TAMPEREEN TEKNILLINEN KORKEAKOULU

Tietotekniikan osasto

JOHANNES KOSKINEN

MAS-ALGORITMIN TOTEUTUS WINDOWS NT -YMPÄRISTÖSSÄ

Diplomityö

Aihe hyväksytty osastoneuvoston kokouksessa
11.10. 2000
Tarkastajat: FT. Tarja Systä (TTKK)
Prof. Kai Koskimies (TTKK)

ALKULAUSE

Olen tehnyt diplomityöni toimiessani tutkimusapulaisena Tampereen teknillisen korkeakoulun Digitaalisen median instituutissa. Työni on osa DMI:ssä toimivaa ATOS-projektia.
Esitän kiitokseni työni tarkastajille, FT Tarja Syställe ja prof. Kai Koskimiehelle. Lisäksi kiitän erityisesti FT Erkki Mäkistä (Tampereen yliopisto) teoriaosuuden ja kieliasun huolellisesta tarkastuksesta sekä yleisistä kommenteista työhöni liittyen. Kiitän myös vaimoani Anua ja tytärtäni Kaisaa henkisestä tuesta kotona.

Tampereella 4.12. 2000

Johannes Koskinen

Näyttelijänkatu 19 E 18

33720 TAMPERE

050-3202372

Sisällysluettelo

ALKULAUSE II

SISÄLLYSLUETTELO III

TIIVISTELMÄ VI

ABSTRACT VII

LYHENNELUETTELO JA TERMINOLOGIA VIII

1. JOHDANTO 1

2. KÄYTTÄYTYMISEN MALLINTAMINEN JA SYNTETISOINTIALGORITMIT 3

2.1 KÄYTTÄYTYMISEN MALLINTAMINEN UML-NOTAATIOLLA 3

2.2 SYNTETISOINTIALGORITMIT YLEISESTI 7

2.3 VUOROVAIKUTTEISET SYNTETISOINTIALGORITMIT MAT JA MAS 10

2.3.1 Minimally Adequate Teacher (MAT) 10

2.3.2 Minimally Adequate Synthesizer (MAS) 14

3. SYNTETISOINTIOHJELMAN TOIMINTAYMPÄRISTÖ 19

3.1 YLEISTÄ YMPÄRISTÖSTÄ 19

3.2 HAJAUTETUT KOMPONENTIT 19

3.3 TDE EDITOR (TED) 24

3.3.1 TEDin toiminta 25

3.3.2 Syntetisointiohjelman liittyminen TEDiin 26

4. OHJELMISTON TOTEUTUS 28

4.1 KEHITYSYMPÄRISTÖ 28

4.2 Luokka- ja moduulijako 28

4.3 Testaus- ja kehitysalusta 29

4.4 Syntetisoijan toteutus 29

4.4.1 Syntetisoijan luokat 30

4.4.2 Tilakoneen luokat 33

4.4.3 Syntetisoijaluokan käyttö 34

4.4.4 Tilakoneluokan käyttö 36

4.4.5 Muita luokkia 38

4.5 Käyttöliittymän toteutus 41

4.5.1 Käyttöliittymän eri osat 41

4.5.2 Käyttöliittymä ennen algoritmin suoritusta 42

4.5.3 Käyttöliittymä syntetisointialgoritmin suorituksen aikana 43

4.5.4 Käyttöliittymä syntetisointialgoritmin suorituksen jälkeen 44

5. KOKEMUKSIA JÄRJESTELMÄN TOIMINNASTA 46

5.1 Syntetisointiohjelman ajoesimerkki 46

5.2 Suorituskyky käytännön aineistolla 54

5.2.1 Nopeuden pullonkaulat 55

5.3 Syntetisoinnin teoreettinen nopeus 57

6. SYNTETISOINTIALGORITMIEN VERTAILU 60

6.1 SCED ja SMS 60

6.2 Ajoitetut skenaariot 62

6.3 Lisätietoa syntetisointiin OCL:llä 63

6.4 Reaaliaikainen mallinnus (ROOM) 63

7. JATKOKEHITYS 65

7.1 KÄYTTÖLIITTYMÄN JATKOKEHITYS 65

7.1.1 Käyttöliittymän integrointi *65*

7.1.2 Käyttäjäkysymykset *66*

7.1.3 Muita parannusehdotuksia *68*

7.2 SYNTETISOINNIN JATKOKEHITYS 68

7.2.1 Kielletyt merkkijonot *68*

7.2.2 Oletukset *69*

7.2.3 Epävarmat vastaukset *69*

7.3 TOTEUTUKSEN JATKOKEHITYS 70

7.3.1 Ohjelman ulkoinen käyttö *70*

7.3.2 Ohjelman suoritusnopeus *72*

8. YHTEENVETO 73

LÄHDELUETTELO 75

LIITTEET 79

LIITE 1 80

LIITE 2 81

LIITE 3 83

LIITE 4 84

TIIVISTELMÄ

TAMPEREEN TEKNILLINEN KORKEAKOULU

Tietotekniikan osasto

Ohjelmistotekniikka

KOSKINEN, JOHANNES: MAS-algoritmin toteutus Windows NT -ympäristössä

Diplomityö, **78** s., **11** liites.

Tarkastajat: FT Tarja Systä, prof. Kai Koskimies

Joulukuu 2000

Avainsanat: Software design, Minimally adequate teacher, Minimally adequate synthesizer, Synthesis algorithm, Sequence Diagram, Statechart Diagram

Olioperustainen määrittely ja suunnittelu (OOAD) sekä sitä tukeva mallinnuskieli Unified Modeling Language (UML) ovat laajasti käytössä ohjelmistotuotannossa. Suunnittelijan työtä helpottamaan on kehitetty erilaisia automaattisia menetelmiä, joilla voidaan muuntaa suunnittelussa käytettäviä kaavioita toiseksi ja tarkistaa kaavioiden välisiä riippuvuuksia. Minimally Adequate Synthesizer (MAS), joka perustuu Angluinin Minimally Adequate Teacher -algoritmiin, on yksi tällainen algoritmi. Se muuntaa UML-sekvenssikaavioita (skenaarioita) tilakaavioiksi. MAS:in etuna muihin algoritmeihin on sen vuorovaikutteisuus käyttäjän kanssa, millä vältetään syntetisointialgoritmeissa yleisesti esiintyvä tilojen liiallinen yleistys.

Diplomityössä toteutettiin MAS-algoritmi Windows NT-ympäristöön. Sen lähtö- ja tulostietoihin käytettiin Nokian TED-työkalua, joka on monen käyttäjän UML-mallinnusohjelma. Lisäksi toteutettiin kaksi erilaista käyttöliittymää syntetisointialgoritmin käyttöön. Yhteydet TED:iin hoidettiin COM-rajapinnan kautta.

Ajettaessa algoritmilla erilaisia testiskenaarioita huomattiin, että MAS:in sisäinen toteutus mahdollistaa toimimisen lähes kokonaan ilman käyttäjän antamia syötteitä. Lisäksi havaittiin, että kysymysten määrää voidaan vähentää antamalla erilaisia oletuksia algoritmille. Toisaalta käyttöliittymän toimivuudessa TED:in kanssa on ongelmia. Mallinnusohjelma ja syntetisointiohjelma toimivat toisistaan täysin erillisinä, jolloin helppokäyttöisen liittymän tekeminen on vaikeaa. Kyselyiden vähentämiseksi ja käyttöliittymän yhdistämiseksi mallinnusohjelman kanssa on työssä esitetty parannusehdotuksia.

ABSTRACT

TAMPERE UNIVERSITY OF TECHNOLOGY

Department of Information Technology

Software Systems Laboratory

KOSKINEN, JOHANNES: Implementing MAS on Windows NT

Master of Science Thesis, **78** pages, **11** enclosure pages.

Examiner: Ph.D. Tarja Systä, prof. Kai Koskimies

December 2000

Keywords: Software design, Minimally adequate teacher, Minimally adequate synthesizer, Synthesis algorithm, Sequence Diagram, Statechart Diagram

Object-oriented analysis and design (OOAD) and Unified Modeling Language (UML) are widespread in software design. There are a number of different synthesis algorithms, which are used to translate diagrams. Minimally Adequate Synthesizer (MAS), which is based on Angluin's algorithm of Minimally Adequate Teacher, is such an algorithm. It is used to synthesize statechart diagrams from a set of sequence diagrams (scenarios). One of the advantages of MAS algorithm is that it is interactive, so we can avoid overgeneralizations, which are common with other algorithms.

