasanoman, voi DTS Proto lähettää seuraavan sanoman.

2. Transfer DataMsg to Modem

Yhteenveto: MC lähettää DTS Protolle sanoman, josta DTS Proto muodostaa datasanoman modeemille.

Aktorit: MC

Esiehdot: DTS Proto sekä rajapinnat MC:lle ja modeemille toimivat normaalisti.

Kuvaus: MC ilmoittaa lähettäneensä uuden sanoman [Poikkeus 1], jonka jälkeen DTS Proto lukee sanoman sisällön ja muodostaa siitä uuden salatun modeemisanoman. MC lähettää myös sanoman pituuden [Poikkeus 2]. Muodostettuaan uuden modeemisanoman, DTS Proto tallettaa sen seuraavaa lähetystiedonkyselyä varten.

Poikkeukset: [1] Jos 1553-väylä DTS Protolle ei ole toiminnassa, MC käynnistää DTS Proton uudelleen. [2] Jos pituus on nolla, ei sanomaa muodosteta.

Jälkiehdot: DTS Proto on valmis antamaan modeemin lähetystiedonkyselyyn vasteeksi uuden sanoman.

3. Response to TxQuery

Yhteenveto: Modeemi pyytää DTS Protoa antamaan seuraavaksi lähetysvuossa olevan datasanoman.

Aktorit: Modeemi

Esiehdot: DTS Proto ja rajapinta modeemille ovat toiminnassa.

Kuvaus: Modeemi lähettää DTS Protolle lähetystiedonkyselyn [Poikkeus 1]. DTS Proton tulee lukea kysely rajapinnasta niin nopeasti, että se ehtii antamaan vasteen alle 50ms:ssa. DTS Proto vastaa modeemille lähettämällä viimeisimmän modeemisanoman [Poikkeukset 2,3].

Poikkeukset: [1] Jos DTS Proto ei ole toimintakunnossa, modeemi hylkää lähetystiedon kyselyn. Ongelman havaittuaan MC yrittää DTS Proton uudelleen käynnistämistä. MC havaitsee ongelman vain, jos DTS Proto lopettaa 1553-väylälle vastaamisen (Jatkossa ongelma voidaan havaita myös tilakyselyiden avulla). Modeemi ei suorita toimenpiteitä DTS Proton käynnistämiseksi. [2] Sanoma hylätään, jos modeemi ei vastaanota sitä. [3] Jos DTS Protolla ei ole mitään lähetettävää, tulee vasteeksi modeemille antaa tieto siitä.

Jälkiehdot: Vaste lähetetty modeemille. DTS Proto jatkaa normaalia toimintaa.

4. Startup of the system

Yhteenveto: Laitteiston käynnistämiseen liittyvät tehtävät.

Aktorit: DTS Proton käyttäjä.

Esiehdot: DTS Proto ei ole toiminnassa.

Kuvaus:

1. Laitteiston alustus.
2. Käyttöjärjestelmän käynnistys (maksimi kestoaika < 1000 ms).
3. Sanomajonojen luonti modeemirajapintaan.
4. Sovelluksen olioiden ja tehtäväsäikeiden luonti.
5. Modeemirajapinnan sanomajonotunnisteiden lisäys nimitietokantaan.
6. Automaattinen alkutestaus.
7. (Tilakyselyjen aloitus, ei tässä versiossa).
8. Modeemin käynnistyssanoman odottaminen.
9. Toimintaparametrien asetus.

Poikkeukset: Jonkin käynnistysvaiheen epäonnistuttua aloitetaan käynnistäminen alusta uudelleen.

Jälkiehdot: DTS Proto toiminnassa.

Kuvissa L.2-L.5 esitetään sanallisia kuvauksia vastaavat UML-sekvenssikaaviot ja kuvassa L.4 esitetään vertailun vuoksi kuvassa L.3 esitetty käyttötapaussekvenssi myös yhteistoimintakaavion avulla. Kaavioita vertailtaessa on helppo havaita sekvenssikaavion parempi selkeys tapahtumien ajallisen suoritusjärjestyksen ja aikarajoitteiden kuvaamisessa. Kuvas-

sa L.5 on esitetty käynnistysvaihe luvun 6 prosessimallista poiketen tilakaaviolla, koska toiminnot ovat lähinnä järjestelmän sisäisiä.

7.1.5 Laitteistorajoitteet

DTS Proto on sulautettu sovellus, jolle toimintaympäristö aiheuttaa suunnittelurajoitteita, jotka tulee huomioida myöhemmin suunnittelun eri vaiheissa. Tässä kappaleessa kuvataan DTS Protoon liittyvät ennaltamäärätyt laitteistoratkaisut, joiden rajoissa suunnittelua voidaan lähteä viemään eteenpäin.

DTS Proto toteutetaan DY4:n valmistamalle SVME/DMV-179 SBC-korttille. Kortilla toimii PowerPC 750 -prosessori, joka on 32-bittisellä osoiteväylällä ja 64-bittisellä dataväylällä varustettu RISC-tyyppinen prosessori [Mot97]. DTS Proto liittyy MIL-STD-1553-väylään PMC-601-tyyppisen liityntäkortin kautta. Myös modeemin laitealustana toimii SVME/DMV-179-prosessorikortti. DTS Proto kommunikoi modeemin kanssa VME64–tyyppisen väyläliitynnän kautta. Huoltopääteyhteys toteutetaan SVME/DMV-179-kortilla olevien RS-232- ja RS-422/485-tyyppisten sarjaliityntäporttien kautta.[DY402]

7.1.6 Käyttöjärjestelmäratkaisu

DTS Proto ei käytä laitteiston tarjoamia palveluita suoraan, vaan prosessorikortti ja sovellus kommunikoivat VxWorks 5.4 -reaaliaikakäyttöjärjestelmän kautta. Näin voidaan piilottaa laitteistoyksityiskohdat sovelluskerroksen suunnittelijalta. Käyttöjärjestelmän käyttöä voidaan perustella sillä, että sovellus ei ole *hard*-tyyppinen reaaliaikasovellus. Jos sovelluksella olisi enemmän tiukkoja aikarajoitteita, voitaisiin harkita ”kevyempää” rajapintaa sovelluksen ja laitteiston välille.

VxWorks on pre-emptiivisellä skeduloinnilla varustettu skaalattava käyttöjärjestelmä, joka mahdollistaa reaaliaikasovelluksen toteuttamisen joukkona toisistaan riippumattomia säikeitä. Käyttöjärjestelmä tarjoaa tuen mm. säikeiden väliseen kommunikointiin, keskeytysten käsittelyyn ja muistinhallintaan. Säikeiden välinen kommunikointi on mahdollista toteuttaa esimerkiksi jaetun muistin, semaforien, sanomajonojen, sanomaputkien tai signaalien avulla. I/O-toiminnot voidaan toteuttaa laitteistoriippumattomasti VxWorks:n C-

kirjaston avulla. VxWorks:iä voidaan laajentaa tukemaan erilaisia laitteistoratkaisuja ns. laitteistotukipaketeilla (engl. *board support package*). Tässä järjestelmässä käyttöjärjestelmää laajennetaan PMC-601 MIL-STD-1553-laitteistoajuripaketilla, joka tarjoaa ohjelmistorajapinnan 1553-väyläkortille.[VxW99a, VxW99b]

7.1.7 Yhteenveto vaatimusmäärittelyvaiheista

DTS Proton EMV-näkymän luominen kontekstikaavion avulla onnistui hyvin ja se antaa selkeän kuvan sovelluksen toimintaympäristöstä. Kontekstikaaviota pystyttiin käyttämään hyödyksi määriteltäessä sovelluksen käyttötapauksille aktoreita, sillä kuten aiemmin on mainittu, sulautetussa ympäristössä sovellusta eivät välttämättä käytä vain ihmiset. EMV-näkymän kautta siirryttiin määrittelemään sovelluksen toiminnallisia vaatimuksia eli UMV-näkymää. Se onnistui käyttötapausten muodossa.

Käyttötapausten määrittelyssä esiintyi jonkin verran tulkintaongelmia siinä, kuinka käyttötapaukset rajataan. Esimerkkinä voidaan mainita 1553-väylän lukuoperaatiot. 1553-väylän lukuoperaatiot voitaisiin tulkita erilliseksi käyttötapaukseksi, koska sovelluksen tehtävä on suorittaa niitä periodisesti välittämättä siitä, lähettääkö MC uusia sanomia vai ei. Tämä johtuu 1553-väylän toimintaperiaatteesta. Väyläsanomat liikkuvat 1553-väylällä sanomakohtaisilla taajuuksilla välittämättä siitä, päivittääkö MC niiden sisältöjä. Tällöin yksi vaihtoehto olisi tehdä erillinen käyttötapaus 1553-väylän lukuoperaatiosta ja sisällyttää se `<<include>>`-riippuvuussuhteella nykyiseen datasanoman välitysoperaatioon MC:ltä modeemille. Tässä työssä käyttötapaus mallinnettiin kuitenkin yhtenäisenä toimintosekvenssinä, jossa DTS Proto ikään kuin odottaa datasanomaa 1553-väylältä, vaikka sen todellisuudessa tulee lukea kaikki 1553-sanomat sanomakohtaisen taajuuden määräämällä nopeudella. Tällä menettelyllä saatiin käyttötapauksista kuitenkin toteutusriippumattomampia. Ongelma on kuitenkin se, että 1553-väylän lukemiseen liittyvät aikarajoitteet eivät tule huomioitua kovinkaan selkeästi vaatimusmäärittelyssä. Vaatimus näkyy lähinnä vain 1553-väylään liittyvänä teknisenä rajoitteena.

Käyttötapausten toiminnallisuuden dokumentointi tehtiin sekä sanallisesti että esimerkkisekvenssien avulla. Sanallisiin kuvauksiin määriteltiin myös käyttötapauksiin liittyvät poikkeustapaukset. Aikarajoitteita valituilla käyttötapauksilla ei ole kovin monia, mutta niiden liittäminen sekvenssikaavioon onnistuu hyvin rajoitelauseiden avulla. Poikkeustapausten määrittely jätettiin vain sanallisiin kuvauksiin, koska sekvenssikaaviot eivät suoraan tue niiden kuvaamista. Vaihtoehtona olisi sekvenssien kuvaaminen myös ehtorakenteet sisältävien aktiviteettikaavioiden avulla, mutta niiden lisääminen dokumentaatioon ei ole järkevää ylläpidettävyydenkään takia. Kaavioiden määrä kasvaisi liian suureksi. Vaatimusmäärittelyvaiheen tuotosten avulla siirrytään seuraavaksi yleissuunnitteluvaiheeseen.

7.2 DTS Proto –sovelluksen yleissuunnittelu

Yleissuunnitteluvaihe aloitetaan sovelluksen jakamisella alijärjestelmiin. Alijärjestelmäjaon yhteydessä määritellään, mitkä käyttötapaukset liittyvät mihinkin alijärjestelmään, ja se, kuinka alijärjestelmät ovat toisistaan riippuvaisia. Alijärjestelmäjaon jälkeen valitaan tiedonsiirto-osuutta kuvaava alijärjestelmä tarkemmin mallinnettavaksi. Alijärjestelmän sisäistä arkkitehtuuria kuvataan luokkakaavioiden avulla. Yleissuunnitteluvaiheessa pyritään löytämään toteutustavasta riippumaton ylemmän tason malli sille, minkälaisella luokkarakenteella tiedonsiirtoa vastaavat käyttötapaukset voidaan toteuttaa. Tähän rakenteeseen pitäisi myöhemmin olla liitettävissä/laajennettavissa myös muut, nyt mallinnuksesta poisjätetyt käyttötapaukset. Luokkakaaviosta jätetään vielä tässä vaiheessa pois esimerkiksi luokkien välisiä kommunikointimenetelmiä koskevat ratkaisut. Niitä tarkennetaan myöhemmin yksityiskohtaisen suunnittelun vaiheessa. Lisäksi yleissuunnittelussa ohitetaan aktiivisten tehtäväsäieluokkien tarkempi määrittely. Myös siihen palataan yksityiskohtaisessa suunnittelussa.

7.2.1 Alijärjestelmät

Kuvan L.6 luokkakaaviossa esitetään DTS Proton alijärjestelmät UML-paketteina. Alijärjestelmien väliset rajapinnat on kuvattu riippuvuussuhteina. Kuvasta nähdään, että tiedonsiirrosta vastaava *Communication*-alijärjestelmä ei ole riippuvainen *Maintenance*-alijärjestelmän toiminnasta. *Communication*-alijärjestelmän toiminta on riippuvainen vain

MainControl-alijärjestelmästä, koska se huolehtii käynnistykseen ym. järjestelmän ylläpitoon liittyvistä asioista. Varsinainen tiedonsiirtotoiminta ei kuitenkaan tarvitse *MainControl*-toimintoja, joten *Communication*-alijärjestelmä voidaan mallintaa itsenäisesti. *Communication*-alijärjestelmän vastuuna on luoda yhteys 1553-rajapinnan ja modeemirajapinnan välille sekä palvella MC:ltä ja Modeemilta tulevia palvelupyyntöjä. Tässä luvussa mallinnetaan *Communication Subsystem* -arkkitehtuurista Modeemin ja MC:n välisestä tiedonsiirrosta vastaavat osuudet. *MainControl*-alijärjestelmän tehtäväksi jää käynnistysvaiheeseen liittyvien toimintojen kontrollointi, järjestelmän tilan valvonta ja järjestelmän toimintaa ohjaavien tietojen ylläpito.

