Samanaikaisuuden mallintaminen oliokeskeisessä ohjelmistokehitysmenetelmässä

Mikko Piipponen

Tampereen yliopisto
Tietojenkäsittelytieteiden laitos
Tietojenkäsittelyoppi
Pro gradu -tutkielma
Ohjaaja: Jyrki Nummenmaa
Huhtikuu 2008

Tampereen yliopisto
Tietojenkäsittelytieteiden laitos
Tietojenkäsittelyoppi
Mikko Piipponen: Samanaikaisuuden mallintaminen oliokeskeisessä ohjelmistokehitysmenetelmässä
Pro gradu -tutkielma, 79 sivua, 2 liitesivua
Huhtikuu 2008

Vaikka erilaisia ohjelmistojen suunnittelumenetelmiä on kehitetty vuosikymmeniä ja rinnakkaisten prosessien vuorovaikutustakin on tutkittu jo 60-luvulta lähtien, ei niiden välille ole onnistuttu rakentamaan selkeää siltaa. Samanaikaisuutta on käsitelty usein *joko* sen teorian ja formaalien metodien, *tai* sen ohjelmoinnin näkökulmasta. Tässä tutkielmassa perehdytään samanaikaisuuden suunnittelun periaatteisiin ja siihen liittyviin ongelmiin sekä samanaikaisuuden mallintamisessa käytettäviin tekniikoihin. Lisäksi esitellään, kuinka nämä periaatteet ja tekniikat voidaan ottaa osaksi yleiskäyttöistä, koko ohjelmiston elinkaaren kattavaa menetelmää. Erityisesti käsitellään OMT++-ohjelmistokehitysmenetelmän vaihetuotteiden sisältämän informaation käyttöä samanaikaisuuden mallintamisessa tarvittavan informaation lähteenä. Samanaikaisuuden mallintamiseen tässä tutkielmassa käytetään FSP-notaatiota ja sen rinnalla käytettäväksi tarkoitettua LTSA-työkalua.

Avainsanat ja -sanonnat: samanaikaisuus, rinnakkaisuus, reaaliaikajärjestelmät, oliokeskeiset menetelmät, OMT++.

Sisällys

1. Johdanto

Ohjelmistokehitystä käsittelevässä kirjallisuudessa on yksimielisyys siitä, että ohjelmiston tuotannossa tapahtuvien virheiden korjaaminen on sitä kalliimpaa, mitä aikaisemmin virhe tehdään ja mitä myöhäisemmin se havaitaan. Usein virheet huomataan vasta ohjelmiston testausvaiheessa, ja virheen korjaaminen saattaa vaatia muutoksia yllättävänkin suureen osaan jo toteutettua järjestelmää ja voi täten tulla kalliiksi.

Ohjelmistoteollisuudessa tätä ongelmaa ratkaisemaan on kehitetty erilaisia suunnittelumenetelmiä, joilla pyritään kehittämään mahdollisimman virheettömiä ohjelmistoja. Menetelmien kirjo on laaja, eikä yhtä oikeaa, kaikenlaisten ohjelmistojen tuottamiseen tarkoitettua menetelmää ole, eikä myöskään tule, johtuen sovellusalueiden hyvinkin erilaisista vaatimuksista. Ohjelmistoyritysten laadunhallinnan kautta yritetään myös omalta osaltaan karsia virheet mahdollisimman varhaisessa vaiheessa käyttäen erilaisia laadunvarmistusmenetelmiä, esimerkiksi katselmointeja.

Ohjelmistot ovat jatkuvasti kasvaneet ja tulleet monimutkaisemmiksi. Samalla niiltä vaaditaan aina vain parempaa luotettavuutta ja suorituskykyä. Ohjelmistojen suuruus on pakottanut jakamaan niitä pienempiin, helpommin hallittavimpiin osiin. Suurta osaa tällä hetkellä kehitetyistä ohjelmistoista on tarkoitus ajaa ympäristöissä, jotka sallivat useamman prosessin samanaikaisen suorituksen. Älypuhelinohjelmistojen kehityksessä käytetyn Symbian-ohjelmistoalustan sekä Java-ohjelmointikielen yleistymisen myötä yhä useammat taidoiltaan ja kokemukseltaan eritasoiset ohjelmoijat kirjoittavat ohjelmia välineillä, jotka tarjoavat ohjelmoijille prosessien hallintaan vaadittavat rakenteet ja toiminnallisuuden ohjelmointikielen tasolla.

Edellä esitetyn valossa onkin hämmästyttävää havaita, ettei ohjelmistoteollisuudessa ole olemassa ensimmäistäkään yleisesti hyväksyttyä standardia samanaikaisuuden suunnitteluun ja mallintamiseen. Teollisuudessa on käytössä lähinnä joitakin erilaisia tietovuo- ja oliokeskeisten suunnittelumenetelmien samanaikaisuuden paremmin huomioonottavia laajennoksia, kun taas akateemisessa maailmassa on lähinnä keskitytty erilaisten formaalien mallien ja niiden syntaksin ja semantiikan kehittämiseen.

Tässä tutkielmassa perehdytään samanaikaisuuden mallintamiseen osana yleiskäyttöistä ohjelmistokehitysmenetelmää. Esimerkiksi valittiin OMT++-menetelmä, jota on todistettavasti ja menestyksekkäästi käytetty suurten reaaliaikaisten matkapuhelinverkkojen hallintajärjestelmien suunnitteluun. Jotta samanaikaisuuden mallintamista osana laajempaa suunnittelumenetelmää

voidaan tutkia, on ensin tutustuttava tarkemmin sekä samanaikaisuuden käsitteeseen että ohjelmistojen yleisiin suunnittelutekniikoihin.

Tämä tutkielma on jaettu kahdeksaan lukuun. Luvussa 2 tutustutaan samanaikaisuuden suunnitteluun yleisesti. Lisäksi esitetään joitain samanaikaisuuden peruskäsitteitä ja ongelmia. Luvussa 3 esitellään muutamia tutkielman kannalta olennaisimpia ohjelmistojen suunnittelussa käytetyistä notaatioista. Erityisesti tutustutaan tapahtumasekvenssikaavioihin ja tilakaavioihin, sekä selvitetään tilakaavioiden merkitystä samanaikaisuuden mallintamisessa. Luvussa 4 perehdytään samanaikaisuuden mallintamiseen tässä tutkielmassa käytettyyn FSP-notaatioon. Luvussa 5 esitellään OMT++, yleiskäyttöinen olioperustainen ohjelmistonkehitysmenetelmä. Luvussa 6 perehdytään käyttäytymismallien laatimiseen skenaariopohjaisten ohjelmistosuunnittelumenetelmien yhteydessä. Luvussa 7 käsitellään OMT++-menetelmän vaihetuotteiden sisältämän informaation käyttöä käyttäytymismalleissa tarvittavan informaation lähteenä. Luvussa 8 tehdään yhteenveto tutkielmassa esitetyistä asioista.

2. Samanaikaisuus

Useimmat monimutkaiset tehtävät ja järjestelmät voidaan jakaa joukoksi yksinkertaisia osakokonaisuuksia. Näihin osakokonaisuuksiin liittyvät toiminnot eivät aina tapahdu peräkkäisesti yksi toisensa jälkeen, vaan ne voivat tapahtua ajallisesti päällekkäin, samanaikaisesti. Aivan kuten talonrakennusprojektissa voidaan samaan aikaan suorittaa talon sähkö- ja putkiasennustehtäviä, voidaan laaja ohjelmisto jakaa useampaan samaan aikaan suoritettavaan osaan. Samanaikaisuus on käyttökelpoinen monenlaisissa ohjelmissa, joissa vasteaika ja suoritusteho ovat ensiarvoisen tärkeitä. Esimerkiksi käyttöliittymän toimiminen katkeilematta parantaa järjestelmän käytettävyyttä, ja sen voi saavuttaa siirtämällä raskaita ja aikaavaativia tehtäviä taustalla suoritettaviin erillisiin prosesseihin.

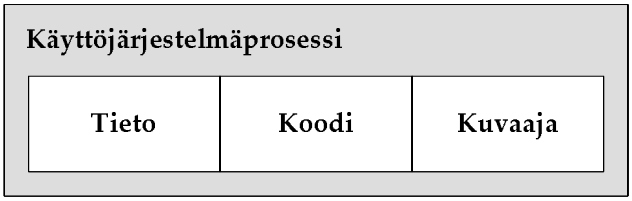
Samanaikaisuudesta saatavat hyödyt voivat kuitenkin hautautua järjestelmän monimutkaistumisen aiheuttamiin ongelmiin. Tämän tutkielman tarkoituksena onkin esittää, että samanaikaisuuden tuoman monimutkaisuuden hallitseminen on mahdollista käyttämällä järjestelmällistä tapaa suunnitella ja toteuttaa samanaikaisuutta sisältävä ohjelmisto.

Tässä luvussa esitellään samanaikaisuuden peruskäsitteitä ja samanaikaisuuteen liittyviä erityisongelmia sekä samanaikaisuuden suunnitteluun ja mallintamiseen kuuluvia periaatteita. Ensimmäisenä esitellään sekä tutkielman että kaiken tietokoneella tapahtuvan prosessoinnin kannalta keskeisin käsite, *prosessi*.

2.1. Prosessi ja sen rakenne

Tietokoneella tapahtuvaa ohjelman suorittamista kutsutaan prosessiksi ja samanaikainen ohjelma koostuu useasta prosessista. Atk-sanakirja [Atk-sanakirja, 2003] määrittelee prosessin seuraavasti: 1) Tapahtumasarja, jolla on tai katsotaan olevan tietty suunta, tarkoitus, vaikutus tai päämäärä tai 2) Kokonaisuus, jollaisena käyttöjärjestelmä hallitsee ohjelman suoritusta ja joka sisältää tiedon suoritettavana olevasta konekielisestä ohjelmasta, käytettävästä datasta ja tietorakenteista sekä suorituksen vaiheesta.

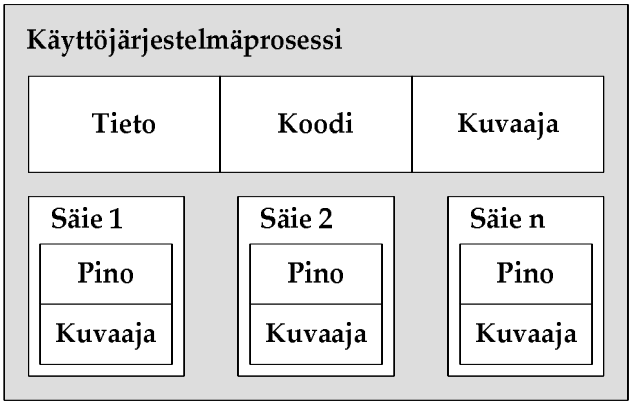
Termi ”prosessi” on lähtöisin käyttöjärjestelmien suunnittelua käsittelevästä kirjallisuudesta. Käyttöjärjestelmän kannalta prosessi on suoritinajan ja käytettävän muistin jakamisen perusyksikkö. Prosessiin sisältyykin sen koodi (code), sen tarvitsema tieto (data) ja eri rekistereiden tilat (state of the machine registers). Prosessin tieto-osa jaetaan globaaleihin muuttujiin ja paikallisiin muuttujiin, jotka on järjestetty pinoon (stack).



Kuva 2.1. Käyttöjärjestelmäprosessi

Yleensä käyttöjärjestelmäprosesseilla on oma osoiteavaruutensa, jonka sisältämään tietoon muut prosessit eivät pääse käsiksi. Sovellusprosessin ajamiseen kuuluvat seuraavat toimenpiteet: muistin jakaminen prosessin globaaleille muuttujille ja pinolle, sovelluksen koodin lataaminen tietokoneen muistiin sekä koodin ajaminen lataamalla ajettavan ohjelman ensimmäiseksi suoritettava käsky ohjelmalaskurirekisteriin, pinon muistiosoite pino-osoitinrekisteriin ja niin edelleen. Käyttöjärjestelmä ylläpitää sisäistä tietorakennetta, jota kutsutaan prosessikuvaukseksi (process descriptor) ja johon se tallentaa prosessiin liittyvää tietoa prosessin prioriteetista, varatusta muistitilasta sekä rekistereiden tilasta prosessin ollessa pois suorituksesta (kuva 2.1).

Prosessit voidaan jakaa kahteen luokkaan: *raskaisiin* (heavy-weight) käyttöjärjestelmäprosesseihin ja *kevyihin* (light-weight) prosesseihin, joita kutsutaan myös nimellä *säie* (thread). Perinteisellä käyttöjärjestelmäprosessilla on vain yksi *ohjaussäie* (thread of control) ts. sillä ei ole sisäistä samanaikaisuutta. Monet nykyisistä yleiskäyttöisistä käyttöjärjestelmistä, kuten Unix, Windows NT tai OS/2, sallivat yhden prosessin omistaa useampia samanaikaisia ohjaussäikeitä. Sen mahdollistavat samanaikaisesti suoritettavat kevyet prosessit eli säikeet. Säikeet ovat käyttöjärjestelmän kannalta prosesseja, jotka toimivat samassa osoiteavaruudessa (kuva 2.2).

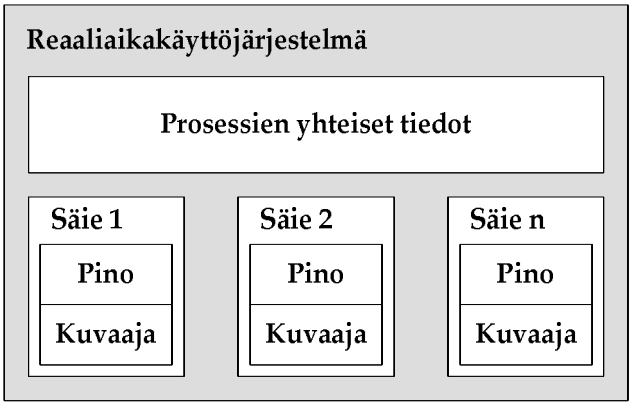


Kuva 2.2. Säikeitä omistava käyttöjärjestelmäprosessi

2.2. Reaaliaikakäyttöjärjestelmät

Reaaliaikakäyttöjärjestelmä (real-time operating system, RTOS) mahdollistaa sovelluksen toteuttamisen joukkona keskenään kommunikoivia prosesseja. Prosessit poikkeavat yleensä "normaaleista" yleiskäyttöisen käyttöjärjestelmän prosesseista siinä, että muistin suojausta ei ole olemassa, ja mikä tahansa prosessi voi viitata minkä tahansa muun prosessin alueelle (kuva 2.3). Tämä saattaa aiheuttaa hankalasti havaittavia virhetilanteita. Suojauksen puuttumiseen on kuitenkin yleensä hyviä syitä, mm. käytettävän prosessorin alkeellisuus ja/tai suojaukset poistamalla saavutettava tehokkuus. Reaaliaikakäyttöjärjestelmän prosessien prioriteetit ovat yleensä kiinteitä ja ne määrätään prosessin käynnistämisen yhteydessä. Reaaliaikajärjestelmän prosesseja kutsutaan usein myös *tehtäviksi* (task), säikeiksi (thread) tai kevytprosesseiksi (light weight process).

Reaaliaikakäyttöjärjestelmät (esim. RMX, OS/9, VRTX, QP, QNX) tarjoavat peruspalveluita mm. prosessien hallintaan (luominen, lopettaminen, prioriteetit), kommunikointiin (sanomanvälitys), synkronointiin ja poissulkemiseen (semafori-operaatiot tai vastaavat) sekä oheispiirien ohjaukseen. Jotkin reaaliaikakäyttöjärjestelmistä sisältävät myös ohjelmistojen testausta helpottavia apuneuvoja [Haikala ja Märijärvi, 2000].

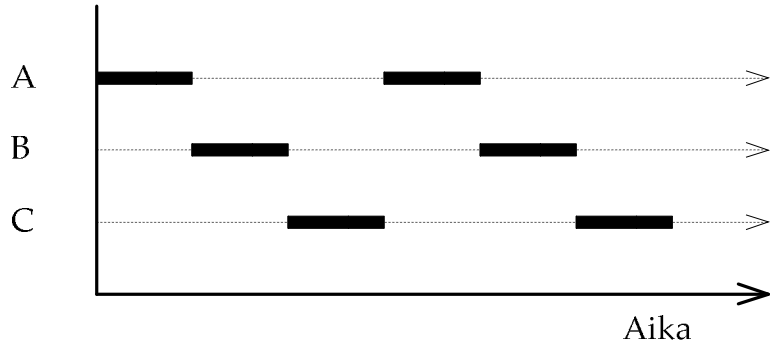


Kuva 2.3. Reaaliaikakäyttöjärjestelmä ja sen prosessit

2.3. Samanaikainen suoritus

Samanaikaisen ohjelman suorituksessa (concurrent execution) useat prosessit ovat aktiivisia (active) yhtä aikaa. Jokainen näistä prosesseista on peräkkäisen ohjelman suorittamista. Prosessi etenee antamalla käskyn kerrallaan prosessorille suoritettavaksi. Jos ohjelmaa suorittavassa tietokoneessa on useampia prosessoreja, voidaan siinä suorittaa yhtäaikaa niin montaa prosessia, kuin siinä on prosessoreja. Tätä kutsutaan usein prosessien *rinnakkaiseksi* (parallel execution, real concurrent execution) suorittamiseksi. On

kuitenkin huomattavasti yleisempää, että ohjelmaa suoritettaessa aktiivisia prosesseja on useampia kuin tietokoneessa on prosessoreja. Tällaisissa tapauksissa käytössä olevat prosessorit, tai yleisemmin yksi ainoa prosessori, suorittaa limittäin näitä aktiivisia prosesseja. Kuvassa 2.4 esitetään tapaus, jossa yksi prosessori suorittaa kolmea prosessia (A, B ja C). Paksut viivat esittävät prosessilta tulevien käskyjen suorittamista prosessorilla. Kuvasta huomataan, että kaikki kolme prosessia edistyvät, mutta ainoastaan yksi prosessi kerrallaan voi antaa prosessorille käskyjä suoritettavaksi.



Kuva 2.4. Prosessin vaihto.

Vaihdot prosessien välillä tapahtuvat vapaaehtoisesti tai vasteena suorituksen aikana tapahtuviin *keskeytyksiin*. Keskeytyksillä tarkoitetaan järjestelmän ulkopuolelta saapuvia signaaleja, jollaisia ovat esimerkiksi I/O-operaation valmistuminen tai kellokeskeytys. Kuten kuvasta 2.4 voidaan nähdä, ei suoritettavan prosessin vaihdolla ole vaikutusta kunkin prosessin prosessorille suoritettavaksi annettavien käskyjen järjestykseen. Prosessori suorittaa kultakin prosessilta tulevia käskyjä järjestyksessä, mutta *limittäin* (interleaving) toisilta prosesseilta saamiinsa käskyihin nähden. Tämänkaltaista limitykseen perustuvaa prosessien samanaikaista suorittamista kutsutaan usein pseudo-samanaikaiseksi (pseudo-concurrent) suorittamiseksi, sillä eri prosesseilta tulevia käskyjä ei suoriteta yhtäaikaa vaan limittäin. Tässä tutkielmassa käytetään termejä rinnakkaisuus ja samanaikaisuus samaa tarkoittavina asioina, eikä tässä yhteydessä tehdä eroa prosessien tosi- ja pseudo-samanaikaisen suorittamisen välillä, sillä yleisesti ottaen samat suunnittelu- ja ohjelmointiperiaatteet pätevät yhtä lailla sekä rinnakkaiseen kuin limitettyynkin suoritukseen. Itse asiassa, kuten myöhemmin tässä luvussa esitetään, on erään samanaikaisuuden suunnitteluperiaatteen mukaista esittää prosessien samanaikainen suoritus aina limitetysti, vaikka todellisuudessa prosesseja suoritettaisiinkin rinnakkaisesti.

2.4. Reaaliaikajärjestelmät

Useimmat nykyaikaisista ohjelmistoista ovat luonteeltaan interaktiivisia. Järjestelmät kommunikoivat ulkoisten elementtien, kuten käyttäjien, tietokantojen, sensoreiden ja verkkojen kanssa. Nämä elementit kommunikoivat järjestelmän kanssa lähettäen ja vastaanottaen erilaisia tapahtumia (sanomia). Tapahtumia syntyy satunnaisessa järjestyksessä (ja ennalta-arvaamattomana ajankohtana) ja järjestelmän täytyy olla aina valmis käsittelemään ne. Ohjelmistosuunnittelijan tärkein tehtävä onkin kehittää järjestelmiä, jotka kykenevät kommunikoimaan ympäristönsä kanssa sujuvasti ja odotetulla tavalla.

Alan kirjallisuudessa samanaikaisuus yhdistetäänkin usein ns. *reaaliaikajärjestelmien* (real time system) toteuttamiseen. Reaaliaikajärjestelmät tuottavat jonkin toiminnon vasteena järjestelmän ulkopuolelta tulevaan tapahtumaan, ja niitä kutsutaan tästä syystä usein myös *reaktiivisiksi järjestelmiksi*. Reaaliaikajärjestelmän täytyy pystyä vastaamaan näihin tapahtumiin tietyn aikajakson sisällä. Koska reaaliaikajärjestelmälle asetetaan tiukat suorituskykyvaatimukset, ohjaavat sen ohjelmistosuunnittelua normaaleiden sovellusvaatimusten, ohjelmointikieleen liittyvien riippuvuuksien ja muun tavanomaisen suunnittelun lisäksi sekä laitteisto- että ohjelmistoarkkitehtuuri ja käytetyn käyttöjärjestelmän ominaisuudet.

Reaaliaikajärjestelmät voidaan jakaa karkeasti ns. pehmeisiin ja koviin reaaliaikajärjestelmiin. Pehmeissä reaaliaikajärjestelmissä (soft real-time system) vasteaikavaatimukset eivät ole kovin tiukkoja, eikä järjestelmän ”hidastelu” johda kohtalokkaisiin seuraamuksiin. Esimerkkejä tällaisista järjestelmistä ovat esimerkiksi pankkiautomaatti tai tekstinkäsittelyohjelman interaktiivinen käyttöliittymä. Kovissa reaaliaikajärjestelmissä (hard real-time system) ”hidastelu” voi johtaa virhetoimintoon, joka voi edelleen johtaa vakaviin seuraamuksiin. Kovuus ei niinkään liity vasteajan nopeuteen kuin sen täsmällisyyteen: vaste ei saa koskaan ylittää määriteltyä maksimiarvoa. Myös määrätyn vasteajan alittaminen saattaa joissain tapauksissa olla virhetilanne. Tyypillisiä esimerkkejä kovista reaaliaikajärjestelmistä ovat erilaisten mekaanisten laitteiden ohjausjärjestelmät, esimerkiksi sotilaallisessa käytössä ohjuksen automaattista maaliinhakeutumista ohjaava järjestelmä sekä monet prosessinohjausjärjestelmät (esimerkiksi matkapuhelinverkossa).

Vaikka suuri ja monimutkainen reaaliaikajärjestelmä saattaa sisältää jopa useampia miljoonia koodirivejä, on aikariippuvaisten osien osuus kokonaisuudesta yleensä hyvinkin pieni. Tämä pieni osa on samalla lähes poikkeuksetta se, joka on algoritmisesti kaikkein monimutkaisin [Everett, 1995].

Usein reaaliaikajärjestelmät, tai ainakin jotkin niiden osat, toimivat ilman niitä ohjaavaa käyttäjää. Tästä syystä reaaliaikajärjestelmiin on rakennettava kyky havaita virheisiin johtavia ongelmia ja kyky toipua automaattisesti järjestelmän ajautuessa virheelliseen tilaan ennen kuin järjestelmän käsittelemille tiedoille ja laitteille aiheutuu vahinkoja. Näin ollen reaaliaikajärjestelmille asetetaan usein myös tiukat luotettavuusvaatimukset.

2.5. Samanaikaisuuden erityisongelmia

Samanaikaisen ohjelmiston toteutuksessa huomioon otettavia erityispiirteitä ovat mm. *poissulkeminen* (mutual exclusion), *synkronointi* (synchronization), *lukkiutuminen* (deadlock) ja *ajastukset* (timing). Näitä samanaikaiseen ohjelmointiin liittyviä perusasioita käsitellään lähes jokaisessa käyttöjärjestelmiin tai samanaikaiseen ohjelmointiin liittyvässä perusoppikirjassa (esim. [Haikala ja Järvinen, 1994]). Tarkastelemme niitä kuitenkin seuraavassa lyhyesti kuvitteellisen kaupan kassapääte-varastokirjanpito -esimerkin avulla [Haikala ja Märijärvi, 2000].

Poissulkemisella tarkoitetaan tilannetta, jossa usean prosessin samanaikainen tietorakenteen käsittely on suojattava. Esimerkiksi em. kassapäätejärjestelmässä on varmistettava, että kun samaa tuotetta myydään samanaikaisesti usealta eri kassapäätteeltä, varastosaldon päivitys sujuu oikein. Ellei asiaan kiinnitetä toteutuksessa huomiota, voi esimerkiksi syntyä virhetilanne, jossa kaksi kassapääteprosessia hakee samanaikaisesti varastosaldon 1037, vähentää sitä yhdellä ja tallentaa sen paikalleen. Lopputuloksena on saldo 1036, eikä todellista tilannetta vastaava 1035. Kannattaa huomata, että virhetilanteen havaitseminen esimerkiksi testaamalla on erittäin hankalaa, ja se saattaa tulla tuotteessa esille vasta vuosien käytön jälkeen (jolloin se todennäköisesti kirjataan satunnaiseksi häiriöksi tai varastohävikiksi).

Synkronointitilanteessa prosessi joutuu odottamaan jotain (yleensä toisen prosessin aiheuttamaa) tapahtumaa. Esimerkkijärjestelmässä synkronointia tarvitaan kassapääteprosesseissa niiden odottaessa kassapäätteen näppäinpainalluksia. Synkronointi kuuluu yleensä reaaliaikakäyttöjärjestelmän vakiopalveluihin.

Lukkiutumistilanteessa prosessi odottaa tapahtumaa, joka ei tapahdu koskaan. Tyypillinen esimerkki on tilanne, jossa muistitila loppuu kesken ja lopulta kaikki prosessit odottavat muistitilan vapautumista. Poissulkemisen ja synkronoinnin yhteydessä tehdyt ohjelmointivirheet ovat myös tavallisia lukkiutumisen lähteitä.

Ajastusta käytetään esimerkkijärjestelmässä varmuuskopiointiprosessin käynnistämiseen kymmenen sekunnin välein. Tyypillistä ajastukselle on myös

ylärajan liittäminen tapahtuman odotteluun. Esimerkiksi tietoliikennesovelluksessa vastaussanoman odotteluun joudutaan yleensä vastaussanoman viipymisen varalta asettamaan yläraja. Reaaliaikakäyttöjärjestelmissä on yleensä välineet sekä määräajan mittaiseen odotteluun että aikaylärajan asettamiseen odottelulle.

2.6. Prosessirakenteen määrääminen

Reaaliaikaisen ohjelmiston modularisointi tapahtuu periaatteessa samojen periaatteiden mukaan kuin tavallisenkin ohjelman. Prosessi voidaan ymmärtää moduuliksi (tai moduulin aktiiviseksi osaksi), jonka rajapinnan määrittelevät sen vastaanottamat ja lähettämät sanomat. Reaaliaikainen ohjelmisto koostuu ”aktiivisista olioista” eli prosesseista sekä passiivisista olioista eli tavallisista moduuleista. Jos järjestelmän suorituksessa ei tarvita samaan aikaan aktiivisina olevia olioita, ei järjestelmällä myöskään ole tarvetta samanaikaiseen prosessointiin ja se voidaan toteuttaa perinteiseen peräkkäissuoritukseen perustuvana järjestelmänä [Pressman, 1997].

Lähes kaikki kaupalliset järjestelmät koostuvat useista tällaisista aktiivisista olioista, toistensa kanssa kommunikoivista ja samanaikaisesti suoritettavista komponenteista, prosesseista tai säikeistä. Järjestelmän jakamisella useisiin prosesseihin voidaan tavoitella erilaisia hyötyjä. Prosessijako voi yksinkertaistaa järjestelmää loogisesti. Kiireellisiä toimintoja varten tarvitaan korkean prioriteetin omaavia prosesseja ja määrätyin aikavälein toistuvat tehtävät voidaan hoitaa erillisessä prosessissa. Poissulkeminen voidaan toteuttaa palvelinprosessina, jolloin poissulkemisongelmaa ei ole, koska jaettua resurssia käsittelee vain yksi prosessi. Moniprosessoriympäristössä suoritustehoa voidaan lisätä prosesseja lisäämällä. Hajautetussa järjestelmässä ohjelmisto on pakko jakaa eri laitteissa toimiviin prosesseihin [Haikala ja Märijärvi, 2000].

Prosessijaolla on myös haittapuolia, eikä järjestelmää kannata jakaa tarpeettomaan moneen prosessiin. Mahdollisimman pientä prosessien määrää puoltavat monet seikat. Prosessien välinen kommunikointi on yleensä muutamaa kertaluokkaa hitaampaa kuin prosessin sisäiset funktiokutsut, koska kommunikointiin liittyy suoritettavan prosessin vaihtaminen ja sanoman välittäminen. Lisäksi jokainen luotava prosessi varaa yleensä jonkin verran muistitilaa ja käyttöjärjestelmän toiminnot saattavat hidastua prosessien määrän kasvaessa. Suuri prosessien määrä yleensä myös vaikeuttaa järjestelmän testausta.

Lähtökohtana kannattaa siis pitää mahdollisimman pientä prosessien määrää ja prosesseihin jaolle tulee aina olla hyvät perusteet. Hyötyjen suuruus tulee aina olla suurempi kuin siitä aiheutuvat haitat [Jaaksi *et al.*, 1999].

2.7. Samanaikaisten ohjelmistojen testauksesta

Perinteisesti ohjelmistoja on tarkasteltu algoritmeina, jotka muokkaavat syötteensä tulosteiksi. Tällaisessa ns. erätyyppisessä toiminnassa syötteiden voi ajatella olevan täysin tiedossa laskennan alkaessa, eikä esimerkiksi prosessorin nopeus vaikuta laskennan lopputulokseen. Reaaliaikajärjestelmässä ohjelman toimintalogiikkaa ohjaavat useista eri lähteistä tulevat tieto- ja tapahtumavirrat. Tapahtumien keskinäiset ajoitukset eivät yleensä ole ainakaan tarkasti etukäteen tiedossa [Haikala ja Märijärvi, 2000]. Peräkkäissuorituksiin perustuvan ohjelmiston testaaminen perustuu pitkälti ongelmia tuottavien tehtäväsarjojen suorittamiseen eri syötteillä. Samanaikaisten ohjelmistojen testaaminen on ollut erittäin vaikeata, koska ohjelmistossa tapahtuvien toimintojen järjestystä ei ole sidottu, ja näin ollen erilaisten mahdollisten suoritusten määrä kasvaa huimasti [Magee and Kramer, 2006].

Samanaikaisuutta sisältävien ohjelmistojen testaus on tavanomaistakin monimutkaisempaa johtuen niille tyypillisestä epädeterministisyydestä. Yhteenlaskuohjelman tapauksessa tämä tarkoittaisi, että testattaessa ohjelmaa useita kertoja täsmälleen samoilla lukuparijonoilla, eri kerroilla voitaisiin saada erilaisia tuloksia. Tavallisessa sarjallisessa ohjelmassa tämä ei onneksi ole yleensä mahdollista, ja testaus voidaan perustaa siihen, että testit ovat toistettavissa. Reaaliaikajärjestelmissä ohjelmiston toiminta saattaa riippua ajoituksista, jotka eri suorituskerroilla saattavat jonkin verran vaihdella täsmälleen samoillakin syötteillä. Reaaliaikajärjestelmien testauksen suurin ongelma onkin se, että virhetilanteiden toistaminen niiden syiden tutkimiseksi on joskus tavattoman vaikeaa.

Testauksen avulla on mahdollista osoittaa, että ohjelmassa on virheitä; ohjelman virheettömyyttä sillä ei sen sijaan ole mahdollista osoittaa edes yksinkertaisissa tapauksissa. Käytännössä ohjelman testauksessa pystytään kattamaan vain häviävän pieni murto-osa kaikista mahdollisista tilanteista ja reaaliaikajärjestelmien tapauksessa testauksen ”todistusvoima” on vieläkin vähäisempi. Tämä ei tarkoita sitä, että testaukseen ei kannattaisi panostaa, vaan sitä, että ohjelman toimivuuteen ei kannata liiaksi luottaa hyvistä testaustuloksista huolimatta. Tarkastelun perusteella korostuu myös ohjelman huolellisen suunnittelun ja ohjelmoinnin merkitys.