In this thesis MAS algorithm was implemented within Windows NT environment. For the input and output of the algorithm, we use TED (a multi user UML modeling tool) by Nokia Research Center. We also implement two user interfaces for the synthesis algorithm. Connections to TED are handled with COM interface.

During the test runs we noticed that internal structures of MAS algorithm allow MAS to operate almost without any help by the user. We also found out that by using special presumptions for the algorithm, we can further decrease the need for user help. On the other hand, there were problems with the user interface and TED. Modeling software and synthesizer are completely separate and thus integrating user interfaces is difficult. The present paper includes some suggestive improvements.

Lyhenneluettelo ja terminologia

ATOS	Advanced Tools for Object-oriented Software Development
bMSC	basic MSC
CORBA	Common Object Request Broker Architecture
COM	Component Object Model
CS-RCS	ComponentSoftware RCS
DepUml	Design Environment Plugin for UML
DCOM	Distributed Component Object Model
GUID	Globally Unique Identifier
HMSC	High-Level MSC
IDL	Interface Definition Language
IID	Interface ID
MSC	Message Sequence Chart
MFC	Microsoft Foundation Class Library
MAS	Minimally Adequate Synthesizer
MAT	Minimally Adequate Teacher
OCL	Object Constraint Language
OMG	Object Management Group
OOAD	Object-Oriented Analysis and Design
RCS	A Revision Control System
ROOM	Real-time Object-Oriented Modeling
SCED	SCenario EDitor
STL	Standard Template Library
SMS	State Machine Synthesis
TED	Tde EDitor
UML	Unified Modeling Language

1. Johdanto

Olioperustainen määrittely ja suunnittelu (OOAD) ja standardiksi muodostunut UML-notaatio ovat nykyisin käytössä lähes kaikkialla ohjelmistotuotannossa. Ohjelmistosuunnittelu on vielä pitkälti käsityötä, eikä markkinamiesten lupaamia yleiskäyttöisiä ja täysin automaattisia koodintuottamisvälineitä ole kovinkaan paljon ilmaantunut ohjelmoijien avuksi. Suunnitteluvaiheessa tarvittavia mallinnustyökaluja on kuitenkin pystytty automatisoimaan esimerkiksi erilaisten kaaviomuunnosten osalta.

TTKK:lla toimiva ATOS-projekti (Advanced Tools for Object-oriented Software Development) pyrkii kehittämään edistyneitä apuvälineitä juuri UML-pohjaiseen suunnitteluun. Sen yhtenä tavoitteena on löytää mekanismit UML-mallien muunnokseen ja syntetisointiin. Osana tätä projektia on kehitetty vuorovaikutteinen MAS-algoritmi, joka syntetisoi annetuista UML-sekvenssikaavioista tilakoneen.

Tämän työn päätarkoituksena oli toteuttaa edellä mainittu MAS tietokoneelle siten, että itse algoritmin testaus olisi mahdollista. Käytössä oli lisäksi Nokian TED-editori, johon syntetisointialgoritmi oli tarkoitus yhdistää. Tarkempia reunaehtoja ei työn alussa annettu vaan toteutustapa jätettiin tarkoituksella avoimeksi.

Koska työ keskittyy algoritmin toteutukseen, ei esimerkiksi käyttöliittymään ole kiinnitetty kovinkaan suurta huomiota. Jos algoritmi siirretään toimimaan jonkin ohjelman alaisuudessa, muuttuu käyttöliittymä tietysti ratkaisevasti. Siksi nykyisen käyttöliittymän toteutus on tarkoitettu tukemaan nimenomaan algoritmin tutkimuskäyttöä.

Tämä työ alkaa teoriaosuudella (luku 2), jossa esitellään käytetty UML-notaatio ja syntetisointialgoritmien yleiset periaatteet. Lisäksi kerrotaan tarkemmin työssä käytetyn MAS-algoritmin toimintaperiaatteet sekä tarvittavat syötteet ja algoritmista saatavat tulosteet.

Kolmannessa luvussa esitellään se ohjelmistoympäristö, jossa algoritmi toteutettiin. Syntetisointiohjelma on tarkoitettu toimimaan osana laajempaa kokonaisuutta, joten eri ohjelmien periaatteet ja se, miten nämä ohjelmat kommunikoivat keskenään, on esitetty tässä luvussa.

Neljäs luku on varattu toteutukselle. Tarkkaa ohjelmistodokumentaatiota ei tästä työstä ole kuitenkaan tarkoitus tehdä, vaan luvussa kerrotaan pääpiirteittäin se, miten algoritmi on toteutettu, ja annetaan esimerkkejä siitä, mitä tällä toteutuksella voi tehdä.

Viidennessä luvussa kuvataan, miten syntetisointi toteutetulla ohjelmalla etenee annetuista lähtötiedoista haluttuun lopputulokseen. Lisäksi viidennessä luvussa tutkitaan ohjelman ja algoritmin suorituskykyä sekä teoriassa että käytännössä. Samalla esitetään nopeuden pullonkaulat eli se, mistä kannattaa suoritusnopeutta lähteä parantamaan.

Erilaisia syntetisointiohjelmia vertaillaan kuudennessa luvussa. Lukuun on koottu samantapaisia algoritmeja kuin mikä tässä on toteutettu. Algoritmit ovat kuitenkin hieman erilaisia esimerkiksi lähtötietojen tai painotustensa osalta, eikä täysin vastaavaa algoritmia löytynyt.

Toteutuksen ja testauksena aikana on tullut esille asioita, joita kannattaisi tutkia enemmän. Lisäksi nykyinen toteutus ei välttämättä ole vielä valmis. Jatkokehitystä varten viimeiseen lukuun onkin koottu eri osa-alueiden parannusehdotuksia.

Työssä on pyritty käyttämään vakiintuneita suomennoksia [7,34] englanninkielisille termeille. Kuitenkin myös alkuperäinen nimi on esitetty suluissa, silloin kun se on katsottu tarpeelliseksi.

2. Käyttäytymisen mallintaminen ja syntetisointialgoritmit

Tässä luvussa käsitellään olioperusteisen määrittelyn ja suunnittelun (OOAD) ympäristöä. Tärkein näistä on kuvauksessa käytetty merkintätapa, jota käsitellään ensimmäisessä kohdassa. Toisessa kohdassa on lyhyesti kerrottu syntetisointialgoritmien yleisiä periaatteita. Lopuksi esitetään tarkemmin vuorovaikutteinen algoritmi MAS.

Peruskäsitteet, jotka liittyvät formaaleihin kieliin (esimerkiksi *säännölliset kielet*, *äärellinen automaatti* ja *tilakone*) ja algoritmien analysointiin (kuten *aika- ja tilakompleksisuus*) oletetaan tässä työssä tunnetuiksi. Niistä käytetään samoja suomenkielisiä nimityksiä kuin lähteissä [7,34,36].

2.1 Käyttäytymisen mallintaminen UML-notaatiolla

Ohjelmistojen suunnitteluun ja dokumentointiin on vuosien varrella kehitetty useita eri mallinnusmenetelmiä. Oliokeskeisille menetelmille tärkeimpinä ovat olleet erilaiset luokka-, sekvenssi- ja tilakaaviot. Kuitenkin eri menetelmien käyttämät notaatiot ovat vaihdelleet melkoisesti. Tällä hetkellä UML on laajasti hyväksytty teollisuusstandardiksi oliokeskeisten ohjelmien mallinnusvälineeksi, ja se sopiikin varsin hyvin käytettäväksi melkein minkä tahansa menetelmän kanssa. Object Management Group (OMG) hyväksyi [27] vuonna 1997 UML:n standardiksi.

UML on yhdistelmä vuosikymmenien varrella kehitetyistä mallinnusmenetelmistä (kuten ER-kaaviot ja SDL) ja se muistuttaa erityisesti aikaisemmin yleisesti käytettyä OMT-notaatiota [34]. Se sisältää laajan valikoiman erilaisia notaatioita, jotka tukevat ohjelmistokehitystä vaatimusten määrittelystä toteutukseen.