7.2.2 DTS Proton tiedonsiirron toteuttavat luokat

Tyypillisesti oliomallinnus etenee vaatimusmäärittelystä olioanalyysivaiheen luokkakandidaattien tunnistamiseen vaatimusmäärittelyn perusteella (esimerkkejä olioiden tunnistusmenetelmistä esitellään mm. lähteissä [Lee97, Dou99, Rin00]). Luokkien tunnistamisessa auttaa niiden ryhmittely esimerkiksi tyypillisiin luokkaryhmiin: rajapintaluokat (engl. *boundary*), tiedonsäilytysluokat (engl. *entity*) ja toimintaa ohjaavat luokat (engl. *control*). Tässä luvussa olioanalyysivaiheen luokkakaaviota kutsutaan yleissuunnitteluvaiheen luokkakaavioksi luvun 6 mukaisesti.

DTS Proton yleissuunnitteluvaiheen luokkakaavion tarkoituksena on luoda perusrakenne, jossa tiedonsiirtoon tarvittavien toimintojen vastuut jaetaan luokkakandidaateille. Yleissuunnitteluvaiheen luokkakaavion ei alkuvaiheessa tarvitse ottaa kantaa siihen, ovatko oliot aktiivisia (eli säikeitä tai prosesseja) vai passiivisia. Toisaalta yksi yleissuunnitteluvaiheen luokkakandidaatti voidaan myöhemmin toteuttaa useammankin aktiivisen olion avulla. Kuvassa L.7 on tiedonsiirtoon osallistuvat luokkakandidaatit. Luokkakaaviossa ei vielä oteta kantaa siihen, kuinka luokkien välinen kommunikointi toteutetaan. *Communication*-stereotyyppillä kuvataan kuitenkin sitä, että luokat kommunikoivat keskenään, jolloin niiden välille täytyy myöhemmin suunnitella tiedonsiirtomekanismit. Kaaviossa ei vielä tarkenneta koostesuhteiden tyyppiä, mutta pyrkimyksenä on välttää dynaamista muistinvarausta suorituksen aikana. Syy tähän on se, että muistinkäyttöä ei pyritä ennustamaan pa-

himman tapauksen varalta, jolloin turvallisinta on varata kaikki sovelluksen tarvitsema muisti heti käynnistyksen yhteydessä.

Yleissuunnitteluvaiheen luokkakaavion luokkakandidaattien riittävyyttä arvioidaan sekvenssikaavioiden avulla kuvissa L.8 ja L.9. Sekvenssikaavioissa käydään läpi vastaavat sekvenssit kuin kuvattaessa käyttötapaussekvenssejä mustalaatikkoperiaatteella (kuvat L.2 ja L.3), tosin sillä erotuksella, että nyt DTS Proto on ensimmäisen kerran paloiteltu pienempiin osiin. Sekvenssikaavioiden piirtämisen yhteydessä saadaan samalla liitettyä käyttötapaukset ja luokkakandidaatit toisiinsa, jolla selviää, mitkä luokat osallistuvat kuhunkin käyttötapaukseen. Tätä tietoa voidaan hyödyntää siinä vaiheessa, jos jonkin käyttötapauksen kohdalla ilmenee muutos-/korjaustarpeita myöhemmin. Tällöin voidaan sekvenssikaavion avulla selvittää, mitkä luokat muodostavat käyttötapauksen toiminnallisuuden. Sen jälkeen, kun sekvenssikaaviot osoittavat luokkakandidaattien riittävyyden, voidaan siirtyä yksityiskohtaisen suunnittelun vaiheeseen.

7.2.3 *Communication Subsystem* –luokkakuvaukset

C1553InterfaceControl-luokan tehtävä on muodostaa rajapinta 1553-väylän ja sovelluksen välille. Se on *control*-tyyppinen toimintaa ohjaava luokka ja ainoa luokka, joka käyttää väyläkortin ajuriohjelmiston palveluita. Luokan tehtävä on vastaanottaa 1553-sanomia väylältä ja kirjoittaa sinne lähetettäviä 1553-sanomia. Kun väylärajapintaan saapuu uusia sanomia, *C1553InterfaceControl* välittää sanoman sisältämän datan sen käsittelystä vastaavalle luokalle. Väylälle kirjoitettavat 1553-sanomat *C1553InterfaceControl*-luokka saa muilta luokilta, mutta esimerkiksi väyläsanomien perillemenosta kertovia kuittauksia luokka voi käsitellä itsekin.

Luokka kommunikoi *CDataMsgHandler*-luokan kanssa, joka vastaa lähetettävien ja vastaanotettujen datasanomien sisällöstä. Kun *C1553InterfaceControl*-luokka vastaanottaa datasanoman *CDataMsgHandler*-luokalta, tulee sen kirjoittaa sanoma väylälle mahdollisimman nopeasti. Luokkakaavioon on kuvattu kommunikointi myös *CStatusControl*-luokan kanssa. *CStatusControl*-luokka vastaa DTS Proton ja modeemin tilatietojen välittämisestä MC:lle, mutta sitä ei mallinneta tässä työssä vaan se jätetään seuraaviin ver-

sioihin. Kommunikointi täytyy olla asynkronista, koska 1553-väylän luku- ja kirjoitusoperaatioita ei saa keskeyttää, sillä sanomakohtaisia luku-/kirjoitustaajuuksia on noudatettava. 1553-väylän sanomat päivittyvät sanomakohtaisesti määrätyillä taajuuksilla, riippumatta siitä, ovatko muut väylän käyttäjät lähettäneet uutta dataa.

C1553InterfaceControl-luokka koostuu *C1553Device*-luokasta, joka puolestaan alustaa tarvittavat *C1553Message*-luokan oliot. *C1553Message*-luokka sisältää vain yksittäisen 1553-sanoman tiedot. *C1553Device*-luokka kuvataan *boundary*-tyyppiseksi luokaksi, koska sen kautta käytetään 1553-väyläkortin ajuriohjelmiston palveluita 1553-väylän alustamiseen, lukemiseen ja väylälle kirjoittamiseen.

CDataMsgHandler-luokan tehtävä on tulkita modeemilta tai MC:lta vastaanotettujen sanomien sisältöjä ja huolehtia niihin liittyvistä salausasioista. Luokka on toimintaa ohjaava *Control*-tyyppinen luokka. Luokka koostuu kahdesta algoritmityyppisestä luokasta, *CCoder* ja *CDecoder*, jotka tarjoavat palvelut sanomien salaamiseksi tai salausten purkamiseksi.

CDataMsgHandler kommunikoi asynkronisesti sekä 1553-liikenteestä vastaavan *C1553InterfaceControl*-luokan kanssa, kuin myös modeemirajapinnan liikenteestä vastaavan *CModemInterfaceControl*-luokan kanssa. Kommunikointimekanismit määritellään myös tälle välille vasta yksityiskohtaisen suunnittelun vaiheessa. Jo tässä vaiheessa on hyvä huomata, että ***CDataMsgHandler***-luokan tehtävien prioriteetit eivät ole yhtä korkealla tasolla kuin rajapintoja käsittelevien luokkien. Tämä seuraa siitä, että 1553-väyläoperaatiot täytyy pystyä suorittamaan sanomakohtaisten taajuuksien asettamissa rajoissa ja modeemille tulee vastata alle 50 ms:ssa sen jälkeen, kun tilakysely on saapunut (ks. kuva L.3).

CModemInterfaceControl-luokka vastaa modeemirajapinnan vastaanotto- ja lähetysoperaatioista. Se vastaanottaa modeemirajapinnassa olevien sanomajonojen kautta MC:lle välitettäviä datasanomia ja lähetystiedonkyselyitä. Lisäksi se lähettää rajapintaan vasteita lähetystiedonkyselyihin, joko uudella datasanomalla tai ”ei lähetettävää”-tiedolla. Datasanoman saapuessa modeemilta *CModemInterfaceControl* välittää sen välittömästi *CDataMsgHandler*-luokalle, joka purkaa sanoman salauksen ja välittää sen eteenpäin *C1553InterfaceControl*-luokalle. *CModemInterfaceControl*-luokka ei itse tulkitse

sanomien sisältöjä, koska sen on ehdittävä vastaamaan myös lähetystiedonkyselyhin vasteaikarajoissa. Tästä aiheutuu, että modeeminrajapinnan operaatiot täytyy hoitaa korkeammalla prioriteetilla kuin sanomien sisällön käsitteleminen. Siten modeemirajapinnan käsittelyyn tarvitaan vähintään yksi oma suoritussäie.

Luokka kommunikoi asynkronisesti *CDataMsgHandler*-luokan kanssa. *CModemInterfaceControl*-luokka koostuu *MessageQueue*-luokista, joka ovat modeemin ja DTS Proton välisiä sanomajonoja ja tarjoavat sanomajonopalvelut. *CModemInterfaceControl* ohjaa sitä, milloin sanomajonoista vastaanotetaan tai niihin kirjoitetaan sanomia.

7.2.4 Yhteenveto yleissuunnitteluvaiheesta

Yleissuunnitteluvaiheen jälkeen pitäisi olla saavutettuna käsitys siitä, mikä on järjestelmän arkkitehtuuri ylemmällä tasolla ilman toteutukseen liittyviä yksityiskohtia. DTS Proton kohdalla ensimmäisenä suoritettiin sovelluksen jakaminen alijärjestelmiin. Tämä oli suhteellisen suoraviivaista, koska se oli hahmotettavissa melko suoraan käyttötapausten toiminnallisuuksista. Kun DTS Proton alijärjestelmät oli selvillä, voitiin valita tiedonsiirrosta vastaava alijärjestelmä tarkemman mallintamisen kohteeksi. Vaatimusmäärittelyn ja teknisten rajoitteiden (esim. rajapintamäärittelyt) perusteella muodostettiin luokkarakenne, jota testattiin muutamalla sekvenssikaaviolla. Mallina käytettiin luvussa 6 esitettyä menettelytapaa hahmotella järjestelmän staattinen rakenne (SMV-näkymä) ja testata sen riittävyyttä toiminnalliselta kannalta (BMV-näkymä). Lisäksi määriteltiin sanallisesti luokkien vastuut ja tärkeimmät koostesuhteet, joiden perusteella luokkien sisäistä toteutusta lähdetään yksityiskohtaisen suunnittelun vaiheessa tarkentamaan.

7.3 DTS Proto -sovelluksen yksityiskohtainen suunnittelu

Yksityiskohtainen suunnittelu lähtee liikkeelle yleissuunnitteluvaiheen tuottaman luokkakaavion avulla. Tavoitteena on tuottaa toteutuskelpoinen luokkarakenne, jonka avulla koodin implementointi onnistuu mahdollisimman suoraviivaisesti. Tämä tapahtuu tarkentamalla yksittäisten luokkien rakennetta käyttäen apuna yleissuunnitteluvaiheen sekvenssikaavioita, joista voidaan kerätä luokkiin liittyviä operaatioita. Lisäksi

tarkennetaan luokkien välisiä assosiaatioita, tutkitaan mitä luokkia löytyy valmiina ja määritellään luokkien väliset kommunikointimenetelmät. Tässä vaiheessa myös otetaan käyttöön aktiiviset luokat, jotka ovat itsenäisiä suoritussäikeitä, joilla on oma prioriteetti. Niistä luodut oliot ohjaavat sulautetun järjestelmän toimintaa ja päättävät milloin operaatioita suoritetaan. Näiden aktiivisten olioiden toimintasekvenssit mallinnetaan aktiviteettikaavioiden avulla.

7.3.1 Luokkien yksityiskohtainen suunnittelu

Yleissuunnitteluvaiheessa tuotettiin ensimmäinen kuvaus DTS Proton luokkakaaviosta. Sen todettiin riittävän käyttötapauksissa esitettyjen toiminnallisten vaatimusten toteuttamiseen. Yksityiskohtaisen suunnittelun tavoitteena on tuottaa toteutuskelpoinen luokkakaavio, joka kuvaa kommunikointimenetelmät, säieluokat (eli taskit), luokkien metodit ja attribuutit.

Kuvassa L.10 kuvattu yksityiskohtainen luokkakaavio DTS Proton rakenteesta. Se on staattinen kuvaus sovelluksen SMV-näkymästä. Luokkakaaviossa on pyritty laatimaan toteutuskelpoinen malli yleissuunnitteluvaiheen luokista. Kaaviossa kuvataan luokkien välisestä kommunikoinnista sekä niiden suorituksesta vastaavat luokat. Tilakyselyistä vastaava *CStatusControl*-luokka jätetään tässä versiossa toteuttamatta. Kommunikointi toimintaa ohjaavien kontrolliluokkien välillä tapahtuu käyttöjärjestelmän tarjoamien sanomajonoluokkien avulla. Käyttöjärjestelmäksi valittu VxWorks tarjoaa sanomajonoluokan toteutuksen valmiina, joten tässä työssä päädyttiin niiden hyödyntämiseen [VxW99b]. Luokka mahdollistaa myös tiedonsiirron asynkronisesti suoritussäikeiden välillä.