2.8. Ohjelmiston ominaisuuksista

Ohjelmalla voidaan katsoa olevan *ominaisuuksia*. Ohjelman ominaisuus on tosi jokaisella ohjelman mahdollisella ajokerralla. Samanaikaisten ohjelmien oleellisimmat ja tärkeimmät ominaisuuskategoriat ovat *turvallisuus* (safety) ja *elävyys* (liveness, progress). Turvallisuusominaisuudet varmistavat, että mitään

ei-toivottua ei pääse tapahtumaan ohjelmaa suoritettaessa. Elävyydellä taas tarkoitetaan sitä, että jotain toivottua ja tarkoituksenmukaista todella tapahtuu.

Perinteisen peräkkäissuoritukseen perustuvan ohjelman tärkein turvallisuusominaisuus on se, että ohjelman lopetustila on oikein. Elävyysominaisuuden kannalta olennaisinta on ohjelman saapuminen lopetustilaansa, eli ohjelman suorituksen päättyminen. Samanaikaisissa ohjelmissa tärkeitä turvallisuusominaisuuksia ovat onnistunut poissulkeminen sekä lukkiutumistilanteeseen joutumisen mahdottomuus. Sen sijaan elävyyden kannalta tilanne on aivan eri kuin perinteisillä ohjelmilla. Useat samanaikaiset ohjelmat eivät nimittäin koskaan tarkoituksella saavuta lopputilaansa, vaan suoritus jatkuu ikuisessa silmukassa. On siis tärkeää varmistaa, että ohjelma pääsee jossain vaiheessa suoritustaan käsiksi tarvitsemiinsa resursseihin. Sitä voidaan kontrolloida asettamalla ohjelmille *prioriteettitasoja*, joilla tarkoitetaan kiireellisyyden astetta. Korkean prioriteettitason prosessit ovat etusijalla, kun käyttöjärjestelmä jakaa suoritinaikaa sitä tarvitseville prosesseille [Magee and Kramer, 2006].

2.9. Reaaliaikajärjestelmän mallintamisen periaatteet

Reaktiivisten järjestelmien suunnittelun yhteydessä tarve formaaleille menetelmille on vahva. Kuitenkaan ei ole olemassa yhtään yleisesti tunnustettua ”oikeaa” menetelmää, jolla samanaikaisuutta mallinnettaisiin. Reaktiivisille järjestelmille on kehitetty useampia kilpailevia täsmällisiä formalismeja, esimerkkeinä erilaiset prosessialgebrat (process algebra), aikalogiikat (temporal logics), tilamallit ja Petri-verkot.

Kurki-Suonio [1993] antaa seuraavanlaiset yleispätevät mallintamisperiaatteet (modeling principles), jotka pitäisi ottaa huomioon reaaliaikajärjestelmiä suunniteltaessa. Periaatteet ovat kuitenkin käyttökelpoisia ja päteviä kaikissa samanaikaisuutta sisältävissä järjestelmissä.

Eksplisiittinen atomisuus (Explicit atomicity): Osana reaaliaikajärjestelmän suunnittelumallia on välttämätöntä määritellä “atominen toiminto”. Atominen toiminto tai tapahtuma on tarkoin määritelty, rajattu toiminto, jonka voi suorittaa joko yksi tehtävä (task), tai jonka voi samanaikaisesti suorittaa useampi tehtävä rinnakkaisesti. Atomisen toiminnon suorittavat ainoastaan ne tehtävät (”osallistujat”), jotka sitä vaativat, ja sen suorittamisen seuraukset vaikuttavat ainoastaan näihin osallistujiin; sen suorittamisella ei ole mitään vaikutusta muihin järjestelmän osiin.

Limitys (Interleaving): Vaikkakin prosessointi voi olla samanaikaista, tietyn suorituksen historia pitää olla kuvattavissa siten, että se voidaan esittää lineaarisena peräkkäisten toimintojen sarjana. Alkutilasta lähtien ensimmäinen toiminto mahdollistetaan ja suoritetaan. Tämän toiminnon seurauksena

järjestelmän tila muuttuu ja suoritetaan seuraava toiminto. Koska useampikin toiminto on mahdollinen (lähes) jokaisessa tilassa, on mahdollista saada useita erilaisia historioita (trace, jälki), vaikka lähtötilanne olisi kaikissa tapauksissa aivan identtinen. Tämä samanaikaisuuteen liittyvä epädeterminististyys on olennaista myös sen limitetyssä mallintamisessa.

Päättymätömät historiat ja tasapuolisuus (Non-terminating histories and fairness): Suunnittelumallilla pitää pystyä mallintamaan järjestelmä, jonka suoritushistorian oletetaan voivan olla ääretön. Tällä tarkoitetaan sitä, että suoritus jatkuu loputtomasti, tai odottelee jossakin tilassa tapahtumaa tai syötettä, joka saa järjestelmän jatkamaan suoritustaan. Tasapuolisuusvaatimus estää järjestelmän pysähtymisen sattumanvaraisessa kohdassa.

Suljetun systeemin periaate (Closed system principle): Reaaliaikajärjestelmän suunnittelumallin pitäisi kattaa sekä ohjelmisto että ympäristö, jossa ohjelmistoa suoritetaan. Tapahtumat voidaan tällöin jaotella niihin, joista itse järjestelmä on vastuussa, että niihin, joiden oletetaan olevan ympäristön suoritettavissa.

Tilojen rakenteellisuus (Structuring of state): Reaaliaikajärjestelmä voidaan mallintaa joukkona olioita, joilla jokaisella on oma tilansa.

Luvussa neljä esiteltävä FSP-notaatio on näiden Kurki-Suonion esittämien periaatteiden mukainen ja se osoitetaan notaation kunkin periaatteen täyttävän ominaisuuden yhteydessä.

3. Kuvaustekniikoista

Ohjelmistojen tekeminen on spesifikaatioiden tekemistä. Kussakin spesifikaatiossa määritellään ohjelmiston tai sen osan toiminnalliset ominaisuudet, ei-toiminnalliset ominaisuudet sekä reunaehdot ja rajoitteet. Ohjelmistojen ei-toiminnalliset ominaisuudet, reunaehdot ja rajoitteet voidaan yleensä kuvata spesifikaatioissa tavanomaisin keinoin, esimerkiksi suorasanaisena tekstinä, matemaattisina kaavoina tai taulukkoina. Toiminnallisten ominaisuuksien kuvaamiseksi käytetään suorasanaisen kuvailun lisäksi usein erilaisia notaatioita eli kuvaustekniikoita. Yhdistelemällä eri kuvaustekniikoita ja ohjeistamalla niiden käyttöä saadaan erilaisia menetelmiä, mm. luvussa 5 esitetty OMT++-menetelmä. Tässä luvussa käydään läpi muutamia tämän tutkielman kannalta tärkeimpiä spesifikaatioiden tekemisessä käytettyjä notaatioita.

Käsiteltävistä notaatioista on käytännössä esiintynyt monia erilaisia vaihtoehtoja. Tavallisesti erot ovat kuitenkin lähinnä kosmeettisia ja poikkevaan esitystapaan on helppo tottua. Syksyllä 1997 OMG (Object Management Group) hyväksyi notaatiostandardin nimeltä UML (Unified Modeling Language), joka osaltaan on vaikuttanut esitystapojen vakiintumiseen. Standardia on päivitetty säännöllisesti ja viimeisin hyväksytty versio on vuodelta 2007 [UML, 2007]. UML on laaja ja monipuolinen kokoelma erilaisia notaatioita. Tässä tutkielmassa käytetään UML:n mukaista piirtotekniikkaa aina kun se on mahdollista.

3.1. Käyttötapaukset

Ivar Jacobsonin ja kumppanien kirjan Object Oriented Software Engineering – A Use Case Driven Approach [Jacobson *et al.*, 1992] ilmestymisen jälkeen käyttötapaukset on liitetty osaksi useita nykyisin käytössä olevia ohjelmistokehitysmenetelmiä, mm. luvussa 5 esiteltävään OMT++-menetelmään. Niitä pidetään yleisesti hyvänä ratkaisuna erityisesti käyttäjävaatimusten kartoittamiseen. Käyttötapausten idea on kuitenkin vielä niin vakiintumaton, että eri lähteissä siitä esitetään hyvinkin erilaisia näkemyksiä. Jatkossa seurataan lähinnä Jaaksin [1998], Jaaksin ja muiden [1999] ja Erikssonin ja Penkerin [1998] esityksiä.

Käyttötapausten käytöllä pyritään minimoimaan järjestelmälle asetettujen vaatimusten ja sen toimintaan liittyvien väärinkäsitysten mahdollisuutta, sillä ne mahdollistavat ohjelmistosuunnittelijoiden ja loppukäyttäjien keskustelun järjestelmän halutusta toiminnasta. Toisaalta niiden avulla ohjelmistosuunnittelijoiden on myös mahdollista saada tarkempaa tietoa sekä

loppukäyttäjien yleisistä työskentelytavoista että tavoista käyttää tulevaa järjestelmää.

Käyttötapausten idea on yksinkertainen: kuvataan järjestelmän toiminnallisuus joukkona järjestelmän käyttäjien sillä suorittamia tapahtumaketjuja. OMT++-menetelmässä käyttötapaukset kuvataan ainoastaan tekstuaalisesti, mutta muissa menetelmissä apuna saatetaan käyttää muitakin kuvaustekniikoita, esimerkiksi seuraavassa kohdassa esitettäviä tapahtumasekvenssikaavioita. Kuhunkin käyttötapaukseen liittyy yksi tai useampia *suorittajia* (actor), joka suorittaa kyseisen käyttötapauksen.

Käyttötapaus alkaa aina jonkin suorittajan aloitteesta ja päättyy siihen, että järjestelmä on tuottanut ”lisäarvoa” suorittajalleen, ts. suorittaja on saanut jonkin mielekkään tehtäväkokonaisuuden suoritetuksi. Hyvällä käyttötapauksella on mm. seuraavia ominaisuuksia.

- Ymmärrettävyys: käyttötapaus kirjoitetaan niin, että järjestelmän tilaaja ja tulevat käyttäjät pystyvät ymmärtämään ne. Tästä syystä kuvaustavan on oltava mahdollisimman konkreettinen.
- Kuvaavuus asiakkaan vaatimusten kannalta: käyttötapauksessa vältetään ottamasta tarpeettomasti kantaa toteutukseen. Esimerkiksi ”käyttäjä identifioi itsensä” saattaa olla parempi ilmaus kuin ”käyttäjä identifioi itsensä syöttämällä kuusinumeroisen käyttäjätunnuksen ja nelinumeroisen PIN-koodin”. Tämä on usein ristiriidassa edellisen kohdan konkreettisuusvaatimuksen kanssa.
- Testattavuus: käyttötapaukset muodostavat perustan järjestelmätestaukselle. Tästä syystä käyttötapauksen on muodostettava kokonaisuus, joka voidaan ajaa testausvaiheessa yhtenä (tai useampana peräkkäisenä) testitapauksena.
- Koko: käyttötapaus ei saa olla liian laaja. Sopiva koko on yksi A4-arkki.
- Sopiva tarkkuus: käyttötapaukset kattavat tärkeimmät osat toteutuksesta. Kaikkia yksityiskohtia ei voi ottaa käyttötapauksiin mukaan.

Liitteessä 1 on esimerkki käyttötapauksesta. Koska UML ei lainkaan standardoi käyttötapauksen esittämistapaa, noudatetaan esimerkissä Jaaksin [1997] kuvaamaa käytäntöä, jossa poikkeukset normaalista tapahtumien kulusta on merkitty hakasulkeisiin ja niihin reagoiminen on kuvattu erikseen käyttötapauksen alapuolella.

Käyttötapausten ensisijainen tarkoitus on yleensä toimia kommunikointivälineenä kartoitettaessa asiakasvaatimuksia ja kuvattaessa niitä ohjelmistovaatimuksiksi. Ne eivät esimerkiksi korvaa järjestelmän

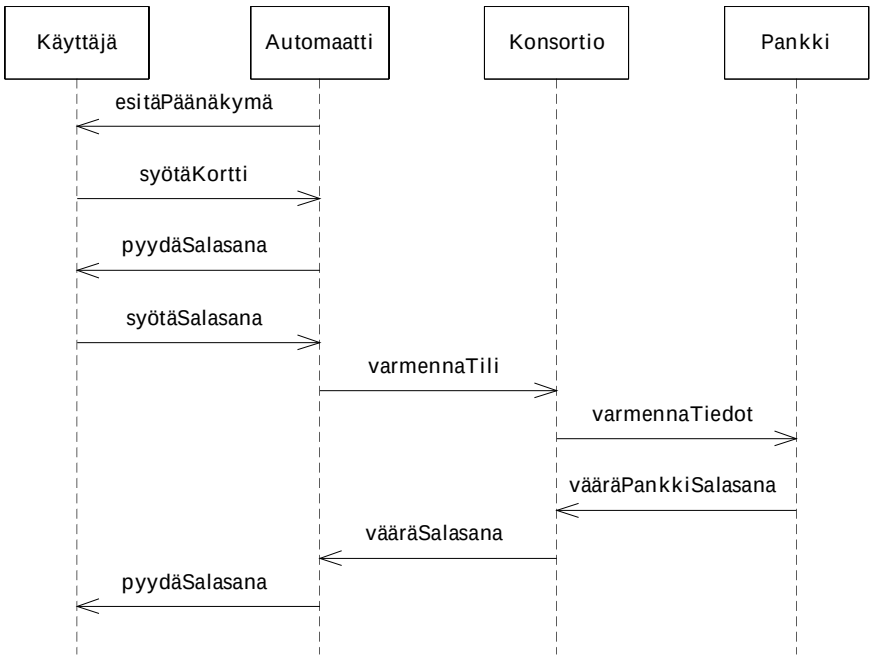
toiminnallisessa määrittelyssä esitettäviä järjestelmän toimintojen kuvauksia. Tästä syystä ei ole mitenkään välttämätöntä, että käyttötapaukset kattavat järjestelmän koko toiminnallisuuden, eikä siihen käytännön syistä johtuen (mm. projektiaikataulut) yleensä pyritäkään.

Käyttötapauksia voidaan hyödyntää määrittelyvaiheen jälkeen myös ohjelmistotyön muissa vaiheissa. Niitä voidaan käyttää esimerkiksi projektisuunnittelun pohjana: ensimmäiseen versioon toteutetaan käyttötapaukset 1 ja 5, toiseen versioon käyttötapaukset 2, 3 ja 4 jne. Käyttötapaukset muodostavat hyvän pohjan myös järjestelmätestaukselle ja käyttöohjeiden kirjoittamiselle. Oliokeskeisissä menetelmissä käyttötapauksia tutkimalla löydetään analyysivaiheen oliomallin keskeiset käsitteet. Toisaalta käyttötapauksia ei voi suoraan käyttää arkkitehtuurisuunnittelun pohjana. Ohjelmassa ei siis voi olla esimerkiksi moduulia ”käyttötapaus a”, eikä projektia myöskään yleensä voi suunnitella siten, että toteutustyö jaetaan suunnittelijoille käyttötapauksittain [Haikala ja Märijärvi, 2000].

3.2. Tapahtumasekvenssikaaviot

Skenaario on kuvaus käyttäjän, järjestelmän komponttien ja ympäristön vuorovaikutuksesta tietyn tapahtumaketjun aikana. Skenaarioiden dokumentoinnin tarpeesta on syntynyt useita erilaisia skenaariopohjaisia notaatioita. Vaikka näillä notaatioilla on erilaisia ominaisuuksia ja niitä on suunniteltu käytettäväksi erilaisten sovellusalueiden yhteydessä, on niillä yhteinen ydin. Tapahtumasekvenssikaaviot (message sequence charts, MSC, communication diagram, event trace diagram, UML:n sequence diagram) ovat yleisimpiä skenaarioiden kuvaamiseen käytettyjä notaatioita. Tapahtumasekvenssikaavioita sanotaan joskus myös skenaarioiksi. OMT++-menetelmässä, samoin kuin tässä tutkielmassa, omaksuttu käytäntö on se, että skenaario on yleisnimi tapahtumasarjan kuvaukselle, ja tapahtumasekvenssikaavio on vain yksi mahdollinen skenaarionkuvaustekniikka. Skenaarion kuvaukseen voidaan käyttää myös jotain muuta dokumentointitapaa, vaikkapa suorasanaista kuvausta (kts. kuvan 3.1 sanallinen selitys).

Tapahtumasekvenssikaaviot soveltuvat erinomaisesti käytettäväksi mitä erilaisimpien tapahtumaketjujen kuvaamiseen. Tapahtumasekvenssikaavion perusidea selviää tarkastelemalla kuvaa 3.1, jossa tapahtumasekvenssikaaviota on käytetty kuvaamaan pankkiautomaatilla tapahtuvaa toimintaa.



Kuva 3.1. Tapahtumasekvenssikaavio pankkiautomaatin käytöstä.

Kaaviossa on kuvattu pystyviivoilla joukko kommunikoivia olioita (käyttäjä, automaatti jne.). Käyttäjän aloittaessa istunnon pankkiautomaatilla, näytetään hänelle automaatin käyttöliittymän päänäkymä. Käyttäjä asettaa luottokortin automaattiin, jolloin automaatti kysyy kortin salasanaa. Käyttäjän annettua salasanan (vahingossa väärin), pyytää automaatti luottokorttikonsortiota varmentamaan käyttäjän kortin ja siihen liittyvän tilin. Konsortiolla ei itsellään ole tietoa tileistä, vaan se pyytää pankkia varmentamaan kyseiset tiedot. Pankki palauttaa tiedon väärästä salasanasta konsortion kautta automaatille, joka pyytää käyttäjää antamaan salasanan uudelleen.

Tapahtumasekvenssikaaviossa tapahtumaketjuun osallistuvat järjestelmän komponentit, ympäristö (järjestelmän ulkoiset elementit) ja käyttäjä ovat kuvattuna pystyviivoina, joita kutsutaan instansseiksi. Instanssien välinen vuorovaikutus kuvataan niiden välillä olevin vaakasuorin nuolin, joita kutsutaan sanomiksi tai viesteiksi (message). Sanomaa kuvaavan nuolen suunta kertoo viestin lähettäjän ja vastaanottajan. Sanomaan liityvä nimi kuvastaa sanoman sisältöä. Niitä instanssien kohtia, joista sanomaa kuvaava nuoli alkaa ja loppuu kutsutaan (lähetys- ja vastaanotto-) tapahtumiksi. Aika etenee tapahtumasekvenssissä ylhäältä alaspäin. Instanssin tapahtuma tapahtuu ennen sen alapuolella samassa instanssissa olevia tapahtumia. Sanoman lähetystapahtuma edeltää aina saman sanoman vastaanottotapahtumaa. Näiden tapahtumasekvenssikaavioiden ytimeen kuuluvien kuvaustapojen lisäksi tapahtumasekvenssikaavioissa voidaan

kuvata mm. erilaisia rajoitteita, viiveitä, instanssien dynaamista luontia ja tuhoamista, komponenttien tilatietoja, parametrillisia sanomia.

Tapahtumasekvenssikaavio on vain esimerkki mahdollisesta kommunikointisekvenssistä. Tavallisesti kommunikointisekvenssillä on useita vaihtoehtoisia etenemispolkuja, joten yksi kaavio ei voi mitenkään kuvata niitä kaikkia. Usein edes mittavakaan joukko kaavioita ei riitä määrittelemään kaikkia mahdollisia tilanteita. Tästä syystä tapahtumasekvenssikaavio ei yleensä yksinään ole riittävä spesifikaatio, vaan pikemminkin havainnollinen kuvaus siitä, miten asioiden on ajateltu normaalitapauksessa tapahtuvan. Tavallinen tapa onkin kuvata vain yksi tai muutamia tyypillisiä tapahtumaketjuja ja mainita muut vaihtoehdot tekstuaalisesti.

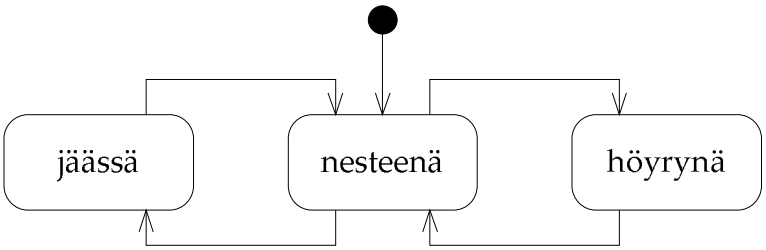
Tapahtumasekvenssikaavioiden käyttö on niiden havainnollisuudesta johtuen jatkuvasti lisääntynyt. UML-notaatiossa kaavioihin on olemassa laajennoksia, esimerkkinä vaikkapa silmukkarakenteen ja ehdollisen rakenteen kuvaukset. Laajennosten käyttö johtaa kuitenkin helposti epähavainnollisiin kuviin.

Tapahtumasekvenssikaaviot ovat läheistä sukua seuraavassa kohdassa esitettäville tilakaavioille. Niitä käytetään kuvaamaan mm. tilakoneiden välistä kommunikointia. Ne ovat yleisesti käytössä mm. tietoliikenteen yhteydessä kuvattaessa sanomanvaihtoa (tähän tarkoitukseen niiden piirtämisestä on olemassa International Telecommunciations Unionin standardikin [ITU Z.120].

3.3. Tilakoneet ja tilakaaviot

Tilakoneet kuvaavat yleensä toiminnan yhden komponentin kannalta tarkasteltuna. Jos toimintaa kuvaamaan tarvitaan useita keskenään kommunikoivia komponentteja, voidaan järjestelmän toiminta kuvata esimerkiksi määrittelemällä kunkin osallistuvan komponentin tilakone. Kuvaus ei useinkaan ole kovin havainnollinen komponenttien välisen kommunikoinnin osalta. Kommunikointia havainnollistamaan voidaan tällaisessa tapauksessa käyttää tapahtumasekvenssikaavioita.

Tilakone eli tila-automaatti on järjestelmä, jolla on joukko toisistaan erotettavissa olevia tiloja ja jonka toiminta on jono tilasiirtymiä näiden tilojen välillä. Tilakone on hyvä spesifioinnin apuväline tilanteissa, joissa järjestelmän toimintaa on mielekästä ajatella jonona tilasiirtymiä tilasta toiseen. Esimerkiksi järjestelmä, jossa käsitellään vettä eri lämpötiloissa, voisi sisältää tilat ”jäässä”, ”nesteenä” ja ”höyrynä” (kuva 3.2).



Kuva 3.2. Yksinkertainen tilakaavio.

Tyypillisiä tilakoneiden sovellusalueita ovat mm. tietoliikenneprotokollien mallintaminen (esimerkiksi tiloja: jouten, sanoma lähetetty, kuittaus saatu...) sekä käsitekaavion yksittäisten käsitteiden ”elinkaaren” analysointi. Tässä luvussa esitellään UML-notaation myötä yleistyneet Harelin *tilasiirtymäkaaviot* (State Transition Diagrams, STD).

3.3.1. Tilakoneen toimintaperiaate

Tilakoneen kannalta oleellisia käsitteitä ovat tilat, tilasiirtymät ja näihin mahdollisesti liittyvät toiminnot. Tilakone odottaa tilassa herätettä, joka aiheuttaa tilasiirtymän johonkin toiseen tilaan (tai takaisin samaan tilaan). Tilasiirtymiin voidaan myös liittää ehtoja, joilla tilasiirtymä voidaan estää tai sallia tilanteesta riippuen. Tilakaavioita on kahta päätyyppiä: Mooren ja Mealyn tila-automaatit [Ward and Mellor, 1985]. Mooren automaatissa tilakoneen toiminnot tapahtuvat automaatin ollessa jossakin tilassa, kun taas Mealyn automaateissa toiminnot tapahtuvat tilasiirtymien yhteydessä. Seuraavissa kohdissa käytetään enimmäkseen alun perin Harelin [1987] kehittämää tilakonenotaatiota. Se on huomattavsti Wardin ja Mellorin tilakonenotaatiota monipuolisempi, ja sillä voidaan kuvata sekä Mooren että Mealyn automaatteja sekä niiden yhdistelmiä. Harelin notaation mukaisten tai sitä läheisesti muistuttavien tilakoneiden käyttö on viime vuosina vähitellen yleistynyt. Niitä käytetään mm. Rumbaughin ja kumppanien [1991] OMT-menetelmässä ja ne on otettu osaksi UML-standardia.

Käytännössä tilojen valinta tilakoneelle on harkinnanvarainen kysymys. Esimerkiksi kuvan 3.2 vettä käsittelevässä järjestelmässä voisi olla kokonaislukumuuttuja ”lämpötila”, jonka arvoalue on -100... +299. Mitä tiloja järjestelmässä pitäisi olla? Tiloja voisi olla esimerkiksi 400 kappaletta – siis yksi joka arvolle. Tällainen tilakone olisi tosin suuruutensa takia käytännön määrittelytyössä todennäköisesti hyödytön. Muita mahdollisia valintoja voisivat olla kaksitilainen järjestelmä: ”jäässä”, ”ei jäässä”, kolmitilainen järjestelmä: ”jäässä”, ”nesteenä”, ”höyrynä” jne. Lisäksi voidaan miettiä, voiko jää muuttua järjestelmässä suoraan höyryksi olematta välillä nesteenä.

Pohdinnassa ei usein ole niinkään kyse luonnonlaeista kuin siitä, miten järjestelmän täytyy pystyä mallintamaan ympäristöään. Tilojen valinta riippuu siis viime kädessä itse sovelluksesta: so. mitä järjestelmän on tiedettävä käsittelemästään vedestä.

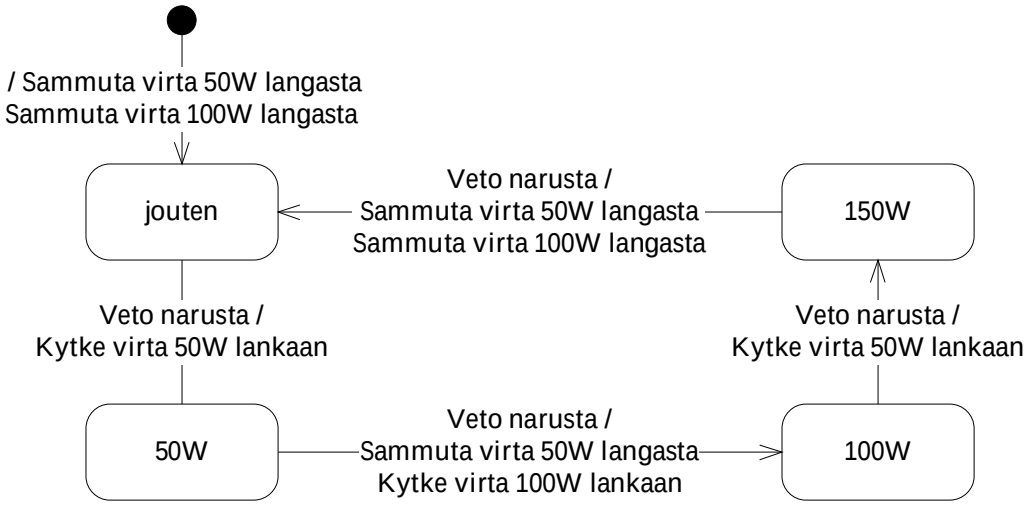
Tilakoneen määrittelemiseksi on lueteltava sen tilat, ja kullekin tilalle on määriteltävä tilan aikana suoritettavat toiminnot (Moore). Edelleen on määriteltävä yksikäsitteisesti, millaisia tilasiirtymiä tilasta voi tapahtua ja mitä toimintoja tilasiirtymiin liittyy (Mealy). Lisäksi tilakoneelle on aina määriteltävä yksikäsitteinen alkutila ja mahdollisesti joukko lopputiloja. UML:n mukaisissa tilakoneissa on mahdollista määritellä myös tilaan saavuttaessa suoritettava toiminto, tilassa ollessa suoritettava toiminto sekä tilasta poistuttaessa suoritettava toiminto.

Tilakoneessa voi olla myös ns. transientteja tiloja. Transientista tilasta lähteviin tilasiirtymiin ei ole asetettu mitään tilasiirtymän laukaisevia signaaleja. Transienttiin tilaan saavuttaessa suoritetaan välittömästi tilasiirtymä seuraavaan tilaan; tilaan ei siis jäädä odottamaan herätettä. Transientteja tiloja käytetään yksinkertaistamaan tilakaavioita.

Tilakone on erittäin havainnollinen ja hyödyllinen spesifioinnin apuväline, kun yritetään selvittää, mitä järjestelmässä pitää voida tapahtua, ja mitä järjestelmässä ei saa tapahtua.

3.3.2. Tilat ja tilasiirtymät

Useimmissa tilakonenotaatioissa tilat kuvataan laatikkoina tai palloina ja tilasiirtymät nuolina. Automaatin toiminnan valaisemiseksi tarkastellaan esimerkkiä, jossa järjestelmän on ohjattava narusta nykäisemällä syttyvää lamppua. Lampussa on kaiksi hehkulankaa: 50W:n lanka ja 100W:n lanka. Ensimmäisellä nykäisyllä on tarkoitus saada 50W:n valaistus, toisella 100W:n valaistus ja kolmannella 150W:n valaistus. Neljäs nykäys sammuttaa valon. Toimintaa vastaava tilakone on esitetty kuvassa 3.3. Yläreunan mustasta pallosta tuleva nuoli osoittaa alkutilan. Tähän tilasiirtymään ei liity mitään ehtoa ja toimintona on tilakoneen alustuksessa tarvittavat toiminnot [Haikala ja Märijärvi, 2000].



Kuva 3.3. Tilakaavio lamppujärjestelmästä.

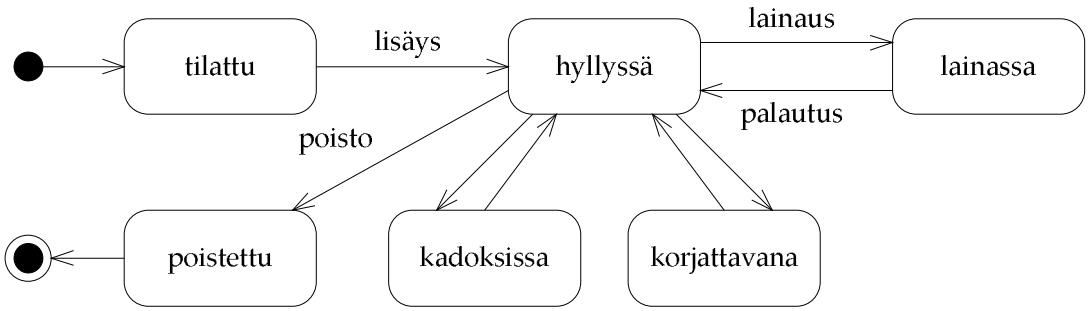
Tilasiirtymän ehtona voi olla jokin tilakoneeseen ulkopuolelta saapuva signaali (”veto narusta”), ajastimen laukeaminen tai ehdollinen lauseke, joka koostuu tilakoneen muuttujista ja mahdollisesti signaalista. Toimintoina voi olla tilakoneesta ulospäin suuntautuva toiminto (”sammuta virta 50W langasta”), tilakoneessa käytettävän muuttujan arvon asettava lauseke tai ajastimen asetus. Jos tilakoneeseen tulee signaali, jonka käsittelytapaa ko. tilassa ei ole määritelty, se ei aiheuta mitään toimintoa.

3.3.3. Tilakoneiden käyttömahdollisuuksia

Tilakoneiden käyttömahdollisuudet ovat monimuotoisemmat kuin ehkä saattaisi arvata. Mitä moninaisimpien laitteiden ja prosessien toimintaa voidaan kuvata tilakoneilla, esimerkkeinä vaikka digitaalikello, matkapuhelin tai pankkiautomaatti. Järjestelmää on kuitenkin yleensä hankala mallintaa pelkästään tilakaaviolla, koska toimintojen määrittelyyn on vain rajoitetut mahdollisuudet. Eräs keino on esittää toiminnon ohjaus tilakaavioiden avulla ja itse toiminnot muulla tavoin; esimerkiksi tapahtumasekvenssikaavioiden avulla. Tilakaaviot toimivat tällaisessa kuvauksessa vain toimintojen käynnistäjinä.

