UML työvälineenä ja tutkimuskohteena

Kai Koskimies, Johannes Koskinen, Mika Maunumaa,
Jari Peltonen, Petri Selonen, Mika Siikarla & Tarja Systä
Tampereen teknillinen yliopisto
Ohjelmistotekniikan laitos
PL 553
33101 Tampere

Tiivistelmä

UML (Unified Modeling Language) on nopeasti yleistynyt graafinen ohjelmistojen mallinnuskieli. UML:ää käytetään sekä ihmisten väliseen kommunikointiin että ihmisen ja koneen väliseen vuorovaikutukseen. Monet työkalut tukevat UML:n käyttöä mallien laatimisessa, koodin tuottamisessa malleista, sekä mallien tuottamisessa koodista. UML:ään liittyvä tutkimus kohdistuu mm. UML:n semantiikan määrittelyyn, mallien transformaatioihin ja mallien analysointiin eri tarkoituksia varten.

1 Johdanto

Ohjelmistojen kehittämisessä käytetään yleisesti erilaisia malleja järjestelmien ja niihin liittyvien ongelma-alueiden kuvaamiseen. Mallit mahdollistavat kohteen yksinkertaistamisen rajoittamalla tarkastelun tiettyyn näkökulmaan, epäolennaisia yksityiskohtia poistamalla ja abstrahoimalla reaalimaailmaa. Malleja voidaan käyttää ongelman ymmärtämisen lisäksi spesifikaatioiden ja dokumenttien laatimiseen, ja ne toimivat ohjelmiston kehittämisen yhteydessä ihmisten välisen kommunikoinnin apuvälineenä.

Mallien esittämiseen käytetään erilaisia kuvaustapoja. Kun tällainen kuvaustapa on täsmällisesti määritelty, voidaan puhua *mallinnuskielestä* (modeling language). Mallinnuskielien esitysmuotona voidaan käyttää esimerkiksi tekstiä, taulukoita tai graafisia kaavioita. Mallinnuskieli voidaan määritellä joko formaalisti tai luonnollisella kielellä, tai käyttäen molempia. Esimerkkejä hyvin tunnetuista ohjelmistotekniikan graafisista mallinnuskielistä ovat ER-kaaviot, tietovuokaaviot, lohkokaaviot ja tilakaaviot.

Mallinnuskielet eivät välttämättä ota kantaa ohjelmistojen rakentamises-

sa käytettävään menetelmään tai prosessiin. *Suunnittelumenetelmät* määrittelevät kuinka mallinnuskieliä sovelletaan jossain tietyssä yhteydessä. Ne kuvaavat usein prosessin, jonka eri vaiheissa tiettyjä mallinnuskieliä käytetään sovituilla, ennalta määrätyillä tavoilla. Menetelmä liittää siis mallinnuskieleen asiayhteydestä riippuvan merkityksen, jolloin samaa mallinnuskieltä voidaan käyttää useissa eri vaiheissa ja jopa useissa eri menetelmissä.

Unified Modeling Language (UML) on Object Management Groupin (OMG) vuonna 1997 standardoima graafinen mallinnuskieli. Standardin [19] mukaan "UML on kieli, joka on tarkoitettu ohjelmistojärjestelmissä esiintyvien artefaktien määrittelyyn, visualisointiin, rakentamiseen ja dokumentointiin, sekä liiketoiminnan ja muiden ohjelmistotekniikan ulkopuolisten järjestelmien mallintamiseen". UML on luonteeltaan enemmänkin standardi kuin innovaatio: se kokoaa yhteen joukon kauan tunnettuja graafisia mallinnuskieliä ja tarkentaa niiden esitysmuodon. UML on siis kokoelma mallinnuskieliä, joista kukin on tarkoitettu käytettäväksi tietyn ohjelmistoihin liittyvän näkökulman mallinnuksessa. Kutsumme UML:ään kuuluvia mallinnuskieliä *kaaviotyypeiksi*.

UML:n luonteen ymmärtäminen edellyttää jossain määrin sen historian tuntemista. UML:n kehitys alkoi vuonna 1994 kun Rational Software Corporationille työskennelleet Grady Booch ja Jim Rumbaugh pyrkivät yhdistämään ja yhtenäistämään aikaisempien menetelmiensä (Booch [2], OMT [27]) ominaisuudet. Vuosi myöhemmin Ivar Jacobson liitti kehittämänsä OOSE-menetelmän edellyttämät mallinnustavat työhön mukaan (erityisesti käyttötapauskaaviot). Vuonna 1996 työstä kiinnostui myös merkittävä määrä eri teollisuusosapuolia, joiden avulla UML:n määrittelyitä selvennettiin ja kielen ilmaisuvoimaa paranneltiin. Sen tuloksena vuonna 1997 OMG:n standardiksi hyväksytty versio 1.1 oli vielä monilta osin raakile, jonka puutteita ja virheitä myöhemmät versiot ovat paikanneet. Monin tavoin kehittyneempi versio 2.0 julkaistiin vuonna 2003 ja se saanee OMG:n standardin aseman vuoden 2004 aikana.

Ohjelmistotuotannon alalla UML on saavuttanut käytännön standardin aseman varsin hyvin. Jokseenkin kaikki varteenotettavat ohjelmistokehitysmenetelmät, niitä tukevat työkalut ja ohjelmistotuotannon oppikirjat käyttävät UML:ää. Yliopistoissa ohjelmistotuotannon opiskelijat joutuvat ainakin pintapuolisesti tutustumaan UML:ään ja sen tuntemista edellytetään yleisesti alan ammattilaisilta. UML:n hyväksymistä ja käyttöönottoa on erityisesti edesauttanut sen joustavuus. Vaikka kaaviotyyppien edellyttämä mallin rakenne määritellään formaalisti, mallien semantiikka on kuvattu luonnollisen kielen avulla. Näin UML jättää paljon tulkinnanvaraa mallien merkitykselle.

UML eroaa useimmista edeltäjistään merkittävästi siten, että se ei ole prosessiin tai menetelmään sidoksissa, eikä siten myöskään tiettyyn organisaatioon, kulttuuriin tai sovellusalueeseen. Tästä seuraavat UML:n merkittävimmät edut: eri menetelmien yhteydessä voidaan hyödyn-

tää samaa, kaikkien ymmärtämää mallintamiskieltä, ja sen työkalutukea. Vaikka tämän kaltaisen rationalisoinnin edut ovat ilmeiset, UML:n kunnianhimoinen tavoite kattaa kaikki mahdolliset tarpeet on johtanut sen ehkä eniten kritisoituihin ominaisuuksiin: laajuuteen ja mutkikkuuteen. Tästä johtuen UML:ää soveltavat yritykset käyttävät tavallisesti vain itselleen tarpeellisia, pienehköjä UML:n osajoukkoja.

Vaikka UML onkin riippumaton yksittäisestä prosessista, sitä suunniteltaessa on ajateltu käyttötapauksiin pohjautuvaa inkrementaalista ohjelmistokehitysprosessia. Tämä ajatus on kiteytynyt UP-prosessimalliksi (Unified Process [9]). UML on myös olioperustainen mallintamiskieli, koska UML:n keskeisin kaaviotyyppi, luokkakaavio, pohjautuu oliolähestymistapaan. Tämä ei kuitenkaan estä käyttämästä UML:ää muidenkin lähestymistapojen yhteydessä käyttämällä luokkakaavioita esimerkiksi sovellusalueen käsitteistön mallintamiseen. Vaikka monet UML:n kaaviotyypit periytyvät oliolähestymistapaa edeltäneistä mallintamisvälineistä, on UML silti parhaimmillaan juuri oliojärjestelmien kehittämisen apuvälineenä.

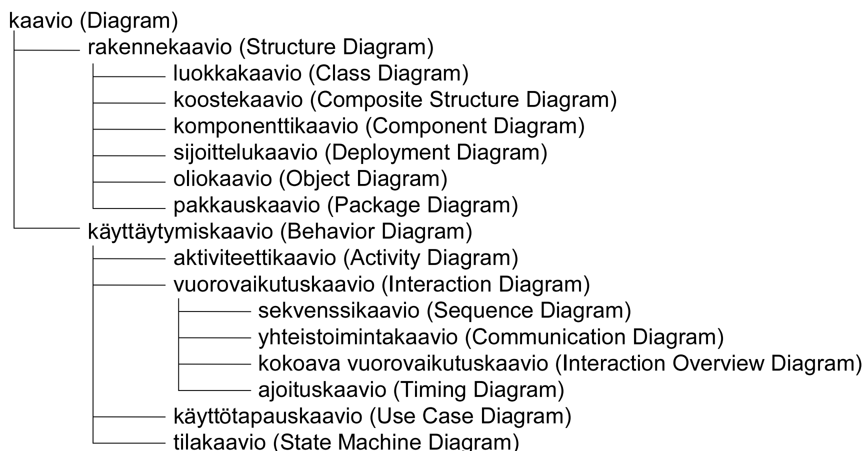
Tässä artikkelissa pyrimme antamaan yleiskuvan UML:sta ja siihen liittyvästä tutkimuksesta. Toivomme, että artikkeli antaa toisaalta lyhyen, helppolukuisen tietoiskun heille, jotka ovat toistaiseksi välttyneet UML:ään tutustumiselta, ja toisaalta virikkeitä esimerkiksi ohjelmistotekniikan väitöskirja-aiheita pohtiville jatko-opiskelijoille.

2 UML pähkinänkuoressa

2.1 UML:n kaaviotyypit

Kaaviotyypin käsite on ollut UML:ssä perinteisesti hieman ongelmallinen. Vaikka UML:n aikaisemmat versiot eivät selkeästi erotelleet eri kaaviotyyppejä, käytännössä UML:n soveltajat ja työkaluvalmistajat ovat kuitenkin olettaneet niiden olemassaolon ja antaneet niille omia tulkintojaan. Kaaviotyypit ovat olleet näin enemmänkin tiettyjä vakiintuneita tapoja käyttää UML:ää kuin täsmällisesti määriteltyjä osakieliä. Tämä on johtanut siihen, että eri lähteissä on jopa erilaisia käsityksiä siitä, mitkä ylipäänsä ovat UML:n kaaviotyypit. Kaikki tämä on haitannut UML:n perimmäistä tarkoitusta, standardointia. Kaaviotyyppeihin liittyvä ongelma juontaa osittain juurensa UML:n hieman idealistiseen ajattelutapaan, jossa kaavio nähdään vain näkymänä koko järjestelmää kuvaavaan loogiseen, abstraktiin malliin. Käytännössä kuitenkin UML-malleja käsitellään kaaviokokoelmina, ja kukin kaavio puolestaan noudattaa kaaviotyyppinsä määrittelemää rakennetta.

UML 2.0 määrittelee 13 eri kaaviotyyppiä, jotka on järjestetty hierarkkisesti kuvan 1 mukaisesti. Kaaviotyypit jaetaan rakennekaavioihin ja käyttäytymiskaavioihin. Rakennekaaviot kuvaavat järjestelmien staattista, ajasta riippumatonta rakennetta eri abstraktiotasoilla, kun taas käyttäytymiskaaviot kuvaavat eri näkökulmista järjestelmän ajoaikaista, ajasta

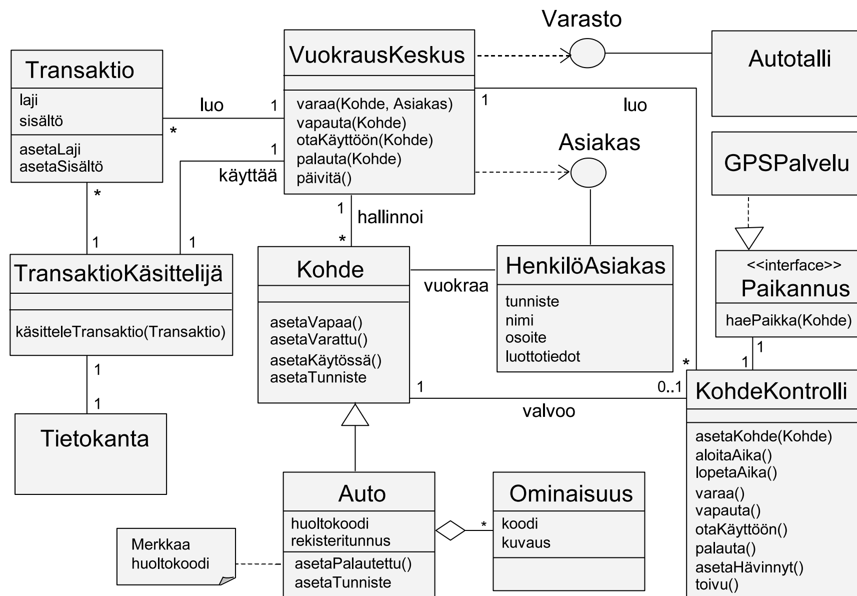


Kuva 1: UML:n kaaviotyypit (UML 2.0)

riippuvaa toimintaa. On kuitenkin muistettava, että tiettyä järjestelmää kuvaavat rakenne- ja käyttäytymiskaaviot ovat monella tavalla toisistaan riippuvia: tietynlainen toiminta edellyttää yleensä tietynlaista rakennetta. Yleisestikin samaa järjestelmää kuvaavilla kaavioilla on paljon keskinäisiä riippuvuuksia, koska ne kuvaavat tiettyä järjestelmää erilaisista, osittain päällekkäisistä näkökulmista.

Kaaviotyypin käsite on — ilmeisen tarkoituksellisesti — jätetty edelleen löyhästi määritellyksi UML 2.0:ssa. Spesifikaatio määrittelee kaaviotyypit vain esitystä helpottavana kaavioiden epätarkkana jaotteluna, ei graafisina osakielinä. Spesifikaatio sallii periaatteessa esimerkiksi rakenne- ja käyttäytymiskaavioiden elementtien käytön samassa kaaviossa, kunhan se vain ei ole ristiriidassa UML:n metamallin kanssa. Palaamme metamallin käsitteeseen myöhemmin.