Jokaisella toimintaa ohjaavalla luokalla on vähintään yksi sanomajono tiedon vastaanottamista varten. Lisäksi *CModemInterfaceControl*-luokalla on sanomajonot jaetun muistin alueella, joiden kautta se kommunikoi toisella prosessorikortilla toimivan modeemin kanssa. Niiden jonojen kautta DTS Proto vastaanottaa datasanomia ja lähetystiedonkyselyjä modeemilta sekä lähettää niihin vasteita. Jonoista toinen on vastaanottoa ja toinen lähetystä varten. Assosiaatiot sanomajonojen ja toimintaa ohjaaviin luokkien välillä ovat vahvoja koosteita. Tämä tarkoittaa sitä, että ne luodaan ja tuhotaan samaan aikaan ne omistavan

olion kanssa. Sama koskee myös säieluokkia. Tämä perustuu siihen, että kaikki suorituksen aikana tarvittava muisti varataan heti käynnistyksen yhteydessä, koska muistinkulutuksen pahinta tapausta ei pyritä ennustamaan. Sanomajonojen assosiaatioihin on kuvattu myös kunkin sanomajono-olion rooli assosiaatiossa. Rooleilla pyritään esittämään se, minkä luokkien välillä kyseinen sanomajono välittää tietoa. Lisäksi assosiaatio lukumäärillä ilmaistaan kuinka monta oliota kyseistä roolia hoitaa.

C1553Device-luokka määrittelee rajapinnan 1553-väylälle ja kapseloi väyläsanomien osoitteet. Se muodostaa käyttöjärjestelmäkutsujen avulla palvelut väyläsanomien luku- ja kirjoitusoperaatioille.

7.3.2 Säikeet eli tehtävät

Kaikilla toimintaa ohjaavilla luokilla on vähintään yksi tehtäväsäie. Säieoliot luodaan käyttöjärjestelmän tarjoamasta *VxWTask*-luokasta [VxW99b]. Kukin *VxWTask*-olion omistajaluokka määrittelee staattisen funktion, jota säie suorittaa ja joka toteuttaa kuvissa L.11-L.16 esitetyt toimintosekvenssit. Toimintosekvenssit kuvaavat järjestelmän sisäistä käyttäytymistä eli BMV-näkymää. Assosiaatiot omistajaluokkien ja säieluokkien välillä esitetään kahdensuuntaisina, koska säikeen täytyy tuntea myös omistajaluokan staattinen funktio sekä tarvittaessa omistajaolion osoite voidakseen muuttaa sen attribuuttien arvoja tai kutsuakseen oliokohtaisia aliohjelmia. Seuraavassa määritellään kunkin säikeen tehtävät sanallisesti.

C1553InterfaceControl-luokan säikeet muodostetaan luokan vastuiden perusteella, joita ovat 1553-sanomien lukeminen ja kirjoittaminen. Kunkin 1553-sanoman lukemisesta vastaa erillinen säie, mutta sanomien *Msg3* ja *Msg5* päivittämisestä vastaan yksi yhteinen säie. Kuvauksissa esiintyvät sanomakohtaiset päivitystaajuudet ja 1553-sanomien bittien merkitykset oletetaan määritellyksi rajapintamäärittelyissä.

Msg2ReaderTask: Säikeen tehtävä on lukea sanomassa *Msg2* saapuvia vastaanottokuittauksia. MC kuittaa datasanoman vastaanotetuksi vaihtamalla sanoman *Msg2* ensimmäisen bitin tilaa. *Msg2ReaderTask* ylläpitää tietoa bitin viimeisimmästä arvosta ja bitin vaihtuessa se asettaa *ReceiveComplete*-lipun arvoksi *true*. Tämän jälkeen datasanomia 1553-

väylälle kirjoittava säie voi lähettää seuraavan 1553-sanoman, tietysti sillä edellytyksellä, että uusia sanomia on lähettävänä. *Msg2ReaderTaskin* prioriteetin tulee olla niin korkea, että se pystyy lukemaan sanomia sanomakohtaisen taajuuden edellyttämällä nopeudella.

Msg4ReaderTask: Säie lukee 1553-väylän sanomaa *Msg4*, jossa vastaanotetaan MC:n lähettämät datasanomat. Säie odottaa sanoman saapumista rajapintaan *waiting*-tilassa. Kun uusi sanoma saapuu, *Msg4ReaderTask*-säie lukee sanoman heti seuraavalla suoritusvuorollaan ja tarkistaa sanoman oikeellisuuden sekä lähettää sen *CDataMsgHandler*-luokan koodattavaksi. Sanomaa ei tarvitse välittää eteenpäin, jos sanoman ensimmäinen validista datasanomasta ilmoittava bitti ei ole asettuneena. Kun sanoma on välitetty eteenpäin (tai hylätty, jos siinä esiintyi virheitä), siirtyy säie takaisin *waiting*-tilaan odottamaan seuraavaa sanomaa. Tämän säikeen täytyy saada suoritusvuoro vähintään sanoman *Msg4* päivitystaajuuden määräämällä taajuudella.

Msg3_5WriterTask: Sanomassa *Msg3* lähetetään MC:lle uusi datasanoma. Datasanoma vastaanotetaan *CDataMsgHandler*-luokalta, joka on purkanut modeemilta vastaanotetusta datasanomasta bitit MC:n ymmärtämään muotoon. *Msg3*-sanoman lisäksi säie kirjoittaa myös sanomaan *Msg5* tiedon uudesta validista datasanomasta. Tämä tapahtuu vaihtamalla sanoman *Msg5* ensimmäisen bitin tilaa. Sen perusteella MC tietää lukea sanoman *Msg3* sisällön. *Msg3WriterTask* voi olla kahdessa eri *waiting*-tilassa. Se voi odottaa joko *ReceiveComplete*-lipun vaihtumista arvoon *true* tai sitten uutta datasanomaa *CDataMsgHandlerilta*. *ReceiveComplete*-lippu ilmaisee MC:n vastaanottaneen edellisen datasanoman. Säie ei saa lähettää seuraavaa datasanomaa ennen kuin MC on kuitannut edellisen vastaanotetuksi. Säikeen täytyy toimia niin korkealla prioriteetilla, että modeemilta vastaanotetut sanomat saadaan toimitettua MC:lle.

CDataMsgHandler-luokan tehtävä on koodata ja dekoodata vastaanotettuja sekä lähettäviä datasanomia. Koska tehtävät ovat toisistaan riippumattomia ja toimivat eri herätteistä, jaetaan ne kahdelle eri säikeelle, jotka esitellään seuraavassa:

TxDataMsgHandlerTask: Säie odottaa *waiting*-tilassa 1553-rajapinnasta vastaanotettuja datasanomia. Kun *C1553InterfaceControl*-luokka lähettää *CDataMsgHandlerin* sanomajonoon uuden datasanoman, *TxDataMsgHandlerTask* lukee sanoman ja pyytää *CCoder*-

luokkaa koodaamaan sen. Kun sanoma on koodattu, lähetetään se *CModemInterfaceControl*-luokan sanomajonoon odottamaan seuraavaa lähetystiedonkyselyä. Säikeen prioriteetti ei saa hidastaa 1553-väyläoperaatioita eikä lähetystiedonkyselyihin vastaamista. Ne sanomat, joita ei ehditä käsitellä, hylätään. Vain viimeisin datasanoma käsitellään.

RxDataMsgHandlerTask: Tämä säie toimii lähes samoin kuin *TxDataMsgHandlerTask*, mutta se käsittelee modeemilta tulevia sanomia. Säie odottaa *waiting*-tilassa modeemilta vastaanotettuja datasanomia. Kun *CModemInterfaceControl*-luokka lähettää *CDataMsgHandlerin* sanomajonoon uuden datasanoman, *RxDataMsgHandlerTask* lukee sanoman ja pyytää *CDecoder*-luokkaa purkamaan sen salauksen. Kun sanoman sisältö on käsitelty, se lähetetään *CModemInterfaceControl*-luokan sanomajonoon odottamaan seuraavaa lähetystiedonkyselyä. Tämän säikeen prioriteetti vastaa *TxDataMsgHandlerTask*-prioriteettia.

CModemInterfaceControl-luokka vastaa datasanomien ja lähetystiedon vastaanottamisesta modeemirajapinnan sanomajonoista sekä lähetystiedonkyselyihin vastaamisesta vaatimusmäärittelyn asettamissa aikarajoissa. Koska kaikki modeemilta tulevat sanomat vastaanotetaan samasta sanomajonosta, suoritetaan tämän luokan tehtävät yhdessä säikeessä, joka esitellään seuraavassa:

ModemInterfaceControllerTask: Säie suorittaa tehtäviä vain modeemin herätteistä. Se siirtyy *waiting*-tilasta suoritustilaan heti, kun modeemirajapinnan vastaanottojonoon saapuu sanoma. Säie tulkitsee sanoman tyypin ja toimii sen tyypin edellyttämällä tavalla. Jos sanoman tyyppi on datasanoma, se välitetään heti *CDataMsgHandler*-luokalle, jotta *ModemInterfaceControllerTask* olisi mahdollisimman pian valmis käsittelemään seuraavaa palvelupyyntöä. Jos sanoman tyyppi on lähetystiedonkysely, säie tarkistaa välittömästi onko lähettäviä datasanomia sanomajonossa. Jos datasanoma löytyy, vastaa säie modeemille lähettämällä sanoman. Jos datasanomia ei ole, vastaa säie modeemille tiedolla ”ei lähetettävää”. Vaste lähetystiedonkyselyyn tulee antaa vaatimusmäärittelyssä määrätyn aikarajoitteen aikana. Aikarajoitteen noudattaminen on kriittistä, jolloin tämän säikeen prioriteetti täytyy olla kaikkien muiden säikeiden prioriteetteja korkeampi. Modeemiraja-

pinnan sanomien sisäinen formaatti oletetaan määritellyksi DTS Proton ja modeemin välisessä rajapintamäärittelyssä.

7.3.3 Tehtävien prioriteetit

Edellisten kuvausten perusteella määritellään säikeille prioriteetit. Käyttöjärjestelmäksi valitussa VxWorks-käyttöjärjestelmässä voidaan *VxWTask*-luokan säikeille antaa prioriteettiarvoja väliltä 0-255 [VxW99a]. Seuraavassa on lueteltu DTS Proton tiedonsiirtoon osallistuvien säikeiden prioriteetit:

ModemInterfaceControllerTask:	10
Msg2ReaderTask:	30
Msg4ReaderTask:	30
Msg3_5WriterTask:	35
TxDataMsgHandlerTask:	50
RxDataMsgHandlerTask:	50

Prioriteettiarvojen välit on määritelty suuriksi, koska jatkossa sovelluksen toiminnallisuutta kehitettäessä tarvitaan lisää prioriteettiarvoja ja osa niistä arvoista voi sijoittua nykyisten arvojen väleihin. Nykyisen priorisoinnin tärkein tehtävä on varmistaa nopea vasteaika lähetystiedonkyselyihin ja riittävän nopea 1553-väyläsanomien lukeminen. Tämä pyritään toteuttamaan antamalla näille toiminnoille korkeimmat prioriteetit.

7.3.4 Yhteenveto yksityiskohtaisesta suunnittelusta

Yksityiskohtainen suunnittelu sai syötteeksi karkean luokkakaavion yleissuunnitteluvaiheesta, jossa oli määritelty toimintaa ohjaavat luokat sekä niiden väliset riippuvuudet. Lisäksi rajapinnat sovelluksesta ulospäin oli määritelty. Toimintaa ohjaavien luokkien toteutuksen määrittely edellytti luokkakaavion tarkentamista. Se onnistui tarkastelemalla yleissuunnitteluvaiheen luokille määriteltyjä vastuita.

Luokkien tehtävien perusteella pystyttiin suunnittelemaan suoritusta ohjaavat säikeet. Suoritussäikeet johdettiin edelleen käyttötapausten perusteella tehdyistä yleissuunnitteluvai-

heen sekvensseistä tarkastelemalla sitä, mistä tehtävistä kukin luokka vastasi käyttötapauksissa. Näin saatiin edelleen pidettyä jonkinasteista jäljitettävyyttä säikeiden toimintosekvenssien, luokkakaavioiden ja käyttötapausten välillä. Tosin saman asian jäljittäminen yksityiskohtaisesta luokkakaaviosta käyttötapauksiin päin ei ole varmastikaan yhtä suoraviivaista. Aikavaatimukset pystyttiin joka tapauksessa tuomaan käyttötapauksista toteutuskelpoisiin säikeisiin asti, jolloin kuvausmenetelmän voidaan todeta toimineen onnistuneesti tässä suhteessa. Säikeet ja sanomajonot voidaan toteuttaa suoraan käyttöjärjestelmän tarjoamista luokista, jolloin voidaan hyödyntää valmista koodia ja säästää resursseja toteutusvaiheessa.

Kokonaisuutena luokkakuvaus ja aktiviteettikaavioihin kuvatut toimintosekvenssit antavat melko hyvän mallin sovelluksen toteutusvaiheelle. Sovelluksen sisäistä toiminnallisuutta kuvaavaa näkökulmaa (BMV-näkymä) voitaisiin tarkentaa piirtämällä vielä tässäkin vaiheessa sekvenssikaavioita, mutta koska tavoitteena on etsiä vain välttämättömimmät kaaviot ja välttää ylimääräistä kuvaamista, ne jätettiin toteuttamatta. Toiminnallisuuden kuvaamisessa päätettiin jäädä aktiviteettikaavioiden tasolle, koska ne sisältävät kaiken käyttötapauksiin liittyvän toiminnallisuuden. Lisäksi niillä on mahdollista kuvata koodin toiminnallista rakennetta (kuten ehtolauseet, silmukat ja tilat).