Tilakaavioita voidaan käyttää myös määrittelyn apuvälineenä järjestelmässä tarvittavien tilatietojen ja toimintojen löytämiseen. Tämä tapahtuu tarkastelemalla järjestelmän olioiden ”elinkaaria”; so. niiden mahdollisia tiloja ja tilasiirtymiä. Koska olion tila muuttuu vain toiminnon seurauksena, voidaan toiminnot löytää tilasiirtymiä tarkastelemalla. Esimerkkinä voidaan tarkastella vaikkapa kirjaston tietojärjestelmän oliota kirja. Yksinkertaisimmillaan tilakaaviossa on vain kaksi tilaa: ”hyllyssä” ja ”lainassa”. Vastaavasti voidaan tunnistaa tilasiirtymät aiheuttavat toiminnot:

”lainaus” ja ”palautus”. Hetken miettimisen jälkeen havaitaan, että tarvitaan myös toiminnot ”lisäys” ja ”poisto”. Jos järjestelmän on tunnistettava myös kirjastoon tilatut kirjat, jotka eivät ole vielä saapuneet, on lisättävä tila ”tilattu”. Edelleen voisi käydä ilmi, että tarvitaan vielä tilat ”kadoksissa”, ”poistettu” ja ”korjattavana”.



Kuva 3.4. Kirjastojärjestelmän tilakaavio.

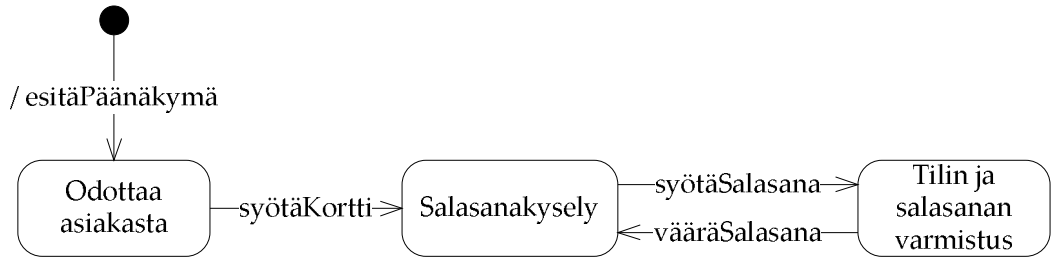
Keskustelussa sovellusalueen asiantuntijoiden kanssa tilakaaviot ovat hyvä tapa havainnollista asioita, herättää keskustelua ja kirjata keskustelun tulokset – toivottavasti kaikkien asianomaisten ymmärtämällä tavalla. Kuvan 3.4 kaavion avulla voi esimerkiksi yrittää lypsää sovellusalueen asiantuntijoilta tietoja kirjan poistamisesta: voidaanko kirja poistaa vain hyllystä, vai pitäisikö se voida poistaa myös joistain muista tiloista, esimerkiksi ”korjattavana”-tilasta. Lopputulos voi olla, että käytännössä kirja voidaan poistaa myös ”korjattavana”-tilasta, mutta tilanteen arvellaan olevan niin harvinainen, että se voidaan käytännössä hoitaa merkitsemällä kirja ensin korjauksesta palautetuksi ja vasta sitten poistetuksi.

Edellinen esimerkki valottaa myös mallintamisen ongelmia: tarkkakaan malli ei juuri koskaan vastaa täysin todellisuutta ja mallia joudutaan vielä tästäkin usein yksinkertaistamaan toteutettavaa järjestelmää mietittäessä. Liian tarkka malli johtaa tarpeettoman monimutkaisen ja kalliin järjestelmän kehittämiseen. Esimerkiksi pienessä kirjastossa saattaisi hyvinkin riittää kolmitilainen järjestelmä: hyllyssä, lainassa ja poistettu. Liian ylimalkainen malli ei taas anna riitävää palvelutasoa. Esimerkiksi kuvan 3.4 mallin mukaisessa järjestelmässä ei voida kirjata tilannetta, jossa kirja on palautettu, mutta ei vielä viety hyllyyn. Suuressa kirjastossa haluttaisiinkin ehkä lisätä tila ”palautushyllyssä” tilojen lainassa ja hyllyssä väliin.

Tilakaavioita voidaan myös käyttää apuna käyttöliittymien määrittelyssä, joka onkin tilakaavioiden ainoa varsinainen käyttötarkoitus OMT++-menetelmässä. Käyttöliittymät ja niiden määrittely eivät kuitenkaan varsinaisesti kuulu tutkielman aihepiiriin, joten tilakaavioiden käyttöä tässä tarkoituksessa ei erikseen esitellä.

3.4. Tilakaavioiden ja tapahtumasekvenssikaavioiden yhteys

Jos tapahtumasekvenssiin osallistuvalle oliolle laaditaan tilakaavio, tapahtuvat tilasiirtymät olioon tulevien nuolien kohdalla. Esimerkkinä tästä on kuvan 3.1 pankkiautomaatin käyttöä kuvaavasta tapahtumasekvenssikaaviosta laadittu automaatin tilakaavio (kuva 3.5).



Kuva 3.5. Automaatin tilakaavio.

Tilakaavioiden avulla voidaan tarkastaa, etteivät tapahtumasekvenssikaaviot sisällä kiellettyjä sekvenssejä: seuraamalla tilakaavion tilasiirtymiä jokaisen saapuvan sanoman kohdalla, voidaan tarkastaa, onko vastaava tilasiirtymä tilakaaviossa mahdollinen. Toisaalta tapahtumasekvenssikaavioiden avulla voidaan hahmotella tilakone, joskaan ei välttämättä yksikäsitteisesti. Toteutuksen kannalta tarkasteltuna tilakaaviot sisältävät siis periaatteessa kaiken sen informaation minkä tapahtumasekvenssikaaviotkin, mutta päinvastainen ei päde: tapahtumasekvenssikaaviot eivät välttämättä määrittele järjestelmän käyttäytymistä yksikäsitteisesti. Toisaalta ne ovat usein tilakaavioita huomattavasti havainnollisempia.

Toteutuksessa tapahtumasekvenssikaavion tapahtumat kuvautuvat usein funktiokutsuiksi, sanoman välitykseksi tai funktiokutsujen palauttamiksi arvoiksi. Kaavioita voidaankin käyttää rajapintojen suunnitteluun: piirretään tapahtumasekvenssikaavioita erilaisista käyttötavoista ja poimitaan rajapinnan funktiot kaavioista.

Oliomenetelmien yhteydessä edellämainittuja kaavioita käytetään usein siten, että ensin laaditaan luokkakaavio. Tämän jälkeen laaditaan tapahtumasekvenssikaavioita, joilla kuvataan olioiden välinen kommunikointi. Luokista ja prosesseista, joilla on ”mielenkiintoista käyttäytymistä”, voidaan laatia lisäksi myös tilakaaviot.

3.5. Tilakoneen esittäminen tekstipohjaisella notaatiolla

Tilakaavioiden graafinen esitystapa rajoittaa vahvasti ongelmakuvauksen monimutkaisuutta. Käyttämällä tekstipohjaista notaatiota tilakoneiden mallintamiseen voimme sen sijaan kuvata suuria ja monimutkaisiakin malleja.

Eräs tällainen notaatio on seuraavassa luvussa tarkemmin esiteltävä FSP (Finite State Processes) [Magee and Kramer, 2006]. Teknisesti katsoen FSP on prosessikalkyyli, joka pohjautuu CCS- (Calculus of Communicating Systems) [Milner, 1989] ja CSP-notaatioihin (Communicating Sequential Processes) [Hoare, 1985]. Kaikilla näillä notaatioilla on algebrallisia ominaisuuksia. FSP:n eroavaisuudet näihin kahteen notaatioon ovat lähinnä syntaktisia; FSP suunniteltiin tietokoneella tulkittavaksi.

Seuraavassa luvussa myöskin esiteltävällä LTSA-työkalulla voidaan FSP:llä kuvatut mallit esittää myös graafisessa muodossa. Lisäksi sillä voidaan yhdistää useita malleja yhdeksi tilakoneeksi. Verrattuna usean samanaikaisen prosessin tilakaavion tutkimiseen paperilla tai näyttöruudulla, on LTSA:n kaltaisen työkalun avulla tehtävä analysointi monin verroin helpompaa. Tutkimuksen kannalta olennaisin asia FSP:ssä kuitenkin on, että sillä määritellyistä malleista voidaan mekaanisesti varmistaa, että malli tyydyttää sille asetetut turvallisuus- ja elävyysvaatimukset.

4. FSP-notaatio

Malleja käytettäessä on yhtenä tarkoituksena turhien yksityiskohtien piilottaminen, jotta voitaisiin keskittyä olennaisiin asioihin. Ohjelmiston samanaikaisuutta mallinnettaessa voidaan siis piilottaa tai yksinkertaistaa tiedon kuvaamista, käyttäjäinteraktiota ja resurssien varaamista. Näiden yksityiskohtien piilottaminen antaa mahdollisuuden keskittyä olennaiseen, eli ohjelmiston samanaikaisuuden suunnitteluun ja analysointiin. Samanaikaisuutta mallinnettaessa erityisasemassa ovat äärelliset tilakaaviot. Tilakaaviot ovat tuttuja useimmille ohjelmistojen kehittäjille. Niitä käytetään olioiden dynaamisen käyttäytymisen määrittelyyn monissa tunnetuissa ohjelmistokehitysmenetelmissä, joista yleisimmät ovat Booch [Booch, 1986], OMT (Object Modeling Technique) [Rumbaugh *et al.*, 1991] sekä nopeasti yleistynyt UML (Unified Modeling Language) [Booch *et al.*, 1998].

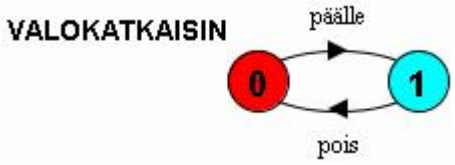
4.1. Prosessin mallintaminen

Prosessin suorittaminen on ohjelman käskyjen peräkkäistä suoritusta. Prosessin tila kullakin ajan hetkellä määräytyy sekä sen eksplisiittisesti määriteltyjen muuttujien arvoista, että sen implisiittisten muuttujien, kuten ohjelmalaskurin ja prosessin tieto- ja osoiterekisterien arvoista. Kun prosessia suoritetaan, muuttuu sen tila suoritettujen käskyjen perusteella. Jokainen käsky koostuu yhdestä tai useammasta atomisesta toiminnosta, joiden suoritus tapahtuu keskeytyksittä. Esimerkkejä tällaisista atomisista toiminnoista ovat mm. rekistereistä tapahtuvat luku- ja kirjoituskäskyt. Abstraktimpi ja yksinkertaisempi tapa kuvata prosessia on ajatella sen aina olevan tilassa, joka muuttuu toiseen tilaan jokaisen atomisen toiminnon suorittamisen seurauksena. Näiden toimintojen sallittu tapahtumajärjestys voidaan määrittää laatimalla ohjelmasta abstrakti kuvaus, joka voidaan esittää tilasiirtymäkaaviolla. Prosessi voidaan siis määritellä äärellisenä tilakoneena.

Kuvassa 4.1 on kuvattuna tilakone valokatkaisimelle, jolla on toiminnot `päälle` ja `pois`. Tässä ja tulevissa kuvissa käytetään seuraavanlaisia käytetylle työkalulle ominaisia kuvauskäytäntöjä: alkutila on aina numeroltaan 0 ja siirtymät on aina piirretty myötäpäivään. Kuvassa 4.1 `päälle` toiminto aiheuttaa siirtymän tilasta 0 tilaan 1, ja toiminto `pois` aiheuttaa siirtymän tilasta 1 tilaan 0. Tämänkaltaista tilakonekuvausta kutsutaan nimellä Labelled Transition System (LTS), sillä siirtymät on aina nimetty toimintojen mukaisesti. Vaikka tämänkaltainen prosessikuvaus on äärellinen, voi prosessin luonne hyvinkin olla ääretön. Tässä esimerkissä tilakaavio antaa mahdollisuuden

seuraavanlaiseen äärettömän pitkään toimintojen ketjuun (vrt. Kurki-Suonion päättymättömien historioiden periaate):

```
päälle -> pois -> päälle -> pois -> päälle ->...
```



Kuva 4.1. Valokatkaisimen LTS-kuvaus.

Luvussa 3 esitetty tilakaavioiden graafinen kuvaus on mainio tapa esittää yksinkertaisia prosesseja. Kun prosessin sisältämien tilojen ja niitä yhdistävien toimintojen määrä kasvaa, muuttuvat kuvaukset nopeasti epäselviksi ja jopa lukukelvottomiksi. Tarvitaan siis yksinkertainen algebrallinen notaatio prosessien kuvaamiseksi. Yksi tällainen notaatio on FSP (Finite State Processes). LTSA:n avulla jokainen FSP-notaatiolla kuvattu prosessi voidaan esittää graafisesti LTS-kuvauksena [Magee and Kramer, 2006].

Tässä luvussa FSP-notaation ominaisuuksien esittämiseen käytetyt esimerkit on tarkoituksellisesti pyritty pitämään mahdollisimman yksinkertaisina, eivätkä ne siksi kuvaa aitoja ohjelmistoesimerkkejä. Myöhemmissä luvuissa notaation avulla mallinnetaan prosesseja, mutta tässä luvussa pääpaino on ainoastaan FSP-notaation ominaisuuksien esittelyssä.

4.1.1. Prosessi, toiminto ja valinta

Tässä kohdassa esittelemme FSP:n peruskäsitteistä prosessin, toiminnon ja valinnan yksinkertaisen kahviautomaatteihin liittyvän esimerkin avulla (kuva 4.2). Kahviautomaatista voi tilata mustan kahvin painamalla punaista nappia. Maitokahvin saa puolestaan painamalla sinistä nappia.

Prosessin tunnuksena on sen alkaminen isolla kirjaimella (**A**UTOMAATTI) ja prosessin määrittely loppuu aina pisteeseen ”.”.

Toiminto-operaattorin ”->” vasemmalla puolella on aina toiminto ja oikealla puolella joko prosessi tai toinen toiminto. Toiminnot erotetaan prosesseista käyttämällä pieniä alkukirjaimia (**p**unainen).

Valinta-operaattorin ”|” käytöllä mahdollistetaan prosessien etenemispolkujen haarautuminen. Ilman valinta-operaattorin mukanaan tuomaa valinnan mahdollisuutta olisi prosessin eteneminen suorituskerrasta riippumatta jo etukäteen tiedossa ja aina sama. Valinta mallintuu tilakoneen tilana, josta on enemmän kuin yksi ulospäin suuntautuva siirtymä. Tällainen tila on esimerkiksi kuvan 4.3 alkutila 0, jolla on kaksi ulospäin suuntautuvaa siirtymää, sininen ja punainen. Kuka tai mikä sitten tekee valinnan näiden

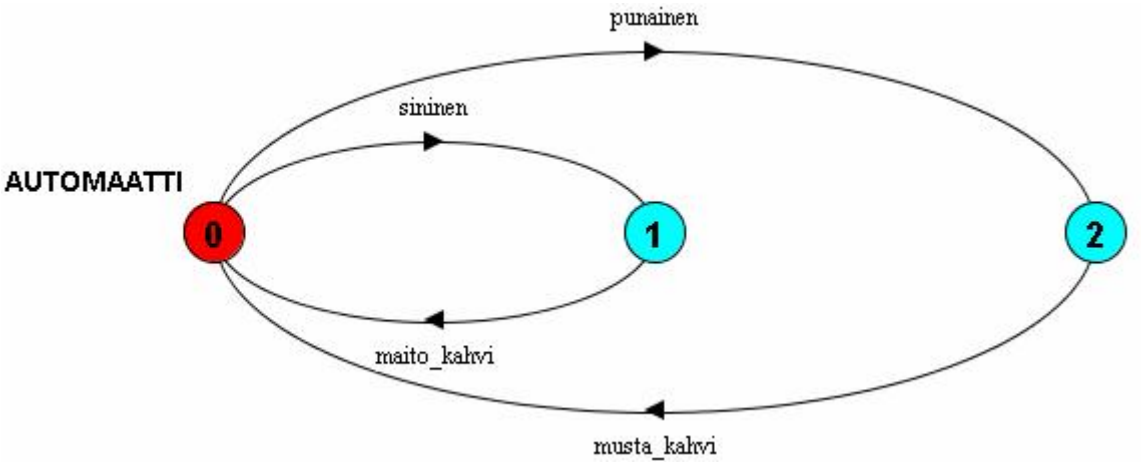
kahden vaihtoehdon välillä? Tässä tapauksessa valinnan tekee ympäristö – joku painaa joko sinistä tai punaista nappia. Valinta-operaattoria käyttämällä voidaan mallintaa valinta myös useamman kuin kahden vaihtoehdon välillä. Automaatin määrittelyyn voitaisiin siis vielä lisätä esimerkiksi `| vihreä -> cappuccino -> AUTOMAATTI`, jolloin ympäristö voisi tehdä valinnan kolmesta vaihtoehdosta punaisen, sinisen tai vihreän napin painalluksella. FSP:llä on mahdollista mallintaa myös epädeterministinen valinta, jolloin valinta tapahtuu satunnaisesti. Epädeterministinen valinta esitetään FSP:ssä samannimisinä toimintoina, jotka johtavat kahteen eri tilaan.

FSP:ssä ei millään lailla erotella prosessin syöte- ja vastetoimintoja. Voidaan kuitenkin ajatella, että ympäristölle tarjottaviin deterministisiin valintoihin kuuluvat toiminnot ovat syötetoimintoja, kun taas vastetoimintojen yhteydessä ympäristölle ei tarjota vaihtoehtoja. Kahviautomaattiesimerkissä syötetoimintoja ovat siis punainen ja sininen, ja vastetoimintoja ovat musta_kahvi ja maito_kahvi.

```
AUTOMAATTI = ( punainen -> musta_kahvi -> AUTOMAATTI
             | sininen -> maito_kahvi -> AUTOMAATTI ).
```

Kuva 4.2. FSP-kahviautomaatti.

Kuvassa 4.3 on esitetty edellä määritellyn automaatin LTS-tilakone. Siitä nähdään, kuinka jokaista FSP-notaation toimintoa vastaa tilakoneen siirtymä tilasta toiseen.



Kuva 4.3. Kahviautomaatin LTS-kuvaus.

Prosessin määrittelyssä voi käyttää apuna lokaaleita prosesseja. Esimerkki lokaaleiden prosessien käytöstä on annettu kuvassa 4.4 kahviautomaatin vaihtoehtoisena prosessimäärittelynä, joka erilaisesta muodostaan huolimatta vastaa täysin alkuperäistä Kahviautomaatti-prosessia. Lokaalit prosessit eivät

ole varsinaisia prosesseja, vaan vastaavat lähinnä jotakin pääprosessin tilaa. Kuvan 4.4 lokaali prosessi SINISTÄ_PAINETTU vastaa siis kuvan 4.3 tilaa 1 ja prosessi PUNAISTA_PAINETTU vastaavasti tilaa 2. Pääprosessi ja sen lokaalit prosessit erotetaan toisistaan pilkulla ”,”.

```
AUTOMAATTI2 = ( punainen -> PUNAISTA_PAINETTU
              | sininen -> SINISTÄ_PAINETTU ),
PUNAISTA_PAINETTU = ( musta_kahvi -> AUTOMAATTI2 ),
SINISTÄ_PAINETTU = ( maito_kahvi -> AUTOMAATTI2 ).
```

Kuva 4.4. Vaihtoehtoinen FSP-kahviautomaatti.

LTS-kaavioita voidaan luoda FSP:hen liittyvän LTS Analysis Tool (LTSA) -ohjelmiston avulla. Ohjelmistoon kuuluu FSP-määritelmien kirjoittamiseen tarkoitetun tekstieditorin lisäksi kääntäjä, jolla FSP-notaatiolla kirjoitetut prosessit voidaan kääntää LTS-kaavioiksi ja joka samalla mekaanisesti tarkastaa prosessien turvallisuus- ja elävyysominaisuudet. Lisäksi ohjelmistoon kuuluu LTSA-Animator niminen työkalu, jonka avulla prosesseja voidaan suorittaa. Kuvassa 4.5 on näytetty, kuinka Animatoria käyttämällä voidaan määrittelystä tuottaa sarja tapahtumia. Käyttäjä voi valita mallin tarjoamista toiminnoista mieleisensä. Mallin toiminnot on listattu kuvan 4.5 oikealla puolella ja kussakin tilassa valittavana olevat toiminnot on merkitty. Kuvan 4.5 vasemmalla puolella esitetty suoritettujen toimintojen sarja on saanut AUTOMAATTI-prosessin tilaan, jossa maito_kahvi on ainoa mahdollinen valittavana oleva toiminto. Tällaista tapahtumien sarjaa, joka muodostuu yhtä tai useampaa prosessia suoritettaessa, kutsutaan suorituksen jäljeksi (trace).

Kuva 4.5. LTSA-Animator.

4.1.2. Indeksoidut toiminnot ja parametrisoidut prosessit

Sekä lokaalit prosessit että toiminnot voidaan indeksoida FSP:llä. Tällä tavoin pystytään mallintamaan prosesseja ja toimintoja, joille voidaan antaa eri arvoja. Parametrisointi ja indeksointi parantavat huomattavasti notaation ilmaisukykyä. Indekseillä täytyy aina olla rajattu arvoalue, jonka sisään indeksien arvojen on mahduttava. Tällä tavoin varmistetaan FSP:llä määriteltyjen mallien äärellisyys, joka taas on vaatimuksena sille, että mallit ovat mekaanisesti analysoitavissa. Kuvan 4.6 prosessi esittää puskuria, joka voi sisältää yhden arvon. Puskurille voidaan antaa syötteenä luku väliltä 0-3, jonka puskuri tämän jälkeen tulostaa.

```
PUSKURI = ( sisään[i:0..3] -> tulosta[i] -> PUSKURI ).
```

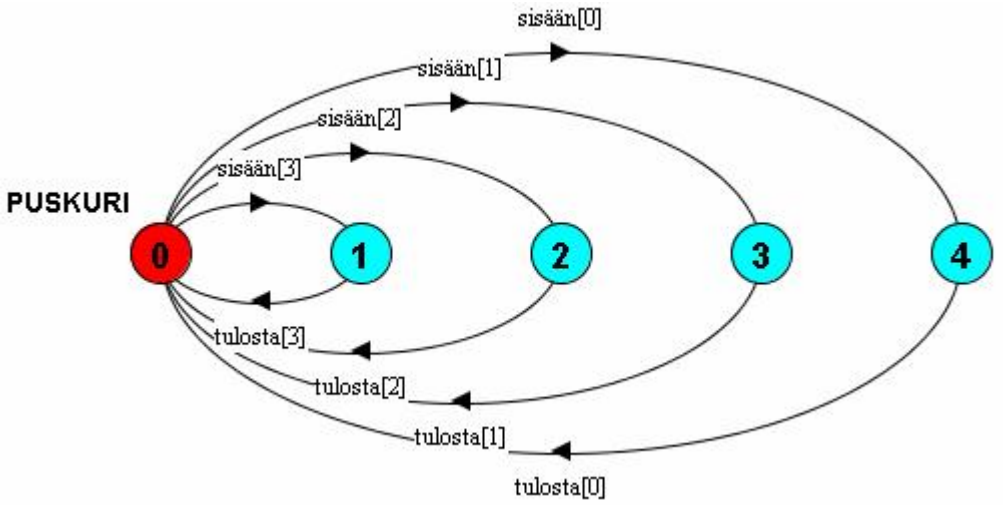
Kuva 4.6. FSP-puskuri.

Edellä kuvatun indeksoituja toimintoja käyttävän tilakoneen voi esittää myös käyttämällä indeksoitua lokaaliprosessia kuvan 4.7 esittämällä tavalla. Koska siinä käytetään kahta saman arvoalueen muuttujaa, määritellään nille yhteinen arvoalue.

```
range R = 0..3
PUSKURI = ( sisään[i:R] -> LUKUPUSKURI[i] ),
LUKUPUSKURI[i:R] = ( tulosta[i] -> PUSKURI ).
```

Kuva 4.7. FSP-puskuri2.

Ylläolevien määritelmien identtisyys on havaittavissa niiden tilakoneista, jotka ovat identtiset (kuva 4.8).



Kuva 4.8. Puskurin LTS-kuvaus.

Sekä prosessille että toiminnolle voidaan määritellä useampikin indeksi. FSP-notaatio tukee indekseillä tehtäviä aritmeettisia operaatioita. Kuvan 4.9 esimerkki havainnollistaa useamman indeksin ja aritmeettisten operaatioiden käyttöä FSP:ssä. Esimerkin prosessi saa syötteenä kaksi kokonaislukua, joista se laskee summan.

```
const N = 1
range T = 0..N
range R = 0..2*N
SUMMA = ( sisään[a:T][b:T] -> YHTEENSÄ[a+b] ),
YHTEENSÄ[s:R] = ( tulosta[s] -> SUMMA ).
```

Kuva 4.9. FSP-summa.

Prosessit voidaan myös parametrisoida, jotta yhdellä kuvauksella voitaisiin mallintaa useammalla eri parametriarvolla alustettavia prosesseja. Esimerkiksi kuvan 4.6 puskuriprosessi voidaan parametrisoida ottamaan syötteinään lukuja 0-N kuvan 4.10 esittämällä tavalla. Parametrit kirjoitetaan isolla alkukirjaimella, ja niillä täytyy olla oletusarvo.

```
PUSKURI(N=3) = (sisään[i:0..N] -> tulosta[i] -> PUSKURI).
```

Kuva 4.10. Parametrisoitu puskuri.

4.1.3. Vahditut toiminnot

Toiminto on usein tarpeen määritellä ehdollisena, jolloin toiminto on suoritettavissa ainoastaan tietyn ehdon täyttyessä. Tällaisina toimintovahteina FSP:ssä voidaan käyttää Boolean-tyyppisiä (tosi/epätosi) muuttujia. FSP tukee ainoastaan kokonaislukutyyppejä ja Boolean-ehdot esitetään C-, C++- ja Java-ohjelmointikielistä tutulla syntaksilla; arvo 0 on epätosi ja muut arvot tosi.

Kuvassa 4.11 on esitetty AJASTIN-prosessi, joka laskee aikaa (lukuja) alaspäin ja päästää hälytysäänen kun aika loppuu. Ajastuksen voi lopettaa milloin tahansa.

```
AJASTIN(N=3) = ( aloita -> AJASTIN[N],
AJASTIN[i:0..N] = ( when(i>0)  tik -> AJASTIN[i-1]
                  | when(i==0) piip -> STOP
                  | lopeta -> STOP ).
```

Kuva 4.11. FSP-ajastin.

Kuvan 4.11 AJASTIN-prosessissa esiintyvä STOP on yksi FSP:ssä määritellyistä erityisprosesseista. STOP-prosessista ei ole ulospäin johtavia toimintoja ja näin ollen prosessin suoritus loppuu sen saavuttua prosessin STOP määrittelemään tilaan.

AJASTIN-prosessin mahdolliset suoritusjäljet on kuvattu alla:

```
aloita -> lopeta
aloita -> tik -> lopeta
aloita -> tik -> tik -> lopeta
aloita -> tik -> tik -> tik -> lopeta
aloita -> tik -> tik -> tik -> piip.
```

Prosessin joutuminen tilaan, josta ei ole ulospäin suuntautuvia tilasiirtymiä, johtaa prosessin lukkiutumiseen. LTSA havainnoi prosessin kääntämisen yhteydessä tällaiset tilat ja esittää lyhimmän lukkiutumistilanteeseen johtavan toimintojen ketjun, joka tämän esimerkin yhteydessä on seuraavanlainen:

```
Trace to DEADLOCK:
    aloita
    stop.
```

4.2. Samanaikaisten prosessien mallintaminen

Edellisessä kohdassa esitettiin, kuinka prosessi voidaan mallintaa abstraktina tilakoneena, jonka tilat vaihtuvat atomisten toimintojen suorituksen myötä. Prosessin suorittamisen yhteydessä syntyy atomisten toimintojen sarja, prosessin jälki. Tässä kohdassa esitetään, miten FSP-notaatiolla voidaan mallintaa useamman samanaikaisen prosessin muodostamia järjestelmiä.

Samanaikaisuuden kannalta on olennaista pohtia prosessien välisen suoritusnopeuden mallintamista. Prosessin suoritusnopeus riippuu monista tekijöistä, kuten esimerkiksi prosessorin nopeudesta, prosessorien lukumäärästä, sekä siitä strategiasta, jonka perusteella käyttöjärjestelmä jakaa prosesseille suoritinaikaa (scheduling). Koska tarkoituksena on kuitenkin suunnitella samanaikaisia ohjelmia, jotka toimivat näistä tekijöistä riippumatta, on FSP:ssä valittu lähestymistapa, jossa ei lainkaan oteta kantaa prosessien väliseen suoritusnopeuteen, vaan prosesseja suoritetaan mielivaltaisella nopeudella. Tällä tarkoitetaan sitä, että prosessi saa käyttää toiminnon suorittamiseen mielivaltaisen pitkän ajan. Näin voidaan abstrahoida ajan kuluminen kokonaan pois mallista. Tällä lähestymistavalla on se huono puoli, ettei mallinnettavan järjestelmän reaaliaikaominaisuuksista voida tehdä mitään päätelmiä. Olennaisempaa on kuitenkin se, että tällä tavoin järjestelmästä voidaan todentaa muita ominaisuuksia tekemättä minkäänlaisia olettamuksia käyttöjärjestelmästä ja laitteistosta. Tämä riippumattomuus on olennaista järjestelmän siirrettävyyden (portability) kannalta.

FSP:ssä prosessien rinnakkaisuutta mallinnetaan limityksen avulla, eikä sen avulla ole mahdollista mallintaa todellista, useammassa prosessorissa yhtä aikaa tapahtuvaa rinnakkaista toimintaa. Toiminnon a katsotaan olevan samanaikainen toiminnon b kanssa, jos mallissa on mahdollista suorittaa toiminnot järjestyksessä a->b tai b->a. Koska FSP:ssä ei mallinneta aikaa, ei toimintojen todellinen rinnakkainen suoritus useammassa prosessorissa vaikuta ohjelman todistettaviin ominaisuuksiin.

Yhden prosessin toiminnot tapahtuvat aina mallissa esitettyjen sääntöjen mukaisessa järjestyksessä. Koska eri prosessien suoritus kuitenkin etenee mielivaltaisella nopeudella, ovat eri prosessien toiminnot mahdollista suorittaa mielivaltaisessa järjestyksessä (vrt. Kurki-Suonion limitys periaate). Tämänkaltainen mielivaltainen limitys vaikuttaa hyvältä tavalta mallintaa prosessien samanaikaista suoritusta, sillä sen avulla voidaan abstrahoida pois prosessorien suorittamiin prosessivaihtoihin liittyvät keskeytykset. Näiden keskeytysten ajoituksia kun ei yleisesti ottaen voida millään tavalla ennakoida.

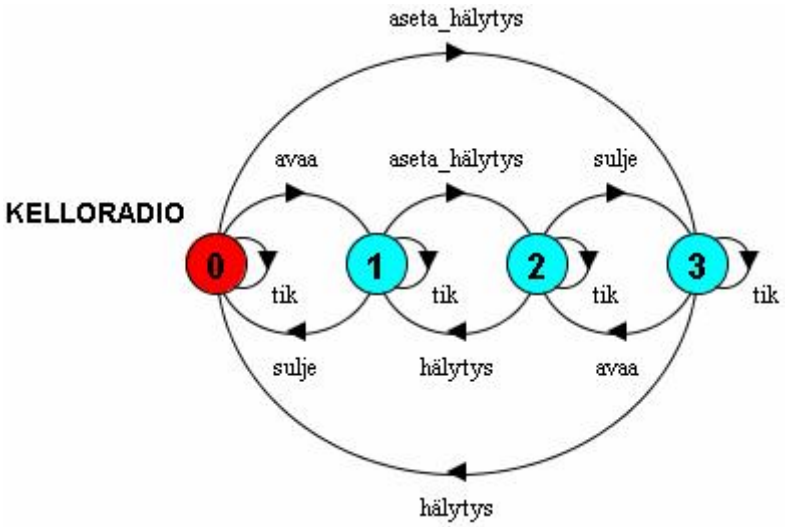
4.2.1. Prosessien samanaikainen suoritus

Prosessien samanaikaista suorittamista varten FSP:ssä on määritelty rinnakkaisen koosteen operaattori ”||” (parallel composition operator). Samanaikaisesti suoritettavat prosessit siis yhdistetään uudeksi koosteprosessiksi kuvassa 4.12 esitetyn esimerkin mukaisesti. Esimerkissä on mallinnettu kelloradio, joka on jaettu kahteen samanaikaisesti suoritettavaan osaan: kelloon, jonka voi asettaa hälyttämään, ja radioon, jonka voi avata ja sulkea.

```
KELLO = ( tik -> KELLO
        | aseta_hälytys -> HÄLYTYS ),
HÄLYTYS = ( tik -> HÄLYTYS
          | hälytys -> KELLO ).
RADIO = ( avaa -> sulje -> RADIO).
||KELLORADIO = ( KELLO || RADIO ).
```

Kuva 4.12. FSP-kelloradio.