Usein järjestelmän ajoaikaisen luonteen ymmärtämistä voidaan helpottaa kuvaamalla esimerkkejä ajoaikana vallitsevista tilanteista, pikemmin kuin pyrkimällä kaikkien mahdollisten tilanteiden täydelliseen kuvaamiseen. Ihmisen on aina helpompi ymmärtää konkreettinen esimerkkitilanne kuin asbtrakti täydellinen spesifikaatio; tämä pätee sekä (ajoaikaiseen) rakenteeseen että käyttäytymiseen. Eräät UML:n kaaviotyypit on tarkoitettu nimenomaan tällaisten esimerkkitilanteiden kuvaamiseen: *oliokaaviolla* (object diagram) kuvataan yksittäinen ajoaikainen oliokokoelma, ja *sekvenssikaaviolla* (sequence diagram) kuvataan järjestelmän mahdollinen suorituspolku järjestelmän osien välisenä vuorovaikutuksena. Edellinen kuvaa näin esimerkkirakenteen, jälkimmäinen esimerkkikäyttäytymisen.



Kuva 2: Autonvuokrausjärjestelmän luokkakaavio

2.2 Rakennekaaviot

Ylivoimaisesti tärkein rakennekaavio on järjestelmän staattista rakennetta luokkien tasolla kuvaava *luokkakaavio* (class diagram), joka periytyy aikaisemmin käytetyistä ER-kaavioista. UML:n luokkakaavion esikuvana on ollut hyvin pitkälle Rumbaugh'in OMT-menetelmässä esitelty kaavionotaatio [27]. Luokkakaavio on verkko, jonka solmuina ovat järjestelmän luokat ja särminä luokkien väliset suhteet. Luokkasolmu esitetään suorakaiteena, jonka sisällä on luokan nimi, luokan attribuutit tyyppeineen sekä metodien otsaketiedot. Luokkien väliset suhteet voivat olla yleistyssuhteita, assosiaatiota, osasuhteita, toteutussuhteita tai riippuvuussuhteita.

Vaikka luokkakaavio kertoo periaatteessa järjestelmän luokkien välisistä suhteista, sen avulla voidaan myös nähdä millaiset oliokonfiguraatiot ovat ajoaikana mahdollisia. Tätä varten luokkakaavion suhteisiin voidaan liittää *kertautumismääreitä* (multiplicity), jotka ilmaisevat kuinka monta oliota voi olla ajoaikana kyseisessä suhteessa jonkin toisen olion kanssa. Kertautumismääre kirjoitetaan suhteen päähän, jolloin se ilmaisee tässä päässä olevan luokan ilmentymien (olioiden) mahdollisen lukumäärän. Kertautumismääre annetaan joko suoraan ab-

soluuttisena lukumääränä tai ala- ja ylärajana, jolloin "*" tarkoittaa rajoittamatonta ylärajaa. Esimerkiksi "0..*" (tai pelkkä "*") tarkoittaa siten mielivaltaista lukumäärää. Jos kertautumismääre puuttuu, noudatetaan yleistä UML:n periaatetta, jonka mukaan puuttuva määritys tulkitaan määrittelemättömäksi (eikä oletusarvoiseksi).

Kuvassa 2 on esitetty yksinkertainen autonvuokrausjärjestelmää koskeva luokkakaavio, joka kuvaa järjestelmän perusrakenteen. Kaaviosta voidaan lukea esimerkiksi, että autoilla voi olla mielivaltainen määrä ominaisuuksia, että paikannuspalvelun toteuttaa esimerkiksi GPS, että vuokrattava kohde on tässä järjestelmässä auto, ja että järjestelmä perustuu yhden tietokannan käyttöön transaktioiden kautta. Pallosymbolit tarkoittavat rajapintaluokkia, jollainen voidaan kuvata myös tavallisella luokkasymbolilla varustamalla se stereotyypillä «interface». Palaamme stereotyyppeihin UML:n laajennusmekanismien yhteydessä. Katkonuoli kuvaa yleistä riippuvuusuhdetta, salmiakkisymboli kuvaa osasuhdetta, ontto nuolisymboli kuvaa yleistys- (so. periytymis-) suhdetta, ja katkoviivalla varustettu ontto nuoli kuvaa (rajapinnan) toteutussuhdetta. Assosiaatio kuvataan viivana, jonka päihin voidaan haluttaessa lisätä nuolenpää osoittamaan assosiaation käyttösuuntaa. Huomaa, että jättämällä kertautumismääreet antamatta suunnittelija ei ole halunnut ottaa kantaa siihen, onko Kohde- ja HenkilöAsiakas-olioiden välillä yksi-moneen vai moni-moneen (tai jokin muu) suhde.

UML 2.0 on tuonut mukanaan joitakin laajennoksia luokkakaavioihin. Luokkiin voidaan liittää nk. portteja (port), joiden kautta luokan ilmentymät ovat vuorovaikutuksessa ympäristönsä kanssa. Portit voidaan puolestaan liittää luokan ilmentymän sisältämiin olioihin tai komponentteihin, joille portin kautta tulevat palvelupyynnöt siirretään tai joilta portille luokan ilmentymältä itseltään lähtevät palvelupyynnöt johdetaan. Porttien avulla voidaan näin mallintaa tarkemmin luokan sisältämien osien rooli palvelujen tarjoamisessa ja pyytämisessä. Portit esitetään graafisesti pieninä neliöinä luokkasymbolin sivuilla. Luokkakaavioita, joissa luokkien sisään on piirretty olioita tai komponentteja, kutsutaan UML 2.0:ssa *koostekaavioiksi* (composite structure diagram).

Oliokaavio (object diagram) kuvaa yhden mahdollisen oliokokoelman, joka järjestelmässä voi ajoaikana esiintyä. Oliokaavio on siten luokkakaavion yksi ilmentymä: luokkaa edustaa oliokaaviossa luokan ilmentymä ja luokkien välistä assosiaatiota edustaa oliokaaviossa assosiaation ilmentymä, olioiden välinen linkki. Oliokaavio on muodoltaan yksinkertaisen luokkakaavion näköinen: oliot ovat suorakaiteita ja linkit näiden välisiä särmiä. Se, että suorakaidesymboli edustaakin oliota eikä luokkaa käy ilmi olion otsakkeessa, jossa näkyy sekä olion mahdollinen oma nimi että tämän luokan nimi alleviivattuna. Oliokaaviot ovat hyödyllisiä, kun halutaan konkreettisesti havainnollistaa järjestelmän oliokonfiguraatioita tyypillisissä tapauksissa purkamalla auki luokkakaavion kertautumismääreet.

Komponenttikaavio (component diagram) kuvaa komponentit ja niiden väliset suhteet. Tässä komponentilla tarkoitetaan hyvin määritellyn rajapinnan toteuttavaa itsenäistä ohjelmistoyksikköä, joka voidaan haluttaessa korvata toisella. Komponenttikaavio koostuu komponenteista ja rajapinnoista sekä näiden välisistä toteutus- ja käyttösuhteista. Komponenttikaavio on tyypillisesti arkkitehtuuritason kuvaus, kun taas luokkakaaviolla kuvataan usein yksityiskohtaisen suunnittelun tulosta. UML 2.0 toi mukanaan myös pienen mutta näppärän visuaalisen lisäyksen komponenttikaavioihin: komponentin tarvitsema rajapinta kuvataan kuppisymbolilla, johon tarjotun rajapinnan pallosymboli intuitiivisesti sopii.

Kuvassa 3 on esitetty yksinkertainen esimerkki komponenttikaaviosta, missä osa kuvan 2 luokkakaaviosta on esitetty komponenttimuodossa. Komponentti esitetään tässä luokkasymbolilla, jonka yläkulmassa on pieni komponentti-ikoni. Esimerkissä on käytetty myös porttisymboleja (pienet neliöt) rajapintojen ja komponenttien liitoskohdissa.

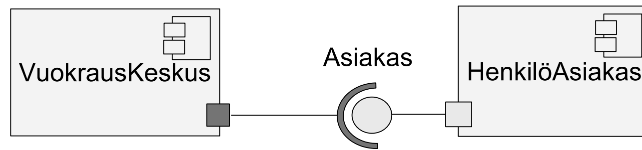
Pakkauskaavio (package diagram) kuvaa järjestelmän pakkausten (tyypillisesti alijärjestelmien) väliset riippuvuussuhteet. *Sijoittelukaavio* (deployment diagram) kuvaa järjestelmän laitearkkitehtuurin ja erilaisten ohjelmistoartefaktien sijoittumisen laitteistoon sekä laitteiden väliset kommunikointiväylät. Tyypillisesti sijoittelukaavio sitoo ohjelmistoarkkitehtuurin verkkoarkkitehtuuriin. Pakkauskaavio ja sijoittelukaavio ovat selvästi arkkitehtuuritason kuvauksia, joissa järjestelmää tarkastellaan korkealla abstraktiotasolla.

2.3 Käyttäytymiskaaviot

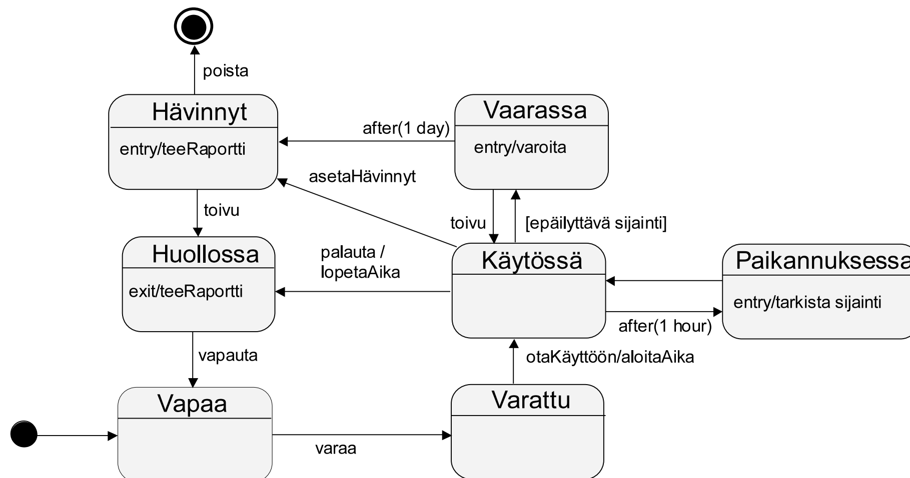
Käyttäytymiskaaviot soveltavat vanhoja, hyvin tunnettuja ohjelmistojen toiminnallisuuden graafisia kuvaustapoja. *Tilakaaviot* (state machine diagram) ovat äärellisen tila-automaatin variaatio, johon on vaikuttanut voimakkaasti David Harelin [6] esittämä rakenteinen, sisäkkäisiä ja rinnakkaisia tiloja salliva tila-automaatti (statechart). Tilakaavio koostuu tiloista ja niiden välisistä siirtymistä; lisäksi tilakaavioon voidaan liittää monia erilaisia mallintamista helpottavia merkintöjä.

Tilakaavio voi kuvata ohjelmayksikön (esim. olio tai prosessi) käyttäytymisen tilakoneena, palveluprotokollan (so. missä järjestyksessä palveluja voidaan pyytää) tai olion elinkaaren. Palveluprotokollan kuvaava tilakaavio voidaan liittää tarkennuksena esimerkiksi rajapintaan, jolloin tilakaavio spesifioi rajapintaan kuuluvien operaatioiden mahdolliset suoritusjärjestykset. Olion elinkaari on usein kätevää kuvata tilakaaviona, vaikka olion varsinaista käyttäytymistä ei voisikaan sellaisella kuvata.

Kuvassa 4 on annettu autonvuokrausjärjestelmän KohdeKontrolli-olion tilakaavio. Siirtymissä olevat operaatioiden nimet kertovat, minkä operaation kutsun seurauksen tilasiirtymä suoritetaan. Ilmaus “after” siirtymässä tarkoittaa, että siirtymä laukeaa tietyn ajan kuluttua. Jos siirtymään ei liitetä tietoa aiheuttajasta, siirtymä tapahtuu automaattisesti kun läh-



Kuva 3: Autonvuokrausjärjestelmään liittyvä komponenttikaavio



Kuva 4: KohdeKontrolli-olion käyttäytymistä kuvaava tilakaavio

tötilassa olevat toimenpiteet saadaan tehtyä. Hakasuluissa oleva ilmaus tarkoittaa ehtoa, jonka toteutumista siirtymän laukeaminen edellyttää. Tiloissa olevat “entry” toimenpiteet suoritetaan tilaan tultaessa ja “exit” toimenpiteet tilasta poistuttaessa. Musta pallo tarkoittaa alkutilaa, häränsilmä lopputilaa.

Tilakaavio on keskeinen UML:n työkalu abstraktin käyttäytymisen kuvaamiseen tilanteissa, joissa järjestelmän osien toiminta on parhaiten ymmärrettävissä siirtyminä tilasta toiseen. Tilakaaviot antavat myös mahdollisuuden järjestelmän toiminnan simulointiin ja animointiin, joita voidaan käyttää järjestelmän kriittisten dynaamisten ominaisuuksien tutkimiseen ennen varsinaista toteutusvaihetta. Koska tämä ei edellytä lopullisen toteutuskoodin olemassaoloa, malliin (esimerkiksi tilakaavioihin) ei välttämättä tarvitse liittää informaatiota toteutustason asioista.

Aktiviteettikaaviot (activity diagram) yhdistävät kaksi vanhaa, hyvin tunnettua toiminnan graafista kuvaustapaa, kontrollivuokaaviot ja tietovuokaaviot. Aktiviteettikaaviot koostuvat solmuista, joita yhdistävät särmät kuvaavat kontrollin tai tiedon siirtymistä solmujen välillä. Solmussa määritelty toiminto käynnistyy, kun sisään tulevissa särmissä oleville tieto- tai kontrollialkioille pätevät tietyt ehdot, ja toiminnon lopputuloksena joihinkin tai kaikkiin ulosmeneviin särmiin syötetään uudet alkiot. Solmu voi olla myös ehdollinen haarautuminen, joka kuvataan tutulla vinoneliösymbolilla, tai vuon yhdistyminen/jakautuminen, joka kuvataan paksuna viivana. Tietovuomainen kuvaus saadaan aikaan, kun käytetään solmuina olioita (suorakaide), jolloin solmu kertoo mitä tietoa särmässä kulkee.