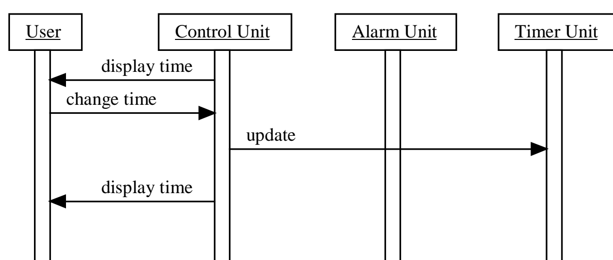
Tapahtumasekvenssikaaviot (Sequence diagrams)

Usein ohjelmistosuunnittelussa on tarve kuvata eri olioiden (luokkien ja rajapintojen ilmentymiä, fyysisiä laitteita) keskinäistä vuorovaikusta. Tällöin käytetään tapahtumase-

kvenssikaaviota (jatkossa sekvenssikaavio). Sekvenssikaaviot ovat olleet käytössä aikaisemminkin erityisesti tietoliikennetekniikassa kuvattaessa eri laitteiden välistä viestitystä.

Sekvenssikaaviot näyttävät olioiden välisen vuorovaikutuksen ajan suhteen, mutta ne eivät kerro mitään olioiden muista suhteista. Tähän tarkoitukseen käytetään muita kaaviotyyppejä, kuten esimerkiksi luokkakaaviota.

Sekvenssikaavioita sanotaan usein myös skenaarioiksi, vaikka skenaario sinänsä on yleisnimi tapahtumasarjan kuvaukselle ja sekvenssikaavio on vain yksi mahdollinen skenaarioiden kuvaustapa. Skenaarioiden kuvaukseen voidaan käyttää myös esimerkiksi yhteistyökaavioita (*collaboration diagrams*) tai suorasanaista kuvausta.



Kuva 2.1. Tapahtumasekvenssikaavio

Kuva 2.1 esittää erästä sekvenssikaaviota. Sekvenssikaavio koostuu olioista (pystyviivat, kuvassa *User*, *Control unit*, *Alarm Unit* ja *Timer Unit*) ja olioiden toisilleen lähettämistä viesteistä (vaakanuolet, kuvassa *display time*, *change time* ja *update*). Viestit voivat olla esimerkiksi funktiokutsuja ja niiden palauttamia paluuarvoja tai esimerkiksi prosessien toisilleen lähettämiä viestejä. Sekvenssikaavioissa aika kulkee ylhäältä alaspäin yleensä siten, että vain suoritusjärjestyksellä on väliä eikä tarkkoja suoritusaikoja anneta.

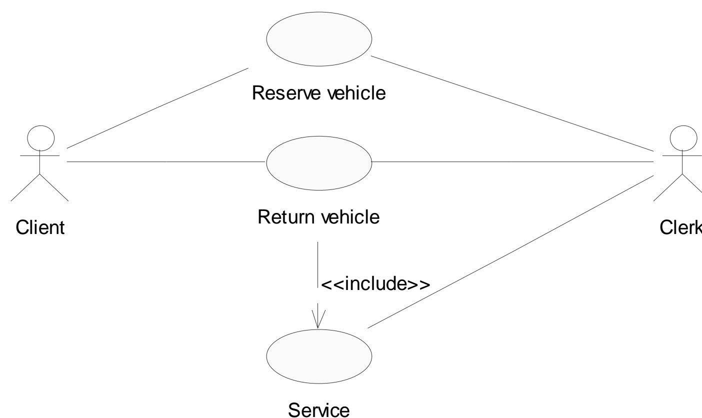
Sekvenssikaavio muodostetaan yleensä vain yhdestä tapauksesta kerrallaan ja esitettävän asian kaikkien mahdollisten etenemispolkujen kuvaaminen onkin yleensä mahdotonta. Tästä syystä UML-lähtöisessä mallinnuksessa käytetään käyttötapauksia, joissa kuvataan esimerkiksi sekvenssikaavioilla ensin onnistuneesti suoritettuja tapauksia (eli miten normaalisti tapahtuu) ja tämän jälkeen esitetään erilaisia epätavallisia (eli esimerkiksi virhetilanteet) tapauksia.

Käyttötapauskaaviot (Use case diagrams)

Käyttötapauksilla mallinnetaan järjestelmän toiminnallisuus joukkona järjestelmän käyttäjien (*actors*) sillä suorittamia tapahtumaketjuja [34]. Käyttötapauksen kuvaus voi olla pelkkää tekstiä tai esimerkiksi joukko sekvenssikaavioita. Järjestelmän eri käyttötapaukset ja käyttäjäroolit kuvataan käyttötapauskaaviossa.

Käyttötapauskaavioon voidaan merkitä myös käyttötapauksien välisiä suhteita (kuten esimerkiksi käyttö- ja laajennussuhde (*include, extend*)). Laajennussuhteella voidaan kuvata esimerkiksi käyttötapauksen erikoistapaus yhtenä käyttötapauksena kun taas käyttösuhteella eri käyttötapauksien yhteinen osa voidaan siirtää omaksi käyttötapaukseksi.

Käyttötapaus alkaa jonkin käyttäjäroolin aloitteesta ja päättyy, kun käyttäjä on saanut tehtäväkokonaisuuden suoritettua. Kuvatut tehtäväkokonaisuudet ovat yleensä laajoja (esimerkiksi hissillä matkustaminen toiseen kerrokseen tai herätyksen asettaminen herätyskellossa). Kuva 2.2 esittää esimerkin käyttötapauskaaviosta.



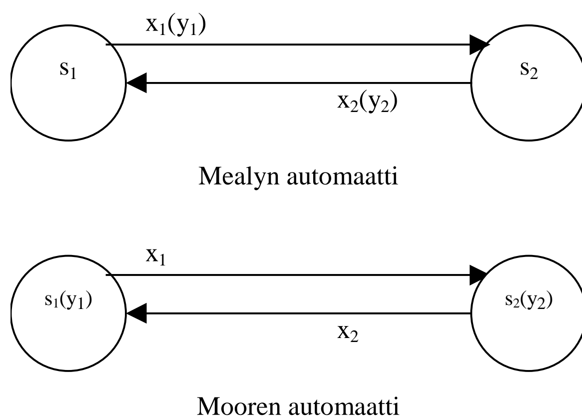
Kuva 2.2. Esimerkki käyttötapauskaaviosta

Tilakaaviot (Statechart diagrams)

Tilakaavio on kaavio, joka kuvaa tilakoneen. Tilakone (tila-automaatti) on järjestelmä, jossa on joukko tiloja (esimerkiksi lamppu päällä/pois) ja tilasiirtymiä (sytytä lamppu, sammuta lamppu) tilojen välillä. Tila-automaattien teoriaan voi tutustua tarkemmin esimerkiksi monisteesta `Automaattiteoria` [36].

Tunnetuimmat tilakonetyypit ovat Mooren ja Mealyn tilakone (katso kuva 2.3). Koska molemmat kuvaavat kuitenkin samaa asiaa, onnistuu tilakoneen mallin muuttaminen kuitenkin suhteellisen helposti [36]. Suurin ero näiden kahden välillä on se, että Mooren automaatissa tulostus (tila-automaateissa puhutaan tulosteista, eli automaatti "tulostaa" merkin, kun taas ohjelmoinnissa tulostusta on järkevämpi ajatella esimerkiksi funktiokutsuna) tapahtuu automaatin tilassa, kun taas Mealyn automaatissa tulostus tapahtuu tilasiirtymän yhteydessä.