Näin saatu koosteprosessi on mahdollista esittää samanlaisena tilakoneena kuin tavallinen yksinkertainen prosessikin. Koostetta kuvaavassa tilakoneessa on esitettynä kaikki mahdolliset prosessien toimintojen välillä tapahtuvat limitykset. Kelloradion tilakone on esitetty kuvassa 4.13.



Kuva 4.13. Kelloradion LTS-kuvaus.

Koosteprosessin tilakone muodostuu sen jäsenten karteesisesta tulosta. Kelloradioesimerkissä prosessin KELLO ollessa aseta_hälytys-toiminnon suorittamisen jälkeen tilassa (1) ja prosessin RADIO ollessa avaa-toiminnon suorittamisen jälkeen tilassa (1), on yhdiste KELLORADIO tilassa (<1,1>). Tätä tilaa kuvaa KELLORADIO-koosteprosessin tila 2, jossa hälytys on asetettu ja radio on avattu. Tilaan on mahdollista saapua kahdesta eri tilasta riipuen toimintojen avaa ja aseta_hälytys suoritusjärjestyksestä (vrt. Kurki-Suonion tilojen rakenteellisuuden periaate).

Kuvasta 4.13 voidaan havaita, että toiminto avaa on samanaikainen toimintojen aseta_hälytys ja hälytys kanssa. Malli mahdollistaa näiden toimintojen suorituksen missä järjestyksessä tahansa, pitäen silti kiinni rajoitteesta, jonka mukaan toiminto aseta_hälytys tapahtuu ennen toimintoa hälytys. Hälytys on siis asetettava ennen hälytyksen tapahtumista, mutta radio on mahdollista avata missä vaiheessa tahansa.

Koska useamman prosessin muodostama rinnakkainen kooste on itsessään prosessi, koosteprosessi, voidaan sitä käyttää muiden koosteprosessien määrittelyssä tavallisen primitiiviprosessin asemesta. Koosteprosessesista voidaan siis koostaa yhä suurempia koosteprosesseja.

Kurki-Suonion suljetun systeemin periaatteen mukainen mallintaminen saavutetaan mallintamalla järjestelmän ulkopuoliset elementit aivan samalla tavoin kuin järjestelmän sisäisetkin prosessit, ja koostamalla ne yhteen järjestelmän sisäisten prosessien kanssa.

4.2.2. Jaetut toiminnot

Edellä esitetyssä Kelloradio-esimerkissä prosesseilla KELLO ja RADIO ei ole yhteisiä, samannimisiä toimintoja. Prosessien yhteisiä toimintoja kutsutaan

jaetuiksi toiminnoiksi (shared actions). Prosessien välinen vuorovaikutus mallinnetaan näillä jaetuilla toiminnoilla. Prosessien jakamattomat toiminnot voivat lomittua mielivaltaisesti, mutta eri prosessien jakaman toiminnon suorittaminen tapahtuu samanaikaisesti toiminnon jakavissa prosesseissa (vrt. Kurki-Suonion eksplisiittisen atomisuuden periaate). Kuvassa 4.14 on esitetty esimerkki jaetusta toiminnosta.

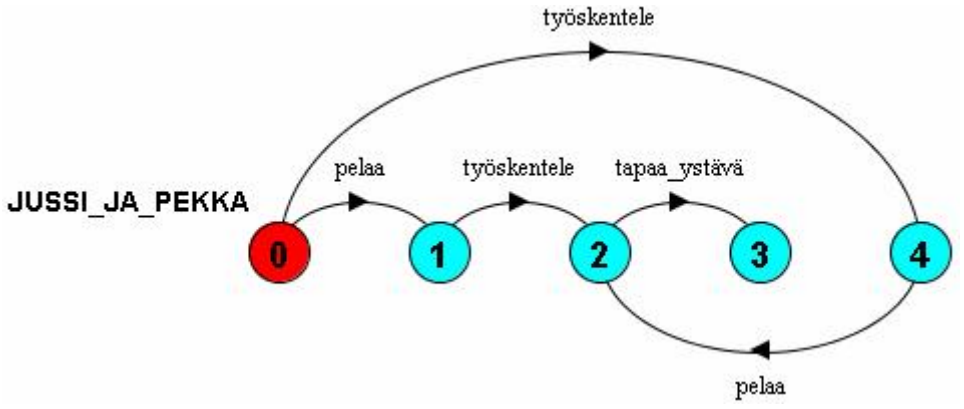
```
JUSSI = ( työskentele -> tapaa_ystävä -> STOP ).
PEKKA = ( pelaa -> tapaa_ystävä -> STOP ).
||JUSSI_JA_PEKKA = ( JUSSI || PEKKA ).
```

Kuva 4.14. FSP-Jussi ja Pekka.

Kuvan 4.14 FSP-määrittelyn JUSSI_JA_PEKKA-koosteen suorituksen mahdolliset jäljet ovat:

```
työskentele -> pelaa -> tapaa_ystävä
pelaa -> työskentele -> tapaa_ystävä
```

Esimerkin jakamattomat toiminnot, `työskentele` ja `pelaa`, ovat samanaikaisia ja kumpi tahansa toiminnoista voi tapahtua ennen toista. Kummankin näistä toiminnoista on kuitenkin tapahduttava ennen jaetun toiminnon `tapaa_ystävä` suorittamista. Jaettu toiminto `tapaa_ystävä` synkronoi prosessien JUSSI ja PEKKA suorituksen. FSP-notaatio ei ota kantaa siihen, minkä prosessin aloiteesta jaettu toiminto suoritetaan. Esimerkin koosteen JUSSI_JA_PEKKA LTS-tilakaavio on esitetty kuvassa 4.15.



Kuva 4.15. JUSSI_JA_PEKKA-prosessin LTS-kuvaus.

Prosessien välistä synkronointia tarvitaan monessa eri yhteydessä, esimerkiksi niin sanottujen kättelyprotokollien mallintamisessa. Kättelyprotokollaa käytetään laajalti prosessien välisessä vuorovaikutuksessa.

Edellä esitetyssä esimerkissä synkronointi tapahtui kahden prosessin välillä. On kuitenkin mahdollista, että useampikin prosessi jakaa yhteisen toiminnon ja ottaa siten osaa synkronointiin.

4.2.3. Prosessien nimeäminen

Järjestelmän mallintamisessa on usein tarpeellista käyttää samasta prosessista useampia instansseja. Oletetaan siis, että prosessi VALOKATKAISIN on määritelty kuvan 4.16 mukaisesti.

```
VALOKATKAISIN = ( päälle -> pois -> VALOKATKAISIN ).
```

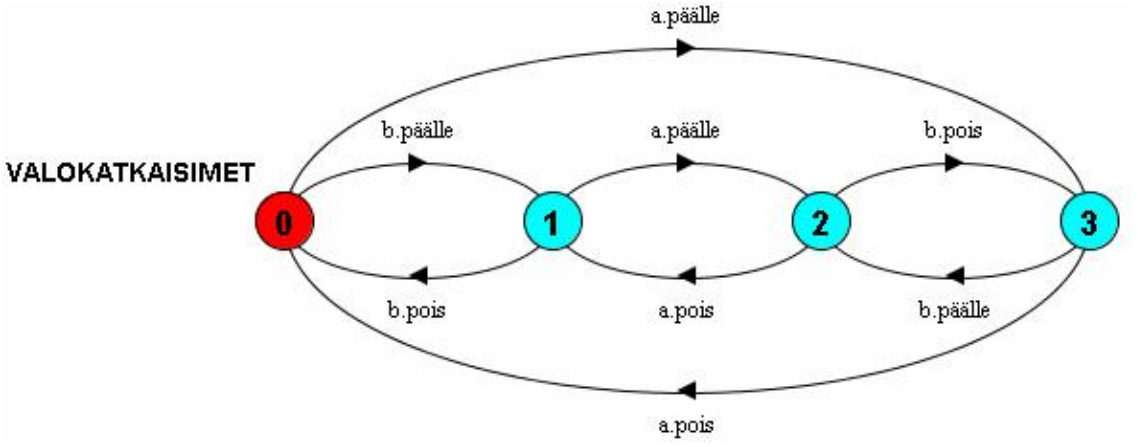
Kuva 4.16. FSP-valokatkaisin.

Jos nyt halutaan mallintaa järjestelmää, jossa on kaksi valokatkaisinta, ei yksinkertainen rinnakkainen kooste (VALOKATKAISIN||VALOKATKAISIN) ole toimiva määrittely. Tällä koosteella on täysin identtinen LTS-tilakone yhden VALOKATKAISIN prosessin kanssa (kuva 4.1). Se johtuu siitä, että prosessien samannimiset toiminnot, päälle ja pois, synkronoivat prosessit. Jotta kummankin prosessin toiminnot tulkittaisiin erillisinä, jakamattomina toimintoina, täytyy varmistaa, että prosessien toiminnot eivät ole samannimiset. FSP:ssä tämä tapahtuu nimeämällä prosessit. Edellä esitetty kahden valokatkaisimen koosteprosessi voidaan määrittää kuvan 4.17 esittämällä tavalla.

```
VALOKATKAISIMET = ( a:VALOKATKAISIN || b:VALOKATKAISIN ).
```

Kuva 4.17. FSP-valokatkaisimet.

Kuva 4.18 esittää VALOKATKAISIMET koosteprosessin LTS-tilakoneen, josta voidaan hyvin havaita nimeämisen vaikutukset tilakoneen toimintaan.



Kuva 4.18. VALOKATKAISIMET-prosessin LTS-kuvaus.

Prosessien nimeäminen voidaan tehdä myös prosessitaulukkoa käyttämällä. Kuvan 4.19 määrittely mallintaa kolmen valokatkaisimen järjestelmän, joissa valokatkaisin prosessien nimet ovat s[1], s[2] ja s[3].

```
||VALOKATKAISIMET(N=3) = (s[i:1..N]:VALOKATKAISIN).
```

Kuva 4.19. Prosessitaulukko.

Toinen tapa prosessin toimintojen nimeämiseen on määritellä prosessille etuliite, joka lisätään jokaisen prosessin toiminnon eteen. Esimerkkinä prosessin etuliitteen käytöstä mallinnetaan yksinkertainen järjestelmä, jossa kaksi käyttäjää jakaa jonkin resurssin siten, että ainoastaan yksi käyttäjä kerrallaan voi käyttää jaettua resurssia. Esimerkin avulla esitetään samalla kuinka FSP:n avulla voidaan mallintaa prosessien välinen poissulkeminen. Käyttäjä- ja resurssiprosessit on esitetty kuvassa 4.20.

```
KÄYTTÄJÄ = ( varaa -> käytä -> vapauta -> KÄYTTÄJÄ ).
RESURSSI = ( varaa -> vapauta -> RESURSSI ).
```

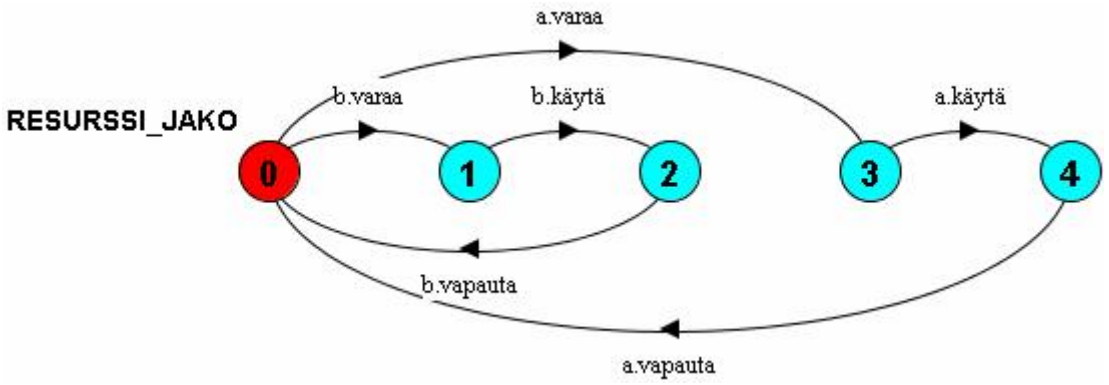
Kuva 4.20. Käyttäjä ja resurssi.

Järjestelmän kaksi käyttäjää voidaan mallintaa nimeämällä käyttäjät edellä esitetyn VALOKATKAISIMET-esimerkin mukaisesti a:KÄYTTÄJÄ ja b:KÄYTTÄJÄ. Tämä johtaa siihen, että resurssia on nyt varaamassa kaksi erinimistä toimintoa (a.varaa ja b.varaa). Näin ollen prosessin RESURSSI toiminnot pitää nimetä uudelleen, jotta ne synkronoituisivat KÄYTTÄJÄ-prosessien vastaavien toimintojen kanssa. Tämä tehdään määrittelemällä RESURSSI-prosessi käyttämään etuliitteitä a ja b kuvan 4.21 mukaisesti.

```
||RESURSSI_JAKO =
      ( a:KÄYTTÄJÄ || b:KÄYTTÄJÄ || {a,b}::RESURSSI ).
```

Kuva 4.21. FSP-resurssijako.

Kuvassa 4.22 esitetystä LTS-tilakaaviosta voidaan havaita, että ainoastaan yksi KÄYTTÄJÄ-prosessi kerrallaan pääsee käyttämään resurssia.



Kuva 4.22. Resurssijaon LTS-kuvaus.

4.2.4. Toimintojen nimeäminen ja piilottaminen

FSP mahdollistaa myös prosessien yksittäisten toimintojen nimeämisen uudelleen. Uudelleennimeämistä käytetään yleensä varmistamaan koosteprosessien tiettyjen toimintojen välistä synkronointia. Toimintoja voidaan nimetä uudelleen niin primitiivi- kuin koosteprosesseissa. Esimerkkinä toimintojen uudelleennimeämisestä mallinnetaan yksinkertainen asiakas-palvelin järjestelmä, jossa asiakasprosessi käyttää palvelinprosessin tarjoamaa palvelua. Asiakas- ja palvelinprosessit ovat esitettynä kuvassa 4.23.

```
ASIAKAS = ( kutsu -> odota -> jatka -> ASIAKAS ).
PALVELIN = ( pyyntö -> palvele -> vastaa -> PALVELIN ).
```

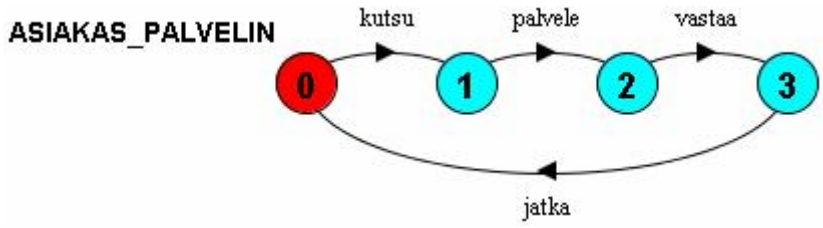
Kuva 4.23. FSP-asiakas ja -palvelin.

Kuten kuvasta 4.23 voidaan havaita, ei ASIAKAS- ja PALVELIN-prosesseilla ole jaettuja toimintoja, eivätkä ne synkronoi minkään toiminnon kohdalla. Käyttämällä toimintojen uudelleennimeämistä voimme liittää ASIAKAS-prosessin kutsu-toiminnon PALVELIN-prosessin pyyntö-toimintoon. Samoin voimme yhdistää odota- ja vastaa-toiminnot. Prosesseista tehty kooste on esitetty kuvassa 4.24 ja sen tilakone kuvassa 4.25.

```
||ASIAKAS_PALVELIN = ( ASIAKAS||PALVELIN )
/ {kutsu/pyyntö, vastaa/odota}.
```

Kuva 4.24. Toimintojen yhdistäminen.

Uudelleennimeämisen vaikutuksesta PALVELIN-prosessin pyyntö-toiminto korvataan toiminnolla kutsu. Toiminto vastaa korvaa vastaavasti ASIAKAS-prosessin odota-toiminnon. Näin prosessit synkronoituvat jaettujen toimintojen kutsu ja vastaa suorittamisen yhteydessä.



Kuva 4.25. ASIAKAS_PALVELIN-prosessin LTS-kuvaus.

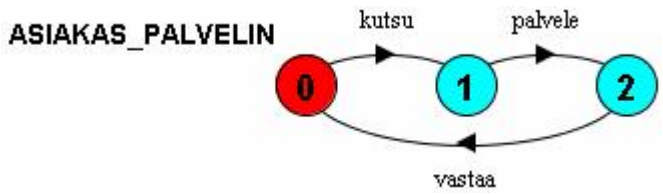
Toimintoja voidaan myös kätkeä. Kätketyt toiminnot, joita kutsutaan myös hiljaisiksi toiminnoiksi, eivät vaikuta millään tavoin toisten prosessien suoritukseen, eikä eri prosessien kätkettyjä toimintoja pidetä jaettuina toimintoina. Toimintojen kätkemisellä pyritään vähentämään suurten mallien

monimutkaisuutta. Toimintoja voidaan kätkeä kahdella tavalla. Voimme määritellä listan prosessin kätkettävistä toiminnoista, mutta toisinaan on helpompi määritellä, mitkä toiminnot eivät ole hiljaisia. Kuvassa 4.26 esitetään nämä kaksi tapaa kätkeä ASIAKAS_PALVELIN-koosteprosessin toiminto `jatka`. Kätkemisen tuloksena saatu minimoitu tilakone on kuvassa 4.27. Kumpikin määrittely tuottaa identtisen tilakoneen.

```
||ASIAKAS_PALVELIN = ( ASIAKAS||PALVELIN )
/ {kutsu/pyyntö, vastaa/odota}.
\ {jatka}. // kätketään toiminto jatka
```

```
||ASIAKAS_PALVELIN = ( ASIAKAS||PALVELIN )
/ {kutsu/pyyntö, vastaa/odota}.
@{kutsu,palvele,vastaa}. // kätketään muut kuin nämä
```

Kuva 4.26. Toiminnon kätkeminen.



Kuva 4.27. Minimoitu ASIAKAS_PALVELIN-prosessin LTS-kuvaus.

Tässä ja edellisessä luvussa esiteltiin tyypillisimmät ohjelmistojen kehitystyössä käytetyt kuvaustekniikat eli käyttötapaukset, tapahtumasekvenssikaaviot ja tilakoneet eri muodoissaan. Luokkakaaviot sivuutettiin, sillä ne eivät sisällä samanaikaisuuden mallintamisen kannalta olennaista tietoa. Seuraavaksi esitellään yksityiskohtaisesti eräs ohjelmistokehitysmenetelmä, jossa näitä erilaisia kuvaustekniikoita käytetään ohjeistetusti suurten reaaliaikaisten ohjelmistojen kehittämisessä.

5. OMT++

Ohjelmistokehitysmenetelmiä on useita ja niiden pohjalta räätälöityjä yrityskohtaisia menetelmiä lähes yhtä monta kuin yrityksiäkin. Harvat mallit ovat todella yleiskäyttöisiä, eikä niitä voi asettaa paremmuusjärjestykseen. Monet asiat vaikuttavat valittuun menetelmään ja sen muokkaamiseen. Yrityksen henkilökunnan koko ja ohjelmiston sovellusalue ovat merkittäviä menetelmän valintaan vaikuttavia tekijöitä. Tässä luvussa keskitytään OMT++-menetelmän kuvaamiseen suurten samanaikaisuutta sisältävien ohjelmistojen kehitysmenetelmänä ja siitä syystä esitystä on pyritty painottamaan ohjelmistojen käyttäytymistä kuvaaviin vaiheisiin.

5.1. Historia

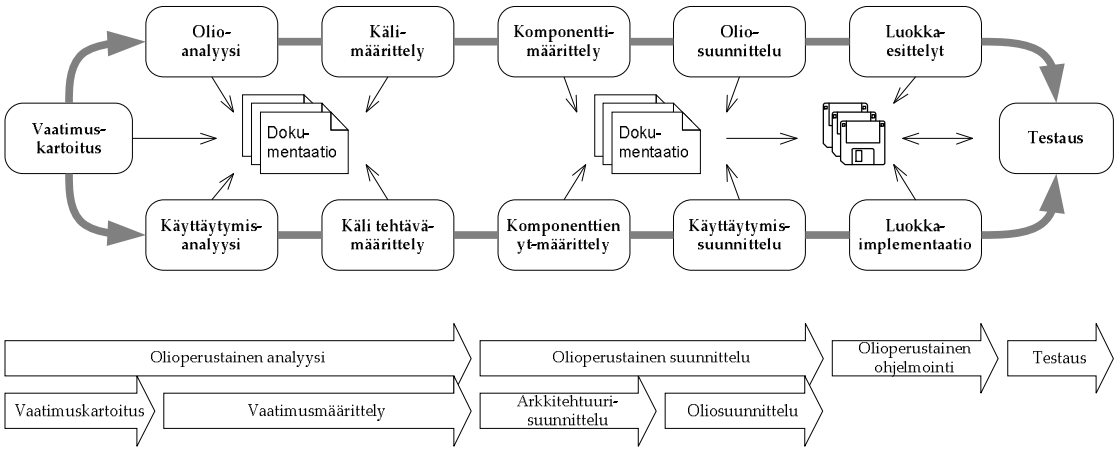
Nokia Telecommunications aloitti olio-ohjelmointiparadigman käytön ohjelmistokehityksessään 90-luvun alussa käyttäen Rumbaughin ja kumppanien [1991] kehittämää OMT-menetelmää (Object Modeling Technique). Pian kuitenkin huomattiin, että menetelmää jouduttaisiin muokkaamaan omiin tarpeisiin sopivaksi. Kehitysmenetelmän tuli olla helppo oppia Nokian kasvaessa nopeasti ja uusien ohjelmoijien, suunnittelijoiden ja arkkitehtien virratessa mukaan kehitystyöhön. Tarvittiin selkeä kehitysprosessi selkeästi erotettavine vaiheineen ja niissä tuotettavien dokumenttien määrittelyineen. Ideoita lainattiin muista menetelmistä, tarpeettomiksi todettuja notaatioita jätettiin pois, ja Nokialla jo käytössä olleita tapoja dokumentoitiin mukaan uuteen menetelmään.

Menetelmän pohjan muodosti siis OMT, sen notaatiot ja nimeämiskäytännöt. Uuden menetelmän nimeksikin tuli OMT++ [Aalto and Jaaksi, 1994]. Menetelmässä siirryttiin varsin sujuvasti UML:n notaatiokäytäntöön, sillä siihen oli jo aiemmin lisätty monia UML:n ominaisuuksia, kuten käyttötapaukset (use cases), osajärjestelmät (subsystems) ja komponentit (components). Erityisen tärkeitä kaavioita OMT++-menetelmässä ovat luokka-, sekvenssi-, komponentti- ja sijoituskaaviot (deployment diagrams) [Jaaksi *et al.*, 1999].

Muista OMT++:an vaikuttaneista menetelmistä kannattaa mainita *Object-Oriented Software Engineering* (OOSE) [Jacobson *et al.*, 1992] ja *Fusion* [Coleman *et al.*, 1994]. OOSE:sta lainattiin käyttötavat ja Fusion-metodista operaatiospesifikaatiot, joilla mallinnetaan järjestelmän kommunikointia ulkoisten elementtien kanssa.

5.2. Yleiskatsaus kehitysprosessiin

OMT++ jakautuu neljään kuvassa 5.1 esitettyyn päävaiheeseen: olioperustaiseen analyysiin, oliperustaiseen suunnitteluun, olioperustaiseen ohjelmointiin ja testaukseen. Jokainen vaihe tarkastelee kehitettävää ohjelmistoa eri abstraktiotasolta. Vaikka vaiheet on tässä lueteltu peräkkäisesti, käytetään laajemmissa ohjelmistoprojekteissa yleisesti iteratiivista lähestymistapaa. Toteutettavaa järjestelmää tarkastellaan jokaisessa vaiheessa sekä staattiselta että toiminnalliselta kannalta. Staattiselta kannalta tarkasteltuna mielenkiinnon kohteena ovat järjestelmän osien muodostamat rakenteet ja hierarkiat. Analyysivaiheessa tällaisia ovat esimerkiksi järjestelmän kohdealueen käsitteet, järjestelmään kuuluvat ja sen kanssa vuorovaikutuksessa olevat ulkoiset laitteet sekä järjestelmän käyttäjät. Suunnitteluvaiheessa näitä staattisia osia ovat ensin mm. suoritettavat prosessit, luokkakirjastot ja laitteet ja lopuksi ohjelmoitavat luokat. Toiminnalliselta kannalta mielenkiinnon kohteena ovat näiden staattisten osien keskinäinen vuorovaikutus ja yhteistoiminta; se millä tavoin järjestelmän osat mahdollistavat järjestelmälle asetettujen vaatimusten täyttävän toiminnallisuuden.



Kuva 5.1. OMT++:n päävaiheet

Analyysivaihe koostuu vaatimuskartoituksesta ja vaatimusmäärittelystä. Vaatimuskartoituksessa kerätään ja dokumentoidaan kaikki kehitettävälle järjestelmälle olennaiset toiminnalliset- ja ei-toiminnalliset *vaatimukset* (requirements) sekä *käyttötapaukset* (use cases). Vaatimusmäärittelyssä kerätyistä vaatimuksista ja käyttötapauksista mallinnetaan sovellusalueen oliomalli ja järjestelmältä vaadittu ulkoinen käyttäytyminen. Analyysivaiheessa siis kerätään, analysoidaan ja jalostetaan vaatimuksia. Siinä tuotetaan vaatimuslauseet, käyttötavat, alustava analyysioliomalli ja

operaatiospesifikaatiot. Lisäksi analyysivaiheessa määritellään järjestelmän käyttöliittymä.

Suunnitteluvaihe koostuu arkkitehtuurisuunnittelusta ja oliosuunnittelusta. Vaiheen materiaalina ovat analyysivaiheen dokumentit, joita käyttäen vaiheen aikana tuotetaan järjestelmän tekniset spesifikaatiot. Tekniset spesifikaatiot määrittelevät toteutettavan järjestelmän niin tarkasti, että se on mahdollista ohjelmoida spesifikaatioiden mukaisesti. Suunnitteluvaiheessa määritellään luokkarakenteet, luokkien rajapinnat sekä luokkien väliset yhteydet. Arkkitehtuurisuunnittelun kohteena ovat kokonaiset suoritettavat ohjelmat, komponentit, luokkakirjastot ja laitteet. Oliosuunnittelun aikana keskitytään nämä osat toteuttavien C++- tai Java-luokkien suunnitteluun. Arkkitehtuurisuunnittelussa siis spesifioidaan suoritettavat prosessit, kirjastot, tietovarastot, oheislaitteisto ja muut järjestelmän komponentit. Oliosuunnittelu tuottaa näkymän arkkitehtuurin sisälle suunnittelutason oliomallin ja sekvenssikaavioiden avulla.

Ohjelmointivaiheen yksityiskohdat riippuvat käytössä olevasta ohjelmointikielestä ja ohjelmointiympäristöstä. Esimerkiksi C++:lla ohjelmoitaessa voitaisiin ensin kirjoittaa luokkien esittelytiedostot ja niiden jälkeen kirjoittaa toteutuskoodi. Javalla ohjelmoitaessa edellisen kaltaista erottelua ei ole. Ohjelmointivaihe sisältää itse koodin kirjoittamisen lisäksi koodin kääntämistä, linkitystä sekä kirjoitetun koodin luokkakohtaista testausta.

Testausvaiheessa järjestelmästä etsitään virheitä ja varmistetaan, että järjestelmä toteuttaa sille asetetut vaatimukset. Osia järjestelmästä voidaan testata edellisten vaiheiden aikana, mutta vasta kun kaikki järjestelmän kompontit on integroitu, voidaan suorittaa koko järjestelmän toiminnallisuuden kattava testaus.

5.3. Analyysivaihe

Kaupallinen ohjelmistokehitysprojekti alkaa lähes poikkeuksetta aina analyysivaiheella. Siinä kerätään toteutettavaan ohjelmistoon liittyvät vaatimukset ja analysoidaan ongelmaa, jonka ratkaisemiseksi ohjelmistoa ollaan kehittämässä. Analyysivaiheessa pyritään rakentamaan ratkaisu käyttäjän näkökulmasta. Siinä pyritään vastaamaan kahteen perustavaa laatua olevaan kysymykseen: ”Mikä on ongelma?” ja ”Minkälaisella ratkaisulla ongelma saadaan selvitettyä?”. Näin ollen analyysivaihe tähtää käyttäjän ongelmat ratkaisevan ohjelmiston kuvaamiseen.

Analyysivaiheessa käytetään järjestelmän käyttäjän näkökulmasta olennaisia käsitteitä. Vaatimukset kerätään pääasiassa järjestelmän tulevilta käyttäjiltä ja niistä keskustellaan toisten käyttäjien ja muiden ryhmien kesken.

Näiden keskustelujen perusteella hahmotellaan ohjelmisto, joka täyttää käyttäjien vaatimukset. Tämä hahmotelma on alustava ratkaisu käsillä olevaan ongelmaan: ohjelmiston spesifikaatio käyttäjien näkökulmasta.

5.3.1. Vaatimuskartoitus

Ennen kuin ohjelmistonkehitysprojekti on virallisesti edes alkanut, on käytössä tavanomaisesti jo paljon tietoa, jota voidaan käyttää tulevaa järjestelmää suuniteltaessa. Tällaista tietoa on esimerkiksi käyttäjiltä tulevissa vaatimuksissa, ohjelmistosuunnittelijoilta tulevissa vaatimuksissa, markkinatutkimuksissa, kokouspöytäkirjoissa ja muissa yhteyksissä esille tulleissa uusissa ajatuksissa. Samoin käytössä voi olla edellisistä samankaltaisista tuotteista tulleita korjausvaatimuksia, kehitysideoita ja asiakasraportteja. Yleensä näissä olevat tiedot, ideat, konseptit ja vaatimukset ovat hyvinkin epämääräisiä, eikä niitä sellaisenaan voi suoraan käyttää ohjelmistoprojektin käynnistämisen lähtökohtana. Niitä pitää ensin työstää ja tarkentaa ja ennen kaikkea dokumentoida. Ainoastaan dokumentoidut vaatimukset ovat jäljitetettävissä.

Vaatimuskartoituksen yhteydessä näistä eri lähteistä tulevat epämääräiset vaatimukset kerätään yhteen ja dokumentoidaan *vaatimuslauseiksi* (requirement statements). Ohjelmistoa kehittävällä organisaatiolla on oltava tarkka ja todenmukainen kuva jokaisen vaatimuksen luonteesta ja tarkoituksesta, ja siksi näitä vaatimuksia jalostetaan pilkkomalla niitä osiin, yhdistelemällä niitä ja muokkaamalla niiden sanamuotoa. Tämän jalostuksen tarkoituksena on saada vaatimuksista eksplisiittisiä, tarkkoja ja mitattavia.

Kun kaikki kehitettävälle ohjelmistolle asetetut vaatimuslauseet on saatu valmiiksi, jaetaan vaatimukset tavallisesti kahteen osaan, toiminnallisiin- ja ei-toiminnallisiin vaatimuksiin. Toiminnalliset vaatimukset kertovat mitä järjestelmä tekee. Ne selittävät kuinka järjestelmä toimii ulkopäin takasteltaessa ja miten järjestelmä on vuorovaikutuksessa järjestelmän ulkopuolisten osien kanssa. Lisäksi toiminnalliset vaatimukset määrittelevät miten käyttäjä käyttää järjestelmää. Ei-toiminnalliset vaatimukset ovat tyypillisesti teknisiä vaatimuksia. Tällaisia voivat olla esimerkiksi vaatimukset järjestelmän nopeudelle, kapasiteetille ja käytettävyydelle. Vaatimuksia voidaan asettaa myös käytetyille ohjelmointikielelle tai laitteille. Ei-toiminnalliset vaatimukset tavallaan asettavat raamit, joiden sisällä toiminnalliset vaatimukset pitää saada toteutetuksi. Kuvassa 5.2 on kuvattu erään järjestelmän vaatimuslauseet.

Toiminnalliset vaatimukset:

1. Käyttäjä voi lähettää tekstiviestejä tietokoneelta siihen liitetyn matkapuhelimen välityksellä.
2. Käyttäjä voi tallentaa ja ladata puhelinnumeroita vastaanottajan nimen perusteella.
3. Käyttäjä voi luoda monen vastaanottajan ”postituslistoja”.
4. Käyttäjä voi tallentaa ja käyttää uudelleen yleisiä sanontoja ja fraaseja.
5. Käyttäjä voi tallentaa lähetettyjä viestejä ja katsella / käyttää niitä uudelleen myöhemmin.