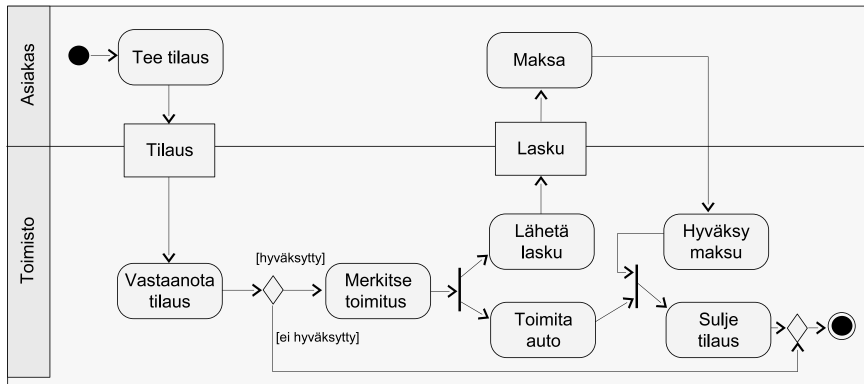
Aktiviteettikaaviot voidaan jakaa osiin toimintojen suorittajan mukaan. Tällaiset osat piirretään pysty- tai vaakasuuntaisina vyöhykkeinä (swimlane) kaavioon. Suorittajaperustainen jako antaa jossain määrin mahdollisuuden kuvata aktiviteettikaaviolla myös vuorovaikutusta; huomaa kuitenkin, että aktiviteettikaavio ei ota kantaa siihen, miten vuorovaikutus tapahtuu, vaan vuorovaikutuksen sisältöön. Kuvassa 5 on annettu esimerkki aktiviteettikaaviosta, jossa kuvataan tilauksen käsittely autonvuokrausjärjestelmässä.

Tyypillisesti aktiviteettikaavioita käytetään tarkentamaan käyttötapauksia, kuvaamaan tietojärjestelmässä tai organisaatiossa tapahtuvaa tiedon prosessointia tai spesifioimaan keskeisiä operaatioita tai algoritmeja. Aktiviteettikaaviot ovat kuitenkin suhteellisen vähän käytetty (ja ehkä myös huonosti ymmärretty) UML:n osa, joka ei ole yhtä selkeästi löytänyt paikkaansa käytännön ohjelmistokehitysprosesseissa kuin esimerkiksi sekvenssikaaviot.

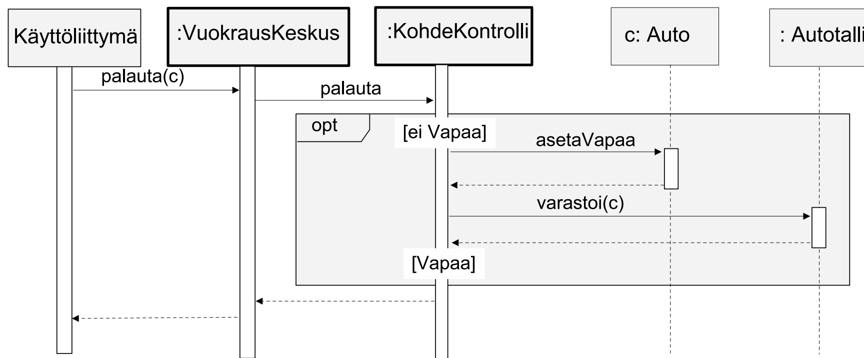
Sekvenssikaaviot (sequence diagram) periytyvät puolestaan nk. MSC-kaavioista (Message Sequence Chart), joilla jo pitkään ennen UML:ää kuvattiin prosessien välistä vuorovaikutusta etenkin tietoliikennemaailmassa. Perinteisesti sekvenssikaavio kuvaa mahdollisen suorituspolun aikana tapahtuvan viestienvaihdon tiettyjen osallistujien (esimerkiksi olioiden) välillä. Muista UML:n kaaviotyypeistä poiketen sekvenssikaavio ei ole verkkomainen esitys, vaan osallistujat kuvataan aikaulottuvuuden suuntaisina viivoina (yleensä pystysuorassa), ja osallistujien välillä kulkevat viestit esitetään nuolina yhdeltä osallistujaviivalta toiselle, aikajärjestyksessä.

Jos olioiden käyttäytyminen on kuvattu tilakaavioina, näiden olioiden välinen sekvenssikaavio kuvaa yhden suorituspolun tilakaavioiden joukossa. Sekvenssikaavio edustaa esimerkkiä oliojoukon käyttäytymisestä, kun taas tilakaavio kuvaa yhden olion täydellisen käyttäytymisen.

UML 2.0 on laajentanut olennaisesti sekvenssikaavion ilmaisuvoimaa lisäämällä siihen mm. rakenteiset haarautumisilmaisut (vapaaehtoisuus, vaihtoehtoisuus ja toisto) ja alisekvenssikaaviot. Nämä saadaan aikaan visuaalisesti rajaamalla osa sekvenssikaaviosta nk. fragmentiksi, joka voidaan jakaa katkoviivalla edel-



Kuva 5: Auton vuokrausta kuvaava aktiviteettikaavio



Kuva 6: Auton palautustoiminnan kuvaava sekvenssikaavio

leen osiin haluttaessa kuvata esimerkiksi vaihtoehtoisuutta. Alisekvenssikaaviot vastaavat eräänlaista makrokutsua: viitattu sekvenssikaavio ajatellaan liimatuksi kutsun paikalle kutsuvaan kaavioon. Laajennokset on tehty seuraten MSC-notaatiota.

Kuvassa 6 on esitetty autonvuokrausjärjestelmään liittyvä sekvenssikaavio. Fragmentti “opt” on valinnainen lohko, joka suoritetaan, jos KohdeKontrolli-olioon liitetty vahti “[ei Vapaa]” on tosi. Toisin sanoen, KohdeKontrolli-olion palauta-operaatio toimii siten, että jos olio

on jossakin muussa tilassa kuin Vapaa, se saatetaan tähän tilaan; muussa tapauksessa ei tehdä mitään. Osanottajien elinkaariviivoihin liitetyt pystypalkit tarkoittavat aktivaatiota: kyseinen osallistuja on tuona aikana suorittamassa jotain omaa operaatiotaan. Nk. aktiivisilla olioilla (olio toimii omassa suoritussäikeessään) aktivaatiopalkit kattavat koko elinkaariviivan. Paluu operaatiosta kuvataan katkoviivanuolella.

Sekvenssikaaviot ovat tärkein muoto UML:n vuorovaikutuskaavioista. Muita vuorovaikutuskaavioita ovat *yhteistoimintakaavio* (communication diagram), *kokoava vuorovaikutuskaavio* (interaction overview diagram) ja *ajoituskaavio* (timing diagram). Yhteistoimintakaavio sisältää olennaisesti saman informaation kuin yksinkertainen sekvenssikaavio (ilman fragmentteja), mutta vuorovaikutus esitetään olioiden välillä ilman aikadimensioon sidottuja elinkaariviivoja. Kaaviossa solmuina ovat oliosymbolit ja särminä olioiden väliset linkit samalla tavalla kuin oliokaaviossa. Linkkeihin liitetään merkintä, joka ilmaisee olion toiselle lähettämän viestin (tai operaatiokutsun) sekä tämän suoritusjärjestysnumeron. Yhteistoimintakaaviolla voidaan siten esittää olioiden vuorovaikutus tietyssä tapauksessa kuten sekvenssikaaviolla, mutta tapahtumien ajallinen järjestys kuvataan numeroilla, ei asettelemalla tapahtumat tiettyyn järjestykseen kaaviossa.

Kokoava vuorovaikutuskaavio on aktiviteettikaavion muunnelma, jossa toimintosolmuina on kokonaisia, esimerkiksi sekvenssikaavioilla kuvattuja vuorovaikutuksia. Tällaisella kaaviolla voidaan siten kuvata esimerkiksi logiikka, jolla eri sekvenssikaaviot seuraavat toisiaan. Ajoituskaavio kuvaa yhden tai useamman olion käyttäytymisen reaaliajan suhteen. Kokoavat vuorovaikutuskaaviot ja ajoituskaaviot ovat uusia UML 2.0:ssa, eikä niiden käytöstä ole juuri kokemuksia.

Käyttötapauskaaviot (use case diagram) kuvaavat käyttötapausten suhteet toisiinsa sekä järjestelmän ulkopuolisiin toimijoihin (kuten käyttäjä). Käyttötapausten välisiä suhteita voivat olla laajennossuhde (käyttötapaus laajentaa toista), sisältymissuhde (käyttötapaus sisältyy toiseen) tai yleistyssuhde (käyttötapaus kuuluu johonkin yleistävään käyttötapauskategoriaan). Käyttötapauskaavio on suhteellisen yksinkertainen koko järjestelmän toiminnallinen malli, joka etenkin ohjelmistokehitysprosessin alkuvaiheessa auttaa tunnistamaan kuka tai mikä käyttää järjestelmää ja liittämään asiakasvaatimukset järjestelmän toimintoihin. Huomaa, että käyttötapauskaavio ei kuvaa varsinaisia käyttötapauksia; tähän tarkoitukseen sopivat esimerkiksi sekvenssikaaviot ja aktiviteettikaaviot.

3 UML:n metamalli ja sen laajentaminen

3.1 Miksi tarvitaan metamalli?

Kun annetaan uusi kieli mihin hyvänsä tarkoitukseen, on määriteltävä sekä sen rakenne että merkitys. Jos kyseessä on tekstuaalinen ohjelmointikieli, kielen raken-

ne voidaan antaa BNF-muotoisella kieliopilla ja sen merkitys voidaan kuvata esimerkiksi rakenteisiin liitetyillä sanallisilla selityksillä. Ohjelmointikielen tapauksessa sekä rakenteen että merkityksen yksiselitteinen, tarkka määrittely on ehdoton vaatimus.

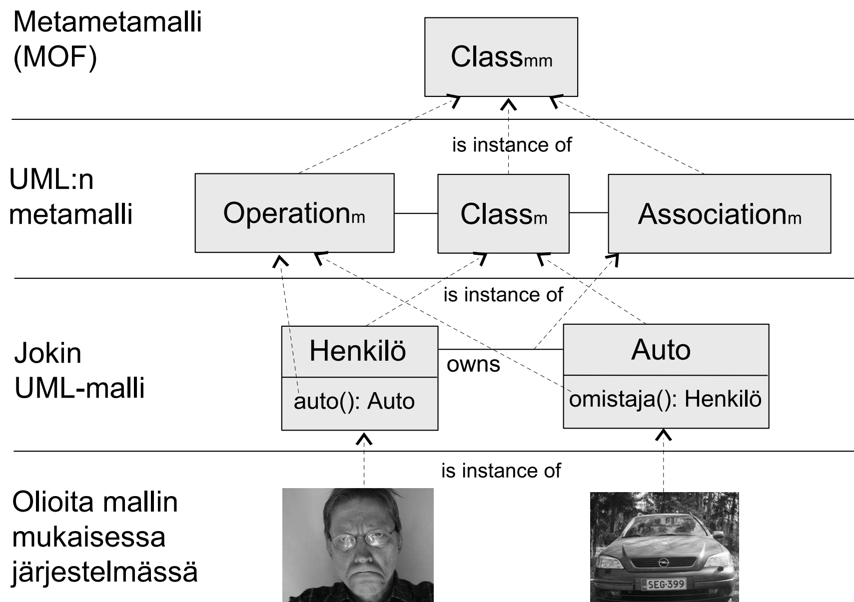
UML:n tapauksessa merkityksen tarkkaa määrittelyä ei ole pidetty käytännössä ehdottoman tärkeänä, koska UML:ää sovelletaan hyvin erilaisissa, usein epäformaaleissa yhteyksissä. Väljästä merkityksen määrittelystä on se etu, että UML:ää voi käyttää varsin vapaasti eri ohjelmistokehityksen vaiheissa ja eri sovellusalueilla tulkitsemalla mallien merkitys sopivasti. UML:ää voi käyttää kuvaamaan myös puutteellisesti ymmärrettyjä järjestelmän käsitteitä, mikä ohjelmistokehityksen alkuvaiheessa on usein tarpeen. Tämä ei toki estä sitä, etteikö joillekin UML:n osille — esimerkiksi tilakaavioille — voisi antaa täsmällistä, formaalia merkitystä ja käyttää sitä hyväksi esimerkiksi järjestelmän analysoinnissa, simuloinnissa tai koodin generoinnissa.

Kielen rakenne voidaan jakaa konkreettiseen rakenteeseen ja abstraktiin rakenteeseen. UML:n tapauksessa edellinen kuvaa UML-kaavioiden visuaalisen ulkoasun, jälkimmäinen mallielementtien loogiset suhteet. UML:n standardi ei määrittele lainkaan konkreettista rakennetta, vaan antaa siitä vain suosituksia. Käytännössä esimerkiksi eri työkalujen tavat visualisoida UML-kaavioita ovat muodostuneet näiden suositusten pohjalta varsin samanlaisiksi. Koska UML:n konkreettinen rakenne on olennainen vain ihmisen ymmärtämisen kannalta, pienet variaatiot konkreettisessa rakenteessa eivät ole ongelma.

Abstraktin rakenteen määrittelyssä on sen sijaan oltava tiukan täsmällinen, koska työkalujen on pystyttävä käsittelemään UML-malleja, vaihtamaan niitä keskenään ja ymmärrettävä UML:n rakenne samalla tavalla. Graafisen kielen abstraktin rakenteen formaalissa määrittelyssä on kaksi lähestymistapaa, metamallit ja erilaiset verkkokieliopit. Edellisessä lähestymistavassa sallitut mallien abstraktit rakenteet kuvataan korkeamman tason mallilla, metamallilla; jälkimmäisessä annetaan joukko tuottosääntöjä, joita soveltamalla saadaan luotua kaikki sallitut mallien abstrakti rakenteet.