Tarkemmin määritellen Mealyn kone on järjestetty viisikko ME=(I,O,S,f,φ), missä I ja O ovat äärelliset syöte- ja tulostemerkkien aakkostot, S äärellinen ei-tyhjä tilojen joukko, siirtymäfunktio f on joukon $S \times I$ kuvaus joukkoon S ja tulostusfunktio φ on joukon $S \times I$ kuvaus joukkoon O. Järjestetty pari (ME,s) on alkutilan omaava Mealyn kone. Mooren kone on järjestetty viisikko MO=(I,O,S,f,ψ), missä I,O,S ja f ovat määritellyt kuten ME:ssä, mutta ψ on joukon S kuvaus joukkoon O. Järjestetty pari (MO,s) on alkutilallinen Mooren kone. [36]

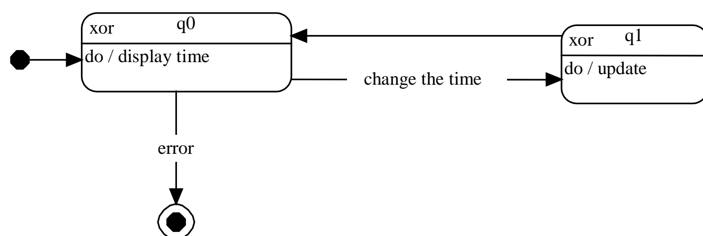


Kuva 2.3. Erilaisia tila-automaatteja

Tilakaavioiden esittämiseen on käytössä lukuisia toisistaan poikkeavia kuvausmenetelmiä, kuten SA-kaaviot (kuvaavat Mealyn automaattia) ja tilasiirtymämatriisit. UML-tilakaaviot perustuvat Harelin tilakaavionotaatioon [8], jolla voidaan kuvata sekä Mealyn että Mooren automaatit ja niiden yhdistelmät. Lisäksi Harelin notaatio sisältää SA-menetelmään verrattuna lukuisia laajennuksia, joista tärkeimpinä mainittakoon rinnakkaiset tilat sekä tilojen yhdistämiseen käytetyt ylitilat. Harelin tilakaaviossa voidaan myös määritellä tarkem-

min, milloin toiminto tapahtuu (eli tapahtuuko se tilaan tullessa, tilassa oltaessa vai tilasta lähtiessä).

Esimerkki UML-tilakaaviosta on alla (kuva 2.4). Tilakaaviossa huomataan myös ehdoton (*quarless*) tilasiirtymä (q_1->q_0), joka suoritetaan automaattisesti heti, kun tilan toiminto on tehty. Vastaavasti nähdään alkutila, josta siirrytään tilaan q_0 ja lopputila (q_0:sta *error*-tilasiirtymä).



Kuva 2.4. UML-notaation mukainen tilakaavio

Tilakoneita käytetään yleensä esittämään järjestelmän jonkin olion käyttäytymistä koko sen elinkaaren aikana. Tilakoneita käytetään erityisesti tietoliikenneprotokollien mallintamiseen (jolloin tiloja ovat esimerkiksi ”viesti lähetetty”, ”kuittaus saatu”, ”aikakatkaisu”, ”virheellinen viesti”). Siinä, missä sekvenssikaavio kuvaa vain yhden skenaarion (tai ei edes sitä) kerrallaan, tilakone näyttää kerralla kaikki sallitut tilasiirtymät, mutta vain yhden sekvenssikaavion olion kannalta kerrallaan. Tilakaavioiden käyttöä skenaarioiden mallinnuksessa on käsitelty tarkemmin lähteessä [5].

2.2 Syntetisointialgoritmit yleisesti

Sekvenssikaavion ja tilakoneen välillä vallitsee selvä sukulaisuussuhde. Sekvenssikaavion yhdelle oliolle voidaan luoda vastaava tilakaavio, jolloin tilasiirtymät tulevat tilakaavion olioon tapahtuvien tilasiirtymien perusteella. Sekvenssikaavion perusteella voidaan luoda tilakaavio, mutta ei aina yksikäsitteisesti. Vastaavasti tilakone voi tuottaa useita erilaisia sekvenssikaavioita.

Suunnittelijan täytyy usein käyttää molempia kaaviotyyppejä, joten automaattinen tilakoneen luonti sekvenssikaavioista (Message Sequence Charts eli MSC) tarjoaa suunnittelijalle mahdollisuuden keskittyä toiseen kaaviotyyppiin ja tarkistaa tulokset helposti tilako-

neella. Tällaisia automaattisia työkaluja onkin kehitetty useita (katso tarkemmin luku 6). Myös sekvenssikaavion ja luokkakaavion sekä luokkakaavion ja tilakaavion [10] välille on kehitetty erilaisia muuntimia. Näitä ei kuitenkaan käsitellä tässä työssä sen tarkemmin.

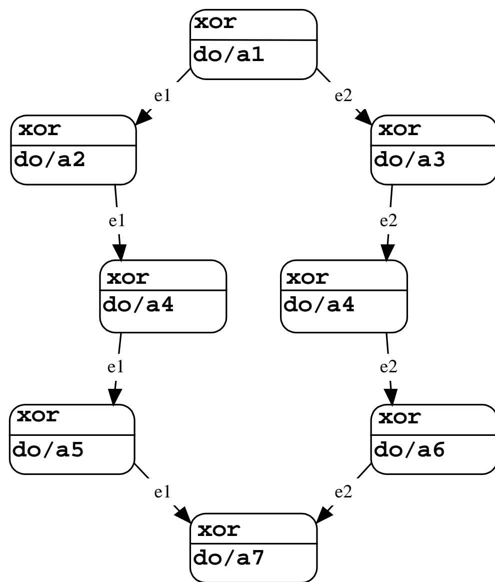
Koska sekvenssikaavio kuvaa vain yhden skenaarion, on tilakone yleistys annetuista sekvenssikaavioista eli tilakone hyväksyy useamman sekvenssikaavion kuin syntetisoinnissa sille on annettu. Vaikka tämä onkin yleensä haluttu lopputulos, ei se välttämättä ole hyvä ominaisuus. Useamman kaavion antaminen tarkentaa annettua tilakonetta useimmissa tapauksissa, mutta liian yleistetyn tilakoneen korjaaminen jää suunnittelijan vastuulle. Lisäksi epätarkat tai virheelliset sekvenssikaaviot tuottavat yleensä täysin erilaisen lopputuloksen kuin mitä suunnittelija on tarkoittanut. Tätä varten onkin kehitetty tässä työssä esitetty vuorovaikutteinen algoritmi, joka kysyy käyttäjältä neuvoa ennen yleistyksiä.

Eräs esimerkki yleistyksestä liittyy tilakoneeseen (kuva 2.6), joka hyväksyy kaksi eri polkua [11]:

$$(a_1, e_1)(a_2, e_1)(a_4, e_1)(a_5, e_1)(a_7, null)$$

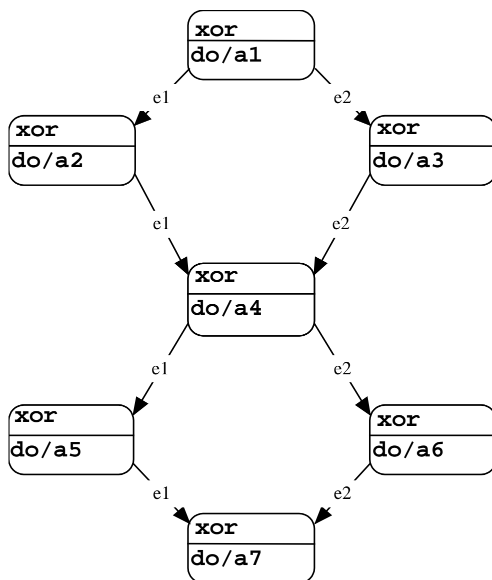
ja

$$(a_1, e_2)(a_3, e_2)(a_4, e_2)(a_6, e_2)(a_7, null)$$



Kuva 2.5. Oikein muodostettu tilakone

Kun algoritmi pyrkii yhdistämään tiloja, se etsii samanlaisia toimintoja. Algoritmi huomaa, että tilan a_4 voikin yhdistää yhdeksi tilaksi, jolloin lopputulos muistuttaa vähemmän oikeaa (kuva 2.6). Syntetisoitu tilakone hyväksyy myös esimerkiksi polun $(a_1, e_2)(a_3, e_2)(a_4, e_1)(a_5, e_1)(a_7, null)$, jota alkuperäinen tilakone ei olisi hyväksynyt. Usein tämäntapainen yleistys on kuitenkin haluttu ominaisuus.



Kuva 2.6. Tilakone, jonka yleistävä syntetisointialgoritmi on luonut

2.3 Vuorovaikutteiset syntetisointialgoritmit MAT ja MAS

Syntetisointia sekvenssikaaviosta tilakaavioon voidaan ajatella myös kielien päättelyongelmana (*language inference problem*). Gold [6] esittää nyt jo klassisen päättelysäännön (*identification in the limit*), jossa syntetisointialgoritmi saa syötteenään loppumattoman jonon merkkijonoja ja jokaisen merkkijonon lukemisen jälkeen tulostaa hypoteesin merkkijonojen esittämästä kielestä.