Ei-toiminnalliset vaatimukset:

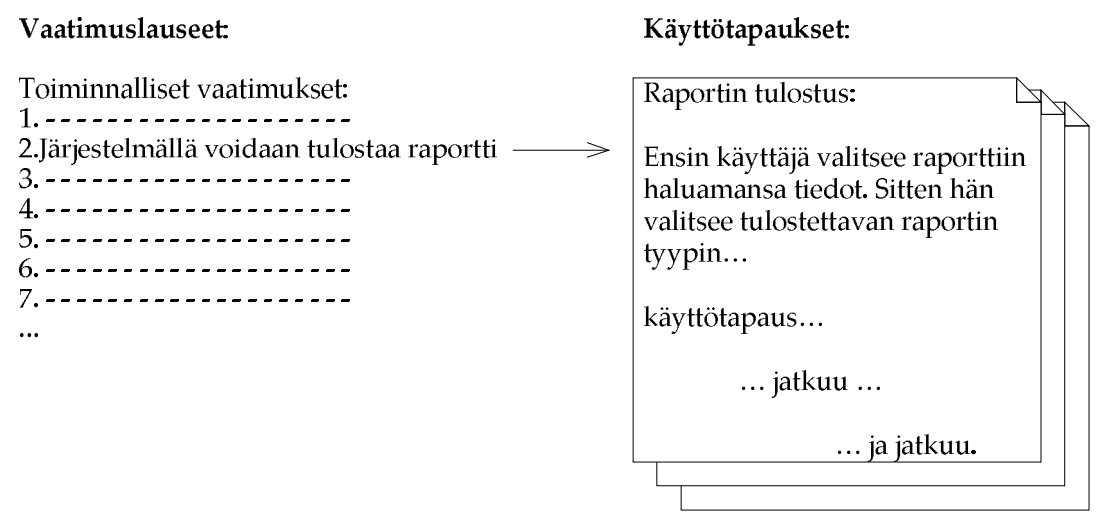
1. Järjestelmää pitää pystyä käyttämään sekä Windows-, että Unix-alustalla.
2. Järjestelmän pitää mahdollistaa monen käyttäjän yhtäaikainen pääsy puhelinnumeroihin ja ryhmiin.
3. Kaikki tieto täytyy tallentaa ASCII-muodossa.

Kuva 5.2. Esimerkki erään järjestelmän vaatimuslauseista.

Vaatimukset on pystyttävä dokumentoimaan tavalla, joka mahdollistaa valmiin ohjelmiston testaamisen niitä vasten. Ohjelmistokehitysprojektin lopuksi jokaisesta vaatimuksesta on pystyttävä toteamaan, onko se täytetty. Suositeltavaa on myös pitää kirjaa jokaisen vaatimuksen alkuperäisestä lähteestä, sillä siitä selviää miksi ja kenen näkökulmasta vaatimus on tärkeä.

5.3.2. Käyttötapaukset

Tärkeimmistä järjestelmälle asetetuista toiminnallisista vaatimuksista laaditaan tämän jälkeen luvussa kolme tarkemmin esitellyt käyttötapaukset. Siinä missä toiminnalliset vaatimukset määrittelevät *mitä* järjestelmän tulee tehdä, määrittelevät käyttötapaukset sen, *miten* järjestelmä tekee halutun toiminnan yhdessä loppukäyttäjän tai muiden järjestelmän ulkopuolisten elementtien kanssa (kuva 5.3).



Kuva 5.3. Käyttötapaukset tarkentavat toiminnallisia vaatimuksia.

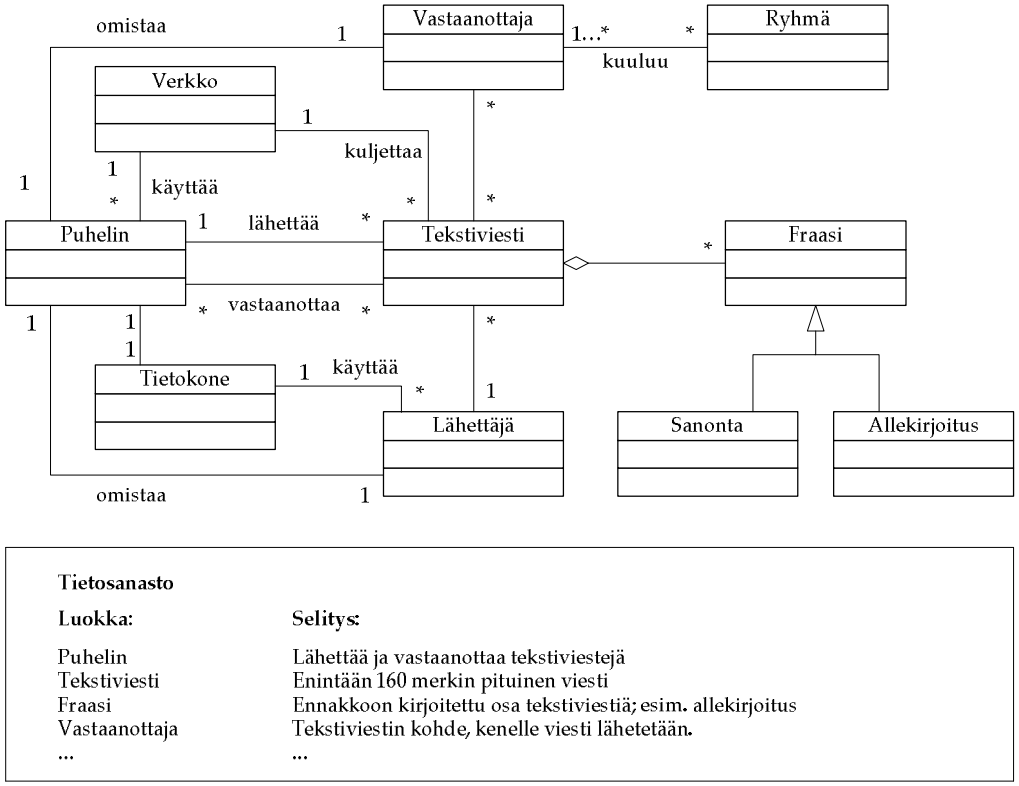
5.3.3. Olioanalyysi

Käyttötapaukset eivät sellaisenaan riitä ohjelmistosuunnittelun lähtökohdaksi, vaan niitä analysoidaan edelleen. Tämä tapahtuu tuottamalla vaatimuslauseiden ja käyttötapausten pohjalta analyysivaiheen luokkakaavio sekä operaatiomäärittely. Luokkakaaviot ja operaatiomäärittelyt ovat ohjelmistosuunnittelijoiden työkaluja, eikä niitä yleensä esitellä loppukäyttäjille. Näiden työkalujen avulla voidaan paljastaa puuttuvia ja epämääräisiä vaatimuksia. Niiden avulla ohjelmistosuunnittelijat tuottavat yksityiskohtaisemman ja tarkemman kuvauksen toteutettavasta järjestelmästä.

Olioanalyysissa tuotetaan analyysivaiheen luokkakaavio, jossa dokumentoidaan järjestelmään liittyvät staattiset avainkäsitteet ja niiden keskinäiset suhteet. Luokkakaavion lähtökohtana ovat dokumentoidut vaatimuslauseet ja käyttötapaukset.

Analyysivaiheen luokkakaavion luokat ovat olioita, jotka ovat merkityksellisiä loppukäyttäjän näkökulmasta. Sellaiset luokat, jotka ovat merkityksellisiä ainoastaan ohjelmoijan näkökulmasta, kuten esimerkiksi tietorakenteet tai käyttöliittymäkomponentit, rajataan ulos analyysivaiheen luokkakaaviosta. Tällaiset luokat tulevat mukaan myöhemmissä vaiheissa määriteltäviin luokkakaavioihin. Sellaiset tekniset luokat, joilla on merkitystä loppukäyttäjän näkökulmasta, kuuluvat kuitenkin tämän vaiheen luokkakaavioon. Esimerkiksi käsite UNIX-prosessi otettaisiin mukaan analyysivaiheen luokkakaavioon suuniteltaessa järjestelmää, jolla hallitaan UNIX-prosesseja.

Analyysivaiheen luokkakaavion luokat eivät tyypillisesti sisällä monia metodeja. Jaaksin ja kumppanien kokemusten mukaan metodien etsintä tässä vaiheessa järjestelmän kehitystä ei ole hyödyllistä. Sen sijaan analyysivaiheen luokkakaavion yhteydessä esitetään aina järjestelmän *tietosanasto* (data dictionary), jossa kuvataan lyhyesti kaavion luokat ja niiden väliset suhteet. Esimerkki analyysivaiheen luokkakaaviosta ja tietosanastosta on kuvassa 5.4.

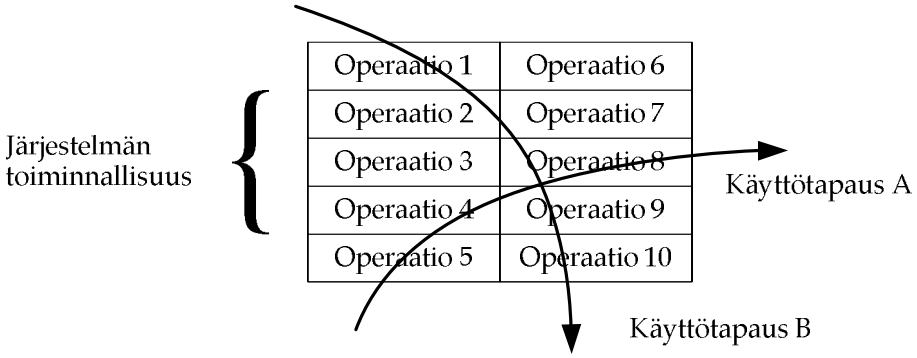


Kuva 5.4. Analyysivaiheen luokkakaavio ja tietosanasto.

5.3.4. Käyttäytymisanalyysi

Analyysivaiheen luokkakaavio ja tietosanasto kuvaavat järjestelmän staattiset avainkäsitteet ja niihin liityvät tiedot. Järjestelmän toiminnallisuus kuvataan analyysivaiheessa ns. käyttäjäoperaatioina. Käyttäytymisanalyysissä käyttötapauksista ja toiminnallisista vaatimuksista erotetaan käyttäjän tekemät operaatiot.

Ensisijainen lähde operaatioiden tunnistamisessa ovat dokumentoidut käyttötapaukset. Käyttötapaukset ovat tyypillisesti useiden peräkkäisten operaatioiden sarjoja, joiden välissä saattaa olla järjestelmän käyttöön liittymättömiä työvaiheita. Käyttötapauksilla on usein kuvattu kuitenkin ainoastaan järjestelmän tärkeimmät toiminnot. Osaa järjestelmän toiminnallisuudesta ei välttämättä ole lainkaan kuvattu osana mitään käyttötapausta. Joskus ainoa maininta järjestelmän tietystä toiminnallisuudesta on kirjattuna järjestelmän toiminnallisiin vaatimuksiin, jotka ovatkin toinen tärkeä operaatioiden lähde. Kuvassa 5.5 on kuvattu käyttötapausten ja operaatioiden suhde.



Kuva 5.5. Käyttötapaukset kuvaavat järjestelmän tärkeimmät toiminnot.

Käyttötapauksissa kuvatut käyttäjän työvaiheet saattavat sisältää keskusteluja kollegoiden kanssa, käyttöohjeiden selaamista, puhelinsoittoja ja muita käyttäjän tekemiä toimintoja, jotka ovat osa käyttäjän työtä, mutta eivät ole osa toteutettavaa järjestelmää. Operaatiomäärittelyssä tämän kaltaisia osia ei ole, vaan niissä keskitytään ainoastaan kehitteillä olevan järjestelmän toiminnallisuuteen.

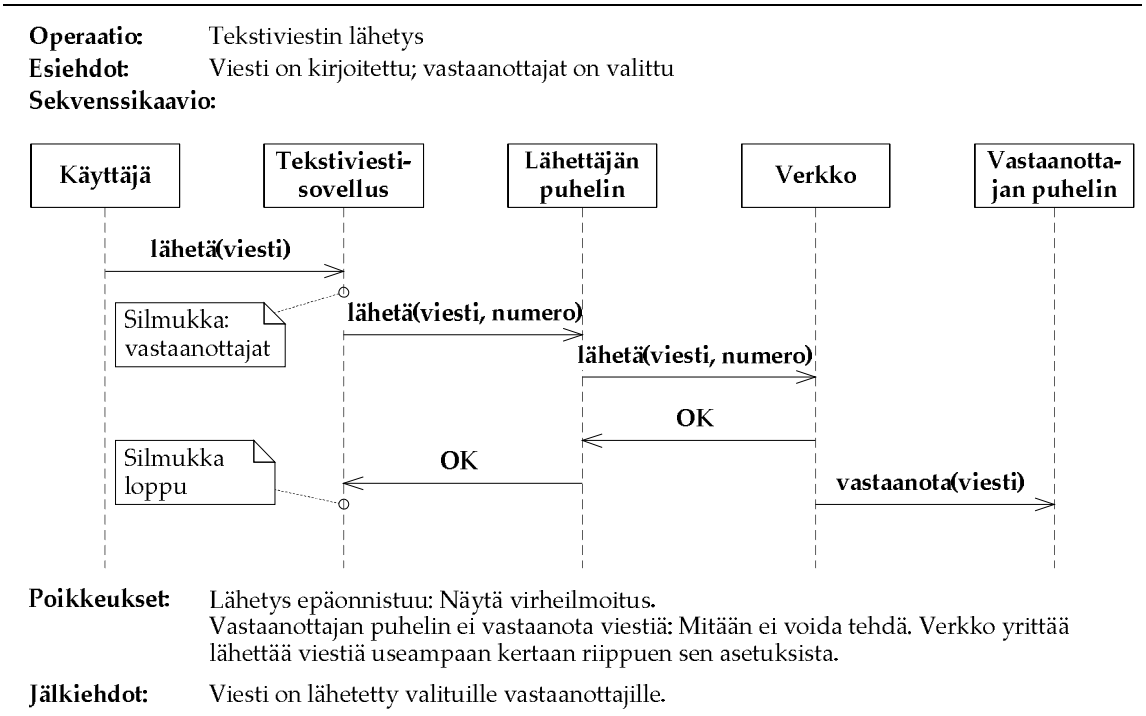
Käyttäytymisanalyysissä luodaan kaksi välituotetta: operaatiolista ja operaatiomäärittely. Operaatiolista (kuva 5.6) on kokoelma operaatioiden nimiä, kuten ”Kirjoita viesti” ja ”Lisää vastaanottajat viestille”. Kun lista on valmis, tutkitaan jokaista operaatiota tarkemmin erikseen. Jos operaatio sisältää kommunikaatiota järjestelmän ulkopuolisten osien kanssa, luodaan siitä operaatiomäärittely.

Tekstiviestisovelluksen operaatiot:

1. Tekstiviestin kirjoitus.
2. Fraasin lisääminen.
3. Vastaanottajien ja ryhmien valinta.
4. Tekstiviestin tallennus.
5. Tekstiviestin lähetys.
6. Tekstiviestin lataaminen.
7. Vastaanottajan tietojen lisääminen.
8. Ryhmän luonti.
9. Ryhmän poistaminen.
10. Vastaanottajien lisääminen ryhmään.
11. Vastaanottajien poisto ryhmästä.
12. Vastaanottajan tietojen poistaminen.

Kuva 5.6. Esimerkki operaatiolistasta.

Operaatiomäärittelyssä operaatioiden kuvaamiseen käytetään UML:n sekvenssikaavioita, joita tarkennetaan operaatiokohtaisilla esi- ja jälkiehdoilla.



Kuva 5.7. Esimerkki operaatiomäärittelystä.

Operaatiomäärittelyt kuvaavat, kuinka järjestelmä kommunikoi käyttäjien ja ulkoisten elementtien kanssa. Kuva 5.7 esittää ainoastaan, että järjestelmä on jollain tavalla saatava lähettämään tekstiviesti. Määrittely ei ota kantaa siihen, miten tämä toiminto saadaan tehtyä käyttöliittymällä. Käyttöliittymää tarkastellaan OMT++-menetelmän myöhemmissä vaiheissa.

5.3.5. Käyttöliittymämäärittely

Analyysivaiheen viimeisenä osana OMT++-menetelmässä on käyttöliittymämäärittely. Koska käyttöliittymämäärittely ei varsinaisesti kuulu tutkielman aihepiiriin, käydään se tässä läpi hyvin pintapuolisesti.

Käyttöliittymämäärittely alkaa etsimällä operaatiomäärittelyistä sellaiset operaatiot, jotka vaativat käyttäjän osallistumista. Normaalikokoiselle sovellukselle saattaa hyvinkin olla määriteltynä 50–70 operaatiota. Näin suuri operaatiomäärä kannattaa jakaa osiin, joita yhdistää jokin tekijä. Jakamisen perustana voi olla esimerkiksi yhteinen aihealue (operaatiot, jotka koskevat tiettyä järjestelmän toiminnallisuutta, esim. lataus- ja tallennusoperaatiot), samoja analyysivaiheen luokkia (esim. tekstiviestiä) koskevat operaatiot tai eri suorittajien suorittamat operaatiot.

Syntyneestä operaatiolistasta etsitään tämän jälkeen käyttäjän kannalta tärkeimmät operaatiot, ns. *ensisijaiset operaatiot* (primary operations). Tärkeimpinä operaatioina voidaan pitää yleisimpiä ja aikaavaativimpia järjestelmällä tehtäviä operaatioita, joita varten järjestelmää ollaan ensisijaisesti

tekemässä. Yleisenä käytettävyystavoitteena pidetään ensisijaisten operaatioiden helppoa ja tehokasta suorittamista. Nykyisten käyttöliittymäparadigmojen mukaan tärkeimmät ja aikaavaativimmat tehtävät tulisi olla suoritettavissa sovelluksen pääikkunassa.

Yleensä sovelluksilla on pääikkunan lisäksi myös useita muita ikkunoita ja dialogeja. Käyttöliittymästä laaditaan dialogikaavio (dialog diagram) käyttäen UML:n tilakaavio-notaatiota. Operaatiot sijoitetaan tämän jälkeen kaavion dialogilaatikoihin (kaavion tilat). Kaaviosta voidaan siten nähdä, missä käyttöliittymän osassa kukin operaatio suoritetaan, ja kuinka käyttäjä pystyy navigoimaan dialogien muodostamassa verkossa.

Kun ensimmäinen versio dialogikaaviosta on valmis, jatketaan määrittelyä suunnittelemalla yksittäiset dialogit tarkemmalla tasolla; valitaan sopivia käyttöliittymäkomponentteja ja sijoitetaan ne dialogeihin. Määrittelyn pohjalta voidaan lopuksi implementoida käyttöliittymäprototyyppi. Dialogikaavioon voidaan tehdä muutoksia dialogisuunnittelussa ja prototyypissä havaittujen puutteiden tai ongelmien johdosta.

5.4. Suunnitteluvaihe

Analyysivaiheessa tuotetaan kehitettävän järjestelmän ongelmakuvaus, vaatimuslauseet sekä alustava ratkaisu käyttäjän näkökulmasta tarkasteltuna. Suunnitteluvaiheessa käytetään jo analyysivaiheesta tuttuja luokka- ja sekvenssikaavioita. Tässä vaiheessa tarkastelun kohteena ovat kuitenkin ohjelmoijille tärkeät käsitteet loppukäyttäjälle olennaisten käsitteiden sijaan.

Jaaksin ja kumppanien mukaan suunnittelu riippuu lähtökohtaisesti toteutuksessa käytetystä ohjelmointiympäristöstä sekä ohjelmiston suorituksessa käytettävästä käyttöjärjestelmästä. Jos järjestelmä toteutettaisiin Java-kielellä käyttäen Visual Age for Java -ohjelmointiympäristöä, tulisi suunnittelun olla erilainen kuin jos järjestelmä toteutettaisiin Microsoftin Visual Basicilla. Unix-käyttöjärjestelmässä suoritettavassa ohjelmistossa voitaisiin suosia prosesseja, kun taas Windowsissa suoritettavassa ohjelmistossa voitaisiin suosia säikeitä. Näin ollen nämä asiat tulee olla päätettyinä ennen suunnittelun aloittamista.

5.4.1. Arkkitehtuurisuunnittelu

Suunnitteluvaihe alkaa arkkitehtuurisuunnittelulla. Arkkitehtuurisuunnittelun tavoitteena on suunnitella järjestelmälle paras mahdollinen komponettirakenne. Arkkitehtuurin suunnittelussa edetään tuttuun tapaan staattista ja toiminnallista polkua. Staattisella polulla tuotetaan järjestelmän komponentti- ja laiterakenne, ja toiminnallisella polulla tarkastellaan näiden komponettien yhteistoimintaa.

OMT++-menetelmässä käytetään hieman muunneltua versiota Philippe B. Kruchtenin esittämästä arkkitehtuurin 4+1 näkymämallista [Kruchten, 1995]. OMT++-menetelmässä käytössä ovat arkkitehtuurin staattisella polulla käytetyt looginen- (logical view), suoritus- (run-time view) ja kehitysnäkymä (development view) sekä toiminnallisella puolella käytetty skenaarionäkymä (scenario view). Käytössä on siis 3+1 näkymämalli.

Looginen näkymä kuvaa järjestelmän korkean tason ositusta sovelluskokonaisuuksiin ja erillisiin sovelluksiin. Näkymää käytetään lähinnä kommunikointi- ja tuotteenhallintatarkoituksiin esimerkiksi keskusteltaessa asiakkaan kanssa ja jaettaessa suurta työtä eri projektien ja tuotelinjojen kesken. Yhden sovelluksen sisältävissä järjestelmissä loogista näkymää ei tarvita lainkaan.

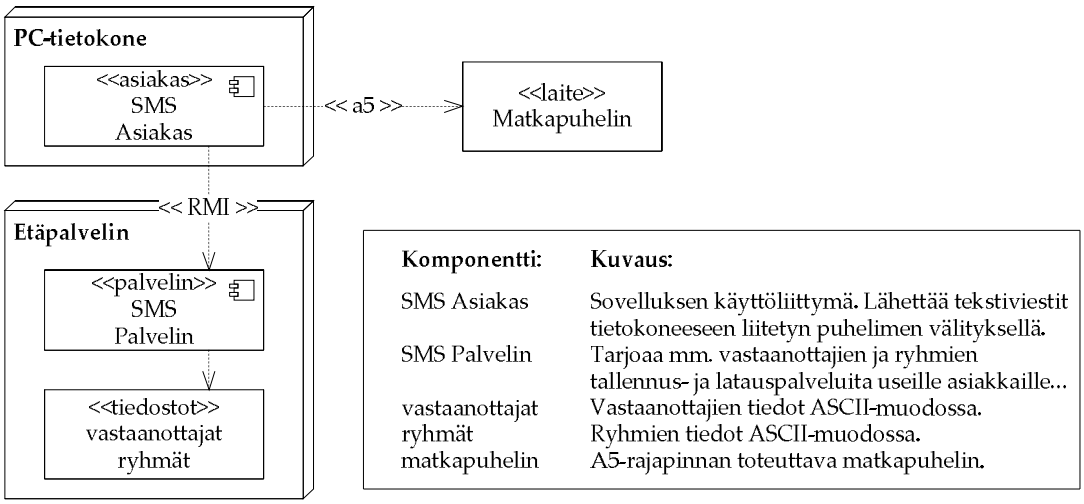
Järjestelmän ohjelmistoarkkitehtuuria suunniteltaessa käytössä ovat suoritus- ja kehitysnäkymät yhdessä skenaarionäkymän kanssa. Suoritusnäkymässä määritellään kaikki järjestelmän suoritettavat komponentit (executables). Kehitysnäkymä puolestaan määrittelee järjestelmän kaikki itsenäisesti kehitettävät komponentit. Skenaarionäkymä antaa kuvan järjestelmän komponenttien yhteistoiminnasta eri käyttötilanteissa.

Komponentin tulisi Jaaksin ja kumppanien mielestä ihanteellisesti olla noin ”ihmisen kokoinen”. Tällä tarkoitetaan sitä, että yhtä komponenttia kehittää ja ylläpitää yksi projektin jäsen. Ohjelmistoarkkitehtuuri jakaa täten järjestelmän osiin ja mahdollistaa kehitystyön jakamisen eri henkilöiden kesken. Esimerkkejä tyypillisistä komponenteista ovat asiakas-sovellus (client application), palvelin-sovellus (server application), tietokantakirjasto ja jokin monimutkaisempi käyttöliittymäkomponentti.

5.4.2. Komponenttimäärittely

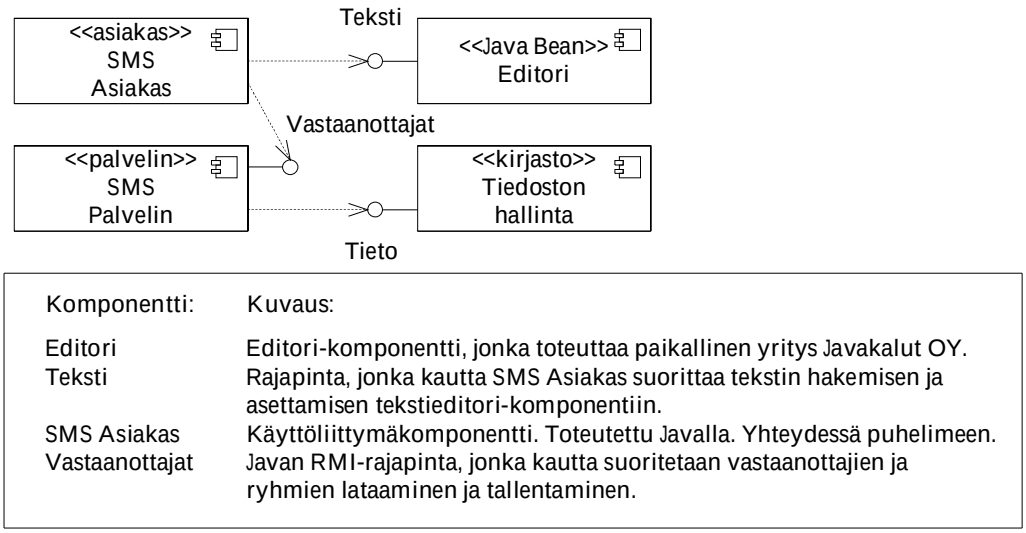
Arkkitehtuurisuunnittelu alkaa yleensä suoritusnäkymän muodostamisella. Siinä määritellään ohjelmiston suoritettavat komponentit sekä järjestelmään kuuluvat sekä siihen yhteydessä olevat laitteet. Nämä laitteet mallinnetaan sellaisenaan, sillä niiden rajapinnat ja toiminnallisuus on yleensä kiinnitetty eikä niitä voida muuttaa. Yleensä nämä ulkoiset laitteet ovat mukana jo analyysivaiheen luokkakaaviossa.

Suoritusnäkymä kuvataan UML:n sijoittelukaaviolla (deployment diagram). Kaavion lisäksi listataan jokainen komponentti ja siihen liittyvä kuvaus. Esimerkki järjestelmän suoritusnäkymästä esitetään kuvassa 5.8.



Kuva 5.8. Esimerkki komponenttiarkkitehtuurin suoritusnäkymästä.

Kehitysnäkymän suunnittelu alkaa samanaikaisesti suoritusnäkymän määrittelyn kanssa. Siinä missä suoritusnäkymä kuvaa ainoastaan järjestelmän suoritettavat komponentit, on kehitysnäkymässä mukana jokainen itsenäisesti kehitettävä komponentti. Korkeatasoinen kehitysarkkitehtuuri mahdollistaa komponenttien samanaikaisen kehitystyön ja hallinnan. Osa komponenteista voidaan mahdollisesti ostaa, osa voidaan saada uudelleenkäytettäväksi vanhoista projekteista ja osa saatetaan ohjelmoida täysin alusta. Kehitysnäkymä kuvataan UML:n komponenttikaaviolla, johon liitetään jälleen tekstuaalinen lista kaaviossa esiintyvistä komponenteista kuvauksineen. Esimerkki järjestelmän kehitysnäkymästä on kuvattuna kuvassa 5.9.



Kuva 5.9. Esimerkki komponenttiarkkitehtuurin kehitysnäkymästä.

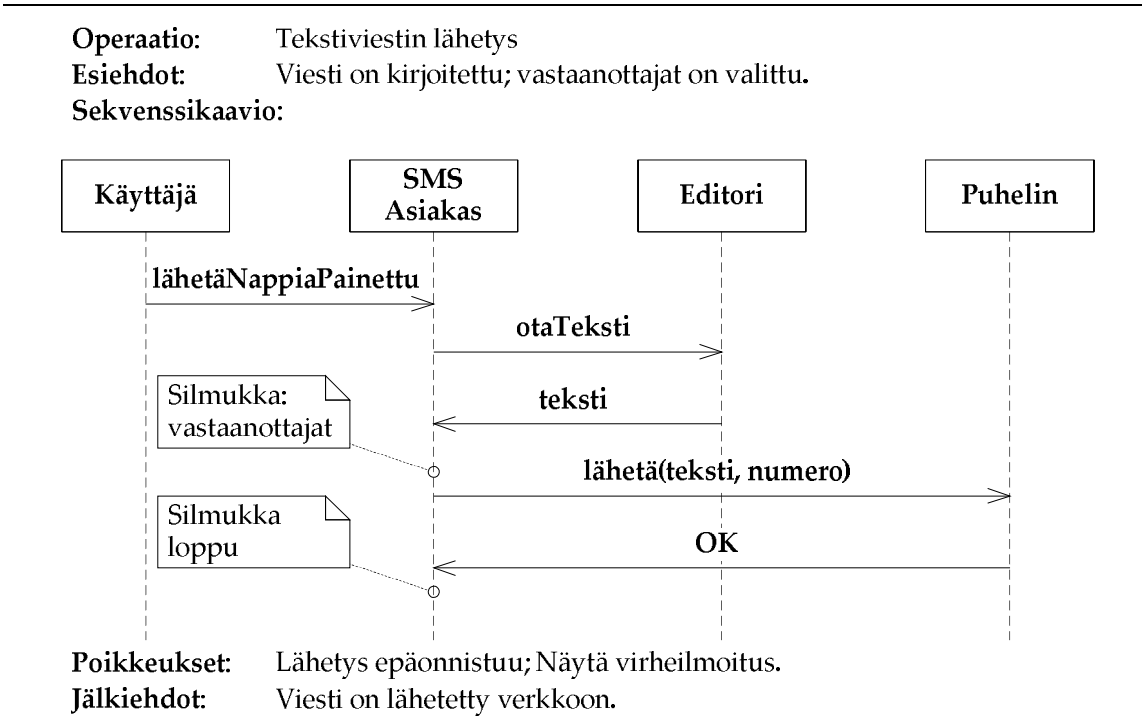
Tässä vaiheessa määritellään myös järjestelmän tietokantaratkaisut tauluineen ja kenttineen. Tietokantaa voidaan pitää jossain määrin järjestelmän yhtenä komponenttina. Tämän tutkielman yhteydessä sivuutamme tietokanta-asiat.

5.4.3. Komponenttien yhteistoimintamäärittely

Järjestelmän komponentit toimivat yhdessä tuottaakseen palveluja järjestelmän ulkopuolisille olioille kuten käyttäjille, muille järjestelmille ja laitteille. Nämä palvelut on jo määritelty analyysivaiheessa järjestelmän operaatioina. Operaatiot ovat lähtökohtana komponenttien yhteistoimintamäärittelylle, joka kuvaa, kuinka komponentit suorittavat järjestelmän operaatiot.

Yhteistoimintaa mallinnetaan sekvenssikaavioilla, joihin liitetään esi- ja jälkiehdot sekä poikkeukset, aivan kuten operaatiomäärittelyssä. Kaaviot antavat arkkitehtuurin skenaarionäkymän. Skenaarionäkymä käsittelee komponenttiarkkitehtuuria toiminnallisesta näkökulmasta ja on siten OMT++-menetelmän toiminnallisella polulla. Kaavioilla voidaan kuvata sekä suoritusnäkymän suoritettavien komponenttien, että kehitysnäkymän sisältämien kaikkien komponettien välistä yhteistoimintaa.

Kuvassa 5.10 on esimerkki komponenttien yhteistoimintaa sisältävästä skenaariosta. Yksi skenaario kuvaa ainoastaan yhden operaation suoritukseen liityvää komponenttien välistä yhteistoimintaa. Kuten aiemmin on mainittu, mielenkiinnon kohteena analyysivaiheessa ovat järjestelmän tuottamat palvelut loppukäyttäjälle. Tässä vaiheessa tavallaan avataan järjestelmä, joka analyysivaiheen sekvenssikaavioissa on kuvattu ainoastaan yhtenä elementtinä, ns. mustana laatikkona. Suunnitteluvaiheessa määritellään, kuinka järjestelmän komponentit yhteistoiminnallaan tuottavat määrittelyn mukaiset palvelut. Kuvaus muuttuu järjestelmän loppukäyttäjän näkökulmasta ohjelmistosuunnittelijan näkökulmaan.