7.4 Yhteenveto DTS Proton mallinnusvaiheista

Luvun yhteenvetona esitetään muutama huomio siitä, kuinka DTS Proton mallinnus eteni luvussa 6 esitetyn prosessimallin ja UML:n avulla. Liikkeelle lähdettiin järjestelmän ympäristöstä kontekstikaavion avulla. Sen jälkeen kerättiin vaatimukset ja tehtiin niiden pohjalta käyttötapaukset. Käyttötapauksia määriteltiin sanallisesti sekä sekvenssikaavioiden avulla, joihin saatiin liitettyä aikavaatimukset. Järjestelmä pysyi näissä kaikissa vaiheissa kuitenkin mustana laatikkona.

Kun käyttötapaukset oli määritelty, ”avattiin” musta laatikko ensimmäisen kerran ja jaettiin järjestelmä toiminnallisuuden perusteella alijärjestelmiin, joista yksi valittiin mallinnuksen kohteeksi. Tälle alijärjestelmälle tehtiin ensimmäinen karkean tason luokkakaavio,

jonka toimivuutta testattiin sekvenssikaavioilla. Tämä vaihe saattaa vaatia useampia iteraatiokierroksia.

Kun yleissuunnitteluvaiheen luokkakaavio todettiin toimivaksi, siirryttiin yksityiskohtaisen suunnittelun vaiheeseen, jossa luokkakaavion toteutusrakennetta tarkennettiin ja määriteltiin säikeet suoritussekvensseineen ja prioriteetteineen. Tämän jälkeen oli valmiina toteutuskelpoinen luokkakaavio ja suoritusta kuvaavat aktiviteettikaaviot.

8 Työn tulosten tarkastelua

Tämän opinnäytetyön tavoitteena oli tutkia UML-notaation ja oliokeskeisen lähestymistavan soveltuvuutta sulautetussa ympäristössä toimivan reaaliaikaisen tiedonsiirtosovelluksen määrittelyyn ja suunnitteluun. Työn aluksi luvussa 2 tarkasteltiin yleisesti sulautettujen järjestelmien piirteitä ja pyrittiin selvittämään sitä, kuinka ne eroavat ns. tavanomaisista tietokoneista. Luvussa 3 perehdyttiin reaaliaikajärjestelmien vaatimuksiin ja ominaisuuksiin. Reaaliaikajärjestelmien tyypillisimpiä ominaisuuksia ovat oikea-aikaisuus ja rinnakkaisuus. Näiden hallitseminen edellyttää prosessien skedulointia, priorisointia ja täsmällistä resurssien hallintaa, joilla vältetään reaaliaikajärjestelmille tyypilliset ongelmatilanteet (kuten aikarajoitteista myöhästymiset ja *deadlock*-tilanteet). Luvussa 3 kuvailtiin myös menetelmiä ja algoritmeja näiden vaatimusten toteuttamiseen sekä sitä, mitä ominaisuuksia nykyaikaiset reaaliaikakäyttöjärjestelmät tarjoavat. Esimerkkinä käytettiin lähinnä DTS Protonkin alustana käytettävää VxWorks-käyttöjärjestelmää [VxW99a, VxW99b].

Reaaliaikakäyttöjärjestelmät tarjoavat nykyään monia reaaliaikasovellusten tarvitsemia palveluita, kuten skedulointi-, kommunikointi- ja synkronointimenetelmiä. Lisäksi nykypäivänä varsinaisen laitteistoläheisen ohjelmoinnin tarve on melko vähäistä, koska käyttöjärjestelmät piilottavat laitteistoyksityiskohdat. Tämän lisäksi käyttöjärjestelmät ovat usein skaalattavia, jolloin uusia laiteresursseja lisättäessä voidaan käyttää laitetukiohjelmistoja helpottamaan niiden ohjelmointia. Luku 4 käsitteli olioajattelun perusasioita, joita käytettiin kuitenkin melko rajallisesti DTS Proto -sovelluksen mallintamisessa. Luvun tarkoitus oli kuitenkin tuoda esille oliokeskeisyyden mahdollisuudet. Sovellusympäristö ei kuitenkaan tuonut esiin tarpeita läheskään kaikille oliomallinnuksen tarjoamille mahdollisuuksille. Luvussa 5 perehdyttiin UML:n kaaviovalikoimaan esimerkkien avulla. Samalla pyrittiin löytämään mahdollisuuksia kuvata reaaliaikaominaisuuksia kaavioiden avulla. Apuna käytettiin myös lähdekirjallisuudessa esitettyjä kokemuksia reaaliaikajärjestelmien mallintamisesta.

Luvussa 6 esiteltiin useista eri lähteistä yhteenvetona kehitetty prosessimalli reaaliaikajärjestelmän mallintamiseksi UML-kuvausmenetelmän avulla. Prosessia voidaan pitää yksin-

kertaistettuna vaihtoehtona OMT++-tyyliselle prosessille. Prosessimallin yhteydessä pyrittiin kuvaamaan myös sitä, mistä näkökulmista sovellusta/järjestelmää kannattaisi eri vaiheissa kuvata ja mitkä UML-kaaviot voisivat tukea näiden näkymien kuvaamista.

Luvussa 7 esiteltiin yksinkertaistetun tiedonsiirtosovelluksen määrittely- ja suunnittelutyön tuloksia luvussa 6 esitetyn prosessimallin avulla. Vaatimusmäärittelyvaiheessa käytettiin apuna käyttötapauskaavioita. Käyttötapausten määrittely sulautetussa ympäristössä toimivalle sovellukselle ei ollut kovin suoraviivaista. Käyttötapausten määrittelyn ohella harkittiin myös ns. EERL-listojen käyttöä vaatimusmäärittelyssä, koska niillä voidaan kerätä kaikki järjestelmän ulkoiset syötteet ja vasteet yhteen listaan, jolloin vaatimukset saadaan yksilöityä numeroiduksi listaksi. EERL-listojen käyttöä on esitelty mm. lähteissä [Ell94, Dou99]. Sovelluksen toimintaympäristön kuvaaminen onnistui hyvin sijoittelukaavioilla. Ulkoiset järjestelmän *black box*-mallille asetetut aikarajoitteet kuvattiin sekvenssikaavioilla mallinnettuihin käyttötapaussekvensseihin, jolloin vasteaikavaatimukset saatiin vaatimusmäärittelyn kuvauksiin mukaan.

Vaatimusmäärittelyn jälkeen siirryttiin yleissuunnitteluvaiheeseen, jossa tavoitteena oli jakaa DTS Proto alijärjestelmiin ja tuottaa karkean tason luokka-arkkitehtuuri sovellukselle. Jatkuvuuden löytäminen vaatimusmäärittelyvaiheen tuotosten ja oliopohjaisen yleissuunnitteluvaiheen välille osoittautui jonkin verran hankalaksi. Lähtökohdaksi otettiin sovelluksen jakaminen käyttötapausten perusteella alijärjestelmiin ja olioanalyysin mukainen luokkakandidaattien määritteleminen. Luokkakandidaateille pyrittiin määrittelemään sellaiset vastuut, että ne kattaisivat valitut käyttötapaukset. Tätä voitiin testata sekvenssikaavioiden avulla.

Järjestelmän rajapintojen sekä niissä kulkevien syötteiden ja vasteiden mallintamiseen UML ei tarjoa välttämättä parhaita menetelmiä. Jatkossa vertailua voisi suorittaa esimerkiksi prosessikeskeisemmän SDL-kuvauskielen kanssa. UML:n ja SDL:n yhteiskäyttöä esitellään mm. lähteessä [DDJ01]. UML ja oliokeskeisyys ohjaa suunnittelua helposti melko staattiseksi, jolloin rinnakkaisuuden huomioiminen jää myöhäisempään vaiheeseen, jolloin sen sovittaminen arkkitehtuuriin ei välttämättä olekaan enää suoraviivaista. DTS Proton kohdalla siinä onnistuttiin, mutta sovellus on melko rajattu ja siinä suoritettavat

tehtävät ovat itsenäisiä suoritussekvenssejä, mikä tekee suunnittelusta kohtuullisen helppoa. Rinnakkain suoritettavia tehtäviä pyrittiin kuvaamaan aktiviteettikaavioiden avulla. Säikeet mallinnettiin yksityiskohtaisen suunnittelun vaiheessa aktiivisiksi luokiksi (stereotyyppi `<<active>>`). Luokkakaavioiden toimivuutta olisi voitu tarkastella lisäksi sekvenssikaavioiden avulla, mutta liiallisen kuvaamisen välttämiseksi se jätettiin tekemättä.

UML-kuvauskielen ja oliokeskeisen suunnittelun pohjalta pystytään tuottamaan myös reaaliaikasovelluksia, mutta lähestymistapa on melko raskas ainakin tässä työssä mallinnetun tiedonsiirtosovellustyypin kohdalla. Kaavioiden määrä kasvaa helposti todella suureksi, jolloin niiden ylläpitäminen on aikaavievää, ja toisaalta niiden ylläpitämättömyys saattaa johtaa ristiriitaisiin tulkintoihin. Tässä työssä käytetyn melko yksinkertaisen esimerkkisovelluksenkin mallintamisessa tuotettujen kaavioiden määrä on melko suuri. On huomioitava, että sovelluksesta mallinnettiin vain yksi alijärjestelmä kolmesta ja sekin vain olennaisimmilta osilta. Lisäksi joissakin kohdin jouduttiin soveltamaan UML-standardia, mikä osaltaan tuottaa erilaisten tulkintojen mahdollisuuksia kaavioihin.

UML-notaatioon on esitetty parannusehdotuksia reaaliaikajärjestelmien kuvaamisen osalta lähteessä [OMG01]. Suurempien sulautettujen reaaliaikajärjestelmäkokonaisuuksien mallintamiseen UML:n avulla löytyy esimerkkejä mm. lähteissä [Dou98, Dou99, Gom00]. Lähteissä esitettyjen esimerkkien perusteella UML soveltuu ainakin tietylle tasolle asti reaaliaikajärjestelmän mallintamiseen, mutta esim. DTS Proton kaltaisen yksinkertaisemman tiedonsiirtosovelluksen kohdalla voisi olla järkevä soveltaa ”kevyempää” suoraviivaisemmin tiedonprosessoinnin mallintamiseen keskittyvää menetelmää. UML-kuvauskielen kaaviovalikoima on melko suuri ja siitä seuraa myös melko suuret kustannukset mallinnusvälineiden hankinnassa. Toisaalta, kuten työssä on aiemmin mainittu, UML-mallinnustyökaluista löytyy myös ilmaisia vaihtoehtoja kuten esimerkiksi [ARG02]. Tässä esitetyt kuvaukset on tuotettu Rational Rose -mallinnusohjelmistolla. Rational Rosesta on olemassa myös reaaliaikaversio nimeltä Real Time Rose, mutta sen soveltuvuutta ei tässä yhteydessä tarkasteltu.

Yhteenvetona työn tuloksista voidaan todeta UML:n mahdollistavan sulautettujen reaaliaikajärjestelmien mallintamisen kohtuullisen hyvin. Tärkeää on kuitenkin valita sopiva pro-

sessimalli ja vain tarpeelliset kuvaustavat. Kriittisesti asiaa tarkastellessa UML-kuvauskieltä ja oliokeskeistä mallinnusta voidaan pitää ehkä hieman raskaana lähestymistapana pienten sulautettujen ohjelmistojen mallintamiseen, koska kaikkia kaavioita ja olio-ominaisuuksia ei välttämättä tarvita suoraviivaisen prosessoinnin ja sen vaatiman ohjelmistoarkkitehtuurin mallintamiseen. Lisäksi liiallinen yritys kuvata sovellusta useilla eri kaavioilla johtaa helposti tilanteeseen, jota voisi kuvata sanalla ”kaaviokaaos”. Tällöin kaavioiden ylläpitäminenkin saattaa osoittautua liian vaativaksi tehtäväksi. Kuitenkin etukäteen mietityillä kaaviovalinnoilla voidaan joka tapauksessa UML:stä saada reaaliaikasovellustenkin kehitykseen selvää hyötyä.

9 Lähteet

[ARG02] ArgoUML project, http://www.ArgoUML.org, luettu 10.10.2002.

[Awa96] Awad Maher, Kuusela Juha, Ziegler Jurgen, Object-Oriented Technology for Real-Time Systems: A Practical Approach Using OMT and Fusion, Prentice Hall, 1996.

[Bal00] Stuart R. Ball, Embedded Microprocessor Systems, Real World Design, Butterworth-Heinemann, 2000.

[Bar99] Barr Michael, Programming Embedded Systems in C and C++, O'Reilly & Associates, January 1999.

[Bec97] Becks Ari, Opeta itsellesi delphi-ohjelmointi, Suomen Atk-kustannus Oy, 1997.

[Boo94] Booch Grady, Object-Oriented Analysis and Design, Benjamin Cummings, 1994.

[Ced02] http://www.cedmagic.com/history, 28.01.2002.