Päättelyssä yritetään etsiä sääntöjä tuntemattomalle kielelle. Esimerkkinä voisi ajatella sääntöä numeroille 1, 2, 4, 8… . Sopivia sääntöjä voisivat olla vaikkapa $x_n = 2^{n-1}$ ja $x_n = 1 + \sum_{i=1}^{n-1} x_i, x_{n<1} = 0$, molemmissa n=1, 2, 3… . Päättelyongelman rajausta varten tarvitaan erilaisia määrityksiä, joista tarkemmin lähteessä [2].

Päättely onnistuu, jos kieli on pääteltävissä (*identified in the limit*) eli jos algoritmin esittämät hypoteesit jonkin äärellisen ajan jälkeen pysyvät muuttumattomina. Jotta säännöllisten kielten perheeseen kuuluvan kielen päättely onnistuisi, täytyy algoritmin lukemien merkkijonojen sisältää merkkijonoja, jotka kuuluvat tunnistettavaan kieleen sekä sellaisia merkkijonoja, jotka eivät kuulu tunnistettavaan kieleen. Pelkillä kieleen kuuluvilla merkkijonoilla säännöllisiä kieliä ei ilman lisärajoitteita voida päätellä luotettavasti Goldin määrittelyn mukaisesti [6]. Eräs tapa esittää lisärajoitteita tilakoneiden päättelyn yhteydessä on käytössä lähteessä [12].

2.3.1 Minimally Adequate Teacher (MAT)

MAT on Angluinin [1] esittämä säännöllisten kielin päättelyalgoritmi, joka koostuu kahdesta osasta: opettaja (*teacher*) ja oppilas (*learner*). Päättely on MAT:issa tapahtuma, jossa opettaja (oraakkeli) opettaa oppilasta (syntetisointialgoritmi) vastaamalla oppilaan esittämiin kysymyksiin koskien pääteltävänä olevaa kieltä.

Opettajan tehtävät

Angluin määrittelee myös tarkemmin, minkälaisia ominaisuuksia opettajalta vaaditaan. Opettajan tehtävä on vastata oikein kahdenlaisiin oppilaan esittämiin kysymyksiin [1]:

- Jäsenkysely (*Membership query*):"Kuuluuko merkkijono t tutkittavana olevaan säännölliseen kieleen (*regular set*) vai ei?"
- Ekvivalenssikysely (equivalence query): ”Onko algoritmin tuottama konjektuuri M haluttu lopputulos?” Jos vastaus on ei, niin opettajan täytyy antaa myös vasta-esimerkki (*counterexample*). Vastaesimerkki w kuuluu joko etsittyyn tuntemattomaan kieleen L (eli positiivinen vastaesimerkki) tai annetun konjektuurin määrittelemään kieleen K, mutta ei etsittyyn kieleen (negatiivinen vastaesimerkki). Toisin sanoen $w \in (L \cup K) \setminus (L \cap K) = (L \setminus K) \cup (K \setminus L)$.

Näistä ensimmäiseen kysymykseen on olemassa vain yksi oikea vastus, joten kaikki algoritmin hyväksymät opettajat vastaavat samoihin jäsenkysymyksiin samalla tavalla. Jälkimmäinen kysymys on hieman ongelmallisempi, koska mahdollisia vastaesimerkkejä on olemassa useita erilaisia.

Observaatiotaulu

Opettaja pitää niin kutsuttua observaatiotaulua, joka sisältää käytössä olevan tiedon tutkittavan kielen jäsenistä ja niistä merkkijonoista, jotka eivät ole kielen jäseniä. Taulu koostuu kolmesta asiasta: merkkijonojoukko S, merkkijonojoukko R (Angluin käyttää joukon nimenä E) ja kuvaus $T : ((S \cup S \cdot A) \cdot R) \rightarrow \{0,1\}$, missä A on tutkittavan kielen määrittelevä aakkosto ja S on prefiksisuljettu joukko A^*:stä.

Observaatiotaulu voidaan visualisoida kaksiulotteisena taulukkona, jolloin sen rivit on nimetty joukon $(S \cup S \cdot A)$ jäsenillä. Vastaavasti taulukon sarakkeiden otsikot ovat suffiksisuljettu joukko R. (Joukko on prefiksisuljettu, joss sen jokaisen jäsenen jokainen etuliite on myös joukossa. Suffiksisuljettu määritellään vastaavasti). Observaatiotaulua voidaan merkitä myös kolmikolla (S,R,T).

Kuvaus $T(u)$ ja u määritellään seuraavasti:

$$u = u_1 u_2, \text{missä}\, u_1 \in S \cup S \cdot A \text{ ja}\, u_2 \in R$$

ja

$$T(u) = \begin{cases} 1, \text{jos } u \text{ kuuluu tutkittavaan kieleen } U \\ 0, \text{jos } u \text{ ei kuulu tutkittavaan kieleen } U. \end{cases}$$

Tässä työssä on käytetty observaatiotaulujen kohdalla merkintätapaa, jossa S on erotettu väliviivalla joukosta $S \cup A$, ja sarakkeiden otsikot on merkitty taulukon alapuolelle. Vastaavaa merkintää on käytetty myös lähteissä [24-26].

Taulukko 2.1. Esimerkki observaatiotaulusta, kun S={λ,member} ja R={λ}

T	r_1
λ	1
member	1
ship	0
membermember	1
membership	1

$r_1 = \lambda$

Observaatiotaulun ominaisuudet

Observaatiotaulun sanotaan olevan *suljettu* (*closed*), kun jokaista merkkijonoa $t \in S \cdot A$ kohden löytyy sellainen $s \in S$, että row(t) = row(s). Vastaavasti observaatiotaulu on *konsistentti* (*consistent*), että kun s_1 ja s_2 ovat joukon S sellaisia alkioita, että row(s_1) = row(s_2), on myös jokaiselle merkille $a \in A$ voimassa row($s_1 \cdot a$) = row($s_2 \cdot a$). Merkintä row(t) tarkoittaa observaatiotaulun rivin t ja sarakkeiden $r \in R$ yhdessä muodostamaa "bittikarttaa", jonka eri bittien arvot muodostuvat funktiolla $T(t \cdot r)$.

Esimerkiksi yllä esitetty observaatiotaulu (taulukko 2.1) ei ole suljettu, koska ehto row(*ship*) = row(t) ei toteudu millään $t \in S$. Se ei myöskään ole konsistentti, koska row(λ) = row(*member*), mutta row(λ·*ship*) ≠ row(*member*·*ship*).

Jos (S,R,T) on suljettu ja konsistentti observaatiotaulu, niin kielen hyväksyvä konjektuuri $M(S,R,T)$ voidaan määritellä seuraavasti [26]:

- Tilajoukko $Q = \{row(s) \mid s \in S\}$
- Alkutila $q_0 = row(\lambda)$
- Lopputilojen joukko $F = \{row(s) \mid s \in S \ \ ja\, T(s) = 1\}$
- Siirtymäfunktio $\delta(row(s), a) = row(s \cdot a)$ jokaiselle $s \in S$ ja $a \in A$ siten että $s \cdot a \cdot t$ tiedetään kuuluvan haluttuun kieleen jollakin $t \in A^*$.

Observaatiotaulun rivit $s \in S$ muodostavat tulevan konjektuurin tilat ja taulun sarakkeet kertovat vastaavasti, voidaanko kaksi riviä yhdistää yhdeksi tilaksi (jos row(s_1) = row(s_2), $s_1, s_2 \in S$ on kyseessä sama tila). Tilasiirtymät muodostetaan joukon $S \cdot A$ riveistä. Angluin on lisäksi todistanut, että $M(S,R,T)$ on pienin (eli siinä on vähinten tiloja) äärellinen automaatti, joka on konsistentti T:n kanssa [1].