Kuva 5.10. Esimerkki komponenttiarkkitehtuurin skenaarionäkymästä.

Kehitysnäkymän komponenttien yhteistoimintaa kuvaavat kaaviot antavat pohjan komponenttien välisten rajapintojen määrittelylle. Rajapinnat ovat komponenttien, ja samalla niistä vastuussa olevien ohjelmistokehittäjien välisiä sopimuksia. Näiden sopimusten mukaan komponentti lupautuu joko toteuttamaan tai käyttämään sovittua rajapintaa kommunikoidessaan toisen komponentin kanssa.

Suoritusnäkymän komponttien yhteistoimintaa kuvaavilla kaavioilla esitetään, kuinka järjestelmän suoritettavat komponentit kommunikoivat keskenään järjestelmän ollessa käytössä. Sen avulla voidaan tutkia, millaista tietoa komponenttien välillä kulkee, sekä kuinka paljon ja kuinka usein tietoa siirtyy komponenteilta toisille. Tämän tiedon kerääminen on välttämätöntä järjestelmän suorituskykyä ja skaalautuvuutta analysoidessa. Se antaa hyvän pohjan järjestelmän lopullisesta prosessijaosta päätettäessä.

Suppeammissa, alle kymmenen komponenttia sisältävissä komponenttiarkkitehtuureissa, kuvattavaksi riittävät yleensä kehitysnäkymässä tapahtuvat skenaariot. Kaavioista pystyy tällöin tulkitsemaan myös suoritettavien komponenttien välisen kommunikoinnin. Laajemmissa arkkitehtuureissa on selvyyden vuoksi hyödyllistä tutkia skenaarioita tämän lisäksi myös suoritusnäkymän sisällä.

Skenaarioita ja niiden sekvenssikaavioita piirtäessä saattaa eteen tulla tilanteita, jotka vaativat muutoksia alustavaan komponenttirakenteeseen. Vasta nyt voidaan nähdä ja ymmärtää, miten erilaiset loppukäyttäjän pyynnöt tulevat

suoritetuksi järjestelmän komponenttien yhteistoiminnan tuloksena. Sekvenssikaavioita tutkittaessa pystytään mahdollisesti havaitsemaan järjestelmän suorituskykyyn ja vakauteen liittyviä ongelmia. Komponenttijakoa voi näistä ongelmista johtuen joutua päivittämään useampaankin kertaan. Mahdollisten muutosten vaikutukset komponenttien yhteistoimintaan täytyy tutkia päivittämällä skenaarioita tehtyjen muutosten mukaisesti.

Kaikista yksinkertaisimmista operaatioista ei edes kannata piirtää sekvenssikaaviota. Arkkitehtuurisuunnittelun kannalta on kuitenkin tärkeää piirtää kaaviot kaikista niistä skenaarioista, joiden suorittaminen vaatii useampien komponenttien välistä yhteistoimintaa.

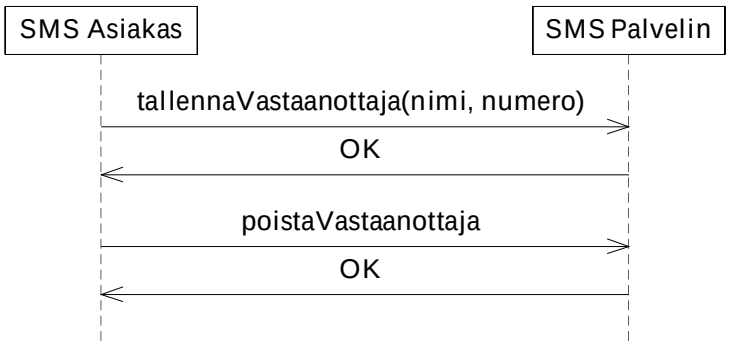
5.4.4. Rajapinnat

Komponentit voivat sijaita saman prosessin sisällä tai eri prosesseissa ja ne kommunikoivat rajapintojen kautta. Rajapintojen kautta kulkeva kommunikaatio on siten joko prosessin sisäistä (in-process) tai prosessien välistä (out-process) kommunikaatiota. Suoritusnäkymän rajapinnat ovat aina prosessien välisiä rajapintoja, mutta kehitysnäkymän rajapinnat voivat olla kumpaakin rajapintatyyppiä.

Rajapinta on sopimus ilman toteutusta. Teoriassa mikä tahansa komponentti voi *toteuttaa* rajapinnan, mikä tarkoittaa sitä, että komponentti toimii rajapintasopimuksen mukaisesti. Rajapintaa käyttävä komponentti on kiinnostunut ainoastaan rajapinnasta ja sen antamista lupauksista. Se, miten rajapinnan lupaamat palvelut on toteutettu, ei ole käyttäjän kannalta olennaista.

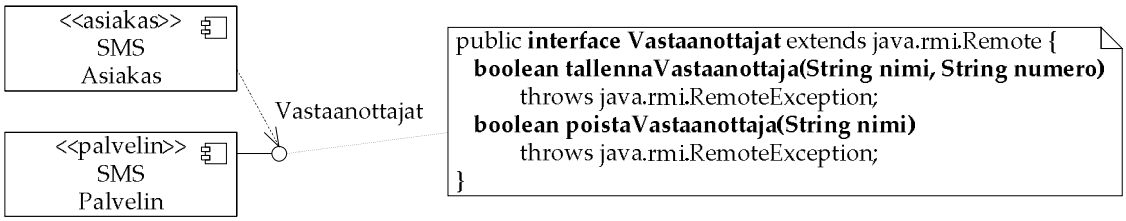
Arkkitehtuurisuunnittelun skenaarioissa tuotetaan rajapintojen metodit, parametrit ja paluuarvot. Sekvenssikaaviot kuvaavat, kuinka komponentit kutsuvat toisia komponentteja ja nämä kutsut menevät aina rajapinnan kautta. Yleisesti ottaen rajapinnat koostuvat ainoastaan funktioiden esittelyistä ja tietojen käsittely eli toteutus on piilotettu. Komponentti, joka toteuttaa annetun rajapinnan, on vastuussa funktion toteuttavasta toteutuksesta. Rajapintaa käyttävä komponentti kutsuu näitä funktiota välittämättä lainkaan itse toteutuksesta tai toteuttavasta komponentista.

Kuvassa 5.11 on esimerkki rajapintojen määrittelystä. Tietyn skenaarion sekvenssidiagrammissa on kuvattu kahden komponentin välistä kommunikaatiota. Komponentti ”SMS Asiakas” kutsuu komponentin ”SMS Palvelin” palveluja vastaanottajien tallentamiseksi ja poistamiseksi. Kutsut menevät rajapinnan ”Vastaanottajat” kautta, sillä kuvassa 5.9 esitetyn kehitysnäkymän perusteella rajapinta ”Vastaanottajat” on ainoa tapa kutsua komponentin SMS Palvelin palveluja.



Kuva 5.11. Osa skenaarion yhteistoimintaa kuvaavasta kaaviosta.

Kuvan 5.11 sekvenssikaavion perusteella voimme nyt määritellä rajapinnan ”Vastaanottajat”, joka on esitetty kuvassa 5.12. Rajapinta sisältää funktiot ”tallennaVastaanottaja” ja ”poistaVastaanottaja”, joita komponetti ”SMS Asiakas” kutsuu. Komponentti ”SMS Palvelin” toteuttaa tämän rajapinnan, ja komponentti ”SMS Asiakas” käyttää komponenttia ”SMS Palvelin” rajapinnan ”Vastaanottajat” kautta.



Kuva 5.12. Vastaanottajat-rajapinta.

OMT++-menetelmässä rajapintojen katsotaan olevan järjestelmän itsenäisiä osia, joita ei koskaan sekoiteta rajapintojen toteutusten kanssa. Ne ovat täysin samanarvoisessa asemassa version- ja muutostenhallinnan kannalta, kuin mikä tahansa järjestelmän muu toteutuksen osa. Käytännössä rajapintoja voidaan kuitenkin usein käsitellä osana rajapinnan toteuttavaa komponenttia.

5.4.5. Komponenttisuunnittelu

Komponenttisuunnittelussa tarkastelun alla on järjestelmän yksi komponentti kerrallaan. Komponentti voi olla joko suoritettava prosessi tai kirjasto. Siinä missä arkkitehtuurisuunnittelussa määritellään komponentit ja laitteet sekä niiden välinen yhteistoiminta, määritellään komponenttisuunnittelussa komponentin sisältämien luokkien toteutus, komponentin luokkarakenne, sekä luokkien välinen yhteistoiminta. Komponenttisuunnittelussa jokainen komponentti siis avataan ja se määritellään toteutettavina luokkina.

Kuten aiemmat vaiheet, etenee komponenttisuunnittelukin sekä staattisella- että toiminnallisella polulla. Staattisella polulla määritellään komponentin

sisältämät luokat ja luokkarakenteet ja toiminnallisella polulla määritellään, millaista yhteistoimintaa näiden luokkien kesken tapahtuu. Staattisella polulla tapahtuvaa suunnittelua kutsutaan oliosuunnitteluksi ja toiminnallisella polulla tapahtuvaa suunnittelua kutsutaan käyttäytymissuunnitteluksi.

5.4.6. Oliosuunnittelu

Oliosuunnittelussa siis määritellään yhden komponentin sisältämät luokat ja niiden muodostamat rakenteet. Käytössä on jälleen luokkakaavio, aivan kuten aiemmissakin staattisella polulla tapahtuneissa suunnitteluvaiheissa. Merkityksiltään kaaviot eroavat kuitenkin huomattavasti. Analyysivaiheen luokkakaaviossa kuvatut luokat ovat kohdealueen käsitteitä. Oliosuunnittelussa kaavion luokat kuvaavat jollakin ohjelmointikielellä toteutettavia luokkia. Vaikka kaavioiden sisältämät luokat olioparadigman mukaisesti läheisesti muistuttavatkin toisiaan, on niiden välillä merkittävä käsitteellinen ero.

Oliosuunnittelu rakentuu analyysivaiheen tuloksien päälle. Siinä pitää kuitenkin ottaa huomioon yleisesti sovittuja ohjeita, kuten *sovelluskehykset* (application frameworks) ja *suunnittelumallit* (design patterns). OMT++-menetelmässä suositellaan käytettäväksi MVC++-mallia oliosuunnittelun pohjana. MVC++-malli määrittelee sovelluksen rakenteen ohjeistamalla suunnittelijaa laatimaan kolmitasoisen luokkakaavion, jossa eriytetään käyttöliittymä (View), sovelluskohtainen toiminnallisuus (Controller) ja ongelma-alueen uudelleenkäytettävä toiminnallisuus (Model, malli). Analyysivaiheen luokkakaavion luokat voidaan yleensä kopioida sellaisenaan olisuunnittelun alustavan luokkakaavion malli-osaan.

Vielä oliosuunnittelussakin pyritään järjestelmän abstrahoimiseen. Aivan kaikkia yksityiskohtia ei siis oteta mukaan oliosuunnittelussakaan. Alimman tason tekniset ratkaisut tehdään vasta ohjelmoinnin yhteydessä, eikä oliosuunnittelussa pyritä luokkien kaikkien metodien ja attribuuttien mallintamiseen. Myös tietorakenteita kuvaavat luokat voidaan jättää määrittelemättä. Näin voidaan tehdä, koska tarkoituksena on tehdä kaaviot luettaviksi ja ymmärrettäviksi, eikä niistä ole tarkoitus generoida ohjelmakoodia. Samalla vältytään koodiin tehtävien pienten muutosten aiheuttamalta kaavioiden jatkuvalta päivitykseltä.

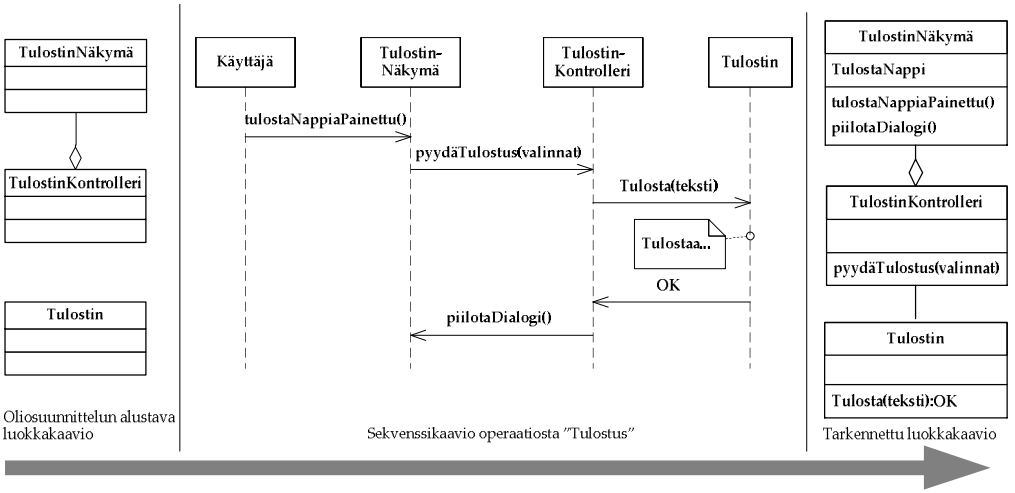
Alustava luokkakaavio vaatii vielä paljon työstämistä ennen lopullista muotoaan. Tässä vaiheessa ei lähdetä arvaamaan luokkien välisiä suhteita ja operaatioita. Todennäköisesti osa luokista häviää jatkosuunnittelun myötä. Aivan yhtä todennäköistä on, että kaavioon tullaan piirtämään kokonaan uusia luokkia ja jäljelle jääneitä tullaan muuttamaan. Nämä muutostarpeet

kartoitetaan käymällä läpi järjestelmän kaikki komponettia koskevat operaatiot.

5.4.7. Käyttäytymissuunnittelu

Käyttäytymissuunnittelussa määritellään komponentin sisältämien olioiden yhteistoiminta. Yhteistoiminta visualisoidaan skenaarioiden avulla. Tuttuun tapaan skenaariot sisältävät sekvenssidiagrammin esi- ja jälkiehtoineen sekä poikkeustapaukset. Käyttäytymissuunnittelun tarkoitus on luokkakaavion viimeistely tuottamalla luokkien väliset suhteet ja jäsenfunktioiden esittelyt.

Jokainen skenaario määrittelee joukon jäsenfunktioita, attribuutteja ja luokkien välisiä suhteita, jotka voidaan lisätä komponentin luokkakaavioon kuvan 5.13 esittämällä tavalla. Luokkakaavioon voidaan skenaarioiden myötä myös lisätä aivan uusia luokkia. Luokkakaavion luokat, jotka eivät esiinny yhdessäkään skenaariossa, voidaan poistaa luokkakaaviosta. Samoin kaavioon merkittyjä reaalimaailman suhteita voidaan muuttaa. Jos kahden luokan välillä ei skenaarioissa tapahdu kommunikointia, voidaan suhde näiden luokkien väliltä poistaa. Kun järjestelmän kaikki operaatiot on mallinnettu skenaarioiden avulla, ja luokkakaavioon on tehty kaikki tarvittavat lisäykset, poistot ja muutokset, on luokkakaavio valmis toteutusta varten.



Kuva 5.13. Luokkakaavion kehittäminen sekvenssikaavion avulla.

Analyysivaiheessa määritellyt operaatiot voivat vaatia suunnittelua useamman komponentin osalta. Esimerkiksi asiakas-palvelin-sovelluksessa on todennäköisesti useita operaatioita, jotka alkavat käyttäjän syötteestä asiakas-komponentille, joka puolestaan käyttää palvelimen palveluja toiminnon toteuttamisessa. Vaikka kyseessä on yksi operaatio, vaatii se läpikäyntiä sekä asiakas- että palvelinkomponenttia suunniteltaessa.

Komponentin käyttäytymissuunnittelun yhteydessä sekvenssikaaviot piirretään ainoastaan niistä operaatioista, joiden toiminta vaatii kyseisen komponentin sisältämien luokkien yhteistoimintaa. Jaaksi ja kumppanit [Jaaksi *et al.,* 1999] tähdentävät, että samoin kuin kaikissa muissakin suunnittelun vaiheissa, triviaalit kaaviot eivät ole pelkästään turhia, vaan jopa vaarallisia. Suuret määrät hyödyttömiä kaavioita hukuttavat alleen tärkeämmät kaaviot.

5.5. Ohjelmointi

Ohjelmoinnin yhteydessä komponenttisuunnittelun vaihetuotteet hienonnetaan ohjelmointikielen tasolle. Ohjelmointia voikin pitää suunnittelun viimeisenä vaiheena. Monet alimman tason päätöksistä tehdään vasta ohjelmointivaiheessa.

Luokkien rajapinnat perustuvat luokkakaavioon ja rajapinnan funktioiden toteutus voidaan perustaa skenaarioiden sekvenssikaavioihin. Jotta luokat ja niiden funktiot voidaan ohjelmoida yksi kerrallaan, on olennaista, että tärkeimmät luokat, rajapinnat ja luokkien välinen yhteistoiminta on tarkasti mallinnettu.

Luokan sisäistä toiminnallisuutta ei yleensä ole tarpeen esittää millään graafisella notaatiolla. Toteutuksessa käytetty ohjelmointikieli itsessään dokumentoi luokkien sisäisen toiminnallisuuden. Joissakin harvinaisissa tapauksissa voidaan monimutkaisimpien ja tilariippuvaisimpien luokkien ohjelmoinnin apuna käyttää tila- tai aktiviteettikaavioita.

5.6. Testaus ja integrointi

Lopuksi järjestelmä testataan ja integroidaan. Testauksen ensimmäisessä vaiheessa suoritetaan *komponenttitestaus* (unit testing, component testing). Tämän vaiheen suorittaa yleensä komponentin suunnittelusta ja ohjelmoinnista vastaava henkilö ja itse asiassa tämän vaiheen katsotaan kuuluvan ohjelmointivaiheeseen. Komponentin testisuunnitelma katselmoidaan ja hyväksytään osana komponentin suunnitteludokumentaatiota. Komponentin katsotaankin olevan valmis vasta kun se on hyväksyttävästi testattu.

Toinen testausvaihe on nimeltään *integraatiotestaus* (integration testing). Integraatiotestaus voi alkaa, kun komponenttitestatut komponentit on yhdistetty sovelluksiksi, ja sovellukset on yhdistetty järjestelmäksi. Ohjelmistokehittäjät suorittavat integraatiotestin testaamalla komponentteja osana koko järjestelmää. Siinä missä komponenttitestauksen voi yleensä suorittaa kehittäjien omissa työpisteissä, suoritetaan integraatiotestaus yleensä valvotussa laboratorioympäristössä.

Komponentti- ja integraatiotestauksessa tarkastetaan suunnittelu- ja ohjelmointivaiheiden tulokset. Testauksessa pyritään löytämään

komponenttien suunnittelun ja ohjelmoinnin aikana tapahtuneita virheitä. Yleensä integraatiotestaus aloitetaan heti, kun järjestelmä on siihen tarpeeksi vakaa. Integraatiotestauksessa toteutettua järjestelmää käydään läpi vertaamalla sitä vaatimuslauseisiin, käyttötapauksiin ja määriteltyihin operaatioihin.

Kolmatta testausvaihetta kutsutaan *järjestelmätestaukseksi* (system testing), jossa pyritään osoittamaan, että järjestelmä täyttää sille asetetut vaatimukset ja sille määritellyn toiminnallisuuden. Testaus suoritetaankin analyysivaiheen määrityksiä, lähinnä käyttötapauksia, vasten. Järjestelmätestaamisen suorittavat järjestelmätestaajat, eivät ohjelmistokehittäjät. He käyttävät järjestelmää loppukäyttäjän näkökulmasta ja pyrkivät löytämään järjestelmästä toiminnallisia puutteita ja vikoja. Myös järjestelmän käytettävyys tulee testattua tässä vaiheessa.

5.7. Kritiikkiä tilakaavioista OMT++:n yhteydessä

Jaaksi ja kumppanit suhtautuvat hyvin kriittisesti tilakaavioiden käyttämiseen järjestelmän suunnittelussa. Heidän mukaansa ohjelmistokehittäjät pitävät sekvenssikaavioita kaikkein intuitiivisimpana järjestelmien käyttäytymisen mallintamiseen käytetyistä kuvaustekniikoista ja ilmoittavat samalla valtaosan kehittäjistä inhoavan tilakaavioita. Tilakaavioiden käytöllä järjestelmien suunnittelussa ei heidän mielestään ole mainittavia hyötyjä lukuunottamatta niiden käyttöä käyttöliittymämäärittelyn dialogikaavioina. Tilakaaviot kuvaavat järjestelmän heidän mielestään paremminkin toistensa kanssa vuorovaikutuksessa olevina ärsykkeisiin reagoivina olioina kuin monimutkaisen tilakäyttäytymisen omaavina osajärjestelminä.

Vaikka he ovatkin perehtyneet tilakaavioita käsittelevään kirjallisuuteen ja tuntevat useita tilakaavioiden nimeen vannovia tutkijoita, pitävät he silti tilakoneita liian ylimainostettuina. He eivät kuitenkaan väitä, että tilakaaviot itsessään olisivat huonoja tai teoreettisesti heikkoja, vaan että tiukkojen aikataulujen puristuksessa ohjelmiston suunnitteluprosessissa on pakko keskittyä olennaiseen, käyttäen ainoastaan parhaiksi todettuja kuvausmenetelmiä [Jaaksi *et al.*, 1999].

Seuraavaksi tutkitaan Jaaksin ja kumppanien väitteiden todenperäisyyttä perehtymällä tilakaavioiden käyttöön järjestelmän toiminnallisuuden kuvaamisessa.

6. Käyttäytymismallit

Useissa skenaariopohjaisissa ohjelmistokehitysmenetelmissä (mm. OMT++:ssa) järjestelmän kuvaamiseen käytetään useita skenaarioita, joilla kuvataan järjestelmän olennaisimmat ja mahdollisesti myös joitakin poikkeuksellisia toimintoja. Jotta järjestelmästä saataisiin mahdollisimman kattava järjestelmäkuvaus, pitää nämä erilliset skenaariot yhdistää yhdeksi kokonaisuudeksi [Uchitel *et al.*, 2003a].

Kuten jo aiemmin todettiin, on skenaarioiden yhteydessä laajimmin levinneitä notaatioita tapahtumasekvenssikaaviot [ITU Z.120] ja UML:n sekvenssidiagrammit [UML, 2007]. Nämä notaatiot ovat yksinkertaisimmillaan hyvin intuitiivisia ja niillä on laajasti hyväksytyt semantiikat. Yksi kaavio sisältää kuitenkin suhteellisen vähän tietoa. Merkityksellisen järjestelmäkuvauksen luomiseen tarvitaan useimmiten useita skenaarioita. Tästä johtuen *skenaariosynteesillä*, yksittäisten skenaarioiden yhdistämisellä kokonaisuudeksi, on keskeinen merkitys järjestelmää suunniteltaessa. Miten joukkoa skenaarioita pitäisi tulkita? Miten skenaariot liittyvät toisiinsa?

Skenaarioiden yhdistämiseen on kaksi tapaa. Skenaarioiden väliset suhteet on joko voitava päätellä jollakin tapaa, tai sitten skenaarioiden laatijoiden on määritteltävä suhteet eksplisiittisesti. Suhteiden ekspliittiselle määrittelemiselle on kehitetty useita eri keinoja. Esimerkiksi ITU:n standardissa käytössä on graafimainen notaatio, korkean tason tapahtumasekvenssikaavio (high-level Message Sequence Chart, hMSC), jota käytetään kuvaamaan järjestelmän siirtymistä skenaariosta toiseen. Idean perustana ovat koosteskenaariot: uusia skenaarioita voidaan määritellä toisten skenaarioiden avulla koostamalla niitä peräkkäisesti, vaihtoehtoisesti ja iteratiivisesti. Tämä mahdollistaa hyvinkin monimutkaisten järjestelmien käyttäytymisen kuvaamisen.

Krüger ja kumppanit [1998] puolestaan esittelevät skenaarioiden yhdistämiseen tarkoitetut *tilatunnisteet* (state condition). Tilatunnisteilla samaistetaan yhtenäisiä tiloja eri skenaarioissa. Samanlaiset tilatunnisteet eri skenaarioissa osoittavat, että skenaarioiden kuvaamissa tapahtumaketjuissa on kohta, jossa järjestelmän eteneminen voidaan esittää vaihtamalla skenaariota.

Välittämättä siitä, miten skenaarioiden väliset suhteet määritellään, on skenaariomäärittelyjen perimmäisenä tarkoituksena kuvata järjestelmän haluttu käyttäytyminen. Siitä syystä järjestelmän käyttäytymisen analysoinnilla pitäisi olla keskeinen rooli skenaariopohjaisten määrittelyiden kehittämisessä. Analysoinnin mahdollistamiseksi on skenaarioiden syntetisointiin kehitetty useita erilaisia algoritmeja, joiden avulla skenaarioista voidaan tuottaa tilakone-perustaisia käyttäytymismalleja. Näiden mallien avulla, järjestelmästä

saatavan erilaisen näkymän lisäksi, voidaan järjestelmän käyttäytymistä tutkia sekä skenaarioiden sisältämien virheiden havaitsemiseksi, että skenaarioiden tarkentamiseksi ja muokkaamiseksi.

Käyttäytymismallit ovat tarkkoja, abstrakteja kuvauksia järjestelmän tarkoitetusta käyttäytymisestä. Nillä on vankat matemaattiset perusteet, jotka mahdollistavat mallien täsmällisen analyysin ja mallin ominaisuuksien mekaanisen todistamisen. Pelkkä järjestelmän kuvaaminen tilakaaviona ei kuitenkaan itsessään tuota näitä hyötyjä. Hyödyn saamiseksi syntetisoinnin tuloksena saatavaa järjestelmän käyttäytymistä kuvaavaa tilakonetta on jollakin tavoin pystyttävä analysoimaan. Luvussa 4 esitetyn LTSA-työkalun mallintarkastus- ja animointiominaisuuksien avulla voidaan osin automaattisestikin analysoida FSP-notaatiolla määriteltyjä LTS-tilakoneita.

Vaikka käyttäytymismallien hyödyllisyydestä monimutkaisten järjestelmien suunnittelussa on saatu konkreettista näyttöä, on niiden leviäminen ohjelmistoteollisuuden käyttöön ollut hidasta. Yksi perustavanlaatuinen syy tähän on käyttäytymismallien laatimisen vaikeus, joka vaatii huomattavaa osaamista sekä ajankäyttöä [Uchitel *et al.*, 2003b].

6.1. Skenaariosynteesi

Laajojen järjestelmien kuvaamiseen käytetään poikkeuksetta useita tapahtumasekvenssikaavioita. Useiden sekvenssikaavioiden yhdistäminen yhdeksi järjestelmää kuvaavaksi tilakoneeksi on huomattavasti vaikeampaa kuin kohdassa 3.4 esitelty yksittäisen tapahtumasekvenssikaavion muuttaminen tilakoneeksi.

Yksittäinen skenaario kuvaa järjestelmän toimintaa tietyn tehtävän suorittamisen yhteydessä. Yhdistettynä skenaariot antavat täydellisemmän kuvauksen järjestelmän odotetusta käyttäytymisestä. Skenaarioiden yhdistämiseen on useita eri menetelmiä, joiden voidaan sanoa perustuvan kolmeen periaatteeseen: skenaarioiden koostamiseen (scenario composition), tilojen tunnistamiseen (state identification) ja laukaisimiin (triggers).

6.1.1. Skenaarioiden koostaminen

ITU:n käytössä olevassa menetelmässä [ITU Z.120] keskitytään monimutkaisuuden hallintaan tarkoitettuihin välineisiin. Yksinkertaiset tapahtumaketjut kuvataan perustapahtumasekvenssikaavioilla (bMSC) (kts. kohta 3.2). Lisäksi esitellään kolme käsitettä bMSC:en yhdistämiseen: vertikaali- ja vaihtoehtoinen koostaminen sekä silmukat. Vertikaalikoosteella voidaan kaksi tapahtumasekvenssikaaviota yhdistää peräkkäisesti. Vaihtoehtokooste määrittelee joukon sekvenssikaavioita, joista järjestelmä voi valita tilanteeseen sopivimman. Silmukoilla tapahtumasekvenssikaavio

voidaan peräkkäisesti yhdistää itseensä. Periaatteena on mahdollistaa skenaarioiden käyttäminen yhä monimutkaisemman käyttäytymisen rakennuspalikoina. Skenaarioista voidaan koostaa uusia, monimutkaisempia skenaarioita [Uchitel *et al.*, 2003a].

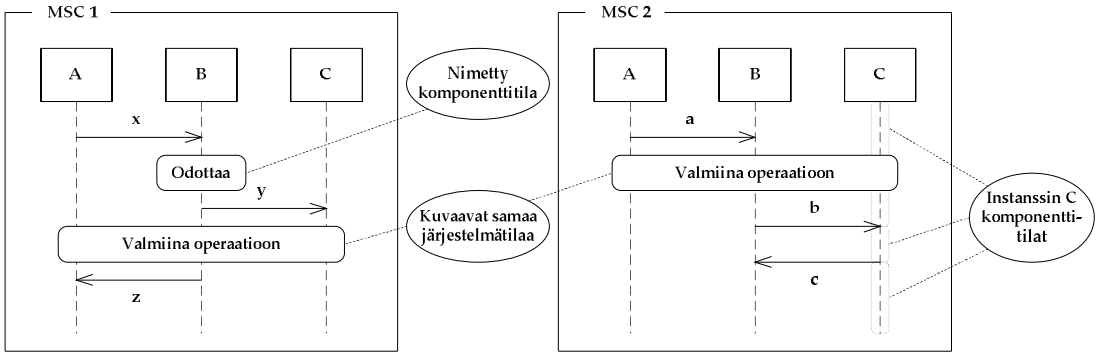
ITU määrittelee joukon syntakseja skenaarioiden koostamiselle, mutta yleisimmin käytössä oleva notaatio on korkean tason tapahtumasekvenssikaavio (hMSC). hMSC on suunnattu graafi, jonka solmut vastaavat joko toista hMSC:tä tai bMSC:tä (esimerkki hMSC:stä esitetään kuvassa 6.2). Solmuja yhdistävät kaaret kuvaavat skenaarioiden kaikki mahdolliset suoritusjärjestykset, ja niiden avulla skenaarioita voidaan yhdistää peräkkäisesti, vaihtoehtoisesti ja silmukoilla. Korkean tason tapahtumasekvenssikaavioilla voidaan skenaariomäärittely jakaa hallittaviin osiin yksinkertaisella ja ymmärrettävällä tavalla, josta samalla käy ilmi skenaarioiden väliset suhteet. Jos järjestelmän käyttäytymisen haarautumiseen ei kuitenkaan ole käytössä muita mekanismeja kuin vaihtoehtokooste, täytyy skenaariot usein jakaa hyvin pieniksi perustapahtumasekvenssikaavioiksi. Näin saatavat skenaariot saattavat olla täysin merkityksettömiä ilman korkean tason tapahtumasekvenssikaaviota niitä selittämässä. Tämä sotii tapahtumasekvenssikaavioiden intuitiivisuuden perusideaa vastaan, jossa yhdellä kaaviolla pystytään selkeällä tavalla kuvaamaan merkittävä osa järjestelmän käyttäytymistä.

6.1.2. Tilojen tunnistaminen

Toinen yleisesti käytössä oleva näkökulma skenaarioiden yhdistämiseen on yleisten komponentti- tai järjestelmätilojen tunnistaminen skenaariojoukosta [Koskimies *et al.*, 1998; Krüger *et al.*, 1998; Whittle and Schumann, 2000]. Lähestymistavan oletuksena on, että skenaariomäärittely itse asiassa kuvaa järjestelmän komponenttien käyttäytymisen mallintavan tilakoneen. Tulkinnan mukaan tapahtumasekvenssikaavio kuvaa siis joukon komponenttien (instanssit) tiloja ja näiden tilojen muuttumiseen johtavia tapahtumia (tilasiirtymiä). Skenaariossa kahden peräkkäisen tapahtuman väliin jäävää tilaa kutsutaan skenaariotilaksi ja tietyn instanssin peräkkäisten tapahtumien väliin jäävää tilaa kutsutaan komponenttitilaksi. Skenaariotila määrittää instanssien sisäisen komponenttitilan tietyllä ajan hetkellä (kuva 6.1).

Komponenttitilojen tunnistamiseen voidaan käyttää kahta erilaista lähestymistapaa. Komponenttitiloja voidaan nimetä, jolloin kahden samannimisen komponenttitilan katsotaan viittaavaan yhteen ja samaan komponentin tilaan (kuva 6.1). Toinen tapa on käyttää sovittuja sääntöjä komponenttitilojen tunnistamiseen. Säännöt perustuvat usein sovellusalueen yleiseen tuntemukseen ja suunniteltavan järjestelmän erityistietämykseen.