[DDJ01] Björkander Morgan, Programming in SDL & UML, s. 93-99, Dr.Dobbs Journal, June 2001.

[Dij65] Dijkstra E.W. Solution of a problem in concurrent program control, Communication of the ACM, 1965.

[Dou98] Douglass Bruce Powell, Real-Time UML: Developing Efficient Objects for Embedded Systems, Addison Wesley Longman, 1998.

[Dou99] Douglass Bruce Powell, Doing Hard Time: Developing Real-Time Systems with UML, Objects, Framework, and Patterns, Addison Wesley, 1999.

[DY402] DY4 Systems, Datasheets, http://www.dy4.com, 07.07.2002.

[Ell94] Ellis John R., Objectifying Real-Time Systems, SIGS Books, April 1994.

[Emb02] 2002 Embedded Linux Consortium, http://www.embedded-linux.org/, 13.06.2002.

[Gal95] Gallmeister Bill O., POSIX.4: Programming for the Real World, O'Reilly & Associates, 1995.

[Gom00] Gomaa Hassan, Designing Concurrent, Distributed, and Real-Time Applications with UML, Addison Wesley, 2000.

[Hie01] Hietanen Päivi, C++ ja olio-ohjelmointi, Docendo Finland Oy, 2001.

[Int02] Intel museum, Processor Hall Of Fame, http://www.intel.com/intel/intelis/museum/exhibit/hist_micro/hof/tspecs.htm, 28.01.2002.

[Jac92] Jacobson Ivar, Christerson Magnus, Jonsson Patrik, Övergaard, Object-Oriented Software Engineering, A Use Case Driven Approach, Addison-Wesley, 1992.

[Lap96] Lappalainen Vesa, Lahdelma Risto, Olio-ohjelmointi ja C++, Jyväskylän yliopisto, Matematiikan laitos, 1996.

[Lep99] Leppänen Mauri, Oliokeskeinen tietojärjestelmien kehittäminen, luentomoniste, Tietojenkäsittelytieteiden laitos, Jyväskylän yliopisto, 1999.

[Liu00] Liu Jane W.S., Real-Time Systems, Prentice-Hall, 2000.

[Men02] Mentor Graphics Corporate Overview, VRTX Real-Time Operating System, http://www.mentor.com/embedded/vrtxos/, 13.06.2002.

[OMG01] Response To The OMG RFP for Schedulability, Performance And Time, OMG Document Number ad/2001-06-14, June 2001.

[Pre00] Pressman Roser S., Software Engineering, Practioner's Approach, European Adaption, Fifth Edition, McGraw-Hill International, 2000.

[Pro01] Snellman Henrik, Reaaliaikaiset käyttöjärjestelmät, Prosessori-lehti, s.50 – 57, 11/2001.

[Qua98] Quatrani Terry, Visual Modelling with Rational Rose and UML, Addison Wesley Longman, 1998.

[Rad02] Radisys corporation 2002, http://www.radisys.com/oem_products/, 13.06.2002.

[Rat02] Crain Anthony, The Rational Edge, Dear Dr. Use Case: *Is the Clock an Actor?*, Rational Software, 2002, www.therationaledge.com, 17.06.2002.

[Rin00] Rintala Matti, Jokinen Jyke, Olioiden ohjelmointi C++ :lla, Satku, 2000.

[Rum91] Rumbaugh J, Object-Oriented Modelling and Design, Prentice-Hall, 1991.

[Sel94] Selic Bran, Gullekson Garth, Ward Paul T., Real-Time Object-Oriented Modelling, John Wiley & Sons, 1994.

[Str95] Stroustrup Bjarne, The C++ programming language, AT&T Bell Telephone Laboratories, 1995.

[Tan92] Tanenbau Andrew, Modern Operating Systems, Prentice-Hall Inc, 1992.

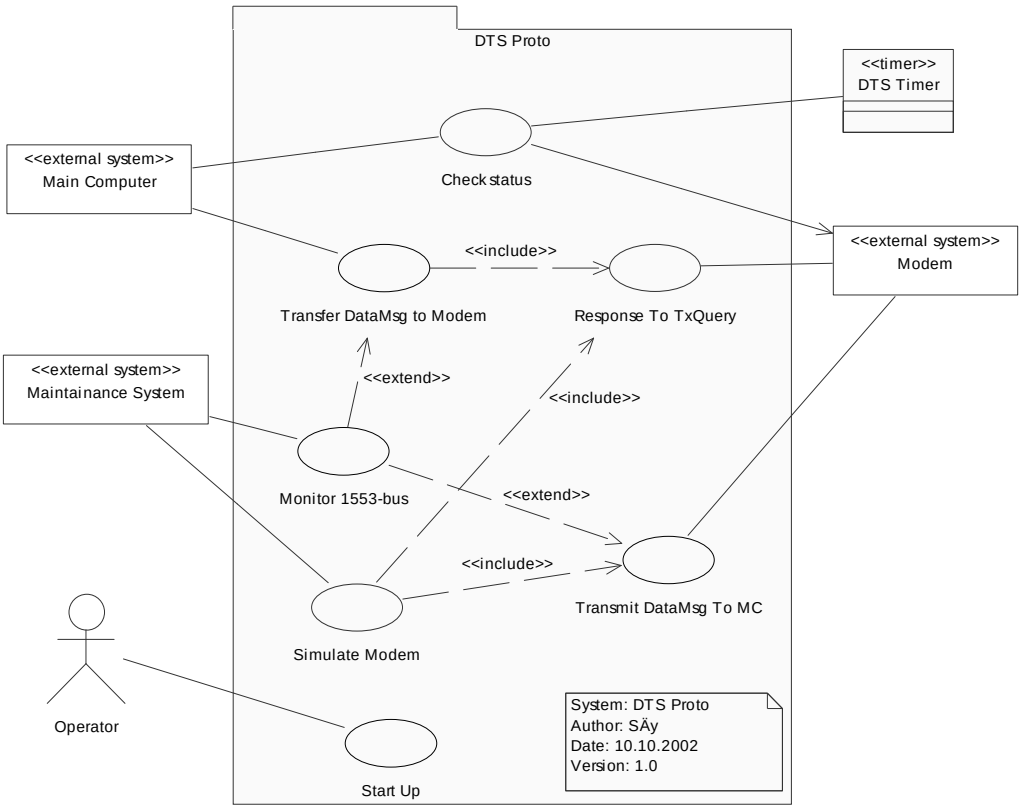
[Vuo01] Vuori Jarkko, TIE342 Real-Time Systems kurssin luentomoniste, Jyväskylän yliopisto, Tietotekniikan laitos, 2001.

[VxW99a] VxWorks, Programmer's Guide 5.4, Edition 1, Wind River Systems, 1999.

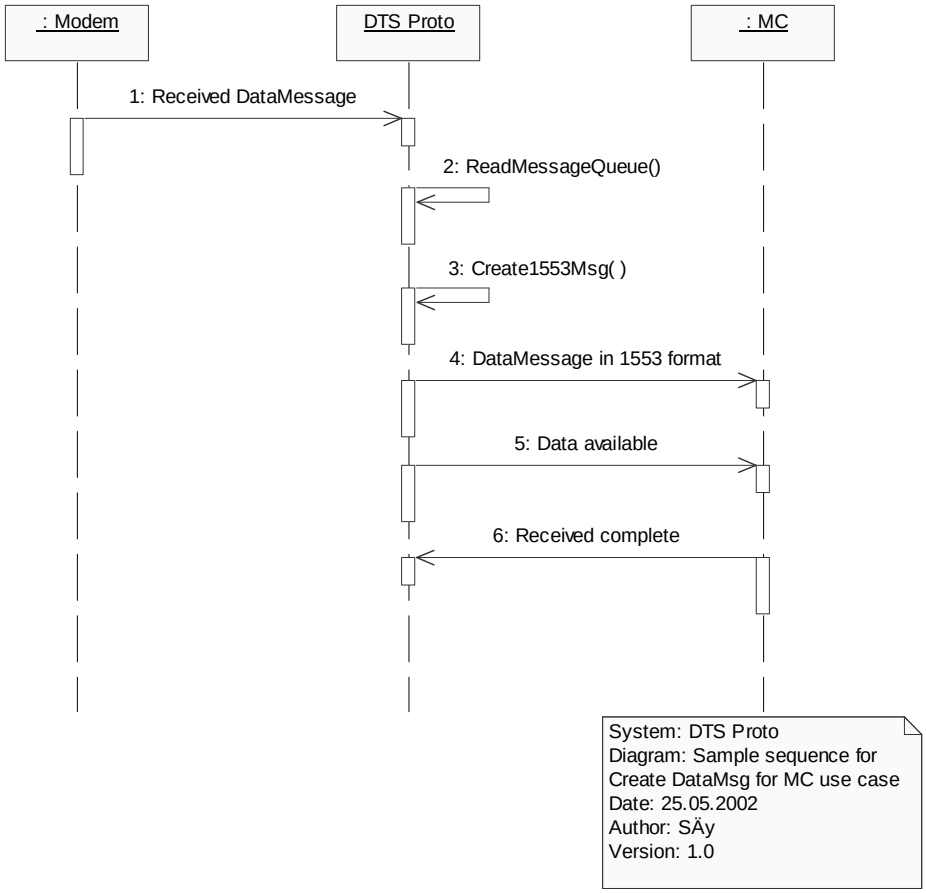
[VxW99b] VxWorks, Reference Manual 5.4, Edition 1, Wind River Systems, 1999.

[Wil97] Williams Michael R., History Of Computing Technology, IEEE Computer Society Press, 1997.