Angluin esittää myös tarkan algoritmin syntetisoinnille [1]:

Algoritmi 1. MAT

```
Aseta S,R={λ}
Kysy jäsenkyselyt λ:lle ja jokaiselle a∈A
Muodosta ensimmäinen observaatiotaulu (S,R,T)

Toista
  Toista niin kauan kun (S,R,T) ei ole suljettu ja konsistentti
    Jos (S,R,T) ei ole konsistentti, niin
          etsi s1,s2∈S, a∈A ja r∈R siten, että row(s1) = row(s2) ja
          T(s1·a·r) ≠ T(s2·a·r).
          Lisää a·r joukkoon R.
          Laajenna T joukoksi (S∪S·A)·R käyttäen jäsenkyselyitä.
    Jos (S,R,T) ei ole suljettu, niin
          etsi s1∈S ja a∈A siten, että row(s1·a) on eri kuin mikään row(s), s∈S.
          Lisää s1 joukkoon S.
          Laajenna T joukoksi (S∪S·A)·R käyttäen jäsenkyselyitä.

  Olkoon M = M(S,R,T). Luo konjektuuri M.
  Jos opettaja vastaa vastaesimerkillä t, niin
    lisää t ja kaikki sen prefiksit joukkoon S.
    Laajenna T joukoksi (S∪S·A)·R käyttäen jäsenkyselyitä.
Kunnes opettaja hyväksyy konjektuurin M.
```

Algoritmin aika- ja tilavaatimukset on esitetty luvussa 5.3.

2.3.2 Minimally Adequate Synthesizer (MAS)

MAS [24-26] on vuorovaikutteinen syntetisointialgoritmi. Se perustuu Angluinin [1] esittämään algoritmiin Minimally Adequate Teacher (MAT), josta se on jatkokehitetty siten, että opettajan paikalla on ohjelmistosuunnittelija (eli syntetisointiohjelman käyttäjä) ja kysymysten määrää on pyritty vähentämään rajusti. Algoritmin perusidea on kuitenkin säilytetty lähes samana.

MAS on tarkoitettu syntetisoimaan UML-sekvenssikaavioista (katso luku 2.1) tilakaavion, joka hyväksyy annetut kaaviot. Käyttämällä jäsenkyselyitä vältytään tekemästä liian yleistettyä tilakonetta, joka ei vastaisi haluttua lopputulosta. Lisäksi käyttäjälle annetaan mahdollisuus vaikuttaa lopputulokseen antamalla joko negatiivisia tai positiivisia vasta-esimerkkejä, joilla ohjataan algoritmin suoritusta haluttuun suuntaan.

MAS käyttää myös observaatiotaulua aivan kuten MAT. Koska MAT (ja siten myös MAS) perustuu säännöllisiin kieliin, täytyy sekvenssikaavio muuttaa ennen käsittelyä merkkijonoiksi. Aakkosto *A* muodostetaan sekvenssikaavioiden tutkittavana olevan olion *C* viestityksestä, siten että *A*:n merkit muodostuvat viestipareista (e_i,e_j), jossa e_i on *C*:n lähettämä viesti jollekin toiselle oliolle. Vastaavasti e_j on *C*:n saama paluuviesti viestiin e_i. Jos jompaa kumpaa viestiä ei lähetetä (eli esimerkiksi vastausta ei ole sekvenssikaaviossa), laitetaan tilalle *NULL*-viesti. [26] Kuva 2.1 muodostaisi siten aakkoston *A*={(*display_time*,*change_time*),(*update,NULL*),(*display_time*,*VOID*)}, kun tutkittavana on *Control Unit* -olio.

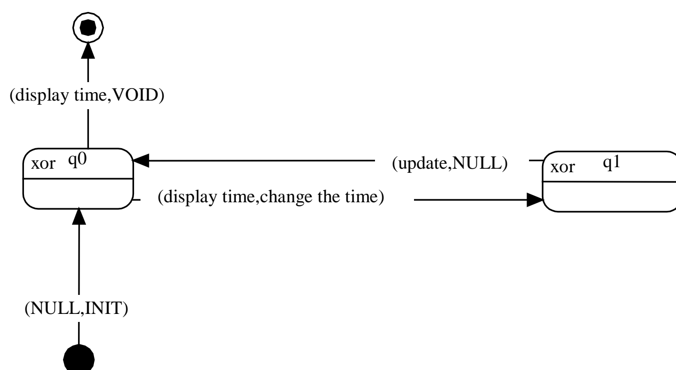
Lähetettyä viestiä voidaan ajatella esimerkiksi funktiokutsuna tai muuna vastaavana toimenpiteenä. Siksi tilakonetta ajatellen on luontevaa laittaa jokainen *C*:n lähettämä viesti tilakoneen tilan toiminnoksi (*action*). Vastaavasti viesti e_j voidaan kuvitella funktion palauttamaksi paluuarvoksi, joka aiheuttaa ohjelman sisäisen tilan vaihtumiseksi toiseksi (esimerkiksi siirtymisen virheenkäsittelyyn). Siten jokainen paluuviesti e_j kuvataan tapahtumaksi, joka aiheuttaa tilasiirtymän toiseen tilaan. Aakkoston merkit koostuvat siis suoraan tilakoneen toiminnoista ja vastaavista tilasiirtymistä.

Koska sekvenssikaaviota syntetisoitaessa keskitytään nimenomaan yhteen olioon, ei viestin vastaanottaja ole tärkeää. Siksi lähetetyille viesteille (eli toiminnoille) ei merkitä tilakoneessa vastaanottajaa.

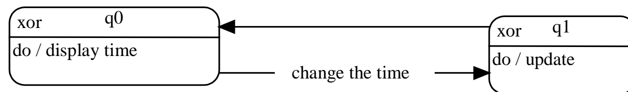
Sekvenssikaavion viestien loppuminen ilmaistaan *VOID*-viestillä. Kun MAS muodostaa käyttäjälle tilakone-ehdotuksen, jätetään lopputila ja siihen johtavat tilasiirtymät näyttämättä, koska sekvenssikaavio ei välttämättä näytä ohjelman koko elinkaarta, vaan kuvaa vain yhden sen osan tapahtumia.

Algoritmin lopputuloksena syntyy äärellinen automaatti, joka hyväksyy annettujen sekvenssikaavioiden viestiparien määrittelemät merkkijonot. Tästä automaatista muokataan aikaisemmin kerrotulla tavalla lopullinen tilakone siten, että äärellisen automaatin siirtymien ensimmäinen viesti määrää tilakoneen toiminnon ja toinen vastaavan tilasiirtymän toiseen tilaan. Alku- ja lopputila sekä niihin liittyvät tilasiirtymät poistetaan. [26]

Kuva 2.7 on äärellinen automaatti, joka on syntynyt, kun MAS-algoritmilla on ajettu esimerkkinä käytetty sekvenssikaavio (kuva 2.1). Kuva 2.8 esittää vastaavasti tästä automaatista muodostettua tilakonetta.



Kuva 2.7. Syntetisoinnissa syntynyt äärellinen automaatti



Kuva 2.8. Äärellisestä automaatista muodostettu tilakone

Eräs MAT:in ongelmista on sen tarvitsemien jäsenkyselyiden määrä. Käytännön OO-työskentelyssä ei olisi kovinkaan käyttökelpoista, jos ohjelmistosuunnittelijalta menisi yhtä kauan aikaa syntetisoijan kysymyksiin vastaamiseen kuin tilakoneen piirtämiseen käsin. Lisäksi suuri määrä kysymyksiä lisää virhemahdollisuuksia ja turhauttaa vastaajaa. MAS-algoritmissa onkin pyritty mahdollisuuksien mukaan vähentämään kysymysten määrää (katso myös luku 7 ja varsinkin alakohta 7.1.2) kolmella oletuksella [26]:

- Koska sekvenssikaavioista muodostetut merkkijonot päättyvät aina symboliin (e_i,VOID) voidaan kaikki muut merkkijonot merkitä suoraan kieleen kuulumattomiksi ilman kyselyä.
- Jos jollain tilalla on tilasiirtymä *NULL*, sillä ei voi olla muita tilasiirtymiä eli jos on olemassa symboli (e_i,NULL) niin e_i:n parina täytyy olla aina NULL kaikissa muissakin symboleissa.
- Jos sekvenssikaaviosta luettu merkkijono on muotoa $e=e_1(e_i,e_j)e_2$ ei muotoa $e=e_1(e_k,e_l)w$ oleville merkkijonoille tarvitse tehdä jäsenkyselyä, jos $k \neq i$. Kyselyn vastaus on aina negatiivinen, koska muuten sama polku e_1 voisi johtaa kahteen eri toimintoon samassa tilassa.

Jos säännöllinen kieli täyttää yllä mainitut kolme oletusta, sanotaan, että kieli on *tiladetermistinen* (*state deterministic*) [26].