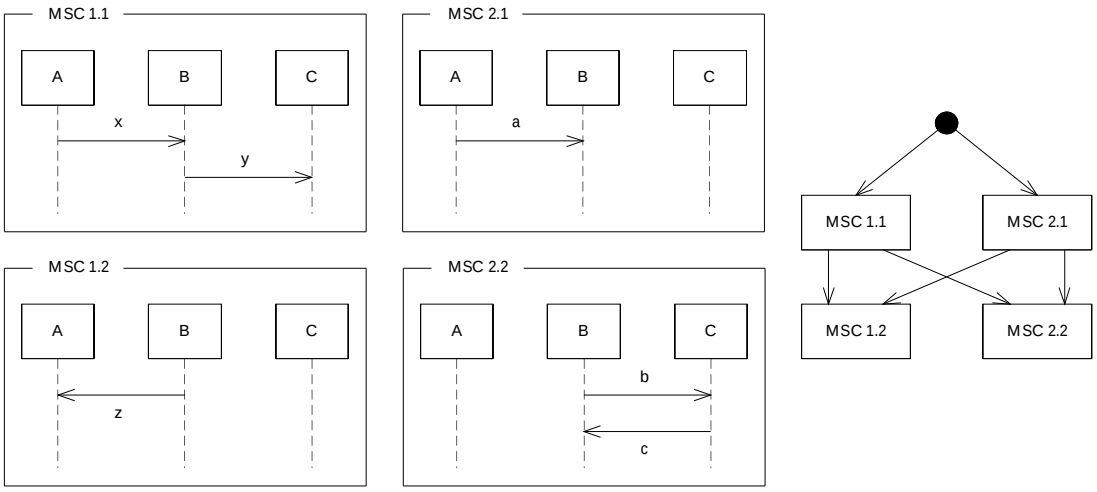
Esimerkiksi Whittle ja Schumann [2000] käyttävät esittämässään menetelmässä UML:n OCL (Object Constraint Language) -määrittelyä, jossa määritellään järjestelmässä lähetettäviin sanomiin liityvät esi- ja jälkiehdot. Käymällä läpi tapahtumasekvenssikaavioita yhdessä OCL-määrittelyn kanssa, arvotetaan komponenttitilat menetelmässä esitettyjen *tilamuuttujien* (state variables) avulla. Samanarvoisten komponenttitilojen katsotaan viittaavan yhteen ja samaan komponentin tilaan.



Kuva 6.1. Komponentti- ja järjestelmätilat

Komponenttitilojen tunnistamisen sijaan skenaarioista voidaan tunnistaa myös järjestelmätiloja. Järjestelmätila määrittelee skenaarion jokaisen instanssin tilan tietyllä ajan hetkellä. Eri skenaarioissa esiintyvät samannimiset järjestelmätilat viittaavat yhteen ja samaan järjestelmän sisäiseen tilaan.

Komponentti- ja järjestelmätilojen määrittämisen avulla voidaan skenaarioiden välille luoda liitoksia, jotka määrittelevät skenaarioiden väliset suhteet toisiinsa. Verrattuna edellä esitettyyn skenaarioiden koostamiseen, voidaan tilojen tunnistamisen avulla kuvata hyvinkin monimutkaista komponentin käyttäytymistä minkä mittaisissa tapahtumasekvenssikaavioissa hyvänsä. Esimerkkinä järjestelmätilojen käyttämisestä vaihtoehtoisen käyttäytymisen kuvaamiseen tutkimme kuvaa 6.1, jonka kummassakin tapahtumasekvenssikaavioissa on järjestelmätila ”Valmiina operaatioon”. Ajatellaan, että järjestelmässä tapahtuu x ja y. Sen sijaan, että järjestelmä etenisi tapahtumaan z, voikin järjestelmässä tapahtua b ja c. Saman asia kuvaaminen skenaarioita koostamalla vaatisi tapahtumasekvenssikaavioiden jakamista ”Valmiina operaatioon” kohdasta, sekä hMSC:n laatimista kuvaamaan järjestelmän vaihtoehtoista käyttäytymistä (kuva 6.2). Kuvasta käy myös hyvin ilmi, miten skenaarioiden paloittelu voi johtaa merkityksettömän kokoisiin tapahtumasekvenssikaavioihin (kuvan 6.2 MSC:t 1.2 ja 2.1).



Kuva 6.2. bMSC:t ja hMSC.

6.1.3. Laukaisimet

Skenaarioita voidaan yhdistää myös käyttämällä laukaisimia tai esiehtoja. Tällöin skenaarioiden yhdistäminen perustuu siihen, milloin skenaariot voivat tapahtua. Tämä lähestymistapa on yleisimmin käytössä informaalien kehitysmenetelmien yhteydessä, joissa skenaarioiden kuvauksiin kuuluvat luonnollisella kielellä esitetty esiehdot [Quatrani, 1998]. Esiehto voi viitata informaalisti tiettyyn tilaan, jossa skenaario voi tapahtua, tai skenaarion laukaisevaan tapahtumaketjuun. Myös edellä mainittua OCL-määrittelyä voidaan käyttää laukaisimien määrittämiseen.

Laukaisimien käytön etuna on skenaarioiden välisten kytkentöjen keveys. Laukaisimet antavat vapauden skenaarioiden määrittelyyn itsenäisinä kokonaisuuksina, toisin kuin esimerkiksi skenaarioiden koostamiseen perustuvissa menetelmissä, joissa skenaarioiden välisten liittymien täytyy olla yhteensopivia. Tämä piirre on samalla myös lähestymistavan haitta, sillä se tekee skenaarioiden yhdistämisestä huomattavan paljon vaikeampaa kuin kummassakaan edellä mainitussa menetelmässä, etenkin jos laukaisimet ovat ainoa skenaarioiden välisten suhteiden määrittämiseen käytetty keino [Uchitel *et al.*, 2003a].

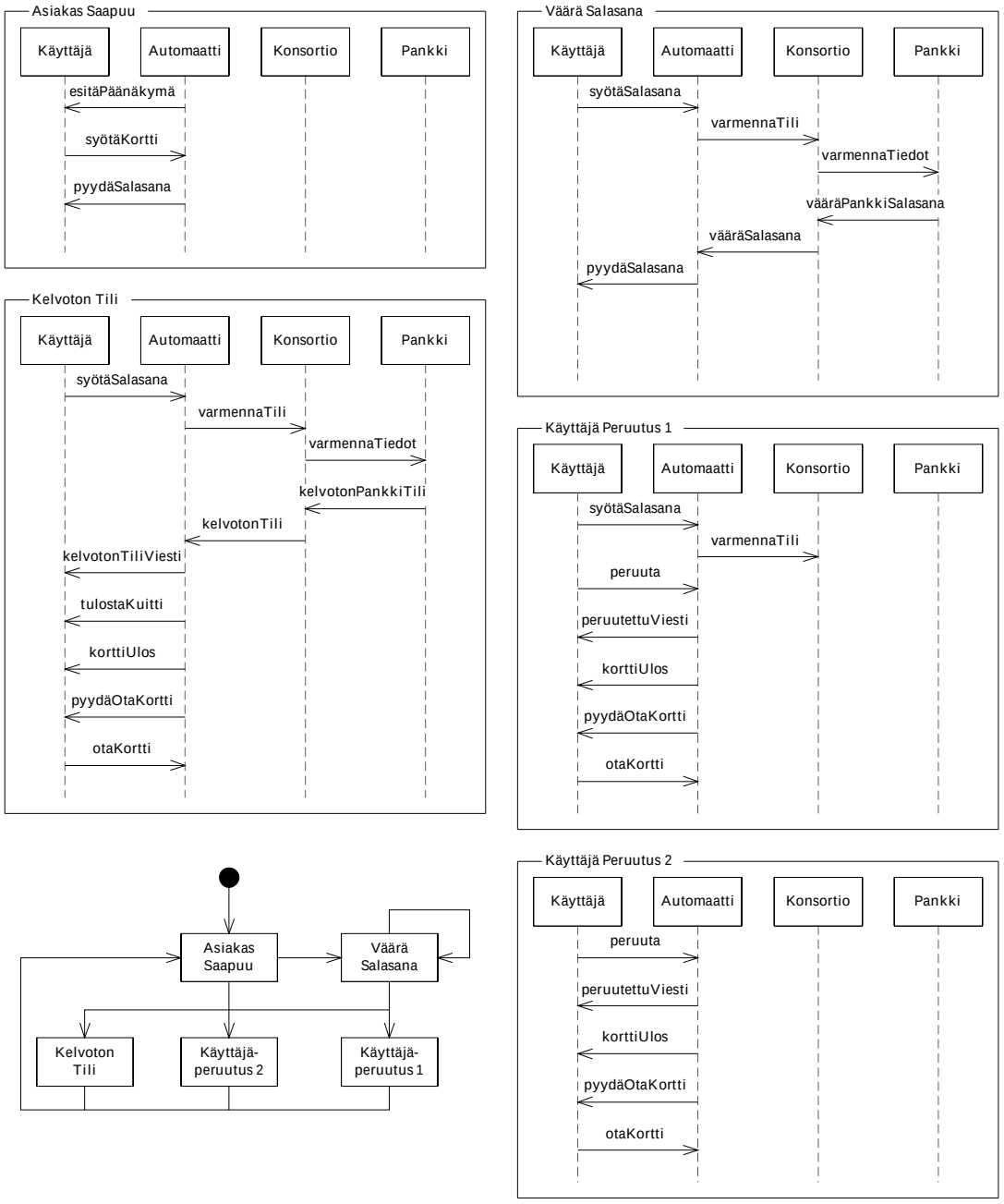
6.2. Esimerkki skenaariosynteesistä: Whittle and Schumann

Whittle ja Schumann [2000] esittelevät tilojen tunnistamiseen perustuvan algoritmin, joka tuottaa tilakaavioita joukosta skenaarioita käyttämällä apuna sanomien esi- ja jälkiehdot kuvaavaa OCL-määritelmää.

```
korttiSisässä, korttiOtettavissa, salasanaAnnettu : Boolean
kortti : Kortti
salasana : Merkkijono

syötäKortti( k : Kortti )
pre  : korttiSisässä = false
post : korttiSisässä = true and kortti = k

syötäSalasana( s : Merkkijono )
pre  : salasanaAnnettu = false
post : salasanaAnnettu = true and salasana = s

otaKortti()
pre  : korttiOtettavissa = true
post : korttiOtettavissa = false and korttiSisässä = false

esitäPäänäkymä()
pre  : korttiSisässä = false and korttiOtettavissa = false
post :

pyydäSalasana()
pre  : salasanaAnnettu = false
post :

korttiUlos()
pre  : korttiSisässä = true
post : korttiSisässä = false and korttiOtettavissa = true and
kortti = null and salasana = null and salasanaAnnettu = false

pyydäOtaKortti()
pre  : korttiOtettavissa = true
post :

peruutettuViesti()
pre  : korttiSisässä = true
post :
```

Kuva 6.3. Pankkiautomaatin OCL.

Seuraavassa havainnollistetaan Whittlen ja Schumannin skenaariosynteesiä. Esimerkiksi otetaan pankkiautomaattijärjestelmä, mutta järjestelmän laajuuden vuoksi tässä keskitytään tarkastelemaan ainoastaan hyvin suppeata ja rajattua osaa järjestelmän toiminnallisuudesta. Periaate on silti käyttökelpoinen laajojenkin järjestelmien kuvaamiseen. Oletetaan siis, että käytössä on kuvassa 6.4 esitetty joukko tapahtumasekvenssikaavioita, jotka kuvaavat, kuinka käyttäjä pääsee käsiksi tiliinsä pankkiautomaattia käyttämällä. Pankkiautomaatti on yhteydessä luottokorttikonsortion tietoverkkoon, josta taas on yhteys pankin tietojärjestelmään.



Kuva 6.4. Pankkiautomaattijärjestelmän sekvenssikaaviot.

Lisäksi oletetaan, että käytössä on sovellusalueen tietämystä hyväksi käyttäen laadittu OCL-määritelmä (kuva 6.3). OCL-määritelmässä kuvataan joukko pankkiautomaattiin liittyviä tilamuuttujia – `korttiSisässä`, `korttiOtettavissa`, `salasanaAnnettu`, `kortti`, `salasana` – yhdessä sanomakohtaisten esi- ja jälkiehtojen kanssa. Sanomiin liittyvät esiehdot määrittävät tilamuuttujan arvot, joiden täytyy olla voimassa, jotta sanoma voisi tapahtua. Jälkiehdot puolestaan määrittävät, miten sanomat muuttavat tilamuuttujien arvoja. Löytämällä ne skenaarioissa kuvatut sanomat, joille on

olemassa esi- ja jälkiehdot, voidaan OCL-tilamuuttujien arvot määritellä tietyissä skenaarioiden kohdissa.

Esimerkiksi tapahtumasekvenssikaaviossa *Kelvoton Tili* ensimmäinen sanoma on `esitäPäänäkymä`. Lisäksi OCL-määritelmästä voidaan nähdä sanomaan liittyvä esiehto, jonka mukaan `korttiSisässä` ja `korttiOtettavissa` ovat epätosia. Näin voidaan päätellä järjestelmän tila tapahtumasekvenssikaavion *Kelvoton Tili* alussa.

Menetelmällä on siis mahdollista arvottaa (joissain kohdissa osittaisesti) tapahtumasekvenssikaavion jokaisen vaiheen kuvaava tilamuuttuja. Whittle ja Schumann käyttävät näitä arvotuksia kahdella tavalla. Ensinnäkin niitä voidaan käyttää silmukoiden löytämiseen käyttäytymisestä. Instanssikohtaisesti kaksi saman arvotuksen omaavaa tilaa määrittävät silmukan. Kaikki sanomat eivät kuitenkaan aiheuta muutosta tilamuuttujassa. Silmukan määrittelevän tilan täytyykin olla tulos tilamuuttujan arvoa muuttavasta sanomasta. Toiseksi, kahden samaa komponenttia kuvaavan instanssin yksittäisten tilojen katsotaan viittaavaan yhteen ja samaan tilaan, kun tiloihin on saavuttu samannimisten tapahtumien suorittamisen jälkeen, ja tilamuuttujien arvot ovat identtiset kyseisissä tiloissa. Havainnollisempi esimerkki Whittlen ja Schumannin menetelmästä esitetään myöhemmin kohdassa 6.4.

6.3. Esimerkki skenaariosynteesistä: Koskimies et al.

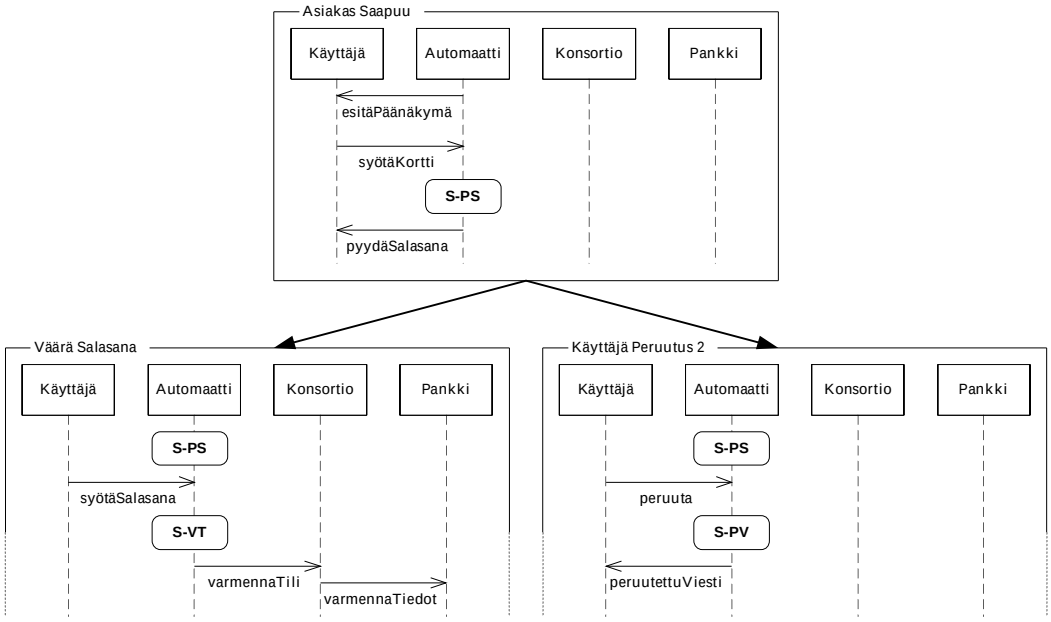
Koskimies ja kumppanit ovat kehittäneet BK-algoritmiin [Biermann and Krishnaswamy, 1976] perustuvaa synteesialgoritmia käyttävän SCED-työkalun komponenttikohtaisten tilakaavioiden tuottamiseen skenaariopohjaisista määrittelyistä [Koskimies *et al.*, 1998]. Työkalun tuottamissa tilakaavioissa tilat on nimetty komponentista lähtevien sanomien mukaan, kun taas tilasiirtymät on nimetty komponenttiin saapuvien sanomien mukaan. Lähestymistavassa käytetty oletus on, että komponentin kyky lähettää tietty sanoma määrittelee yksikäsitteisesti kyseisen komponentin tilan. Jos synteesin yhteydessä jossakin komponentin tilassa esiintyy epädeterminististä käyttäytymistä, ei tiloja kuitenkaan yhdistetä, vaan ne pidetään erillisinä.

6.4. Yksinkertainen skenaariosynteesi käyttäen FSP-notaatiota

Seuraavaksi esitetään yksinkertainen, helppo ja suoraviivainen tekniikka käyttäytymismallin määrittelyyn joukosta tapahtumasekvenssikaavioita käyttäen FSP-notaatiota. Menetelmä yhdistää skenaariosynteesin periaatteita niin skenaarioiden koostamiseen kuin tilatunnisteiden käyttöön perustuvista menetelmistä.

Käyttäytymismallin määrittely aloitetaan laatimalla järjestelmän käyttäytymistä kuvaavista tapahtumasekvenssikaavioista korkean tason tapahtumasekvenssikaavio ITU:n periaatteiden mukaan. Näin saadaan selville, missä järjestyksessä tapahtumasekvenssit voivat järjestelmässä tapahtua. Korkean tason tapahtumasekvenssikaaviota tutkimalla voidaan havaita tapahtumasekvenssikaaviot, joiden tilat ovat kaavioiden alussa samat. Tilat kahden (tai useamman) kaavion alussa ovat samat, jos ne voidaan suorittaa vaihtoehtoisesti toistensa sijaan.

Tämän jälkeen tarkasteluun otetaan yksi tapahtumasekvenssikaavio kerrallaan ja jokaisesta siinä esiintyvästä instanssista luodaan FSP-prosessi, jonka tilat ja tilasiirtymät määräytyvät Koskimiehen ja kumppanien periaatteen mukaan. Instanssista lähtevät sanomat määrittävät yksikäsitteisesti instanssin tilan ennen sanoman lähetystä. LTS ja FSP eivät kuitenkaan tue tilojen nimeämistä. Tämä puute voidaan kiertää käyttämällä lokaaliprosesseja, joita voidaan käyttää simuloimaan tilojen nimeämistä (esim. S_Pyydä_Salasana). Lokaaleiksi prosesseiksi valitaan siis komponentista ulos lähtevät sanomat, jonka ensimmäinen toiminto nimetään sanoman mukaan.



Kuva 6.5. Automaatti-instanssin LTS-kaavio.

Tämä prosessi toistetaan jokaiselle tapahtumasekvenssikaaviolle. Kun eri kaavioiden instanssit lähettävät samannimisen sanoman, voidaan käsiteltävän kaavion tila yhdistää jo mallinnettuun lokaaliprosessiin. Samoin vaihtoehtoisten tapahtumasekvenssikaavioiden aloitustilat voidaan yhdistää. Tätä on pyritty havainnollistamaan kuvassa 6.5, jossa on kuvattu vaihtoehtoisten tapahtumasekvenssikaavioiden alkutilojen yhteneväisyyttä sen

automaatti-instanssin osalta. Tilojen tunnisteina toimivat lyhenteet perustuvat kaavioissa seuraavaksi suoritettaviin toimintoihin ja niiden tarkoitus on ainoastaan havainnollistaa esitystä. Kuvassa 6.4 esitetyn korkean tason tapahtumasekvenssikaavion mukaan *Asiakas Saapuu* -kaavion jälkeen voidaan siirtyä mm kaavioon *Väärä Salasana* tai kaavioon *Käyttäjä Peruutus 2*. Koska viimeisin toiminnon mukaan nimetty tila ensin suoritetusta tapahtumasekvenssistä on S_pyydäSalasana (S-PS), pysyy automaatin tilana kaavioiden *Väärä Salasana* ja *Käyttäjä Peruutus 2* alussa S_pyydäSalasana.

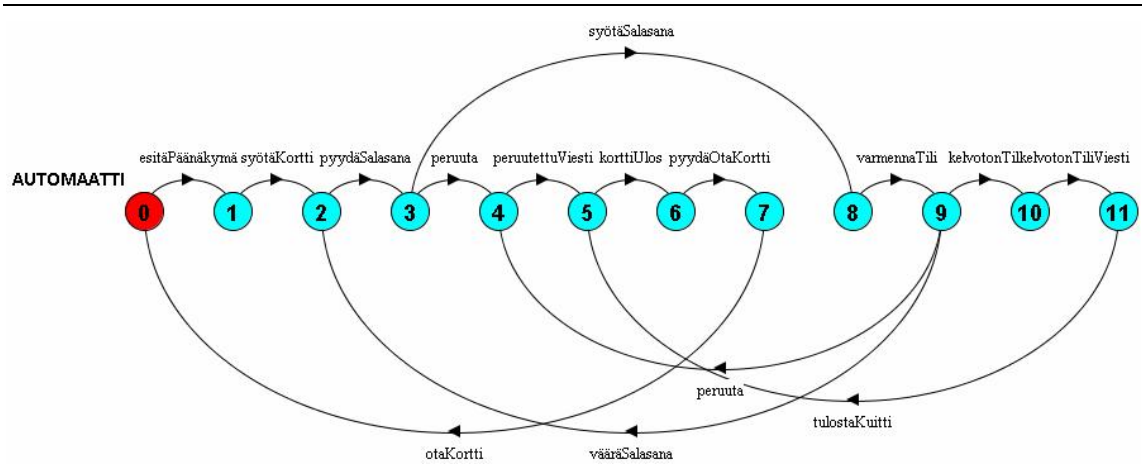
Asian vaikutus FSP-määrittelyyn voidaan havainnoida kuvan 6.6 lokaaliprosessista S_pyydäSalasana. Prosessin määrittelyn mukaan lokaaliprosessissa suoritetaan ensin toiminto `pyydäSalasana`, jonka jälkeen voidaan suorittaa joko toiminto `syötäSalasana` (kaaviosta *Väärä Salasana*) tai toiminto `peruuta` (kaaviosta *Käyttäjä peruutus 2*). Seuraavasta toiminnosta riippuen siirrytään tapahtumasekvenssikaavioiden mukaisesti joko tilaan (lokaaliprosessiin) S_varmennaTili tai tilaan S_peruutettuViesti.

```
AUTOMAATTI = ( esitäPäänäkymä -> AUTOMAATTI2 ),
AUTOMAATTI2 = ( syötäKortti -> S_pyydäSalasana ),
S_pyydäSalasana = ( pyydäSalasana -> ( syötäSalasana -> S_varmennaTili
                                     | peruuta -> S_peruutettuViesti ) ),
S_varmennaTili = ( varmennaTili -> ( kelvotonTili -> S_kelvotonTiliViesti
                                   | vääräSalasana -> S_pyydäSalasana
                                   | peruuta -> S_peruutettuViesti ) ),
S_kelvotonTiliViesti = ( kelvotonTiliViesti -> S_tulostaKuitti ),
S_tulostaKuitti = ( tulostaKuitti -> S_korttiUlos ),
S_korttiUlos = ( korttiUlos -> S_pyydäOtaKortti ),
S_pyydäOtaKortti = ( pyydäOtaKortti -> otaKortti -> AUTOMAATTI ),
S_peruutettuViesti = ( peruutettuViesti -> S_korttiUlos ).
```

Kuva 6.6. Automaatin FSP-määritelmä.

Kuvassa 6.6 on esitetty tällä tekniikalla tehty FSP-määritelmä edellä esitetystä pankkiautomaattijärjestelmän automaatti-instanssista ja kuvassa 6.7 on esitetty automaatin LTS-kaavio. Järjestelmän täydellinen, LTSA:lla tulkittavissa oleva FSP-määrittely, on esitetty liitteessä 2.

Koskimiehen ja kumppanien menetelmään vaikuttava epädeterministisyys voidaan havaita LTS-tilakoneesta tutkimalla tilasta lähteviä tilasiirtymiä. Samannimisten tilasiirtymien lähteminen useampaan eri tilaan on merkkinä tilassa olevasta epädeterministisyydestä, jolloin mallia täytyy muuttaa epädeterministisyyden poistamiseksi.



Kuva 6.7. Automaatti-instanssin LTS-kaavio.

Suorittamalla käyttäytymismallia LTSA-työkalulla saadaan selville, että mallin on mahdollista joutua lukkiutumistilaan. Lukkiutumistilaan päädytään, kun käyttäjä suorittaa `peruuta` toiminnon syötettyään ensin salasanansa, jonka jälkeen seuraavan kerran käyttäjän syötettyä salasanan, ei automaatti pysty suorittamaan toimintoa `varmennaTili`. Toiminnon suorittamisen estää konsortio- ja pankki-instanssien suorittaman tilin varmentamiseen liittyvän tapahtumaketjun suorittamisen kesken jääminen, sillä konsortio ei pysty palauttamaan tilin varmentamiseen liittyvää tulosta (`kelvotonTili` tai `vääräSalasana`) automaatin vaihdettua tilansa `peruuta`-toiminnon yhteydessä. Tällöin automaatti jää odottamaan, että se voisi suorittaa toiminnon varmennaTili, ja samaan aikaan konsortio jää puolestaan odottamaan, että se voisi suorittaa tilin varmentamiseen liittyvän tulostoiminnon. Järjestelmän suoritus ei näin ollen pääse etenemään.

Tilanteen korjaamiseksi voitaisiin esimerkiksi muuttaa automaatin käyttäytymistä siten, että `peruuta`-toiminnon jälkeen konsortiolle lähetettäisiin viesti tilin varmentamiseen liittyvän toiminnallisuuden keskeyttämisestä.

Uchitel ja kumppanit [2003b] määrittelevät kokonaan oman kielen FSP-notaatiolla määriteltyjen käyttäytymismallien laatimiselle tapahtumasekvenssikaavioista, mutta jo tällainen yksinkertainenkin menetelmä pystyttiin osoittamaan toimivaksi ja hyödylliseksi, löydettiinhän sen avulla malliin ja tapahtumasekvenssikaavioihin piiloutunut lukkiutumistila.

6.5. Käyttäytymismallin hyödyntäminen vaatimusmäärittelyssä

Ohjelmistoteollisuudessa on jo pitkään ymmärretty vaatimusmäärittelyn tärkeys kehitettävän ohjelmiston perustana. Hyvä vaatimusmäärittely vaatii järjestelmän käyttäjien osallistumista vaatimusten määrittelyyn. Jotta käyttäjät voisivat ottaa osaa vaatimusten määrittelyyn, on käyttäjien ja suunnittelijoiden

käytössä oltava yhteinen kieli, jolla kehitettävästä järjestelmästä voitaisiin keskustella. Ei siis ole ihme, että skenaariot ja etenkin tapahtumasekvenssikaaviot eri muodoissaan ovat tulleet yleisesti osaksi vaatimusmäärittelyä (OMT++ tosin pitäytyy käyttötapauksissa). Uchitel ja kumppanit [2003b] esittävät mielenkiintoisen tavan hyödyntää käyttäytymismalleja kehitettäessä järjestelmän vaatimusmäärittelyä.

6.5.1. Määrittelemätön käyttäytyminen

Tilakoneisiin perustuvien formalismien (mm. LTS) oletetaan olevan täydellisiä kuvauksia järjestelmän käyttäytymisestä jollakin abstraktiotasolla. Jos LTS-kaavio ei anna mahdollisuutta tietyn tapahtumaketjun suorittamiseen, on perusteltua olettaa, ettei järjestelmän pysty, eikä sen pitäisikään kyetä kyseisen tapahtumaketjun suorittamiseen. Tämä olettamus on periaatteessa oikeutettu järjestelmäsuunnittelun lopussa, kun koko järjestelmän toiminnallisuus on suunniteltu ja mallinnettu. Olettamuksella ei kuitenkaan ole pohjaa järjestelmän suunnittelun ja mallinnuksen ollessa vielä kesken, kuten esim. vaatimusmäärittelyn yhteydessä.

Tässä vaiheessa olisikin tärkeää erottaa toistaiseksi määrittelemätön käyttäytyminen sellaisesta käyttäytymisestä, joka järjestelmältä on tietoisesti estetty. Jos järjestelmän käyttäytymisen kuvauksessa olevat aukot voidaan erottaa järjestelmän tietoisesti estetystä käyttäytymisestä, antaa se mahdollisuuden kehittää määrittelyä asteittain tarkemmaksi. Sillä vaikka määrittelylle sopivasta tarkkuudesta ja täsmällisyydestä voidaankin väitellä loputtomiin, lienee itsestään selvää, että jos järjestelmän käyttäytyminen on laajalti hämärän peitossa, on järjestelmän käyttäytymisen arvailu pelkkien olettamusten varassa hyvinkin vaarallista [Uchitel *et al.*, 2003b].

Jatketaan siis edellämainitun pankkiautomaattijärjestelmän analysointia. Oletetaan, että käytössä on kohdassa 6.7 esitetty pankkiautomaatin LTS-kaavio sekä kohdassa 6.3 esitetty OCL-määrittely järjestelmän tapahtumille (sanomille). LTS-kaaviossa esitetään pankkiautomaatin käyttäytyminen suoritettaessa kuvan 6.4 tapahtumasekvenssikaavioiden kuvaamia tapahtumaketjuja. LTS-kaaviosta voidaan nähdä automaatin kykenevän mm. tapahtumaketjun <esitäPäänäkymä, syötäKortti, pyydäSalasana, syötäSalasana, varmennaTili...> suorittamiseen. Samalla LTS-kaaviosta voidaan havaita myös automaatille mahdottomia tapahtumaketjuja. Automaatti ei kykene suorittamaan esimerkiksi tapahtumaketjua <esitäPäänäkymä, syötäKortti, syötäKortti>, sillä automaatin suoritettua tapahtumat esitäPäänäkymä ja syötäKortti, on LTS-kaavio tilassa 2, josta ei ole ulospäin suuntautuvaa siirtymää syötäKortti. Samasta

syystä automaatti ei kykene tapahtumaketjun <esitäPäänäyttö, syötäKortti, korttiUlos> suorittamiseen.

OCL-määritelmää tarkastelemalla voidaan havaita, että LTS-kaavio ylimäärittelee automaatin toimintaa. Kuvassa 6.8 esitetyssä taulukossa on esitetty OCL-tilamuuttujien arvot tilakaavion kussakin tilassa. Taulukon arvot t, f, s, k ja – esittävät arvoja true (tosi), false (epätosi), salasana, kortti ja null (tyhjä). Sanoman syötäKortti esiehtona on tilamuuttujan korttiSisässä arvon oleminen epätosi, josta voidaan päätellä, että tilassa 2 ei pitäisikään voida käsitellä sanomaa syötäKortti. Toisaalta sanoman korttiUlos esiehtona on tilamuuttujan korttiSisässä oleminen tosi. Tämä ehto on voimassa LTS-kaavion tilassa 2. OCL-määritelmän mukaan ei siis ole mitään syytä estää tapahtumaketjun <esitäPäänäkymä, syötäKortti, korttiUlos> suorittamista automaatissa.

	0	1	2	3	4	5	6	7	8	9	10	11
korttiSisässä	f	f	t	t	t	t	f	f	t	t	t	t
korttiOtettavissa	f	f	f	f	f	f	t	t	f	f	f	f
salasanaAnnettu	t	t	f	f	t/f	t	f	f	t	t	t	t
kortti	-	-	c	c	c	c	-	-	c	c	c	c
salasana	-	-	-	-	p/-	p	-	-	p	p	p	p

Kuva 6.8. Tilamuuttujien arvot LTS-kaaviossa.