UML:n kaltaisen mallinnuskielen tapauksessa metamallipohjainen lähestymistapa on luonteva, koska kieltä itseään voidaan käyttää metamallin antamiseen. UML:n keskeinen kaaviotyyppi, luokkakaavio, on tarkoitettu olioiden muodostamien abstraktien rakenteiden määrittelyyn. Näin on mahdollista käyttää UML:n osajoukkoa määrittelemään UML:n abstrakti rakenne. Tämä määrittely on UML:n metamalli.

UML:n metamalli määrittelee lisäksi joukon nk. hyvinmuodostuneisuussääntöjä (well-formedness rules), jotka edelleen rajoittavat sallittuja UML-malleja. Näitä voitaisiin verrata ohjelmointikielissä erilaisiin staattisiin laillisuussääntöihin, esimerkiksi tyyppisääntöihin, joita ei voi helposti kuvata BNF-kieliopilla. Hyvinmuodostuneisuussäännöt annetaan OCL-kielellä.



Kuva 7: UML:n mallitasot

OMG:n tavoite ei kuitenkaan ole ollut standardoida pelkästään UML, vaan myös infrastruktuuri, jonka pohjalta voidaan kehittää muitakin samaan peruskäsitteistöön nojaavia mallinnuskieliä. Tämä tavoite on johtanut monitasoiseen metamalliarkkitehtuuriin, jota tarkastelemme seuraavassa.

3.2 OMG:n metamalliarkkitehtuuri

OMG:n metamalliarkkitehtuuri perustuu nelitasoiseen rakenteeseen kuvan 7 mukaisesti. Alimman tason rakenteen muodostavat järjestelmän ajoaikaiset oliot ja niiden väliset linkit. Kutsumme tätä tasoa *järjestelmätasoksi*. Seuraavalla tasolla on järjestelmän *malli*, esimerkiksi kuvan 7 mukaisesti luokkakaavio, jonka luokkien ilmentymistä järjestelmätaso muodostuu. Seuraavalla tasolla on vastaavasti hieman yleisempi luokkakaavio, *metamalli*, jonka luokkien (*metaluokkien*) ilmentymiä ovat puolestaan mallitason elementit. Ylimmällä tasolla on edelleen yleisempi luokkakaavio, *metametamalli* (MOF, Meta Object Facility), jonka luokkien ilmentymiä ovat metamallitason elementit. Kun käytännöllisenä tavoitteena on kehys,

jonka puitteissa voidaan määritellä erilaisia samaan peruskäsitteistöön pohjautuvia mallinnuskieliä, nämä neljä tasoa riittävät. Metametamalli on lopulta itsensä eräs ilmentymä.

Eri mallitasoja voidaan konkretisoida UML:n tapauksessa seuraavalla ajatusleikillä. Ajatellaan ensin kaikkia mahdollisia UML-malleja (voidaan yksinkertaistaa hiukan ja rajoittua tässä luokkakaavioihin), jotka halutaan pystyä esittämään UML:llä. Mikä on se UML-luokkakaavio, joka riittää kaikkien näiden esittämiseen olioina? Huomaa, että tarvitsemme selvästi oliot edustamaan esimerkiksi luokkia, näiden attribuutteja ja operaatioita, ja vaikkapa assosiaatioita. Tuloksena saatava luokkakaavio on UML:n metamalli. Koska kyseessä on tietyn maailman — tässä tapauksessa UML:n itsensä — mallintaminen luokkakaaviona, emme tarvitse välttämättä kaikkia UML:n piirteitä tässä luokkakaaviossa.

Nyt voimme jatkaa ajatusleikkiä: mikä luokkakaavio olisi sitten riittävä kaikkien metamallin elementtien esittämiseen olioina? Koska metamalli on vain yksi luokkakaavio, jossa ei käytetä kaikkia UML:n piirteitä, tässä riittää varmastikin metamallia suppeampi luokkakaavio. Tämä on UML:n metametamalli. Todellisuudessa asia ei ole aivan näin yksinkertainen, koska haluamme pystyä esittämään metamalleja muillekin mallinnuskielille kuin UML:lle.

Kuvassa 7 on yksinkertaistaen havainnollistettu UML:n mallitasoja. Kuvan nuolet havainnollistavat ilmentymän suhdetta luokkaansa.

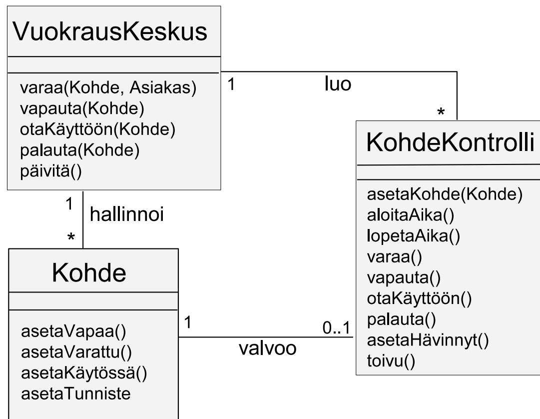
Eräs konkreettinen etu MOF-metametamallista on, että sen avulla on voitu määritellä XML-pohjainen standardi esitysmuoto kaikille malleille, jotka on tehty MOF-pohjaisella mallinnuskielellä. Tätä esitysmuotoa kutsutaan XMI:ksi (XML Metadata Interchange). Palaamme XMI:hin työkalujen yhteydessä luvussa 5.

3.3 OCL

Object Constraint Language (OCL) on UML:n sisältämä formaali deklaratiivinen kieli erilaisten sivuvaikutuksettomien päättelyjen esittämiseen järjestelmän tilasta. Alun perin kieli oli tarkoitettu rajoitteiden lisäämiseen kaavioihin. Usein kaavioissa halutaan esittää sellaisia rajoitteita, joiden ilmaisemiseen ei pelkästään UML:n visuaalisella notaatiolla ole mahdollisuuksia. Esimerkiksi kuvan 8 kaavio kertoo, että jokaiseen *Kohteeseen* ja *KohdeKontrolliin* liittyy yksi *VuokrausKeskus*, ja että jokaiseen *KohdeKontrolliin* liittyy yksi *Kohde*. Graafisesti ei kuitenkaan voida ilmaista, että toisiinsa liittyvillä *Kohteen* ja *KohdeKontrollin* ilmentymillä pitää olla sama *VuokrausKeskus*. Tämä rajoite voidaan esittää OCL:llä esim. seuraavasti:

```
context KohdeKontrolli:
 inv:
  self.vuokrausKeskus =
  self.kohde.vuokrausKeskus
```

Ylläoleva rajoite on sidottu luokkaan *KohdeKontrolli* avainsanalla *context* ja on muodoltaan invariantti. Invariantti on rajoite, jonka ehdon on aina oltava voimassa



Kuva 8: Osa autonvuokrausjärjestelmän luokkakaaviosta

jokaiselle sidotun luokan ilmentymälle. Itse ehdossa vaaditaan, että tarkasteltavana olevan ilmentymän (*self*) *VuokrausKeskus* on sama kuin tarkasteltavan ilmentymän valvoman *Kohteen VuokrausKeskus*.

Ennen OCL:n lisäämistä rajoitteet voitiin esittää vain lisäämällä kaavioihin huomautuksia luonnollisella kielellä tai dokumentoimalla rajoitteet erikseen. Epäformaalien rajoitteiden ongelmana on kuitenkin tulkinnanvaraisuus ja automaattisen käsittelyn puuttuminen: rajoitteita ei voi esimerkiksi automaattisesti tarkastaa annetulle mallille. Toisaalta kovin matemaattisella merkintätavalla kirjoitettujen rajoitteiden pelätään olevan liian vaikeita ymmärtää ja kirjoittaa tavalliselle ohjelmistosuunnittelijalle. OCL:n onkin tarkoitus olla formaali, mutta samalla helppokäyttöinen kieli.

OCL:n ydin koostuu perustyypeistä ja rakenteisista tyypeistä sekä tyypeille määritellyistä operaatioista. Perustyyppejä ovat kokonaisluku, reaaliluku, merkkijono ja totuusarvo. Rakenteisia tyyppejä ovat joukko (*set*), monijoukko (*bag*), järjestetty joukko (*ordered set*), järjestetty monijoukko (*sequence*) ja monikko (*tuple*). Koska kyseessä on puhtaasti määrittelyyn tarkoitettu kieli, tyypien arvoilla tai koolla ei ole ylärajaa. Tyypeille määriteltyjä operaatioita ei ole kovin monia, jopa joitakin melko yksinkertaisia operaatioita puuttuu. Esimerkiksi kokonaisluvuille löytyvät yhteen- ja kertolasku mutta ei potenssiin korotusta. Ytimeen on pyritty keräämään vain tärkeimmät ja useimmin käytetyt operaatiot, jotta kieli pysyisi yksinkertaisena. Joukoille löytyvät mm. olemassaolo- ja kaikkikvanttorit. OCL:ää on myös helppo laajentaa lisäämällä uusia operaatioita ja tyyppejä, joten useimmat puutteet on mahdollista korjata melko vähällä vaivalla.

3.4 Metamallin laajentaminen

Tyypillisellä ohjelmistokehitystä tekevällä yhteisöllä on sovellusalueeseen, toimintakulttuuriin ja käytettäviin ohjelmistoprosesseihin liittyviä rajoitteita ja toimintatapoja. Vaikka UML on määritelmänsä mukaisesti yleiskäyttöinen mallinnuskieli, syntyy usein tilanne, jossa sen ilmaisuvoimaa halutaan rajoittaa tai erikoistaa tietylle yhteisölle sopivaan muotoon. Koska yksi UML:n avainvahvuuksista on sen hyvin määritelty standardi metamalli, ei metamallin mielivaltainen muokkaaminen kunkin tarpeita vastaavaksi ole käyttökelpoinen lähestymistapa.

UML vastaa laajennustarpeisiin antamalla tavan erikoistaa UML:n metamallia *profiilien* avulla. Profiili voidaan ymmärtää metamallin laajennoksena, joka on saatu alkuperäisestä metamallista lisäämällä siihen metaluokkien erikoistuksia, nk. *stereotyyppejä*, sekä ylimääräisiä (OCL-) rajoitteita. Kun mallintaminen perustuu tällaiseen profiiliin, suunnittelijalla on käytössään valikoima toimintaympäristöönsä liittyviä käsitteitä suoraan uusina mallielementteinä, ja rajoitteet ohjaavat käyttämään näitä käsitteitä tarkoitetuilla tavoilla.

Stereotyyppejä voidaan antaa periaatteessa kaikille metaluokille. Stereotyypin S ilmentymä näkyy mallissa samanlaisena elementtinä kuin sen erikoistaman metaluokan ilmentymä, mutta stereotyyppi merkitään näkyviin kirjoittamalla elementin nimen eteen «S». Koska stereotyyppi ei muuta tai poista mitään metaluokkien ominaisuuksia, työkalut voivat käsitellä stereotyyppien ilmentymiä kuten vastaavien metaluokkien ilmentymiä.

Esimerkiksi sellainen termi kuin “transaktio” on vakiintunut tarkoittamaan tiettyä yleistä käsitettä, jota tarvitaan lähes kaikissa liiketoimintajärjestelmissä. Niinpä voimme antaa tällaisille järjestelmille profiilin, johon tämä käsite sisältyy metaluokan “Class” stereotyyppinä. Tällöin voimme käyttää autonvuokrausjärjestelmän luokkakaaviossa esimerkiksi kuvassa 9 annettua stereotyypin ilmentymää.

Stereotyyppien (ja profiilien) käytöstä on monta etua. Stereotyypit antavat mallielementeille tarkemman (vaikkakin silti epäformaalin) semanttisen tulkinnan ja helpottavat siten mallien tekemistä ja ymmärtämistä oikealla tavalla. Näin stereotyypit ovat kevyt tapa tehdä tiettyyn tarkoitukseen oma mallintamiskieli, jolla on omat käsitteensä ja rajoitteensa, poistumatta silti UML-maailmasta ja sen antamasta työkalutuesta. Kun lisäksi stereotyypille voidaan antaa oma ulkoinen esitysmuoto, stereotyyppien avulla on mahdollista luoda UML-pohjainen sovellusaluekohtainen mallintamiskieli, jolla annetut kaaviot kuvaavat intuitiivisin symbolein sovellusalueen käsitteitä, muistuttamatta lainkaan UML:n kaavioita.

Työkalu voi toisaalta antaa stereotyypille täsmällisen merkityksen liittämällä tietyn stereotyypin ilmentymien prosessointiin jotain erityistä toiminnallisuutta. Esimerkkinä tästä voisi olla vaikkapa toteutusalustakohtainen (esim. J2EE) stereotyyppi, jonka avulla työkalu voi tuottaa tälle alustalle koodia UML-mallin perusteella. Työkalu voi myös suorit-

Kuva 9: Stereotyypin soveltaminen autonvuokrausjärjestelmässä

taa erilaisia rakenteen oikeellisuustarkistuksia käyttäen hyväkseen stereotyyppeihin liitettyjä rajoitteita.

Profiili voi kuvata myös järjestelmän yleisen arkkitehtuurityylin tai jonkin ohjelmistoalustan edellyttämät konventiot, joita suunnittelijoiden on UML-malleissaan seurattava. Tutkimusryhmämme on soveltanut profiileita tuotelinja-arkkitehtuurikohtaisten konventioiden ja rajoitteiden esittämiseen ja automatisoituun tarkistamiseen [28]. Profiili voi antaa myös tietyn järjestelmäkategorian edellyttämät peruskäsitteet, joita mallintamisessa tarvitaan. Esimerkkinä tällaisesta on OMG:n tukema reaaliaikajärjestelmien mallintamiseen tarkoitettu profiili.