MAS pitää yllä tehokasta tietorakennetta (puuta), josta se käy jokaisen jäsenkyselyn kohdalla varmistamassa riittävätkö aikaisemmin annetut tiedot vastauksen päättelyyn. Vain, jos lisätietoa tarvitaan, kysytään jäsenkysymys käyttäjältä.

Jäsenkysely merkkijonolle w puusta W antaa kolme erilaista mahdollisuutta [24]:

- Puun oksia voidaan seurata juuresta lehteen, jolloin w:tä vastaava merkkijono on jo aikaisemmin sijoitettu W:hen ja vastaus jäsenkysymykseen on *kyllä*.
- w on muotoa $w=w_1(e_i,f)w_2$, missä w_1 on pisin mahdollinen yhteinen alku w:n ja puussa olevan merkkijonon (y) suhteen. Jos y jatkuu symbolilla (e_j,g), missä $j \neq i$, tiedetään, että w ei voi kuulua kieleen ja vastaus kyselyyn on *ei*.
- w on muotoa $w=w_1(e_i,f)w_2$, missä w_1 on pisin mahdollinen yhteinen alku w:n ja puussa olevan merkkijonon (y) suhteen. Jos y jatkuu symbolilla (e_j,g), missä $j=i$ (ja siten $f \neq g$), algoritmi ei voi päätellä vastausta ja kysely siirretään käyttäjälle.

Toisin kuin MAT, MAS ei tiedä koko aakkostoa etukäteen, vaan uusia merkkejä voi tulla A:han riippuen esimerkkeinä annetuista sekvenssikaavioista. Siksi jokaisen opettajan (käyttäjän) antaman uuden esimerkin jälkeen tarkistetaan hyväksytty aakkosto. Kun käytetään vain tarvittavan kokoista aakkostoa, voidaan observaatiotaulu pitää pienenä.

MAS eroaa myös muutamalta muulta kohdin MAT:ista. Algoritmin syötteenä käytetään käyttäjän antamia positiivisia esimerkkejä kielestä. Nämä esimerkit tallennetaan observaatiotauluun. Esimerkeille ei tarvitse tehdä jäsenkyselyjä, koska tiedetään, että merkkijonot kuuluvat kieleen ja niiden prefiksit eivät. Lisäksi laajennettaessa observaatiotaulua T jäsenkyselyitä käytetään vain, kun vastausta ei muuten pystytä päättelemään. Viimeinen eroavaisuus on, että kieleen kuuluvien merkkijonojen joukko täytyy täyttää tiladeterminismin ehdot.

Algoritmi on lähes samanlainen kuin aikaisemmin esitetty MAT (katso algoritmi 1). Algoritmissa käytetty pref(I) tarkoittaa I:n prefiksejä. Alkuperäinen algoritmi on esitetty lähteessä [26].

Algoritmi 2. MAS

```
I sisältää käyttäjän antamat esimerkit kieleen kuuluvista merkkijonoista
Aseta S={λ}∪I∪pref(I), R={λ}
Tarkista tiladeterministisyys joukolle I
Aakkosto A on joukossa I käytettyjen symbolien joukko
Kysy jäsenkyselyt λ:lle ja jokaiselle a∈A
Laajenna T joukoksi (S∪S·A)·R käyttäen tarvittaessa jäsenkyselyitä

Toista
  Toista niin kauan kun (S,R,T) ei ole suljettu ja konsistentti
      Jos (S,R,T) ei ole konsistentti
                Etsi s1,s2∈S, a∈A ja r∈R siten, että row(s1) = row(s2) ja
                T(s1·a·r) ≠ T(s2·a·r).
                Lisää a·r joukkoon R.
                Laajenna T joukoksi (S∪S·A)·R käyttäen tarvittaessa jäsenkyselyitä
      Jos (S,R,T) ei ole suljettu
                Etsi s1∈S ja a∈A siten, että row(s1·a) on eri kuin mikään row(s), s∈S.
                Lisää s1 joukkoon S.
                Laajenna T joukoksi (S∪S·A)·R käyttäen tarvittaessa jäsenkyselyitä

  Olkoon M = M(S,R,T). Luo konjektuuri M.
  Jos opettaja vastaa vastaesimerkillä t niin
      lisää t ja kaikki sen prefiksit joukkoon S
      Tarkista aakkosto ja lisää t:n uudet symbolit aakkostoon A
      Lisää tarvittaessa t esimerkkijoukkoon I ja tarkista tiladeterminismi.
      Laajenna T joukoksi (S∪S·A)·R käyttäen tarvittaessa jäsenkyselyitä
Kunnes opettaja hyväksyy konjektuurin M.
```

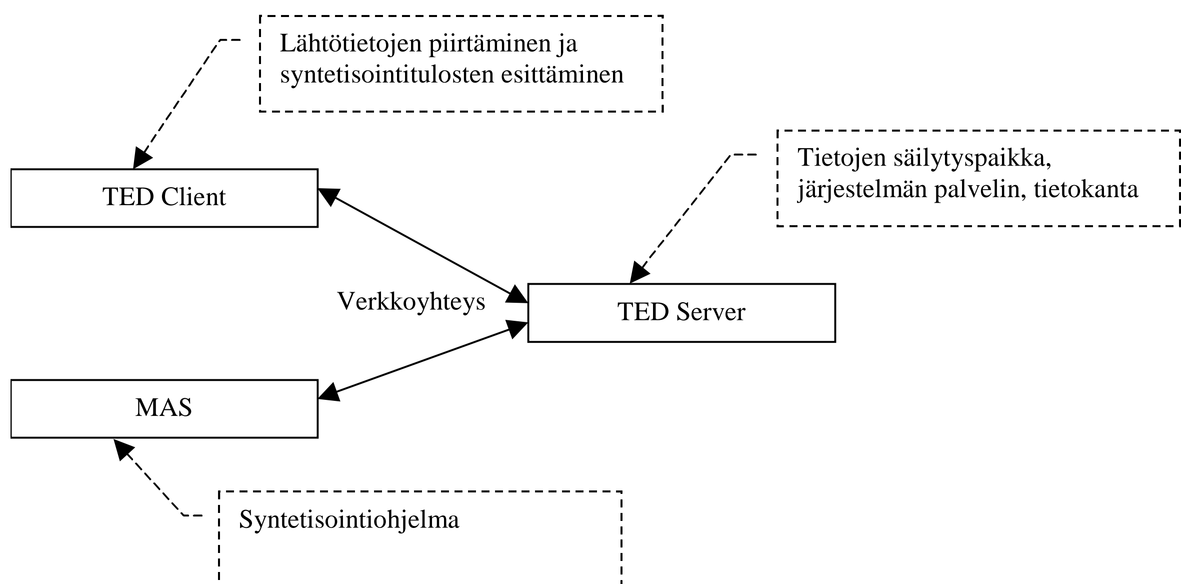
MAS voidaan laajentaa hyväksymään kaikki UML-sekvenssikaavion notaatiot [24]. Yllähän on käsitelty vain yksinkertaisia olio+viestitys –kaavioita. UML-notaatio antaa mahdollisuuden myös ehdollisiin haarautumisiin, iteraation, rekursioon, eksplisiittisiin paluuviesteihin ja olioiden tuhoamisiin.

3. Syntetisointiohjelman toimintaympäristö

Tässä luvussa käsitellään toteutetun syntetisointiohjelman toimintaympäristön erikoispiirteitä. Itse algoritmi voidaan kääntää mille ympäristölle tahansa, mutta graafinen käyttöliittymä ja käyttäjän syötteissä tarvittavat oheisohjelmat ovat riippuvaisia käyttöjärjestelmän palveluista.

3.1 Yleistä ympäristöstä

Syntetisointiohjelma toimii Microsoft Windows NT 4.0 -käyttöjärjestelmässä, johon on asennettu uusin korjaustiedosto Service Pack 6a. Kuva 3.1 esittää eri ohjelmien tehtävät järjestelmässä.



Kuva 3.1. Ympäristön eri ohjelmat

3.2 Hajautetut komponentit

Olio-ohjelmoinnin keskeinen piirre on luokkien uudelleenkäytettävyys. Eri ohjelmointikielet (kuten C++) tarjoavat omia keinojaan kuten luokkakirjastot ja periytyvyys, mutta eri kielten tai kääntäjien eri versioiden välillä luokkia ei ole voitu helposti siirtää. Myös luok-

kien välinen kommunikointi on ollut toteutuskohtaisten ratkaisujen varassa ja komponenttien käyttö on rajoittunut saman prosessin sisälle.