Käytössä olevat tiedot näistä kahdesta tapahtumaketjusta ovat siis erilaiset. Tapahtumaketjua <esitäPäänäkymä, syötäKortti, syötäKortti> ei pitäisikään pystyä suorittamaan, sillä se rikkoo sanomille asetettuja esiehtoja. Tapahtumaketjun <esitäPäänäkymä, syötäKortti, korttiUlos> suorittaminen ei sen sijaan riko mitään ehtoa ja voisikin periaatteessa olla automaatilta täysin sallittua käyttäytymistä. Voikin olla, ettei kyseistä tapahtumaketjun mahdollisuutta ole lainkaan ajateltu käyttäytymismäärittelyä tehdessä. Kyseessä on kuitenkin selkeästi määrittelemätön kohta järjestelmän toiminnallisuudessa, jonka selvittämisellä järjestelmästä olevan tiedon määrää on mahdollista kasvattaa ja määrittelyä mahdollista tarkentaa.

Kun kuvan 6.3 tilamuuttujien esiehtoja verrataan kuvan 6.8 tilamuuttujien tilakohtaisiin arvotuksiin, voidaan helposti päätellä sellaiset sanomat, joita tiloissa saa tapahtua. Esimerkiksi sanoman syötäKortti esiehdon mukaan sanoman ei pitäisi voida tapahtua tiloissa 2-5 ja 8-11.

Tilakaaviosta saadaan puolestaan kerättyä tieto tiloissa sallituista sanomista (tilasiirtymistä). Kun nämä tiedot yhdistetään, saadaan tulokseksi kuvan 6.9 taulukko, jossa sallitut tilasiirtymät on merkitty kirjaimella ”s”, kielletyt tilasiirtymät kirjaimella ”k” ja jäljelle jääneet taulukon tyhjät alkiot varustetaan kysymysmerkillä ”?”.

	0	1	2	3	4	5	6	7	8	9	10	11
syötäKortti	?	s	k	k	k	k	?	?	k	k	k	k
syötäSalasana	?	?	?	s	k	k	k	k	k	k	k	k
otaKortti	k	k	k	k	k	k	?	s	k	k	k	k
esitäPäänäkymä	s	?	k	k	k	k	k	k	k	k	k	k
pyydäSalasana	?	?	s	?	k	k	k	k	k	k	k	k
korttiUlos	k	k	?	?	?	s	k	k	?	?	?	?
peruutettuViesti	k	k	?	?	s	?	k	k	?	?	?	?
pyydäOtaKortti	k	k	k	k	k	k	s	?	k	k	k	k

Kuva 6.9. Automaatin tilojen vertailu.

Kysymysmerkein varustetut alkiot kuvaavat järjestelmään jäänyttä määrittelemätöntä käyttäytymistä, eli sellaisia tila-tilasiirtymä pareja, joissa sanoma olisi esiehtojensa mukaan mahdollinen, mutta vastaavaa tilasiirtymää ei ole määritelty tilasta lähteväksi. Tapahtuma `syötäKortti` voisi siis esiehtojensa mukaan tapahtua myös tiloissa 0, 6 ja 7, mutta järjestelmän tila tapahtuman jälkeen ei ole toistaiseksi selvillä. Kuvan 6.9 taulukon perusteella suunnittelija voisi suorittaa lisäselvityksia täyttääkseen järjestelmän käyttäytymisen määrittelyyn jääneet aukot. Tämän esimerkin mukaan voitaisiin siis selvittää, onko asiakkaan mahdollista aloittaa uusi istunto automaatilla työntämällä pankkikortti uudelleen automaattiin ottamatta sitä ensin kokonaan pois.

6.5.2. Määrittelemättömän käyttäytymisen tunnistaminen

LTSA-työkalua voidaan käyttää apuna käyttäytymismalleissa esiintyvien määrittelyssä olevien aukkojen löytämisessä. Määrittelemättömän käyttäytymisen tunnistaminen FSP-määritelmiin perustuvista LTS-tilakoneista perustuu LTSA työkalun FLTL (Fluent Linear Temporal Logic) -mallien tarkistusominaisuuteen [Magee and Kramer, 2006]. FLTL on lineaarinen aikalogiikka, jonka avulla voidaan tutkia mallissa tapahtuvien toimintojen vaikutusta mallin tilaan.

Tässä FLTL-määritelmiä käytetään kuvan 6.3 OCL-määritelmän tilamuuttujien arvojen kuvaamiseen. Esimerkiksi tilamuuttuja `korttiSisässä` voidaan määritellä seuraavasti:

```
fluent KORTTISISÄSSÄ=<syötäKortti,{korttiUlos, otaKortti}>
```

Määritelmän mukaan `KORTTISISÄSSÄ`-tilamuuttuja saa arvon tosi toiminnon `syötäKortti` seurauksena. Tilamuuttujan arvoksi tulee epätosi toiminnon `korttiUlos` tai toiminnon `otaKortti` tapahtumisen myötä, josta arvon voi jälleen muuttaa todeksi toiminnon `syötäKortti` suorittaminen.

Tilamuuttujien FLTL-määritelmät, kuten KORTTISISÄSSÄ, voidaan johtaa suoraan OCL-määritelmän sisältämistä sanomien jälkiehtojen kuvauksista. Esimerkiksi tilamuuttujan korttiSisässä arvo esiintyy kolmen kuvassa 6.3 esitetyn sanoman jälkiehdoissa: syötäKortti, korttiUlos ja otaKortti, jotka toimivat FLTL-määritelmän syötteenä.

Kun tilamuuttujat on määritelty FLTL:n avulla, voidaan tilakaaviosta löytää sellaiset tilat, joista ei lähde esiehdot täyttäviä sanomia (tilasiirtymiä). Esimerkiksi pankkiautomaattijärjestelmästä voidaan automaattisesti selvittää selvittää sellaiset järjestelmän tilat, joista sanoman esiehtojen täyttymisestä huolimatta ei lähde tilasiirtymää syötäKortti, esittämällä seuraava FLTL-kaava järjestelmän FSP-määrittelyn yhteydessä:

```
assert MÄÄRITTELEMÄTÖN_SYÖTÄKORTTI =
    [ ] ( !KORTTISISÄSSÄ -> X syötäKortti ).
```

Edellä esitetyn kaavan avulla LTSA esittää sellaiseen tilaan johtavan toimintoketjun jäljen, jossa tilamuuttuja KORTTISISÄSSÄ on saanut arvon epätosi, mutta josta ei voida jatkaa toiminnolla syötäKortti. LTSA tulostaa tarkistuksen tuloksen seuraavassa muodossa:

```
Trace to property violation in MÄÄRITTELEMÄTÖN_SYÖTÄKORTTI
        esitäPäänäkymä
        syötäKortti         KORTTISISÄSSÄ
        pyydäSalasana       KORTTISISÄSSÄ
        peruuta             KORTTISISÄSSÄ
        peruutettuViesti    KORTTISISÄSSÄ
        korttiUlos
        pyydäOtaKortti
    Analysed in: 16ms.
```

Vasen sarake on edellä mainittuun tilaan johtavan toimintoketjun jälki. FLTL-vaatimuksen rikkomus tapahtuu välittömästi tilamuuttujan KORTTSISÄSSÄ arvon epätodeksi muuttavan toiminnon korttiUlos jälkeen suoritettavan toiminnon yhteydessä, jos se ei ole syötäKortti. Toimintojen oikealla puolella on teksti KORTTISISÄSSÄ, kun ollaan tilassa, jossa tilamuuttujan arvo on tosi. Siitä voidaan nähdä tilamuuttujan olevan epätosi toiminnon syötäKortti tapahtumiseen saakka. Tämän jälkeen tilamuuttujan arvo on tosi, kunnes se toiminnon korttiUlos tapahtumisen myötä saa jälleen arvon epätosi [Uchitel *et al.*, 2003b].

7. Käyttäytymismallit ja OMT++-ohjelmistokehitysmenetelmä

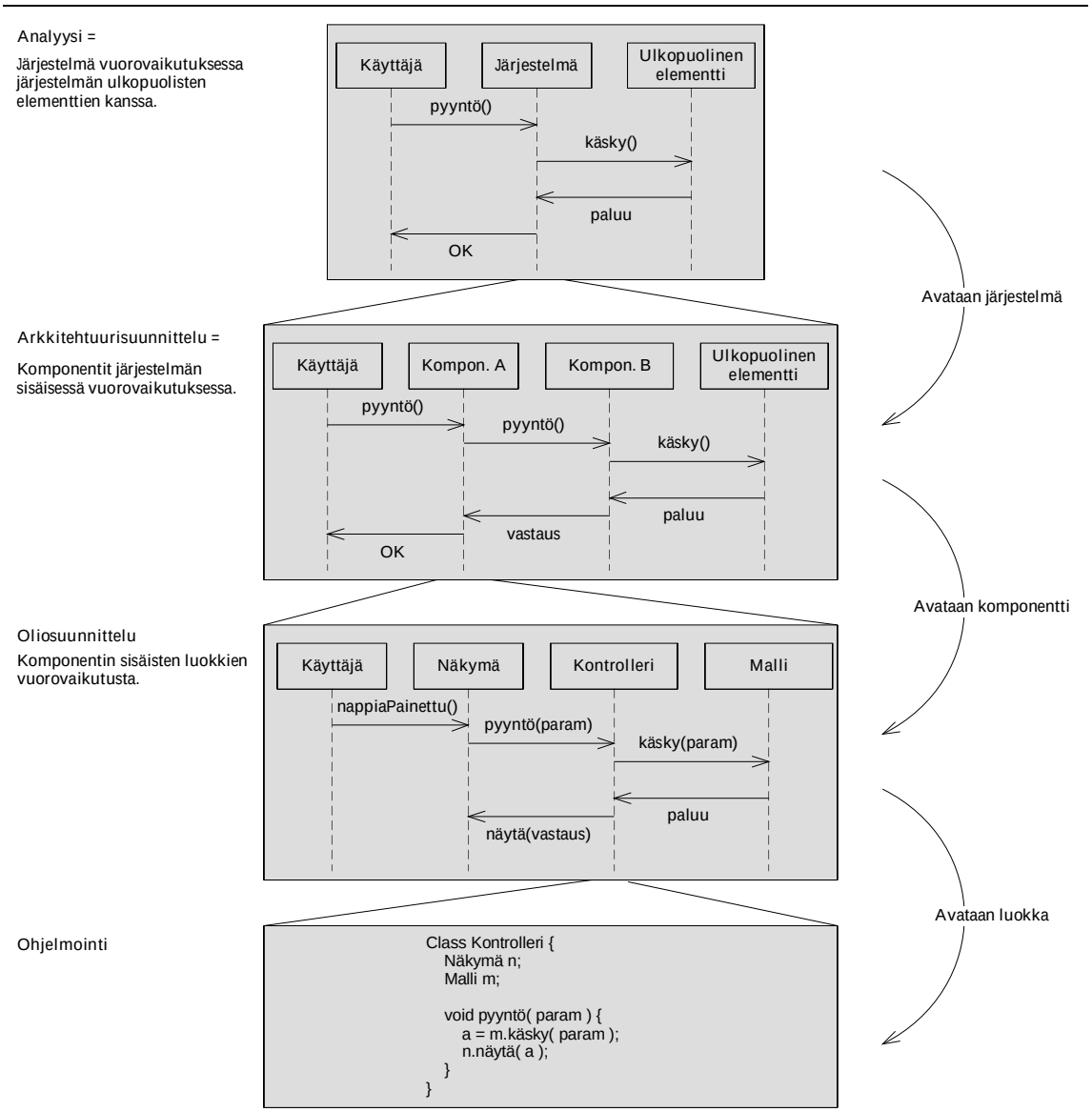
Edellisessä luvussa esitettiin järjestelmän käyttäytymisen analysointiin tarkoitettujen käyttäytymismallien avulla saatavia kiistattomia hyötyjä. Esimerkkeinä toimivat sekä järjestelmän määrittelemättömän käyttäytymisen tunnistaminen että mallinnetusta järjestelmästä löytynyt lukkiutumistilanne, jonka löytäminen perinteisten katselmointien yhteydessä olisi hyvin vaikeaa, ellei jopa mahdotonta.

7.1. Käyttäytymismallin laatiminen OMT++-menetelmässä

Jaaksin ja kumppanien varsin kriittinen suhtautuminen tilakaavioihin ja tilakoneisiin olisi kuitenkin ymmärrettävää, jos niiden laatiminen veisi kauan aikaa tai vaatisi useiden vaikeiden notaatioiden ja kalliiden ohjelmistotyökalujen käyttöönottoa. FSP-notaatio itsessään ei kuitenkaan ole kovin vaikea semantiikaltaan tai syntaksiltaan, eikä sen yksinkertaisimpien, mutta käyttäytymismallien laatimiseen riittävien ominaisuuksien opettelu vie pitkää aikaa. Kaiken lisäksi käyttäytymismallin tulkintaan ja analysointiin tarkoitettu LTSA-työkalu on ominaisuuksiinsa nähden kevyt ja helppokäyttöinen ohjelmisto.

FSP-notaatiolla kuvattujen käyttäytymismallien laatiminen OMT++-ohjelmistokehitysmenetelmällä tehdyn suunnittelun tueksi on itse asiassa varsin vaivatonta. Käyttäytymismallin laatimiseksi kohdassa 6.4 esitetyn menetelmän avulla tarvitaan ainoastaan kattava kuvaus mallinnettavan järjestelmän toiminnallisuudesta sekvenssikaavioilla esitettynä. Ja kuten luvussa viisi selkeästi esitettiin, ovat sekvenssikaaviot luokkakaavioiden lisäksi toinen pääasiallinen OMT++-menetelmässä käytetty kaaviotyyppi. Menetelmä itsessään siis tuottaa raaka-aineet käyttäytymismallin laatimiseen.

OMT++-menetelmässä sekvenssikaavioita käytetään systemaattisesti. Analyysivaiheessa niillä kuvataan, miten järjestelmä kommunikoi ulkoisten elementtien kanssa. Arkkitehtuurisuunnittelussa sekvenssikaavioita käytetään komponenttien yhteistoiminnan kuvaamiseen. Komponenttisuunnittelussa niillä kuvataan komponentin sisältämien luokkien välistä yhteistoimintaa. Sekvenssikaavioita voidaan vielä käyttää apuna luokkien ohjelmoinnissa. Suunnittelun edetessä analyysivaiheessa mustana laatikkona esitetty järjestelmä avataan aina oliotasolle asti (kuva 7.1). Järjestelmässä tapahtuvat operaatiot tarkentuvat askel askeleelta analyysivaiheesta aina ohjelmointitasolle asti.



Kuva 7.1. Sekvenssikaavioiden käyttö funktionaalisella polulla.

Käyttäytymismalleja voidaan OMT++-menetelmän yhteydessä rakentaa usealla eri abstraktiotasolla aina analyysivaiheesta komponentin sisäiseen oliosuunnitteluun asti. Samanaikaisuuden kannalta keskeisimmät vaiheet ovat analyysivaihe ja arkkitehtuurisuunnittelu, joskin komponenttisuunnittelunkin yhteydessä voidaan joutua suunnittelemaan aktiivisia olioita, jolloin samanaikaisuutta voidaan perustellusti mallintaa myös siinä vaiheessa.

OMT++ on ehkä tahtomattaankin onnistunut luomaan hyvän pohjan samanaikaisuuden mallintamiselle. Siinä painotetaan jokaisen sekvenssikaavioita tuottavan vaiheen yhteydessä kaavioiden laatimista juuri samanaikaisuuden kannalta olennaisimmista operaatioista, vaikka muuten kaavioiden lukumäärä pyritään pitämään mahdollisimman pienenä. Analyysivaiheen käyttäytymisanalyysin kohdalla sekvenssikaavion sisältävät operaatiomäärittelyt laaditaan ainakin niistä järjestelmän operaatioista, jotka

sisältävät kommunikaatiota järjestelmän ulkopuolisten osien kanssa. Suunnitteluvaiheessa komponenttien yhteistoimintamäärittelyn skenaarionäkymässä esitetään sekvenssikaaviot ainakin niistä järjestelmän operaatioista, joiden suorittaminen vaatii useampien komponenttien välistä yhteistoimintaa. Komponentin sisäisen suunnittelun yhteydessä suoritettavan käyttäytymissuunnittelun yhteydessä laaditaan jälleen sekvenssikaaviot, tällä kertaa niistä operaatioista, joiden toiminta vaatii kyseisen komponentin sisältämien luokkien yhteistoimintaa.

Kaikissa OMT++-menetelmän vaiheissa laadittavissa skenaarioissa on sekvenssikaavion lisäksi aina määritelty skenaarioon liittyvät esi- ja jälkiehdot, joita voidaan hyödyntää käyttäytymismallin rakentamista helpottavan korkean tason tapahtumasekvenssikaavion laatimiseksi. Skenaarion esiehdoissa voidaan esimerkiksi määritellä muita skenaarioita, joiden täytyy olla suoritettuna ennen kyseisen skenaarion suorittamista. Skenaarioiden suoritusjärjestyksen määrittelemiseksi saatetaan joutua myös käyttötapausten tarkempaan tutkimiseen. Joka tapauksessa kaiken käyttäytymismallin rakentamiseen tarvittavan tiedon pitäisi löytyä OMT++-menetelmän mukaisesti suunnitellun järjestelmän suunnitteludokumentaatiosta.

7.2. Analyysivaiheen käyttäytymismallin hyödyntäminen

Analyysivaiheen käyttäytymismallia voidaan hyödyntää kahdella pääasiallisella tavalla. Sen avulla voidaan etsiä edellisen luvun loppupuolella esitetyn menetelmän mukaisesti järjestelmän toiminnallisuuden määrittelyyn jääneitä aukkoja. Järjestelmän määrittelemättömän käyttäytymisen analysointi antaa mahdollisuuden järjestelmän vaatimusten ja operaatiomäärittelyiden tarkentamiseen. Esitellyssä menetelmässä määrittelemättömän käyttäytymisen etsimiseen käytetään käyttäytymismallin lisäksi OCL-määrittelyä, jossa määritellään järjestelmässä lähetettäviin sanomiin liityvät esi- ja jälkiehdot. Tällaisia tietoja ei OMT++-menetelmässä erikseen kerätä, mutta vastaavien tietojen määrittely hyvin dokumentoitujen järjestelmävaatimusten ja operaatiomäärittelyiden perusteella ei pitäisi tuottaa suuria ongelmia.

Toinen tapa hyödyntää analyysivaiheen käyttäytymismallia on käyttää sitä järjestelmän analyysivaiheen suunnittelun oikeellisuuden varmentamiseen suorittamalla käyttäytymismallia asiakkaan edustajien läsnäollessa. Tämänkaltaisen esittelyn yhteydessä saattaa tulla ilmi suuriakin väärinkäsityksiä kehitettävän ohjelmiston toiminnallisuudessa. Mitä aikaisemmin väärinkäsitykset selvitetään, sen halvemmiksi niiden aiheuttamien ongelmien ratkaisu tulee.

7.3. Arkkitehtuurisuunnittelun käyttäytymismallin hyödyntäminen

Ohjelmistokehityksen arkkitehtuurisuunnittelussa tehtyjen ratkaisuiden oikeellisuutta on erittäin vaikea, ellei jopa mahdotonta varmistaa. FSP:n ja LTSA:n avulla on kuitenkin mahdollista ja jopa helppoa tarkistaa, toteuttaako suunniteltu järjestelmä sille asetetut samanaikaisuutta koskevat vaatimukset. Kun prosessit kuvataan FSP:n avulla, voidaan LTSA-työkalua käyttämällä todeta prosessien turvallisuus- ja elävyysominaisuudet.

Arkkitehtuurisuunnittelun käyttäytymismallin analysoinnin myötä on mahdollista paljastaa järjestelmässä piilevät synkronointiongelmat ja lukkiutumistilanteet, jotka voidaan korjata jo suunnittelutasolla ennen kuin komponenttien toteutusta on vielä ehditty aloittamaan. Mallin avulla on mahdollista tutkia myös komponenttien välillä kulkevien sanomien ja tiedon määrää, jonka selvittäminen on välttämätöntä järjestelmän suorituskykyä ja skaalautuvuutta analysoidessa sekä järjestelmän lopullista prosessijakoa määriteltäessä.

8. Yhteenveto

Tutkielman tavoitteena oli selvittää, miten samanaikaisuuden mallintaminen voitaisiin ottaa osaksi yleiskäyttöistä ohjelmistokehitysmenetelmää. Tavoitteen täyttämiseksi järjestelmien mallintamiseen perehdyttiin sekä yleisesti käytössä olevien perinteisten suunnittelutekniikoiden että erityisesti samanaikaisuuden mallintamisen näkökulmista. Esimerkiksi tällaisesta menetelmästä tarkastelun kohteeksi valittiin OMT++, jota on menestyksekkäästi käytetty suurten reaaliaikaisten ohjelmistojen kehittämisessä. Samanaikaisuuden mallintamiseen tässä tutkielmassa käytettiin FSP-notaatiota.

Yleisimpiä ohjelmistojen suunnittelussa käytössä olevia tekniikoita tarkasteltiin erikseen ja erityisesti esitettiin tapahtumasekvenssikaavioiden ja tilakaavioiden välinen yhteys. Notaatioiden käyttöön osana laajempaa ohjelmistokehitysmenetelmää perehdyttiin käymällä läpi OMT++-menetelmä ja sen avulla kehitettävästä järjestelmästä tuotettavat suunnitteluspesifikaatiot yksityiskohtaisesti.

Tutkielmassa tarkasteltiin samanaikaisuuden mallintamiseen käytetyn FSP-notaation ominaisuuksia ja sen osoitettiin olevan yleisten samanaikaisuuden mallintamisperiaatteiden mukainen. Samanaikaisuuden erityisongelmien suhteen notaation osoitettiin ottavan huomioon synkronoinnin, poissulkemisen ja lukkiutumistilanteet. Ajastukseen liittyviä ongelmia notaation avulla ei kuitenkaan voida selvittää.

Ohjelmistokehitysmenetelmien ja samanaikaisuuden mallintamisen yhteenliittämiseksi tutkielmassa tutustuttiin käyttäytymismalleihin, erityisesti niiden laatimiseen järjestelmän toiminnallisuutta kuvaavista skenaariospesifikaatioista erilaisten skenaariosynteesimenetelmien avulla. Tässä yhteydessä esitettiin yksinkertainen menetelmä käyttäytymismallin laatimiselle FSP-notaatiolla useiden tapahtumasekvenssikaavioiden pohjalta. Käyttäymismallien hyödyllisyys osoitettiin pankkiautomaattijärjestelmän lukkiutumistilanteen ja määrittelemättömän käyttäytymisen avulla ja OMT++-ohjelmistokehitysmenetelmän todettiin tarjoavan sellaisenaan varsin hyvät lähtökohdat eri abstraktiotasojen käyttäytymismallien rakentamiselle sen vaihetuotteiden sisältämien tietojen pohjalta.

Tutkielmassa esitettyjen esimerkkien valossa voidaan todeta nykyisin käytössä olevien samanaikaisuuden mallintamiseen tarkoitettujen menetelmien ja työkalujen hyödyllisyys. Vain aika voi näyttää, johtavatko ne samanaikaisuuden mallintamisen yleistymiseen ohjelmistoteollisuuden käytössä olevien ohjelmistokehitysmenetelmien yhteydessä.

Viiteluettelo

[Aalto and Jaaksi, 1994] Juha-Markus Aalto and Ari Jaaksi, Object oriented development of interactive systems with OMT++. *TOOLS 14, Technology of Object-Oriented Languages & Systems*, edited by Ege R., Singh M., Meyer B., 205-218. Prentice-Hall, 1994.

[Atk-sanakirja, 2003] *Atk-sanakirja*. Talentum, Helsinki, 2003.

[Biermann and Krishnaswamy, 1976] A.W. Biermann and R. Krishnaswamy, Constructing programs from example computations. *IEEE Trans. Software Eng.,* 2, 3 (May 1976), 141-153.

[Booch, 1986] G. Booch, Object oriented development. *IEEE Trans. Software Eng.,* 12, 2 (Feb. 2003), 211-221.

[Booch *et al.*, 1998] G. Booch, J. Rumbaugh, and I. Jacobson, *Unified Modeling Language User Guide*. Addison-Wesley, 1998.

[Coleman *et al.*, 1994] D. Coleman, P. Arnold, S. Bodoff, C. Dollin, H. Gilchrist, F. Hayes, and P. Jeremaes, *Object-Oriented Development. The Fusion Method*. Prentice-Hall, 1994.

[Eriksson and Penker, 1998] H.-E. Eriksson and M. Penker, *UML Toolkit*. John Wiley & Sons, 1998.

[Everett, 1995] W.W. Everett, Reliability and safety of real-time systems. *Computer* 12, 3 (May 1995), 13-16.

[Haikala ja Järvinen, 1994] Ilkka Haikala ja Hannu-Matti Järvinen, *Käyttöjärjestelmät*. Modeemi ry., 1994.

[Haikala ja Märijärvi, 2000] Ilkka Haikala ja Jukka Märijärvi, *Ohjelmistotuotanto*. Satku, Helsinki, toinen painos, 2000.

[Harel, 1987] David Harel, Statecharts: a visual formalism for complex systems. *Science of Computer Programming* 8, 3 (June, 1987), 231-274.

[Hoare, 1985] C.A.R. Hoare, *Communicating Sequential Processes*. Prentice-Hall, 1985.

[ITU Z.120, 1996] *Message Sequence Charts.* Recommendation X.120, Int l Telecomm. Union., Telecomm. Standardization Sector, 1996.

[Jaaksi, 1997] Ari Jaaksi, *Object-Oriented Development of Interactive Software*. Tampere University of Technology, Ph.D Thesis, 1997.

[Jaaksi, 1998] Ari Jaaksi, Our cases with use cases. *JOOP 10,* 9 (Feb. 1998), 58-65.

[Jaaksi *et al*., 1999] Ari Jaaksi, Juha-Markus Aalto, Ari Aalto, and Kimmo Vättö, *Tried & True Object Development – Industry-Proven Approaches with UML*. Cambridge University Press, SIGS Books, Cambridge, 1999.

[Jacobson *et al.*, 1992] I. Jacobson, M. Christerson, P. Jonsson, and G. Övergaard, *Object-Oriented Software Engineering: A Use Case Driven Approach*. Addison-Wesley, 1992.

[Koskimies *et al.*, 1998] Kai Koskimies, Tatu Männistö, Tarja Systä, and Jyrki Tuomi, Automated support for modeling OO software. *IEEE Software,* 15, 1 (Jan. 1998), 87-94.

[Kurki-Suonio, 1993] Reino Kurki-Suonio, Stepwise design of real-time systems. *IEEE Trans. Software Eng.,* 19, 1 (Jan. 1993), 56-69.

[Kruchten, 1995] P.B. Kruchten, The 4+1 view model of architecture. *IEEE Software,* 12, 6 (Nov. 1995), 42-50.

[Krüger *et al.*, 1998] I. Krüger, R. Grosu, P. Scholz, and M. Broy, From MSCs to Statecharts. In: *Proc. of the IFIP WG10.3/WG10.5 International Workshop on Distributed and Parallel Embedded Systems*, 1998, 61-71.

[Magee and Kramer, 2006] Jeff Magee and Jeff Kramer, *Concurrency – State Models & Java Programming, second edition*. John Wiley & Sons Ltd., 2006.

[Milner, 1989] Robin Milner, *Communication and Concurrency*. Prentice-Hall, 1989.

[Pressman, 1997] Roger S. Pressman, *Software Engineering: A Practitioner's Approach, fourth edition*. McGraw-Hill Book Corporation, 1997.

[Quatrani, 1998] T. Quatrani, *Visual Modeling with Rational Rose 2000 and UML*. Addison-Wesley, 1998.

[Rumbaugh *et al*., 1991] James R. Rumbaugh, Michael R. Blaha, William Lorensen, Frederick Eddy, William Premerlani, *Object-Oriented Modeling and Design*. Prentice-Hall, 1991.

[Uchitel *et al.*, 2003a] Sebastian Uchitel, Jeff Kramer, and Jeff Magee, Synthesis of behavioral models from scenarios. *IEEE Trans. Software Eng.,* 29, 2 (Feb. 2003), 99-115.

[Uchitel *et al.*, 2003b] Sebastian Uchitel, Jeff Kramer, and Jeff Magee, Behaviour model elaboration using partial labelled transition systems. In: *Proc. of the 9th European Software Engineering Conference*, 2003, 19-27.

[UML, 2007] Unified Modeling Language version 2.1.2 at http://www.omg.org/spec/UML/2.1.2/

[Ward and Mellor, 1985] P.T. Ward and S.J. Mellor, *Structured Development for Real-Time Systems* – Vol. 1-3. Prentice-Hall, 1985.

[Whittle and Schumann, 2000] J. Whittle and J. Schumann, Generating statechart designs from scenarios. In: *Proc. of the 22nd International Conference on Software Engineering (ICSE '00)*, 2000, 314-323.

Liite 1

Esimerkki käyttötapauksesta

Käyttötapaus:	Tekstiviestin lähetys
Suorittaja:	Tavallinen matkapuhelimen käyttäjä
Käytettävyysvaatimukset:	Käyttäjän pitää pystyä huomaamaan, että viesti on onnistuneesti lähetetty.
Esiehdot:	Käyttäjällä on järjestelmän käyttöoikeudet. Järjestelmään on tallennettu vastaanottajia, ryhmiä ja usein käytettyjä sanontoja.
Kuvaus:	Käyttäjä kirjoittaa tekstiviestin [poikkeus: viestin lataaminen] ja lisää viestin loppuun oman allekirjoituksensa. Käyttäjä valitsee viestin vastaanottajiksi kaksi yksittäistä vastaanottajaa ja kaksi ryhmää [poikkeus: numeroita ei ole saatavilla]. Tämän jälkeen hän tallentaa viestin arkistointia varten. Sitten hän lähettää tekstiviestin valitsemilleen vastaanottajille. Lopuksi sovellus ilmoittaa, että viestin lähetys verkkoon on onnistunut.
Poikkeukset:	Viestin lataaminen: Käyttäjä voi ladata aiemmin tallentamansa viestin jatkokäsittelyä varten. Numeroita ei ole saatavilla: Käyttäjän täytyy ensin syöttää vastaanottajien tiedot järjestelmään.
Jälkiehdot:	Tekstiviesti on lähetetty ja tallennettu järjestelmään.

Liite 2

Pankkiautomaattijärjestelmän FSP-määrittely

```
AUTOMAATTI = ( esitäPäänäkymä -> AUTOMAATTI2 ),
AUTOMAATTI2 = ( syötäKortti -> S_pyydäSalasana ),
S_pyydäSalasana = ( pyydäSalasana -> ( syötäSalasana -> S_varmennaTili
                                     | peruuta -> S_peruutettuViesti ) ),
S_varmennaTili = ( varmennaTili -> ( kelvotonTili -> S_kelvotonTiliViesti
                                   | vääräSalasana -> S_pyydäSalasana
                                   | peruuta -> S_peruutettuViesti ) ),
S_kelvotonTiliViesti = ( kelvotonTiliViesti -> S_tulostaKuitti ),
S_tulostaKuitti = ( tulostaKuitti -> S_korttiUlos ),
S_korttiUlos = ( korttiUlos -> S_pyydäOtaKortti ),
S_pyydäOtaKortti = ( pyydäOtaKortti -> otaKortti -> AUTOMAATTI ),
S_peruutettuViesti = ( peruutettuViesti -> S_korttiUlos ).

KÄYTTÄJÄ = ( esitäPäänäkymä -> S_syötäKortti ),
S_syötäKortti = ( pyydäSalasana -> S_syötäSalasana ),
S_syötäSalasana = ( syötäSalasana -> S_peruuta ),
S_peruuta = ( pyydäSalasana -> S_syötäSalasana
            | kelvotonTiliViesti -> tulostaKuitti -> korttiUlos -> pyydäOtaKortti ->
                    S_otaKortti
            | peruuta -> peruutettuViesti -> korttiUlos -> pyydäOtaKortti ->
                    S_otaKortti),
S_otaKortti = ( otaKortti -> KÄYTTÄJÄ ).

KONSORTIO = ( varmennaTili -> S_varmennaTiedot ),
S_varmennaTiedot = ( varmennaTiedot -> ( kelvotonPankkiTili -> S_kelvotonTili
                                       | vääräPankkiSalasana -> S_vääräSalasana ) ),
S_kelvotonTili = ( kelvotonTili -> KONSORTIO ),
S_vääräSalasana = ( vääräSalasana -> KONSORTIO ).

PANKKI = ( varmennaTiedot -> ( kelvotonPankkiTili -> PANKKI
                             | vääräPankkiSalasana -> PANKKI ) ).

|| JÄRJESTELMÄ = ( AUTOMAATTI || KÄYTTÄJÄ || KONSORTIO || PANKKI ).
```