4 UML:n käyttötapoja

4.1 UML on yleiskäyttöinen

Monista kirjoista, artikkeleista ja UML-työkaluista saa helposti sellaisen kuvan, että UML tai sen eri kaaviotyypit olisi tarkoitettu käytettäväksi nimenomaan johonkin tiettyyn tarkoitukseen tai jollain tietyllä tavalla. Esimerkiksi eri elementtien merkitys tai kaavioiden luontitapa on kuvattu hyvinkin yksityiskohtaisesti ja annettu ymmärtää, että tämä merkitys tai käyttötapa on ainoa mahdollinen. Samoin esimerkiksi eri kaaviotyyppien väleillä oletetaan olevan tietynlaisia suhteita. Nämä kaikki sitovat UML:n tiettyyn käyttötarkoitukseen, tapaan, metodiin tai prosessiin, joka ei kuitenkaan todellisuudessa ole ainoa oikea tapa käyttää UML:aa. Pitää muistaa, että UML on vain standardoitu kuvaustekniikka, jonka käyttökohteita tai käyttötapoja ei ole itse standardissa rajattu.

Eri yhteyksissä UML:ää voidaan käyttää hyvinkin eri tavoin. Niinpä esimerkiksi UML:n luokkakaaviota voidaan käyttää kaikissa ohjelmistokehityksen vaiheissa eri tavoin riippuen kulloisestakin ohjelmistokehitysprosessista. Tyypillisesti sitä käytetään oliokeskeisissä ohjelmistokehitysprosesseissa mm. käsiteanalyysissä, olioanalyysissä, arkkitehtuurisuunnittelussa ja yksityiskohtaisessa suunnittelussa.

Käsiteanalyysissä ei olla kiinnostuneita järjestelmästä vaan sen kohdemaa-

ilmasta: siinä toteutusnäkökohdat, kuten suorituskyky tai arkkitehtuuri, jätetään huomiotta. *Olioanalyysissä* taas kuvataan, millaisia olioita järjestelmässä on ja miten niiden luokat liittyvät toisiinsa, ottamatta huomioon arkkitehtuuriin liittyviä päätöksiä. *Arkkitehtuurisuunnittelussa* keskitytään kuvaamaan arkkitehtuurin kannalta merkittävien luokkien (ja komponenttien) suhteita. Lopulta *yksityiskohtaisessa suunnittelussa* kuvataan lopullisen koodin rakennetta yksittäisten luokkien tasolla. Kaikissa käyttökohteissa kaaviot näyttävät hyvin samanlaisilta, mutta elementtien ja niiden välisten suhteiden merkitys vaihtelee käyttökohteesta riippuen.

Vastaavasti käyttäytymiskaavioita voidaan soveltaa eri ohjelmistokehityksen vaiheissa. Esimerkiksi sekvenssikaavio voi aluksi kuvata pelkästään käyttäjän ja järjestelmän välisen vuorovaikutuksen tietyn käyttötapauksen yhteydessä. Tällainen kuvaus voidaan liittää järjestelmän vaatimuksiin. Sekvenssikaavio voidaan myöhemmin tarkentaa jakamalla järjestelmää kuvaava osallistuja ensin osajärjestelmiin, sitten komponentteihin ja lopulta olioihin, joiden vuorovaikutus kertoo miten käyttötapaus toteutetaan alimmalla tasolla. Kun sekvenssikaavioissa voi käyttää myös ehto- ja toistorakanteita, sekvenssikaavioilla voidaan jopa hahmottaa operaatioiden yleistä toteutusta.

Samoin kuin UML:n käyttökohteet, myös sen käyttötavat vaihtelevat. UML:n yleiset käyttötavat voidaan jakaa karkeasti viiteen eri luokkaan: luonnosteluun, dokumentointiin, mallin rakentamiseen, visuaaliseen ohjelmointiin ja järjestelmien takaisinmallinnukseen. Erot syntyvät lähinnä käyttötavan muodollisuudesta sekä siitä, onko kyse ihmisten välisestä, koneiden välisestä vai ihmisen ja koneen välisestä kommunikoinnista.

4.2 UML:n käyttö ihmisten väliseen kommunikointiin

Luonnostelu on kohdealueen, esimerkiksi järjestelmän arkkitehtuurin, vapaamuotoista hahmottelua ja se on tarkoitettu ihmisten väliseen kommunikointiin. Useimmiten luonnostelun tarkoituksena on ajatusten hahmotteleminen, selventäminen ja esittäminen, tai yksinkertaisesti muistiinpanojen tekeminen. Kuvaukset ovat yleensä erittäin abstrakteja piirroksia, vain mallin tulkinnan kannalta kaikkein oleellisimmat asiat esitetään. Lisäksi näin tehdyt mallit on tarkoitettu yleensä vain pienen ryhmän nähtäväksi ja pois heitettäväksi, joten esimerkiksi elementeille ja suhteille voidaan antaa merkitykset suullisesti. Luonnosten yhteydessä UML:n etuna on, että muutkin kuin kuvauksen tekijä voivat kohtuullisen helposti ymmärtää hahmotelmat ja asioiden esittämistapaa ei tarvitse joka kerta erikseen miettiä. Luonnostelu ei välttämättä tarvitse lainkaan työkalutukea, vaan kynä ja paperi ovat useimmiten parhaat välineet kuvausten tekemiseen.

Myös *dokumentointi* on tarkoitettu ihmisten väliseen kommunikointiin, mutta se eroaa luonnostelusta mm. sen vuoksi, että kysymyksessä on muodollisempi tapa kuvata kohdealuetta. Koska kuvattava asia pitää olla ymmärrettävissä dokumen-

tin perusteella, myös sen sisältämien mallien pitää kuvata kohdealueen kaikki olennaiset piirteet. Esimerkiksi määrittelydokumentin tulee kuvata määritelty järjestelmä siten, että järjestelmä voidaan suunnitella ja toteuttaa sen perusteella. Siten dokumentointiin käytetyt mallit sisältävät luonnoksiin verrattuna huomattavan määrän yksityiskohtia ja selitystä. Dokumentointiin käytetyt mallit ovat myös yleensä tarkoitetut suhteellisen pysyviksi — dokumentointihan on osa järjestelmää.

Koska dokumentoinnissa käytettävien mallien pitää olla selkeitä ja niitä joudutaan yleensä ylläpitämään, ne tehdään useimmiten tietokoneella. Kuitenkin niin luonnosten kuin dokumentointiinkin käytettävien mallien tapauksessa tärkeitä ovat nimenomaan kaaviot, siis ihmisen tulkittavaksi tarkoitetut kuvaukset. Niinpä "mallinnustyökaluksi" riittää periaatteessa mikä tahansa piirto-ohjelma, jolla kaaviot ylipäänsä saa tehtyä.

4.3 UML:n käyttö ihmisen ja koneen väliseen kommunikointiin

Mallien rakentamisella viitataan tässä yhteydessä UML:n käyttötapaan, jossa kaaviot itsessään ovat vain välineitä, joiden avulla ihmiset voivat tuottaa malleja tietokoneelle. Kaaviot ovat tällöin vain näkymiä malliin, eikä niiden tarvitse sisältää kaikkea mallissa olevaa informaatiota. Tässäkin käyttötavassa malli on usein alussa luonnosmainen ja syntynyttä tulosta käytetään mahdollisesti myös dokumentointiin. Pääasiallisena tarkoituksena on kuitenkin rakentaa malli tietokoneen ymmärtämään muotoon, jolloin mallin hallintaan tai käsittelyyn liittyviä tehtäviä voidaan tukea työkaluilla. Näin automatisoitaviin tehtäviin voi kuulua esimerkiksi mallin oikeellisuuden tarkistaminen ja erilaiset muunnokset. Työkaluissa yleisesti tuettu esimerkki muunnoksesta on UML:n luokkakaavion avulla tehdyn tietosisällön kuvauksen muuntaminen SQL-kielisiksi luontilauseiksi.

On huomattava, että automatisointi ei välttämättä vaadi yhtenäistä mallia, vaan suuri osa tehtävistä voitaisiin automatisoida myös kaavioiden (tai muiden mallin osien) perusteella. Yleinen käsitys kuitenkin on, että tällainen yhtenäinen malli on automatisoinnin kannalta tarpeen. Tällöin UML:n metamalli nousee merkittävään asemaan: se voidaan rinnastaa tietokantakaavioon, joka kuvaa yhtenäisen tavan tallentaa ja käsitellä mallia. Yhtenäisesti hallittavaan malliin liittyy automatisoinnin lisäksi muitakin potentiaalisia etuja, kuten mahdollisuus generoida mallista tarpeen mukaan erilaisia näkymiä, monessa mielessä helpompi hallittavuus (esim. kokonaisuus on helpompi alistaa versionhallinnalle), sekä erilaiset hakutoiminnot malliin. Näitä etuja voidaan tietysti hyödyntää vain, jos työkalu tukee näihin liittyvää toiminnallisuutta.

Visuaalisen ohjelmoinnin voidaan ajatella olevan UML:n käyttötapana edellisen erikoistapaus, jossa mallista saadaan suoritettava ohjelma lähes automaattisesti (vrt. ohjelman kääntäminen). Kysymys on siis ihmisen ja tietokoneen välises-

tä kommunikoinnista, jossa ihmisen pitää kuvata kohdejärjestelmä niin hyvin ja yksityiskohtaisesti, että tietokone pystyy suorittamaan ohjelman annetun kuvauksen perusteella. Tämä lähestymistapa vaatii luonnollisesti voimakasta työkalutukea ja useimmissa tapauksissa myös UML:n ulkopuolista informaatiota.

Monet työkalut antavatkin mahdollisuuden suoritettavan koodin (puoli)automaattiseen tuottamiseen mallista eri kohdekieliin. Jos esimerkiksi luokkakaavioita on sovellettu yksityiskohtaisen suunnittelun tasolla, niistä voidaan tuottaa järjestelmän rakenteen toteuttava koodirunko. Luokkakaavioissa ei ole kuitenkaan informaatiota, jonka perusteella esimerkiksi metodien runkoja voitaisiin tuottaa, joten näin tuotettua koodia joudutaan yleensä täydentämään käyttämällä hyväksi jotain muuta informaatiota. Lisäksi on huomattava, että luokkakaavion suhteet eivät vastaa yksi-yhteen lähdekoodissa käytettävissä olevia mekanismeja, joten tällainen koodin tuottaminen perustuu aina jonkinlaiseen luokkakaavion tulkintaan, joka voi olla joko täysin työkalun kiinnittämä tai suunnittelijan säädeltävissä. Pahimmassa tapauksessa suunnittelija joutuu korjaamaan työkalun tulkintoja vielä lähdekoodissa.

Järjestelmän käyttäytymiseen liittyvän koodin (esimerkiksi metodien rungot) tuottaminen voi perustua informaatioon, jonka suunnittelija on antanut käyttäytymiskaavioissa (esim. sekvenssi- tai tilakaavioissa). Tilakaavioista voidaan tuottaa suhteellisen suoraviivaisesti toteutuskoodi, jos tilakaavio kuvaa esimerkiksi tietyn luokan olioiden käyttäytymisen riittävän tarkasti. Tämä edellyttää kuitenkin yleensä, että tilakaavioon liitetään täsmällisiä ilmaisuja kuvaamaan niitä primitiivisiä toimenpiteitä, joita tiloissa tai siirtymissä halutaan tehdä.

Tätä tarkoitusta varten UML:ään on lisätty aktiosemantiikka (action semantics), jota voidaan verrata abstraktilla tasolla määriteltyyn suoritettavaan ohjelmointikieleen. Aktiosemantiikan uutuudesta johtuen sen määrittely tulee muuttumaan, eikä käyttökokemuksiakaan vielä varsinaisesti ole. Käytännöllisempi ratkaisu on joidenkin työkalujen tapa sallia varsinaisen toteutuskielen ilmaisuja mallissa. Tämä tosin sekoittaa mallin ja toteutuksen eroa ja vie siinä mielessä pohjaa koko mallintamiselta. Lisäksi on huomattava, että useimmissa tapauksissa olion käyttäytymisen kuvaaminen tilakaaviona ei ole tarpeellista eikä edes luontevaa. Aktiviteettikaavioista voidaan tuottaa koodia kuten tilakaavioistakin, olettaen että primitiivitoiminnot on kuvattu riittävän täsmällisesti jollakin tunnetulla formalismilla.

Myös sekvenssikaavioista voidaan tuottaa suoritettavaa koodia. Jos sekvenssikaaviossa on käytetty kontrollirakenteita, sekvenssikaavio voi kuvata itse asiassa algoritmin useiden olioiden vuorovaikutuksena, mistä koodi voidaan periaatteessa tuottaa varsin suoraviivaisesti — etenkin, jos primitiiviset toimenpiteet (esim. sijoitukset) on merkitty kaavioon sopivalla tavalla.

Yleisesti voidaan kuitenkin sanoa, että koodin automaattisella tuottamisella on

ainakin toistaiseksi suhteellisen vähäinen merkitys UML:n käytössä teollisuudessa, vaikka se on tutkimuksen kannalta kiinnostava ongelma. UML:n käyttö "visuaalisena ohjelmointikielenä" saattaa kuitenkin lisääntyä työkalutuen kehittyessä.

4.4 Takaisinmallinnus

Mallin osia voidaan tuottaa myös automaattisesti työkalun avulla olemassa olevasta järjestelmästä. Tätä kutsutaan *takaisinmallinnukseksi* (reverse engineering). Takaisinmallinnus on kaaviotyyppikohtaista: työkalu tuottaa tavallisesti tietyn kaaviotyypin mukaisen esityksen järjestelmästä, ja sitä vastaavan malli-informaation. Takaisinmallinnuksen tavoitteena voi olla järjestelmän parempi ymmärtäminen mallin avulla, järjestelmän dokumentoinnin täydentäminen kaavioilla tai järjestelmän automaattinen analysointi tuotetun mallin avulla. Kahdessa ensimmäisessä tapauksessa on kyse UML:n käytöstä kommunikointiin koneelta ihmiselle, jälkimmäisessä koneelta koneelle.