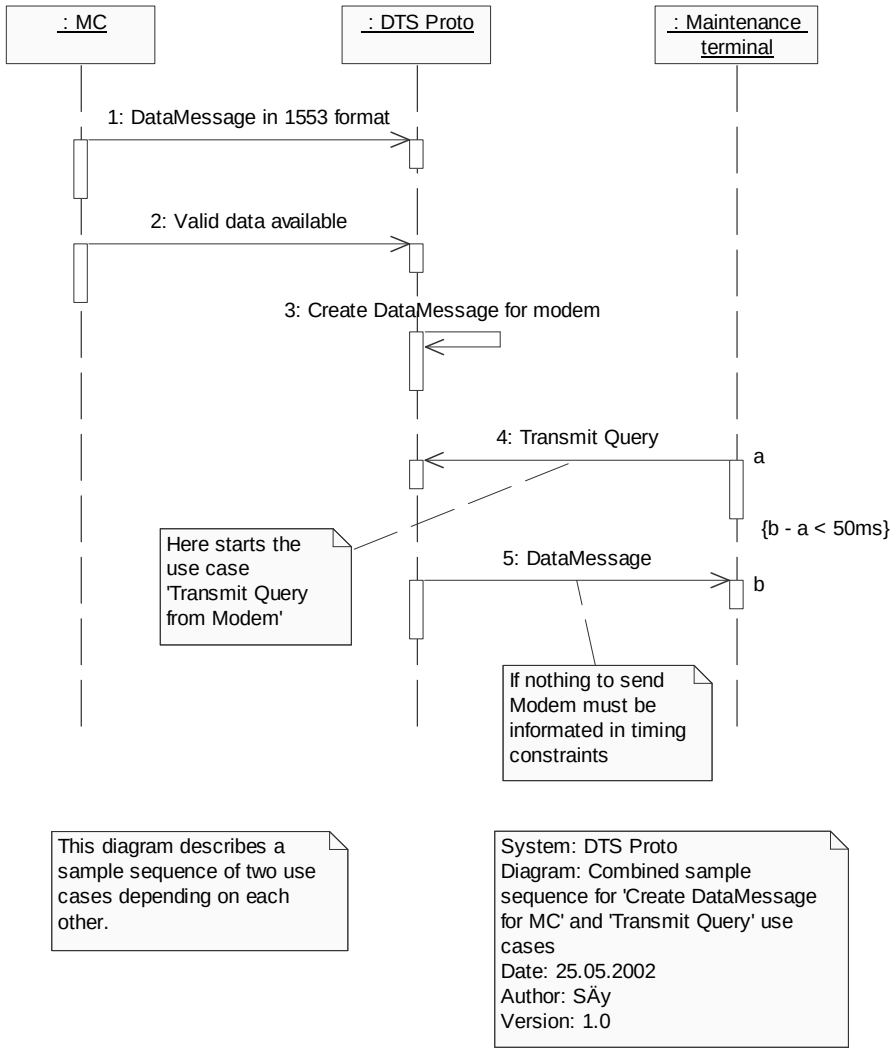
10 Liitteet



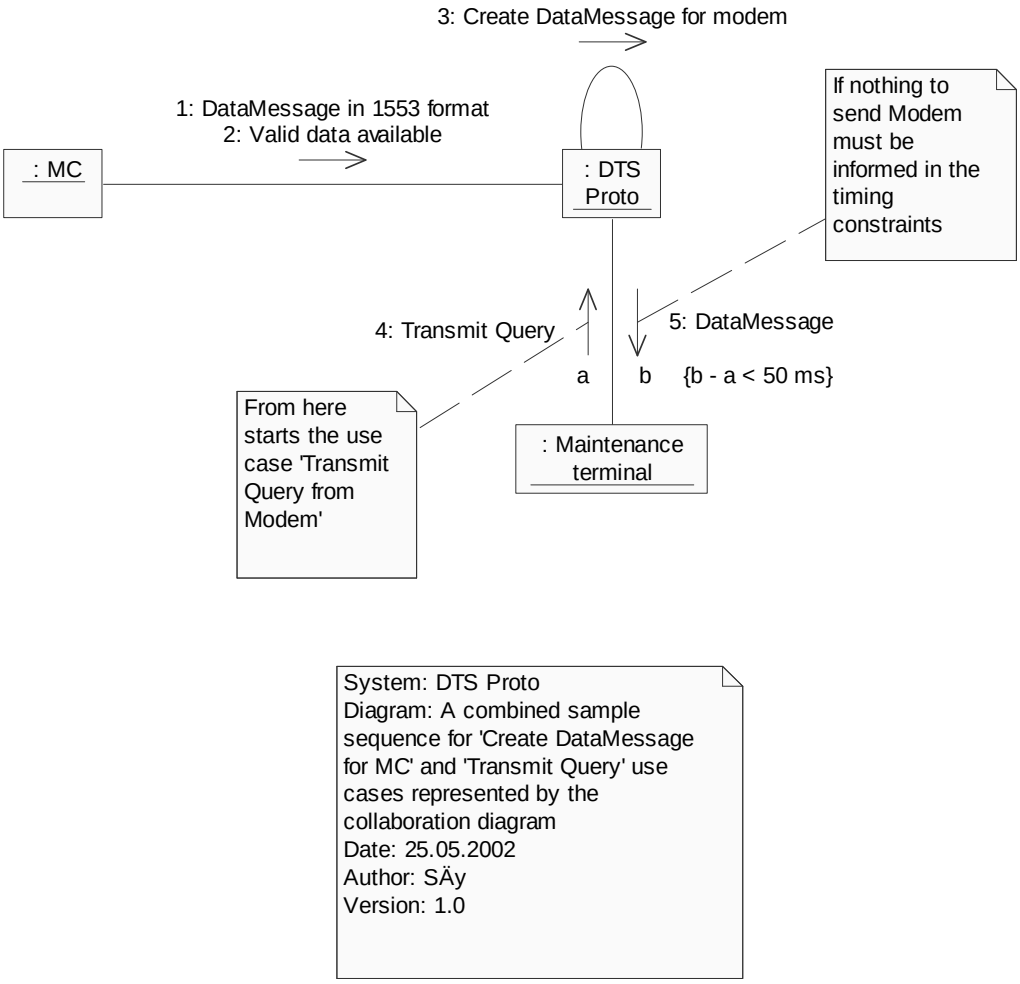
Kuva L.1. DTS Proton käyttötapauskaavio.



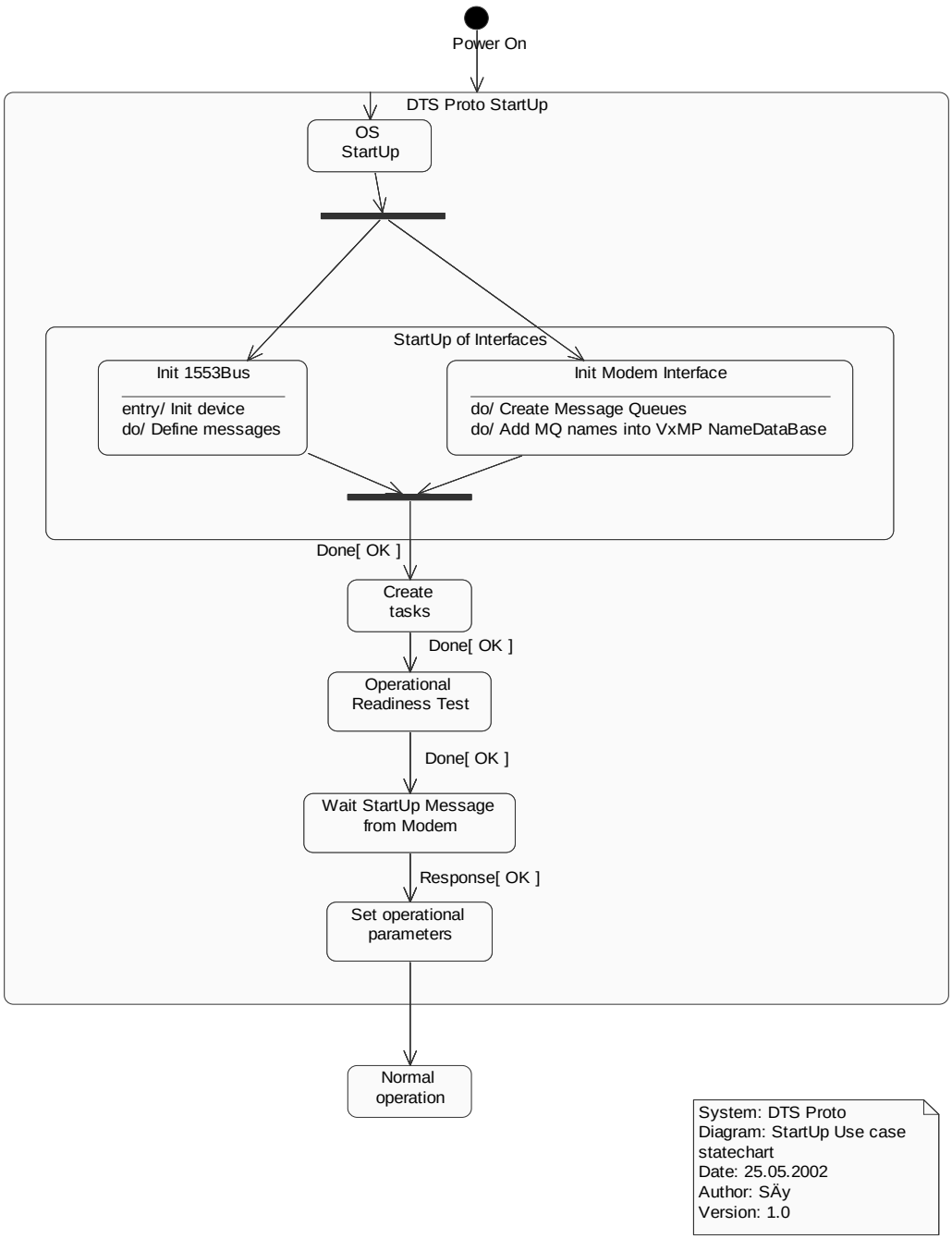
Kuva L.2. Sekvenssikaavio vastaanotetun datasanoman välittämisestä Modeemilta MC:lle.



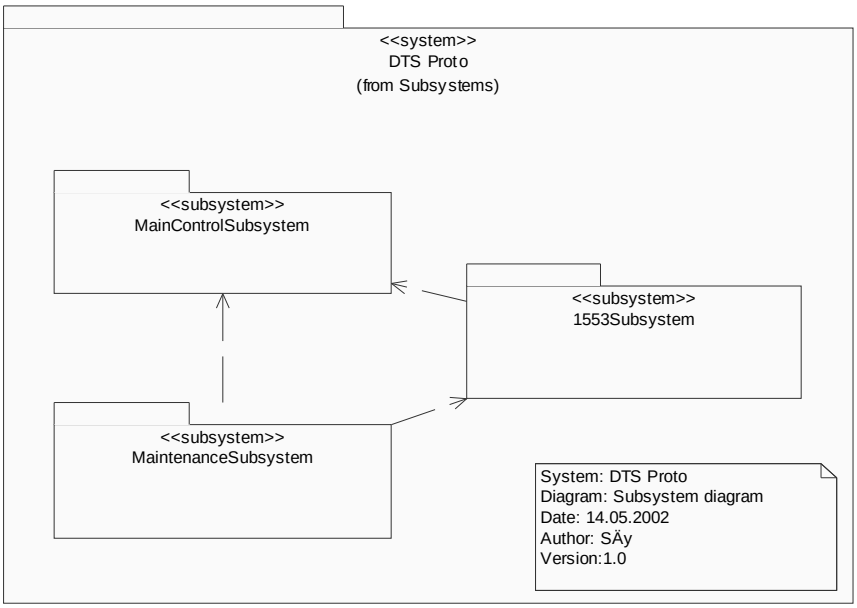
Kuva L.3. Sekvenssikaavio datasanoman välittämisestä MC:lta Modeemille.



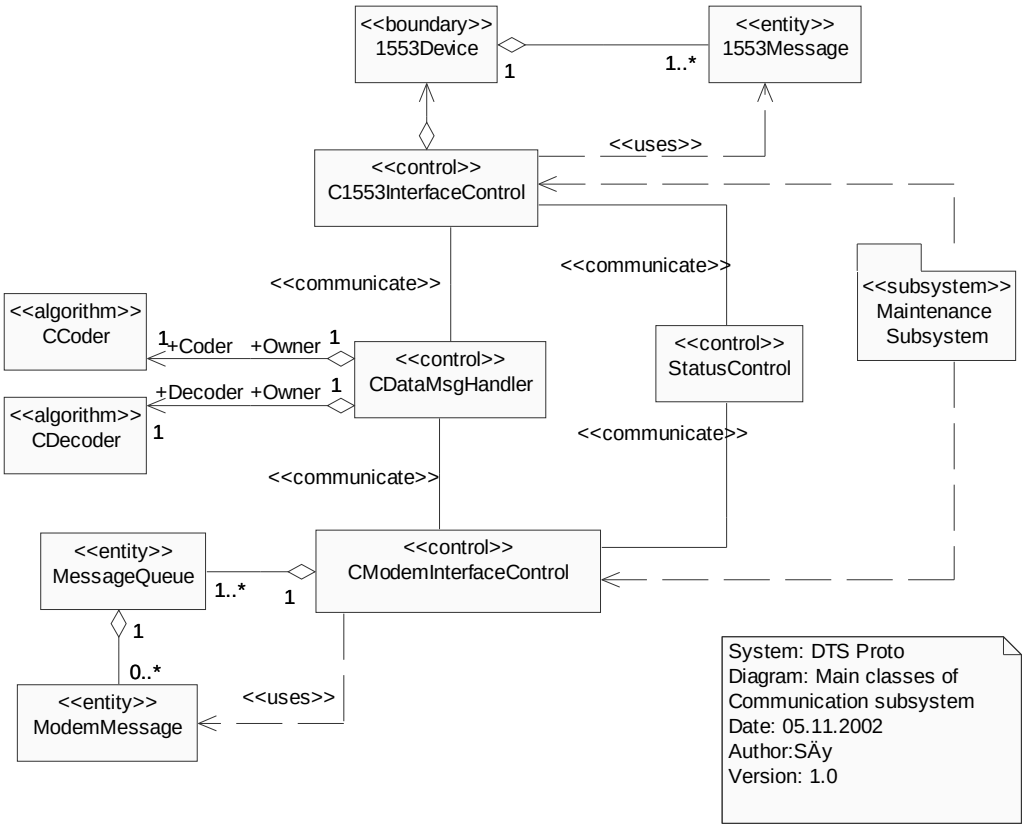
Kuva L.4. Yhteistoimintakaavio datasanoman välittämisestä MC:lta Modeemille.



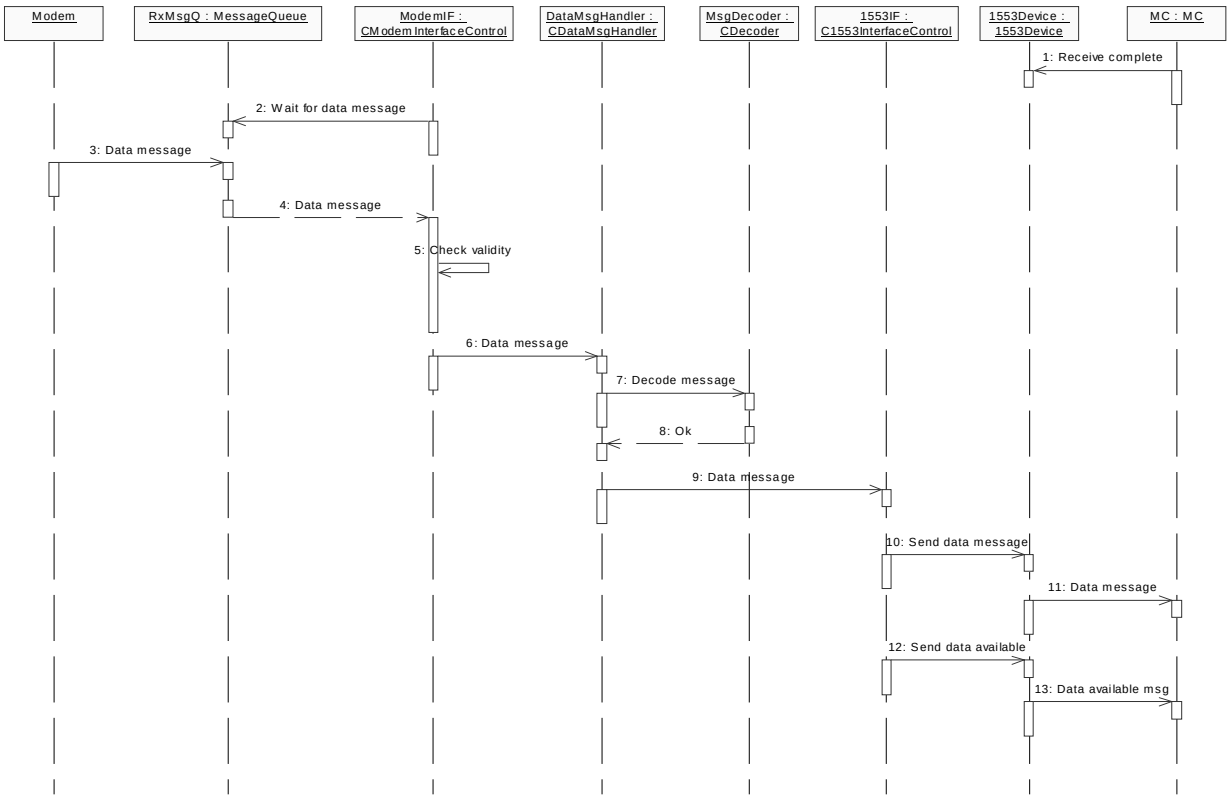
Kuva L.5. Tilakaavio DTS Proton käynnistyksestä.