Yleisimmin takaisinmallinnus koskee luokkakaavioiden tuottamista olemassa olevasta lähdekoodista. Tällöin luokkien väliset suhteet päätellään luokkien keskinäisistä viittauksista koodissa. Näin muodostettu luokkakaavio edustaa lähdekoodin hieman abstrahoitua visualisointia, ja se auttaa hahmottamaan järjestelmän luokkarakennetta paremmin kuin tekstuaalinen lähdekoodi. Takaisinmallinnuksessa on kuitenkin sama ongelma kuin koodin tuottamisessa: koska koodin ja luokkakaavion välillä ei ole suoraa käsitteellistä vastaavuutta, takaisinmallinnuksessa joudutaan sopimaan erilaisista konventioista koskien sitä, miten tietyt koodissa olevat asiat esitetään luokkakaaviossa. Lisäksi joitakin luokkakaavioon olennaisesti kuuluvia asioita (esim. kertautumismääreet) on yleisessä tapauksessa mahdotonta päätellä koodista. Takaisinmallinnettu luokkakaavio voi kuitenkin olla varsin hyödyllinen apuväline etenkin järjestelmän analysoinnissa, ylläpidossa ja dokumentoinnin päivittämisessä. Eri CASE-työkalut takaisimallintavatkin luokkakaavioita tyypillisesti hieman eri periaatteita hyödyntäen. Tämä puolestaan johtaa siihen, että eri työkalut tuottavat samasta lähdekoodista toisistaan poikkeavia luokkakaavioita [11].

Sekvenssikaavioita on myös mahdollista tuottaa ajettavasta ohjelmasta. Tämä edellyttää, että ohjelma on instrumentoitu sijoittamalla sopiviin kohtiin ylimääräisiä toimintoja, jotka tuottavat ajoaikana suoritusjäljen. Tällainen suoritusjälki voidaan puolestaan muuntaa sekvenssikaavioiksi. Kun toisaalta joukko sekvenssikaavioita on mahdollista muuntaa tilakaavioksi [7, 30, 32], on periaatteessa mahdollista tuottaa myös suoritettavaa ohjelmanosaa kuvaavia tilakaavioesityksiä automaattisesti. Tällaista ajoaikaisen informaation hyväksikäyttöä mallien syntetisoinnissa kutsutaan *dynaamiseksi takaisinmallintamiseksi*.

5 UML:n työkalutuki

5.1 Työkalujen nykytila

Koska UML on laaja-alainen mallinnuskieli eikä ota kantaa suunnittelumenetelmään, työkaluvalmistajat ovat ottaneet huomattavia vapauksia sen suhteen, mitkä asiat UML:ssä ovat olennaisia ja mitkä vähemmän olennaisia. Eri työkalut tarjoavat erilaisia ratkaisuja mm. tuettujen kaaviotyyppien ja niiden piirteiden suhteen, laajennettavuuden suhteen, mallin sisäisen esityksen ja sen ohjelmointirajapinnan suhteen sekä metamallin noudattamisen suhteen.

Työkaluvalmistajat eivät toistaiseksi ole suhtautuneet UML:n standardinäkökulmaan kovinkaan vakavasti, koska UML on vielä suhteellisen liikkuva maali uusine versioineen. Kun UML 2.0 tuo mukanaan runsain mitoin uusia, osittain melko huonosti määriteltyjä piirteitä tilanteessa, jossa työkalut eivät tarkkaan ottaen tue vielä edes versiota 1.1, kestää hetken aikaa ennen kuin tulemme näkemään täysin UML 2.0 -yhteensopivaa työkalua, päinvastaisista vakuutteluista huolimatta.

Edellä kuvatuista UML:n käyttötavoista varsinaiset UML-työkalut tukevat tyypillisesti mallien rakentamista, visuaalista ohjelmointia ja takaisinmallinnusta. On olemassa myös prototyyppityökaluja, jotka sallivat UML-kaavioiden luonnostelun valkotaululle siten, että erityisellä kynällä piirretyt kuvat tulkitaan ja siirretään koneelle automaattisesti UML-kaavioina [3]. Tällainen työkalu muuntaa luonnostelun lopulta ihmiseltä koneelle kommunikoinniksi. Työkalua voidaan käyttää esimerkiksi ketterien (agile) ohjelmistokehitysmenetelmien tukemiseen: näissä menetelmissä halutaan mahdollisimman kevyt ja interaktiivinen, ihmisten välistä kommunikointia tukeva mallinnustyökalu. Tähän rinnastettava prototyyppityökalu mahdollistaa UML-kaavioiden muodostamisen puhutun luonnollisen kielen perusteella [14].

UML-mallien rakentamiseen tarkoitetut työkalut tarjoavat nykyisin jo suhteellisen hyvät graafiset editointiominaisuudet. Työkalujen käyttöliittymä perustuu yleensä kaaviotyyppeihin: kaavioikkunaan liittyy symbolivalikko, joka sallii tietyn kaaviotyypin piirtämisen UML:n sääntöjen mukaisesti. Kaavioista muodostetaan looginen malli, joka voidaan näyttää erillisessä puunäkymässä. Malli ja kaavioihin liittyvä visuaalinen informaatio tallennetaan mallikannaksi, joka voi olla yksittäinen tiedosto tai kokonainen tietokanta. Eräät työkalut tukevat hajautettua suunnittelua: malli tallennetaan keskitettyyn tietokantaan, johon eri puolilla maailmaa toimivat suunnittelijat voivat kytkeytyä luomaan ja muokkaamaan UML-malleja. Ongelmana on tällöin huolehtia siitä, että eri suunnittelijat voivat kontrolloidusti käsitellä yhtäaikaisesti samaa mallia.

Yleiset graafiset piirtotyökalut ja UML-työkalut alkavat lähestyä toisiaan ja jakaa toistensa ominaisuuksia. Piirtotyökaluja voidaan usein erikoistaa UML-mallinnusta tukevalla piirtopohjalla, jonka avulla UML-mallinnuselementit ovat esimääriteltyinä suoraan käytettävissä [15].

Tällä hetkellä mallinnustyökalujen kehitykseen vaikuttaa etenkin tarve tukea vi-

suaalista ohjelmointia, takaisinmallinnusta ja ketteriä menetelmiä. Monista mallintimista on kehittymässä visuaalistekstuaalisia ohjelmointiympäristöjä, jossa on sekä mallin piirto-ominaisuudet että ohjelmointikielen syntaksin tunnistava teksteditori. Tällöin kyetään käyttämään hyväksi mallin visuaalista esitystä havainnollistamaan kokonaisuuksia, mutta toisaalta myös ohjelmoimaan toiminnallisuutta operaatioiden sisälle. Malli ja koodi pidetään jatkuvasti synkronoituna koodin generoinnin ja takaisinmallinnuksen avulla. Mallinnuksen ajan on siten jatkuvasti saatavilla myös mallin toteuttava kieliopillisesti oikea ohjelma.

Jatkuvasti synkronoituva malli ja koodi sopivat mainiosti myös ketteriin menetelmiin, vaikkakaan malli ei ole ketterissä menetelmissä aivan yhtä keskeinen kuin itse koodi. Mallinnustyökalut kuitenkin pyrkivät kehittymään siihen suuntaan, että malli ja koodi saadaan mahdollisimman tehokkaasti muutettua toisikseen, estäen siten mallin rapistuminen ohjelmiston elinkaaren aikana.

5.2 Työkalujen ohjelmointirajapinnat

Järjestelmien mallintamisen lisäksi suunnittelutyökaluja halutaan myös laajentaa käyttäjän omilla toiminnoilla. Tätä tarkoitusta varten suunnittelutyökalut sisältävät lähes poikkeuksetta jonkinlaisen tavan ohjata työkalua ja päästä muokattavaan malliin käsiksi. Yksinkertaisimmillaan tämä tarkoittaa mahdollisuutta nauhoittaa makroja, joilla toistetaan käyttäjän toimenpiteitä. Parhaimmillaan työkalu tarjoaa UML:n metamallin mukaisen rajapinnan, jota työkalu itsekin mahdollisesti käyttää [24]. Näiden kahden ääripään välille sijoittuu erilaisia työkaluvalmistajien omia rajapintoja, joiden käyttämät käsitteet ovat enemmän tai vähemmän työkalukohtaisia.

Työkalun rajapintaa käytetään yleensä tarkoitusta varten tehdyllä ohjelmointikielellä, joka on upotettu itse työkaluun. Monet työkalut tarjoavat sisäisen kielen lisäksi myös ulkoisen komponenttirajapinnan, jolloin suunnittelutyökalu ja sitä ohjaava käyttäjän ohjelma voidaan erottaa täysin erillisiksi ohjelmiksi.

Käyttötarkoituksesta ja rajapinnasta riippuen ohjelma voidaan liittää työkalun käyttöliittymään, jolloin työkalun laajentaminen onnistuu parhaimmassa tapauksessa täysin saumattomasti. Heikoimmillaan käyttöliittymä tarjoaa vain ohjelman käynnistysmahdollisuuden, mutta tämäkin riittää toiminnoille, jotka eivät vaadi syötteitä käyttäjältä (esimerkiksi kaavioiden asettelualgoritmit).

Interaktiiviset toiminnot ja niitä tarjoavat ohjelmat vaativat työkalujen rajapinnoilta mahdollisuutta saada tietoja käyttäjän valinnoista ja mallin muuttumisesta. Useissa rajapinnoissa on jokin tapa pyytää työkalua ilmoittamaan, kun malliin lisätään uusi elementti tai jotain olemassa olevaa elementtiä muokataan. Jos suunnittelutyökalu tarjoaa mahdollisuuden käyttöliittymien tekoon (esim. valikot ja työkalupalkit), tulee myös näistä erilaisia tapahtumia ohjelman käsiteltäväksi.

5.3 UML-mallien siirtäminen työkalujen välillä

XML Metadata Interchange (XMI) [18] on OMG:n kehittämä XML-muotoinen tiedonsiirtomuoto MOF-pohjaisille metamalleille. Koska myös UML on MOF-pohjainen, voidaan XMI:n avulla esittää UML-malleja XML-muodossa. XMI-spesifikaatio kuvaa tuotantosääntöjä minkä tahansa MOF-pohjaisen metamallin muuntamiseksi Document Type Definition (DTD) -muodossa annettavaksi rakennemäärittelyksi. Lisäksi kuvataan tuotantosääntöjä, jotka toteuttamalla varsinaiset metamallin mukaiset mallit (esimerkiksi UML-mallit) voidaan kuvata ko. DTD-määrittelyn mukaisessa XML-muodossa.

XML-pohjaiset kielet johtavat tyypillisesti hyvin laveisiin esityksiin ja niiden käsittely saattaa olla hidasta. Tämä pätee myös XMI:lle. Mikäli siirrettävät UML-mallit ovat isoja, muodostuu tiedostoistakin suuria. Toisaalta kokonaisten mallien ohella sallitaan myös mallifragmenttien kuvaaminen. Tämä ominaisuus on hyödyllinen käytännön tiedonsiirrossa, koska se sallii mallien siirtämisen epätäydellisenäkin.

Toinen tärkeä XMI:n ominaisuus on sen monipuoliset laajennettavuusmahdollisuudet. Koska XMI on nimenomaan tarkoitettu mallitiedon esittämiseen, se ei esimerkiksi sisällä mekanismeja UML-kaavioiden asettelutiedon esittämiseksi. Tämä on kuitenkin tarpeellista haluttaessa välittää kaavioita eri CASE-työkalujen kesken. XMI:n laajennettavuutta onkin käytetty laajasti hyväksi eri työkaluissa juuri kaavioiden asettelutiedon esittämiseksi. Toisaalta laajennettavuus sallii myös uusien työkalukohtaisten kaaviotyyppien tallettamisen yhdessä UML-mallien tai niiden osien kanssa.

Vaikka laajennettavuudella onkin ilmeiset etunsa, aiheuttaa se käytännössä huomattavia ongelmia tiedonsiirrolle. Osittain nämä vaikeudet johtuvat XMI-tuen puutteellisesta toteutuksesta työkaluissa ja osittain siitä, että XMI-määrittely ei millään tavalla rajoita tai määrittele itse laajennoksia tai niiden rakennetta.

XMI pyrkii tukemaan tiedon välitystä mahdollisimman hyvin siinäkin tapauksessa, että työkalun valmistajat ovat tehneet laajennoksia. Tavoitteena on, että muut työkalut voisivat käsitellä ja käyttää XMI-määrittelyn mukaista informaatiota jättäen ko. laajennokset huomioimatta ja tallettaa uudelleenmuokatut mallit jälleen XMI-muodossa hävittämättä alkuperäisiä laajennoksia. Tämä toteutetaan käyttämällä tunnisteita, joilla työkalukohtaiset muunnokset merkataan. Tämä ominaisuus toimii kuitenkin UML-työkaluissa käytännössä puutteellisesti.

Koska UML-kaavioihin liittyvän muunkin kuin malli-informaation siirtäminen on katsottu oleelliseksi, kaavaillaan OMG:ssä parhaillaan tiedonsiirtomuotoa nimeltä UML 2.0 Diagram Interchange, jonka tulisi keskittyä nimenomaan kaavioiden asettelutiedon esittämiseen ja jonka tulisi olla tavalla tai toisella linkitetty XMI:n kanssa.

XMI:n suosio perustuu luonnollisesti sille, että se on OMG:n kehittämä ku-

ten myös MOF ja UML. Onkin oletettavaa, että XMI:n kehitys tulee etenemään yhdessä UML:n kehityksen kanssa. Markkinoiden johtavat UML:ää tukevat työkalut lähes poikkeuksetta nykyään tukevatkin XMI:tä, tosin kukin omalla tavallaan. Puutteistaan huolimatta XMI on vakiinnuttanut asemansa erityisesti UML-pohjaisen mallitiedon tiedonsiirtomuotona.

6 UML-tutkimus

6.1 UML:n formalisoinnit

Esitystavan täydellisellä formalisoinnilla tarkoitetaan sen konkreettisen ja abstraktin kieliopin sekä staattisen ja dynaamisen semantiikan määrittelemistä. UML on visuaalinen kieli, joten tässä tapauksessa konkreettinen kielioppi tarkoittaa kielen graafista ulkoasua, eli rajoitteita sille, millaisia kaavioita ja visuaalisia elementtejä saa piirtää. Abstrakti kielioppi taas kuvaa käsitteelliset elementit ja niiden väliset suhteet. UML:ssä abstrakti kielioppi on kuvattu metamallilla, jossa käsitteellisiä elementtejä kuvataan metaluokilla. Metamallia on lisäksi täydennetty OCL:llä esitetyillä hyvinmuodostuneisuussäännöillä. Staattisella semantiikalla tarkoitetaan kuvausta käsitteellisten elementtien ja niiden välisten suhteiden staattisesta merkityksestä. Dynaaminen semantiikka selittää, mitä elementit kertovat järjestelmän käyttäytymisestä.

UML:ssä molemmat kieliopit on määritelty melko tarkasti, vaikka niiden välinen suhde onkin kuvattu puutteellisesti. Semantiikkaa, varsinkin dynaamista, on kuvattu melko vähän, ja silloinkin usein melko epämääräisesti. Suurin osa formalisointiehdotuksista ja -yrityksistä keskittyykin pelkästään semantiikan määrittelemiseen. Riippuen asianosaisten omista tutkimuskohteista ja mielenkiinnosta saattaa määrittely keskittyä UML:n pienen osajoukon formalisointiin ja silloinkin mahdollisesti vain sen staattiseen tai dynaamiseen semantiikkaan. Myös tällaisissa tapauksissa usein puhutaan UML:n formalisoinnista.

Useimmissa tapauksissa formalisoinnin taustalla ei ole halu tehdä UML:stä tarkkaan määritelty kieli, vaan tarve suorittaa erilaisia analysointi- ja päättelytoimenpiteitä kaavioille (tai niiden esittämille malleille) ja niiden ominaisuuksille. Esimerkiksi oikeellisuus-, turvallisuus- ja resurssienkäyttöominaisuuksien päättely ja todistaminen vaativat hyvin määritellyn semantiikan. Malleja voidaan myös haluta simuloida tai testata kokeellisesti joko havainnollistamista tai verifiointia varten.

Halutuista toimenpiteistä pitkälti riippuu, miten formalisointi suoritetaan. UML:ään joko lisätään valitun formalismin piirteitä tai määritellään kuvaus UML:stä tähän formalismiin. Yleistäen voi sanoa, että mitä monimutkaisempia ja formaalimpeja toimenpiteitä ollaan suorittamassa, sitä yleisemmin valitaan siirtyminen pois UML:stä. Tähän on monia syitä, mutta tärkein lienee se, että yksinkertaisesti tunnetaan joitain algoritmeja tai menetelmiä vaikkapa graafeille ja halutaan soveltaa niitä. Sen sijaan, että muu-

tettaisiin algoritmit toimimaan UML:ssä, kuvataan UML-kaavio graafina.

Formalisointiin on käytetty laajaa valikoimaa erilaisia formalismeja, matemaattisia työkaluja, kieliä, jne. Esimerkiksi luokkakaavioita on täydennetty Object-Z:llä [10] ja metamallia Petri-verkoilla [1] sekä monilla formaalisilla kielillä. Graafeja käytetään usein erilaisiin kaavioiden rakenteen tunnistus- ja analysointitoimenpiteisiin sekä ulkoasun muokkaamiseen. Luonnollisesti myös UML:n omaa rajoitekieltä, OCL:ää, käytetään paljon.

6.2 OCL-tutkimus

OCL on käyttökelpoinen kieli järjestelmän rakenteesta puhumiseen, mutta ei sovellu yhtä hyvin käyttäytymisen kuvaamiseen. Tämä johtuu osittain kielen historiasta, mutta osittain myös itse UML:n painottumisesta rakenteen kuvaamiseen. OCL on rajoitekieli eikä ohjelmointikieli, joten järjestelmän tila ei voi muuttua rajoitteen arvottamisen aikana. Tällaista ajasta irroitettua järjestelmän tilaa kutsumme *otokseksi* (snapshot). OCL-lauseke kohdistuu aina yhteen otokseen, eikä ole mahdollista viitata aikaisempien tai tulevien ajanhetkien otoksiin. Ainoa poikkeus on operaation jälkiehtorajoite, jossa voidaan viitata myös otokseen juuri ennen operaation suorituksen alkamista. Staattisen ja jopa dynaamisen rakenteen kuvaamiseen tämä usein riittää, mutta käyttäytymisen kuvaamisessa tieto ajasta ja menneistä tapahtumista ovat tärkeitä. OCL:ään onkin lisätty lähetettyjen viestien historia käsitteenä.

On siis mahdollista vaatia, että esim. operaation suorituksen päätyttyä on tullut lähetettyä tietyn muotoiset viestit. Ei kuitenkaan ole mahdollista sanoa jonkin muuttuneen “edellisestä” tai aikaisemmasta otoksesta, tai edes tarkastella aikaa, joka on kulunut tapahtumien välillä. Tällä hetkellä suurin osa itse kieleen liittyvästä tutkimuksesta keskittyykin lähinnä ajan ja historian käsitteen sisällyttämiseen kieleen [33, 5]. Tämä on välttämätöntä erityisesti rinnakkaisten järjestelmien ja prosessien kuvaamisessa [29]. Tällöin koko otos-ajatusmalli on vaarassa, koska on otettava kantaa esimerkiksi atomisiksi ajateltujen operaatioiden kestoon ja lomittumiseen.

OCL:n parempaa määrittelyä ei varsinaisesti tutkita, mutta kielen käyttäjät ovat suurimmaksi osaksi tutkijoita, jotka usein ehdottavat erinäköisiä pieniä korjauksia ja lisäyksiä törmätessään kielen rajoihin. Hiljattain kielen ytimeen on lisätty monikkotyyppi ja semantiikan perustaa on tukevoitettu juuri käyttäjien ehdotusten perusteella.

OCL:n kehittymisestä huolimatta sen yleisin käyttötarkoitus on edelleen rajoitteiden kuvaaminen. Sitä on kuitenkin käytetty moniin muihinkin tarkoituksiin, erityisesti erilaisiin haku- ja vertailuoperaatioihin. Vuorovaikutteisessa C++ debuggerissa [8] tulostettavien olioiden hakuehtoja sekä erilaisia vahteja annetaan OCL:llä. OCL for Java -sovellus [31] luo luokkakaavioon lisättyjen rajoitteiden perusteella automaattisesti ajoaikaisia tarkistuksia Java-koodiin. Jotkin käyttötavat ovat melko kaukana kielen alkuperäisestä

tarkoituksesta, esim. dynaamisten HTML-sivujen ohjelmointi OCL-kielellä [22].

6.3 Mallien oikeellisuus

UML tarjoaa yhtenäisen alustan, jonka päällä voidaan soveltaa olemassa olevia menetelmiä laillisuuden ja oikeellisuuden varmistamiseen. Samalla UML tuo mukanaan myös uusia ongelmia.

Määriteltäessä malli UML:n metamallin ilmentymäksi tehdään periaatteessa oletus tämän mallin laillisuudesta: malli tulkitaan hyvin muodostetuksi UML:n metamallin suhteen. Esimerkiksi UML 1.4 [17], joka on viimeisin OMG:n standardoima UML:n versio ja jota suurin osa nykyisistä UML CASE-työvälineistä pyrkii tukemaan, asettaa joukon käytännöllisessä mielessä tarpeettoman voimakkaita rajoitteita mallien rakenteelle. Tämä näkyy esimerkiksi siten, että UML 1.4:n metamalli epäsuorasti ohjaa suunnittelijan esittelemään määrittelytason mallielementit ennen niiden ilmentymiä. Tämä kuitenkin hankaloittaa kaavioiden avulla tapahtuvaa luonnostelua ja on näin ristiriidassa UML:n luonteen kanssa yleisenä mallinnuskielenä.

Tilanteen taustalla vaikuttaa suurempi ongelma: samalla kun UML ohjaa — joskin hienovaraisesti — valittua ohjelmistokehitysprosessia inkrementaaliseen suuntaan, se ei kuitenkaan ota kantaa siihen, kuinka tällaisia inkrementtejä yhdistetään olemassaoleviin malleihin. UML ei myöskään määrittele, millä hetkellä mallin tulee olla laillinen, mikä aiheuttaa hankaluuksia erityisesti työkalutuen toteuttajille. Työkalutuen ja UML:n tulkinnan kannalta keskeinen, joskin vähälle huomiolle jäänyt ongelma on lisäksi kaavioiden ja niitä vastaavien mallien epätarkasti määritelty suhde standardissa.

Edellä esitetyistä syistä johtuen eri CASE-työvälineiden valmistajat ovat ottaneet käyttöönsä erilaisia UML:n metamallin ja sen hyvinmuodostuneisuussääntöjen osajoukkoja, ja joutuneet lisäksi tekemään toisistaan poikkeavia oletuksia myös siitä, miten UML-malleja ja niitä vastaavia näkymiä käytännössä esitetään ja rakennetaan. Työkaluja arvioitaessa ja vertailtaessa tulee näin ollen kiinnittää huomiota myös siihen, voidaanko kaikki UML:n metamallin mukaiset mallit esittää työkaluissa, ja toisaalta voidaanko työkaluilla luoda metamallin kannalta laittomia malleja.

Riippumatta siitä, miten UML-mallien välinen oikeellisuus määritellään, tämän oikeellisuuden tarkastaminen tulee voida suorittaa työkalun tukemana. Yksi tapa tarkastaa mallin oikeellisuutta on asettaa sovellusalueesta riippuen joukko rajoitteita jotka sopiva työkalu voi pyydettäessä evaluoida. Näitä rajoitteita ja konventioita voidaan myös esittää käyttämällä UML:n profiileja.

UML:n oikeellisuutta on käsitelty lukuisissa konferensseissa ja workshopeissa, kuten virallisessa UML-konferenssisarjassa ja tämän yhteydessä pidetyissä mallien eheyteen keskittyneissä workshopeissa (esim. [13]). Lisäksi aihepiirin parissa työskentelee useita tutkimusryhmiä (esim. cUML [4] ja pUML [23]). Tuleva UML 2.0 tulee todennäköisesti korjaa-

maan osan edellä mainituista ongelmista, mutta nähtäväksi jää, missä määrin ja kuinka pian nämä muutokset heijastuvat tarjolla olevaan työkalutukeen.

6.4 Mallien käsittely

Samaa järjestelmää kuvaavat UML-kaaviot ovat tyypillisesti riippuvaisia toisistaan. Nämä riippuvuudet voivat syntyä, paitsi kaavioiden kuvatessa samoja käsitteitä, myös siirryttäessä ohjelmistoprosessin vaiheesta toiseen ja järjestelmän kehittyessä. Erityisesti inkrementaaliset ja iteratiiviset ohjelmistoprosessit korostavat mallien tuottamista vaiheittain ja pieninä lisäyksinä, jolloin on ilmeistä, että suunnittelutyössä kaavioiden välisiä riippuvuuksia tulisi pyrkiä mahdollisuuksien mukaan hyödyntämään — myös työkalutasolla.

UML-kaavioiden merkitys ja tulkinta on voimakkaasti sidoksissa työskentely-ympäristöön, sovellusalueeseen ja käytettyyn ohjelmistoprosessiin. Edellä mainittuja riippuvuuksia voidaan käyttää hyväksi esimerkiksi mallien yhdistämisen, viipaloinnin, syntetisoinnin ja tarkastamisen yhteydessä. Kaavioita yhdistettäessä yhden kaavion sisältämä tieto pyritään tuomaan esiin toiseen, mahdollisesti eri tyyppiseen kaavioon. Kaavioita viipaloidessa kaavioista esitetään vain sellainen osajoukko, joka on viipaloinnin perusteena käytettävän toisen kaavion esittämän näkökulman kannalta tarpeellinen. Kaavioita syntetisoitaessa pyritään luomaan uusia kaavioita olemassaolevien, tyypillisesti eri tyyppisten kaavioiden perusteella. Mallien tarkastaminen puolestaan tarkoittaa tässä yhteydessä annettujen kaavioiden keskinäisen ristiriidattomuuden toteamista.

Edellisen kaltaisia mallien käsittelytapoja tulisi tukea tarjoamalla suunnittelijalle ainakin osittain automatisoitua, räätälöitävissä olevaa työkalutukea. Tämän voidaan olettaa [12] tukevan nopeampaa ja helpompaa mallien luomista, kun kaavioita luodaan jo olemassa olevien kaavioiden pohjalta osin työkaluja käyttämällä. Vastaavasti mallien oikeellisuus voi kasvaa, kun toisaalta vältetään käyttäjän tekemiä virheitä siirrettäessä tietoa kaavioiden välillä ohjelmistoprosessin edetessä, ja toisaalta käyttäjällä on mahdollisuus käyttää työkaluja erilaisten tilapäisten näkymien generointiin, jolloin ymmärtämys suunniteltavasta järjestelmästä ja sitä kuvaavasta mallista kasvaa. Mallien yhdistäminen tukee suoraan inkrementaalisia ohjelmistoprosesseja. Lisäksi tällaiset työkalut, kun ne koostetaan yksinkertaisista primitiivioperaatioista [21], mahdollistavat räätälöitävän työkalutuen kehittämisen [20].

Vaikka tässä kuvatun kaltaiset mallien prosessointioperaatiot ja niistä koostettavat mallien käsittelytavat ja työkalut ovat osoittautuneet hyödyllisiksi [26], niiden määrittelyt eivät itsestään selvästi seuraa UML-standardista. Sekä varsinaisten operaatioiden määrittely että niiden toteutus ja yhdistely ovat, paitsi tutkimuksellisesti kiinnostavia, myös käytännön työkalutuen rakentamisen kannalta haastavia.

6.5 Prosessien tukeminen

UML-työkalut tukevat nykyisellään kehnosti ohjelmistokehitystä. Syynä tähän voidaan pitää osittain sitä, että UML on vain standardoitu kuvaustekniikka, eikä siten sisällä mitään ohjelmistokehitysprosessia tai -menetelmää. Tämä on työkaluvalmistajille siinä mielessä hyvä tilaisuus, että heillä on halutessaan vapaus valita millaisia toimintatapoja heidän työkalunsa tukee, mutta he voivat silti käyttää yleisesti käytössä olevaa kuvaustekniikkaa. Toisaalta työkalujen käyttäjät haluaisivat käyttää ja kehittää prosesseja, jotka sopivat parhaiten kulloiseenkin tarpeeseen.