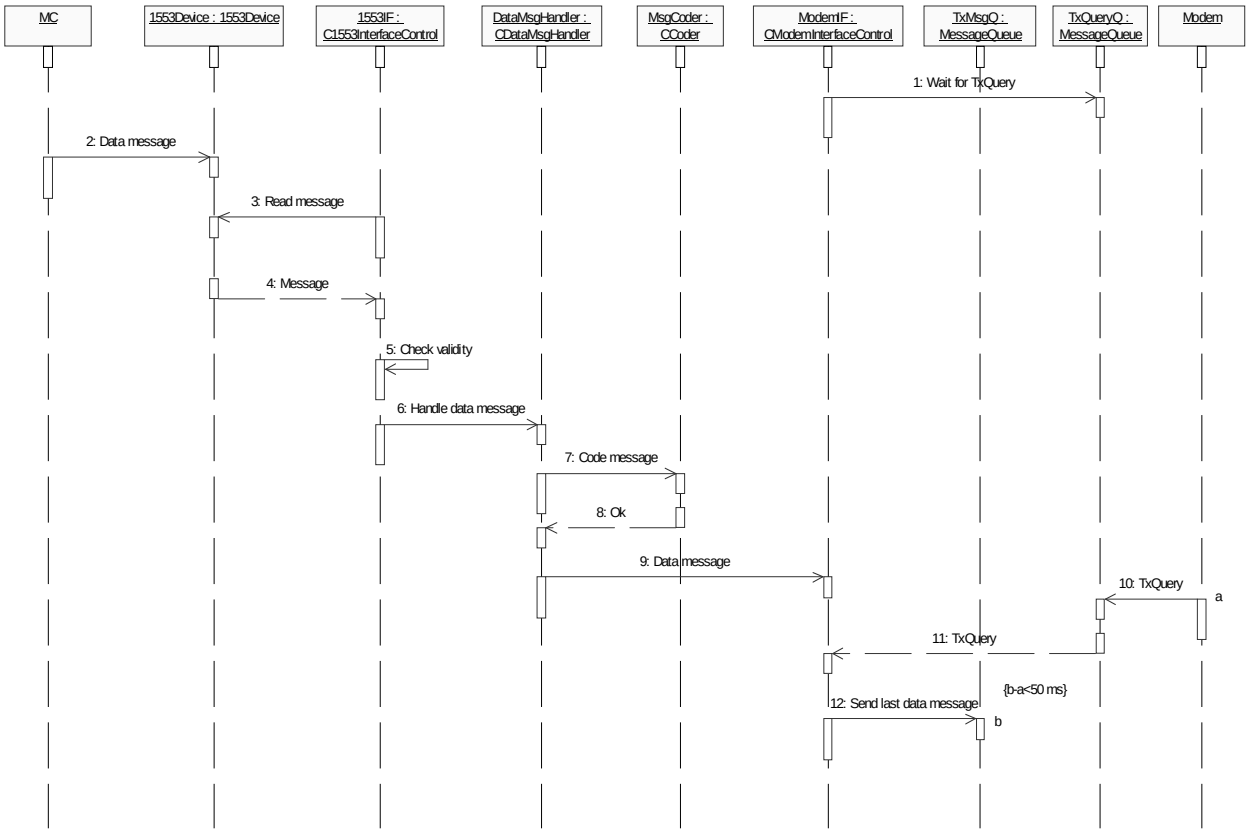
Kuva L.6. DTS Proton alijärjestelmät kuvattuna paketteina luokkakaaviossa.



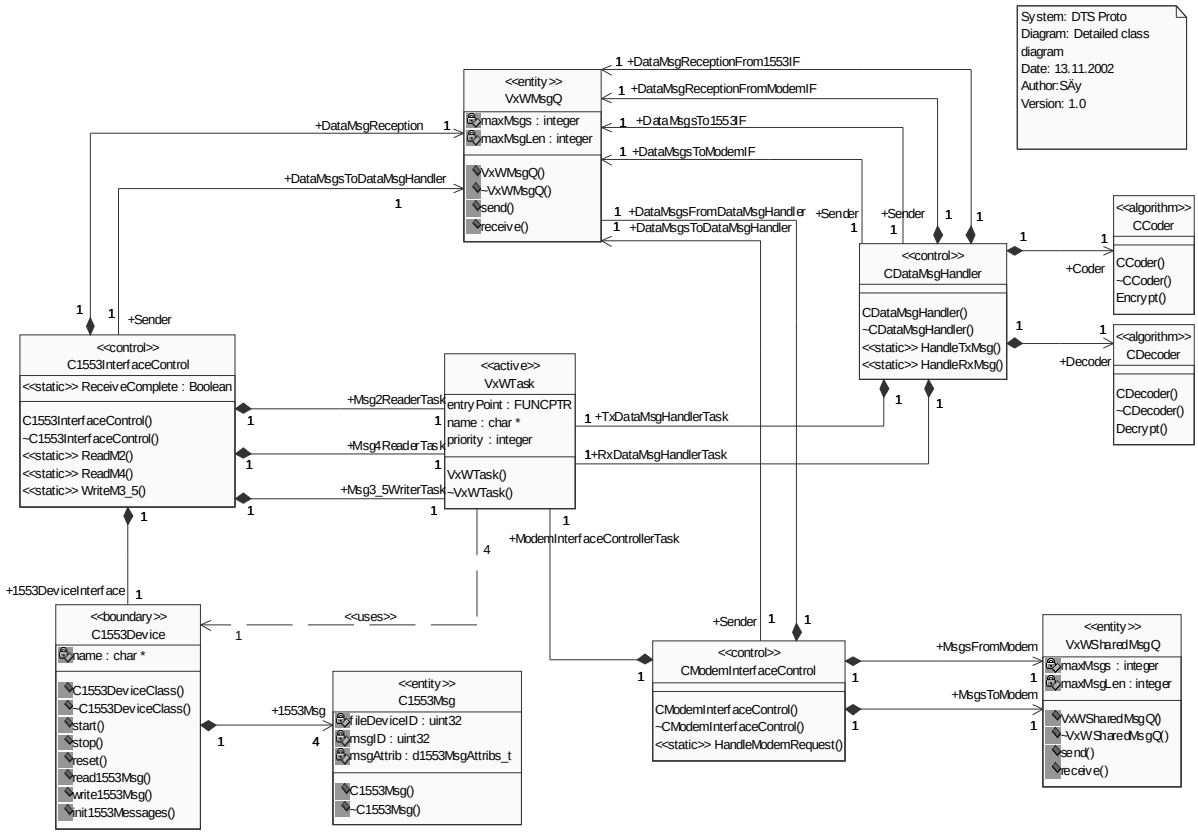
Kuva L.7. Yleissuunnitteluvaiheen luokkakaavio.



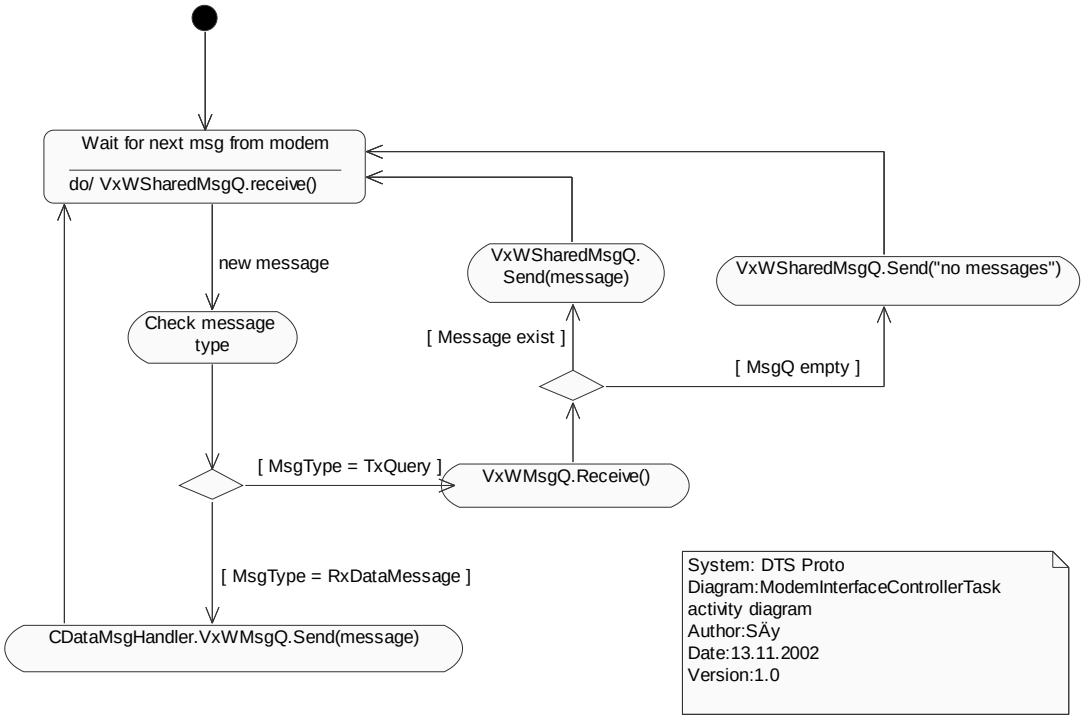
Kuva L.8. Datasanoman välitys modeemilta MC:lle.



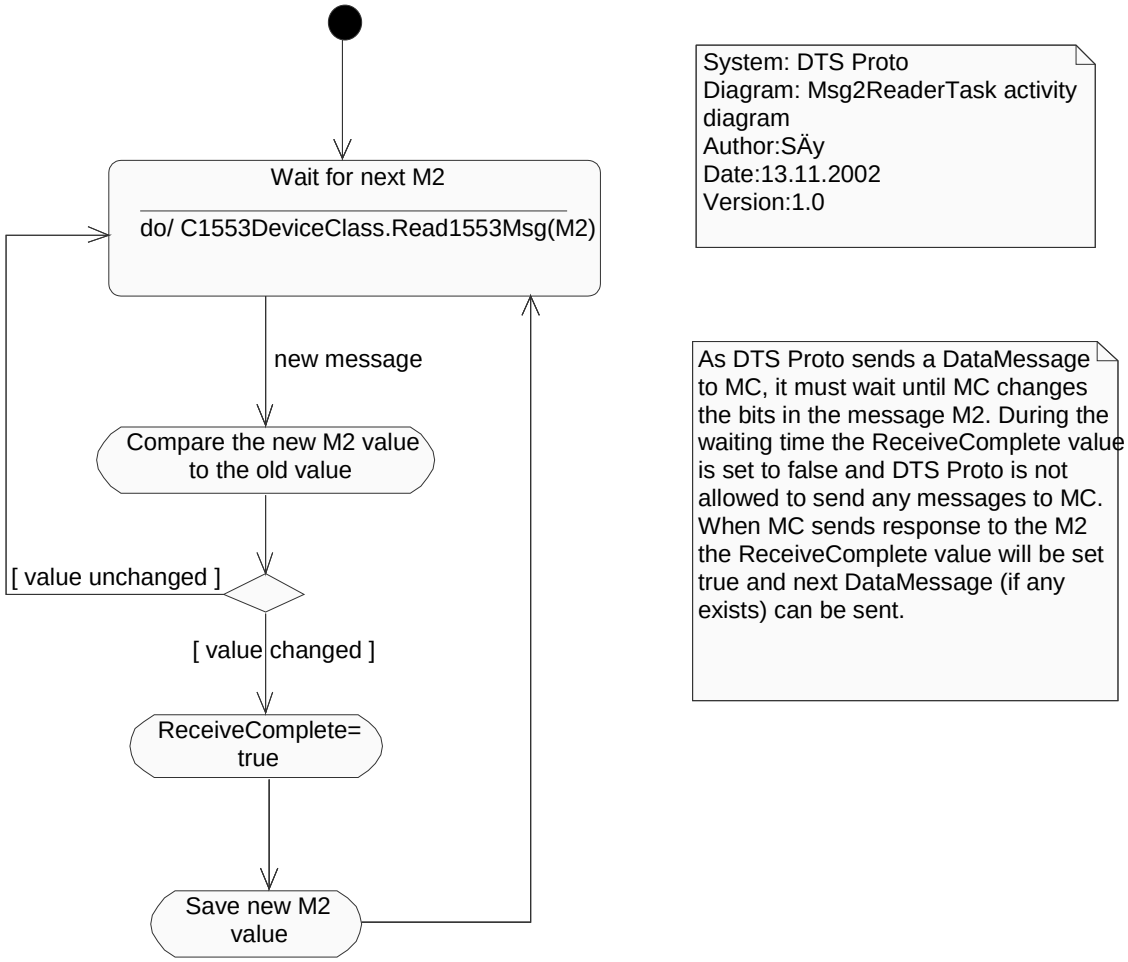
Kuva L.9. Datasanoman välitys MC:ltä Modeemille.



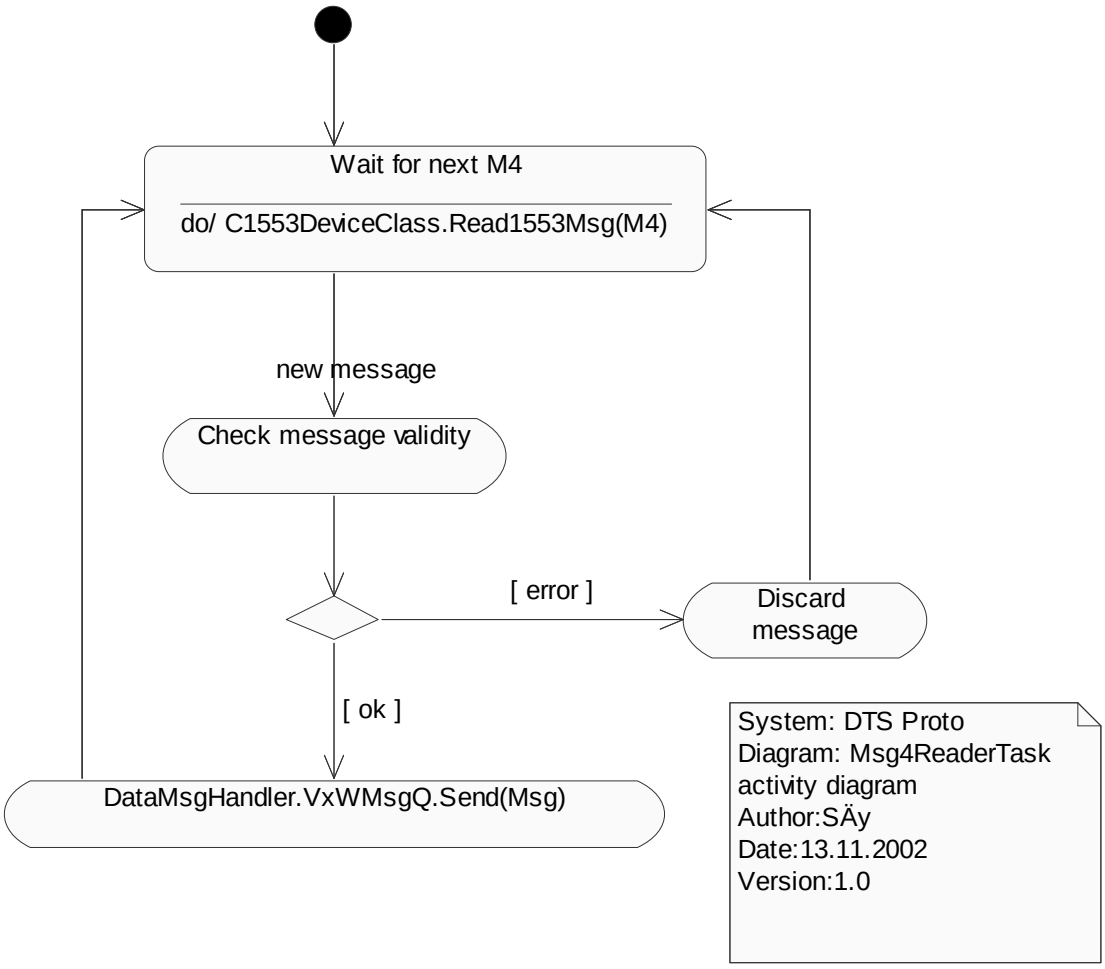
Kuva L.10. DTS Proton yksityiskohtainen luokkakaavio.



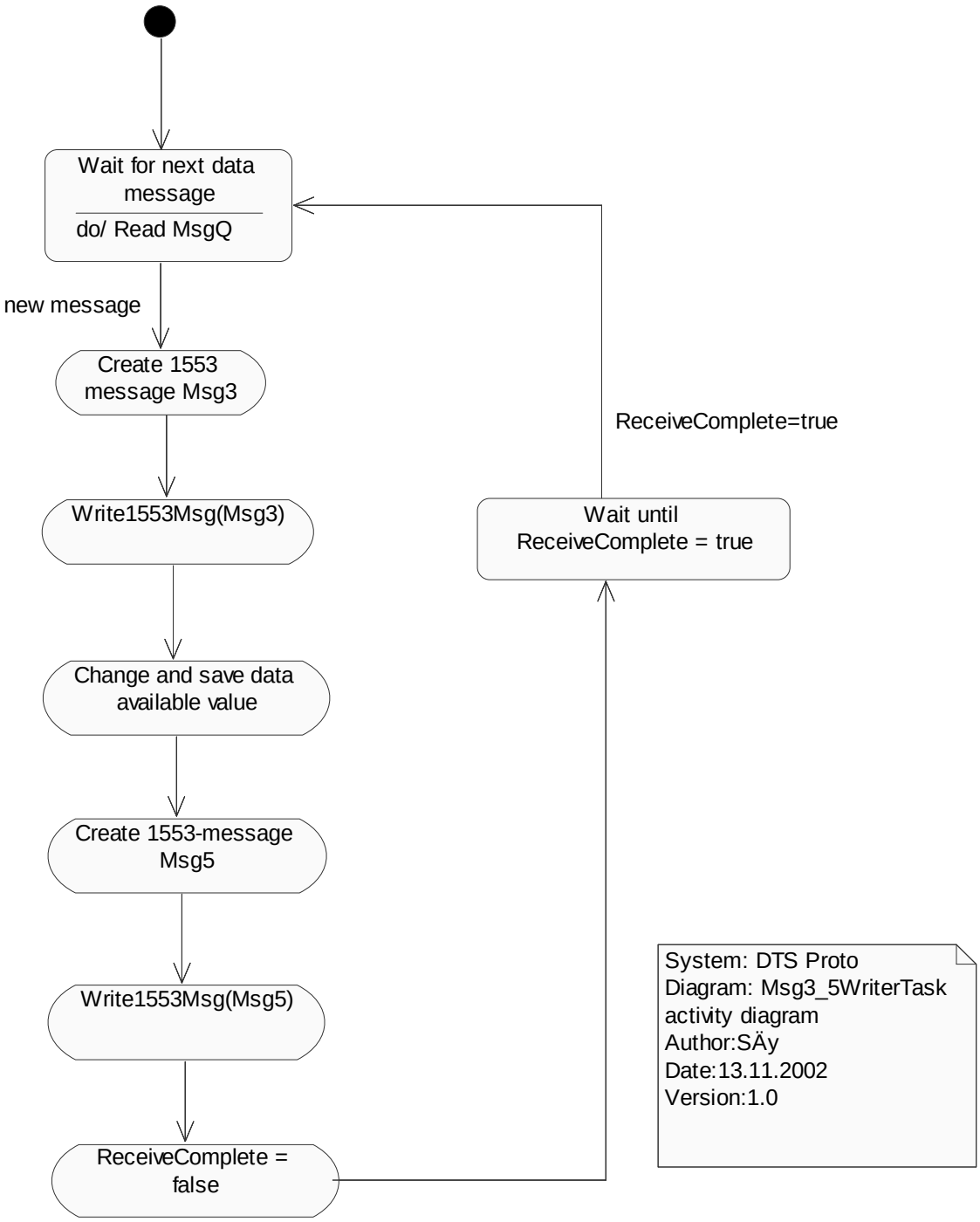
Kuva L.11. ModemInterfaceControllerTask-säikeen aktiviteettikaavio.



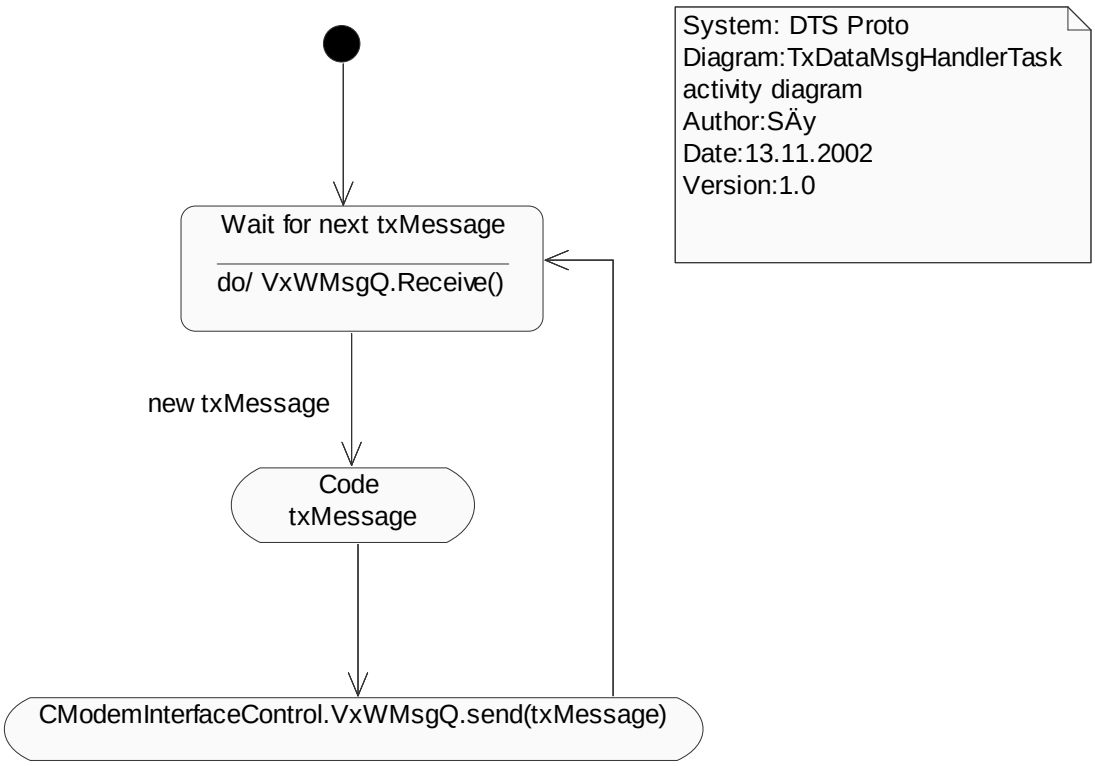
Kuva L.12. Msg2ReaderTask-säikeen aktiviteettikaavio.



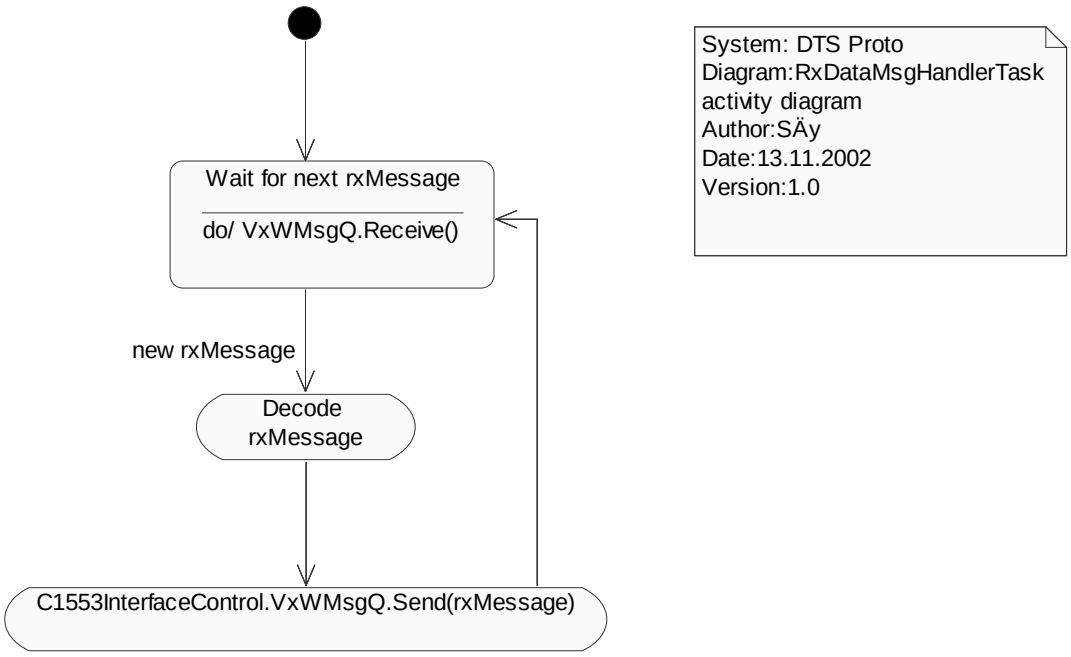
Kuva L.13. Msg4ReaderTask-säikeen aktiviteettikaavio.



Kuva L.14. Msg3_5WriterTask-säikeen aktiviteettikaavio.



Kuva L.15. TxDataMsgHandlerTask-säikeen aktiviteettikaavio.



Kuva L.16. RxDataMsgHandlerTask-säikeen aktiviteettikaavio.