UML ei itsessään sisällä mekanismeja ohjelmistoprosessien kuvaamiseen, mutta OMG on julkistanut tähän tarkoitukseen UML:ään vahvasti pohjautuvan standardin nimeltään Software Process Engineering Metamodel Specification (SPEM) [16], joka on kuvattu paitsi itsenäisenä metamallina, myös UML:n profiilina. Siten SPEM-malleja voidaan periaatteessa luoda ja käsitellä samoilla työkaluilla kuin UML-mallejakin.

SPEM on prosessien kuvaamiseen tarkoitetun kielen kuvaus, samassa mielessä kuin UML on tarkoitettu ohjelmistojen mallintamiseen. Se sisältää kielen prosessin staattisten osien kuvaamiseen ja on sillä tehdyt kuvaukset ajateltu lähinnä ihmisen luettavaksi ja ymmärrettäväksi, joten myös sen instanssit tukevat vain prosessien tai prosessimallien staattisia esityksiä ja mahdollisesti niiden erikoistamista. Esimerkiksi Rational Unified Process (RUP) [25] voidaan nähdä SPEM:n ilmentymänä. Vaikka Rational (IBM) on nimenomaan ohjelmistotyökalutoimittaja, RUP toimii lähinnä dokumenttina kulloisenkin prosessin kulusta sekä siihen liittyvistä artefakteista ja toimijoista, eikä prosessin automatisointiin ole varsinaisesti kiinnitetty sen yhteydessä huomiota. Jos prosessiin liittyykin jotain työkalujen tukemia vaiheita, työkalujen käyttö ei ole osa prosessia. Siten nykyisten työkalujen yhteydessä voidaan puhua prosessien automatisoinnista vain hyvin rajoitetussa mielessä.

Myös UML:ään liittyvä tutkimus on painottunut erillisiin työkaluihin ja operaatioihin, ja ohjelmistokehitystä kokonaisuutena tarkastelevaa tutkimusta on suhteellisen vähän. Periaatteessa esimerkiksi UML-mallien käsittelyä voidaan tukea työkaluilla hyvinkin vahvasti, kun tiedetään millaisen prosessin yhteydessä UML:ää kulloinkin käytetään ja ymmärretään prosessissa esiintyvien vaiheiden, mallien ja muiden artefaktien väliset suhteet. Erityisesti oliopohjaiset ohjelmistokehitysprosessit voidaan nähdä sarjana mallintamistehtäviä, joiden lopputuloksena syntyvillä malleilla on vahvoja riippuvuussuhteita. Tällöin yksittäisiä mallintarkastus tai -käsittelytehtäviä voidaan tukea mm. luvussa 6.4 kuvattujen mekanismien avulla. UML:n yhteydessä yksi prosessitukeen liittyvistä selkeistä kehityskohteista onkin näiden yksittäisten mekanismien ja työkalujen liittäminen sellaiseksi kokonaisuudeksi, että se ohjaa ja automatisoi prosessin suoritusta ja seurantaa, eikä pelkästään yksittäisiä tehtäviä.

7 Lopuksi

UML:ää on monella taholla kritisoitu, eikä täysin syyttä. Usein moititaan UML:n laajuutta ja monimutkaisuutta, mikä onkin ilmeinen ongelma: kieli näyttää kasvavan nopeammin kuin sitä ehditään omaksua. Samaan aikaan kun teollisuudessa opetellaan mallintamista yleensä ja UML:n perusasioita, OMG tuottaa alati kasvavia ja monimutkaistuvia UML:n versioita, joiden ymmärtäminen edellyttää tutkijatason asiantuntemusta. Tähän on reagoitu yrityksissä ottamalla käyttöön omia periaatteita siitä, mitä osaa UML:stä käytetään ja miten.

Jos UML olisi todella visuaalinen ohjelmointikieli, mihin suuntaan eräät tutkijat pyrkivät sitä viemään, se olisi tuskin koskaan saavuttanut merkittävää suosiota. UML:n vahvuus on kuitenkin siinä, että se on nimenomaan löyhästi määritelty mallinnuskieli, joka antaa tilaa monenlaisille käyttäjille ja käyttötavoille. UML:n nopea leviäminen teollisuuteen kertoo siitä, että tällaiselle mallinnuskielelle on ollut todellista tarvetta. UML:stä on tullut erilaisia ohjelmistokehityksen prosesseja ja artefakteja yhtenäistävä standardi, joka on osoittanut mallinnuksen hyödyt ja tehnyt mallinnuksesta yleisesti hyväksytyn osan ohjelmistokehitystä.

UML:n puutteena mainitaan myös usein, että se jättää mallien merkityksen vaille täsmällistä määrittelyä. Kuten edellä on todettu, tätä voidaan pitää myös eräänä UML:n suosion perusteena. Toisaalta OMG:ssa on pyrkimyksiä, jotka näyttävät johtavan tarpeeseen määritellä UML:n semantiikka hieman täsmällisemmin. Tällainen on erityisesti MDA (Model Driven Architecture), joka edustaa OMG:n visiota UML:n tulevaisuuden käyttömuodoista.

MDA:n tavoitteena on luoda infrastruktuuri, jonka avulla on mahdollista tehdä abstrakteja, ohjelmistoalustasta riippumattomia järjestelmämalleja siten, että niistä voidaan automaattisesti tuottaa tietyille ohjelmistoalustoille (esim. J2EE tai .NET) sovitettuja malleja (ja mahdollisesti myös lopullinen toteutus). Tässä toteutuu siis jälleen tietotekniikan yleinen pyrkimys nostaa järjestelmien kuvauksen abstraktiotasoa, tällä kertaa mallintamisen yhteydessä. MDA:han sisältyvä ajatus mallin automaattisesta transformaatiosta alemman abstraktiotason malliksi ja lopulta koodiksi edellyttää kuitenkin mallien semantiikan täsmällistä määrittelyä. Tätä voisi verrata ohjelmointikielen kääntäjään, joka tuottaa konekoodia. Kuinka pitkälle MDA:n visio on mahdollista toteuttaa yleisesti, on edelleen avoin kysymys.

Puutteistaan huolimatta — ja osittain niiden vuoksi — UML on erinomaisen hedelmällinen tutkimuskohde. Järjestelmien mallintaminen on alue, jonka merkitys tulee jatkuvasti kasvamaan järjestelmien monimutkaistuessa. Toisaalta UML on käsitteellisesti rikas maailma, joka kokoaa yhteen joukon tietotekniikan tärkeimpiä kuvaustekniikoita. Näin UML tarjoaa mahdollisuuden tutkia näiden ominaisuuksia ja keskinäisiä suhteita yhtenäisessä kehyksessä. Kun UML on onnistunut jo huomattavassa määrin leviämään teol-

lisuuteen, tutkimustuloksilla on välitöntä käytännöllistä merkitystä, ja niitä voidaan evaluoida todellisten järjestelmien yhteydessä. Tutkimustulokset on usein mahdollista pukea työkalun muotoon, joka helpottaa idean esittämistä ja motivoi tutkijaa. Vaikka UML ei olisikaan itse varsinaisena tutkimuskohteena, sitä voi ja kannattaa käyttää ohjelmistotekniikan tutkimuksen alustana.

Kiitokset: Kiitämme Ilkka Haikalaa ja Antti Valmaria hyödyllisistä kommenteista artikkelin aikaisempaan versioon. Artikkelin kirjoitusta on osittain rahoitettu Suomen Akatemian projektilta 51528.

Viitteet

[1] L. Baresi and M. Pezzè. On formalizing UML with high-level Petri nets. *Concurrent object-oriented programming and Petri nets: advances in Petri nets*, pages 276–304, 2001.

[2] G. Booch. *Object-Oriented Analysis and Design with Applications, 2nd Edition*. Addison-Wesley, 1993.

[3] Q. Chen, J. Grundy, and J. Hosking. An e-whiteboard application to support early design-state sketching of UML diagrams. In *Proceedings of the IEEE Symposium on Human Centric Computing Languages and Environments 2003*, pages 219–226. IEEE Computer Society, October 2003.

[4] Consistency UML Group, http://www.ipd.bth.se/consistencyUML. *Consistency UML Group Homepage*, 2004.

[5] S. Flake and W. Mueller. A UML profile for real-time constraints with the OCL. In *Proceedings of the 5th International Conference on The Unified Modeling Language*, pages 179–195. Springer-Verlag, 2002.

[6] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, June 1987.

[7] D. Harel and H. Kugler. Synthesizing state-based object systems from lsc specifications. *International Journal of Foundations of Computer Science*, 13(1):5–51, Febuary 2002. (Also in *Proc. 5th Int. Conference on Implementation and Application of Automata* (CIAA 2000), Springer-Verlag, pp. 1–33).

[8] C. Hobatr and B. A. Malloy. The design of an OCL query-based debugger for C++. In *Proceedings of the 2001 ACM symposium on Applied computing*, pages 658–662. ACM Press, 2001.

[9] J. R. Ivar Jacobson, Grady Booch. *The Unified Software Development Process*. Addison-Wesley, 1999.

[10] S.-K. Kim and D. Carrington. Formalizing the UML class diagram using Object-Z. In R. France and B. Rumpe, editors, *UML'99 — The Unified Modeling Language. Beyond the Standard. Second International Conference, Fort Collins, CO, USA, October 28–30. 1999, Proceedings*, volume 1723 of *LNCS*, pages 83–98. Springer-Verlag, 1999.

[11] R. Kollman, P. Selonen, E. Stroulia, T. Systä, and A. Zündorf. A study on current state of the art in tool-supported UML-based static reverse engineering.

In *Proceedings of the 9th Working Conference of Reverse Engineering (WCRE'2002)*, pages 22–34. IEEE Computer Society, October-November 2002.

[12] J. Koskinen, J. Peltonen, P. Selonen, T. Systä, and K. Koskimies. Towards tool assisted UML development environments. In T. Gyimóthy, editor, *Proceedings of the 7th Symposium on Programming Languages and Tools (SPLST'01)*, pages 1–15. University of Szeged, June 2001.

[13] L. Kuzniarz, G. Reggio, J. Sorrouille, and Z. Huzar. Proceedings of the Workshop on Consistency Problems in UML-based Software Development. Blekinge Institute of Technology Research Report 2002:06, 2002.

[14] S. Lahtinen and J. Peltonen. Enhancing usability of UML case-tools with speech recognition. In *Proceedings of the IEEE Symposium on Human Centric Computing Languages and Environments 2003*, pages 227–235. IEEE Computer Society, October 2003.

[15] Microsoft, http://office/microsoft.com/visio. *Microsoft Visio*, 2004.

[16] Object Management Group. *Software Process Engineering Metamodel Specification*, November 2002. Version 1.0.

[17] Object Management Group, Inc. *The Unified Modeling Language (UML) Specification (Action Semantics), Final Adopted Specification version 1.4*, January 2002. Version 1.4.

[18] Object Management Group, Inc. *XML Metadata Interchange (XMI) Specification*, 2002. Version 1.2.

[19] Object Management Group, Inc. *Unified Modeling Language: Superstructure version 2.0 Final Adopted Specification ptc/03-08-02*, August 2003. Version 2.0.

[20] J. Peltonen and P. Selonen. Processing UML models with visual scripts. In *Proceedings of the 2001 Human-Centric Computing Languages and Environments (HCC'01)*, pages 264–271. IEEE Computer Society, September 2001.

[21] J. Peltonen and P. Selonen. An approach and a platform for building UML model processing tools. In *Proceedings of the ICSE'04 Workshop on Directions of Software Engineering Environments (WoDiSEE'04)*, May 2004. To appear.

[22] J. Pleumann and S. Haustein. A model-driven runtime environment for web applications. In P. Stevens, J. Whittle, and G. Booch, editors, *UML 2003 - The Unified Modeling Language. Model Languages and Applications. 6th International Conference, San Francisco, CA, USA, October 2003, Proceedings*, volume 2863 of *LNCS*, pages 190–204. Springer-Verlag, 2003.

[23] The precise UML group, http://www.cs.york.ac.uk/puml/. *pUML group homepage*, 2004.

[24] Rational Software, http://www-306.ibm.com/software/awdtools/developer/modeler. *Rational Rose XDE Modeler*, 2004.

[25] Rational Software, http://www-306.ibm.com/software/awdtools/rup. *Rational Unified Process*, 2004.

[26] C. Riva, P. Selonen, T. Systä, A.-P. Tuovinen, J. Xu, and Y. Yang. Establishing a software architecting environment. In *Proceedings of the 4th Working*

IEEE/IFIP Conference on Software Architecture (WICSA'04), June 2004. To appear.

[27] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-oriented Modeling and Design*. Prentice Hall, 1991.

[28] P. Selonen and J. Xu. Validating UML models against architectural profiles. In *Proceedings of the 9th European software engineering conference held jointly with 10th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 58–67. ACM Press, 2003.

[29] S. Sendall and A. Strohmeier. Specifying concurrent system behavior and timing constraints using OCL and UML. In M. Gogolla and C. Kobryn, editors, *UML 2001 - The Unified Modeling Language. Modeling Languages, Concepts, and Tools. 4th International Conference, Toronto, Canada, October 2001, Proceedings*, volume 2185 of *LNCS*, pages 391–405. Springer-Verlag, 2001.

[30] T. Systä. *Static and Dynamic Reverse Engineering Techniques for Java Software Systems*. PhD thesis, University of Tampere, 2000.

[31] Technische Universität Dresden, http://dresden-ocl.sourceforge.net/. *Dresden OCL Toolkit, OCL for Java*, May 2002.

[32] J. Whittle and J. Schumann. Generating statechart designs from scenarios. In *Proceedings of the 22nd international conference on Software engineering*, pages 314–323. ACM Press, 2000.

[33] P. Ziemann and M. Gogolla. OCL Extended with Temporal Logic. In M. Broy and A. Zamulin, editors, *5th International Conference Perspectives of System Informatics (PSI'2003)*, volume 2890 of *LNCS*, pages 351–357. Springer-Verlag, Berlin, 2003.