Ainakin C++ -kielen luokkien uudelleenkäyttöä ovat rajoittaneet myös kaupalliset seikat: esimerkiksi luokan sisäistä toteutusta on ollut vaikea piilottaa ilman, että luokan käyttö hankaloituisi. Näitä seikkoja silmälläpitäen on kehitetty kaksi standardia hajautettujen objektien mallista. Microsoft loi Windows-käyttöjärjestelmälle järjestelmän nimeltään *Distributed Component Object Model* (DCOM) [18], kun taas useammalle järjestelmäalustalle on sovitettu *Common Object Request Broker Architecture* (CORBA) [35]. Tarkempi vertailu näiden kahden standardin välillä on tehty lähteessä [3].

DCOM on toteutettu Microsoftin COM (*Component Object Model*) –mallin [16] pohjalta. COM on prosessien sisäisten ja samalla koneella toimivien komponenttien binäärirakenteen ja keskinäisen viestityksen standardi. Yleisesti ottaen COM-palvelin voidaan helposti toteuttaa toimimaan myös DCOM-palvelimena ja siirtää siten palvelinohjelmisto esimerkiksi toiseen koneeseen.

Tässä luvussa käsitellään lähinnä COMia, koska se on käytössä syntetisointiohjelmassa ja soveltuu CORBAa paremmin, jos ympäristönä on pelkästään Windows.

COM yleisesti

(Distributed) Component Object Model on laitteistoriippumaton, hajautettu, olio-orientoitunut järjestelmä binääristen ohjelmistokomponenttien luomiseksi. Se koostuu palvelinohjelmista ja niitä käyttävistä asiakasohjelmista. Palvelinohjelmat, jotka voivat sijaita myös toisella koneella kuin asiakasohjelmat, ovat ajettavia ohjelmia (EXE) tai dynaamisesti linkitettäviä kirjastoja (DLL). Jokainen palvelinohjelma on rekisteröity omalla yksilöllisellä tunnuksellaan (*Globally Unique Identifier*, GUID) ja tarjoaa asiakasohjelmalle käytettäväksi rajapintoja, jotka ovat myös yksilöity. Kaikki keskustelu asiakkaan ja palvelimen välillä tapahtuu näiden rajapintojen kautta. Koska käytössä on vain rajapinta ja valmiiksi käännetty ohjelma, komponentin sisäinen toteutus voidaan piilottaa kaupallista käyttöä varten.

Komponentti voi sijaita asiakas-palvelin –järjestelmässä kolmella eri tavalla: samassa prosessissa asiakasohjelman kanssa, samassa koneessa ja eri prosessissa sekä kokonaan eri koneessa [17]. Asiakasohjelman toteutuksen kannalta ei palvelimen sijainnilla ole merkitystä, koska COMin järjestelmäkomponentit piilottavat palvelimen lopullisen sijainnin.

Rajapinnat

COM-ajattelun tärkein oivallus on, että standardoimalla komponentin ulkoinen koodirakenne ja tarjoamalla yksinkertaiset välineet rajapintojen suunnitteluun voidaan tehdä uudelleenkäytettäviä komponentteja, jotka eivät ole riippuvaisia toteutustavasta tai -kielestä. Asiakasohjelman ei tarvitse enää välittää siitä, *miten* joku komponentti on toteutettu vaan kysymys kuuluu, *mitä* palveluita komponentti tarjoaa.

Komponenttien rajapinnat määritellään erityisellä kielellä (*Interface Definition Language, IDL).* Tämä määrittely toimii sopimuksena (*contract*) palvelimen ja asiakkaiden välillä. Asiakkaat käyttävät palveluita IDL:ssä määriteltyjen metodien välityksellä. Rajapintoihin on sisällytetty osa olio-ohjelmoinnin ominaisuuksista, kuten tiedon kapselointi, polymorfismi ja periytyvyys.

Rajapinnat näyttävät ulospäin lähes samalta kuin C++:n luokan esittely. Itse asiassa rajapintojen binäärirakenne on sama kuin C++ -kielen virtuaalifunktioiden taulu. Toisin kuin esittelyssä, komponentin rajapinnassa kerrotaan vain luokan tarjoamien palveluiden tiedot – ei siis koko luokan sisäistä rakennetta. Parhaimman kuvan COM-rajapinnasta antaa alla oleva luettelo [19]:

- **Rajapinta ei ole C++ -luokka**
  Rajapinnassa ei ole toteutusta vaan se sisältää vain määrittelyjä. Toteutusosa sijaitsee luokan palvelinohjelmassa. Komponentin rajapinta voidaan toteuttaa myös esimerkiksi Javalla ja Visual Basicilla.
- **Rajapinta ei ole objekti**
  Rajapinta on ryhmä funktioita ja standardi tapa millä asiakasohjelmat ja komponentit keskustelevat keskenään. Rajapinta voidaan toteuttaa millä tahansa tavalla, kunhan se tuottaa rajapinnan vaatimat osoittimet rajapinnan metodeihin.

- **Rajapinta on vahvasti määritelty**

  Jokaisella rajapinnalla on ainutkertainen tunnus, joten eri rajapinnat eroavat toisistaan, eikä sekaannusta voi tulla kahden erilaisen (vaikkakin samannimisen) rajapinnan kanssa.
- **Rajapinnat ovat muuttumattomia**

  Rajapintaa ei voi muuttaa siten, että sen tunnus säilyisi samana. Siten uusi rajapinta ei voi mitenkään olla ristiriidassa vanhan kanssa. Komponentti voi tarjota useita rajapintoja, mutta jokaisella niistä on oma tunnuksensa.

Jokainen rajapinta saa ainutkertaisen tunnuksen (*interface ID,* IID), joten eri versiot samasta rajapinnasta erottuvat automaattisesti. Luokka voi tarjota samanaikaisesti useita eri rajapintoja, joten uudempi rajapinta voi tarjota samasta asiasta tehokkaampaa palveluita, mutta vanhat asiakasohjelmat voivat silti käyttää tuntemiaan vanhempia rajapintoja. Komponentilta voidaan myös kysyä mitä palveluita (ja rajapintoja) sillä on tarjottavanaan.

Kun komponentti tarjoaa käyttöön rajapinnan ja jokainen rajapinnan metodi on implementoitu komponentissa, sanotaan että komponentti toteuttaa rajapinnan. Jokainen komponentti toteuttaa rajapinnan *IUnknown*, joka tarjoaa perusmetodit muiden rajapintojen käyttöön. Lisäksi tässä rajapinnassa on komponentin elinikää säätelevät metodit (katso alakohta Muistinhallinta).

Rajapinnan metodeita käytetään hakemalla aluksi käytettävän rajapinnan perusosoite metodilla `QueryInterface`. Alla olevassa esimerkissä (vertaa [22]) haetaan rajapinnan *IB* osoite.

```
IUnknown* pA = GetMyObject(); /* jostain saatu osoite IUnknown */
CComPtr<IB> pB; /* tämä hoitaa automaattisesti osoittimen muistinhallinan */
HRESULT   hr;
hr = pA->QueryInterface(IID_IB, &pB);
```

Tämän jälkeen voidaan komponenttia käyttää aivan kuten normaalia C++ -luokkaa. Esimerkissä oletetaan, että rajapinta *IB* sisältää metodit `CreateCircle` ja `Draw`.

```
pB->CreateCircle(128,96,100);
pB->Draw(&piirtopinta);
```

Vastaavasti rajapinnan määrittely kuvattuna IDL:llä voisi olla alla olevan kaltainen:

```
[
  object,
  uuid(4c3d7490-a715-11d2-b678-00805ffdb209)
]

interface IB : IUnknown
{
  HRESULT CreateCircle(    [in]DWORD x,
                           [in]DWORD y,
                           [in]DWORD rad);

  HRESULT Draw(            [in]IPiirtopinta* drw);
};
```

Esimerkkimäärittelyssä [in] tarkoittaa sisään tulevaa argumenttia, vastaavasti määritellään myös esimerkiksi paluuarvo [out]. Kuten huomataan, itse komponentin toteutus ei näy ulos mitenkään.

Muistinhallinta ja komponentin elinikä

Koska komponentti on ulkoinen ohjelma, joka voi sijaita myös kokonaan eri koneessa kuin asiakasohjelma, ei kääntäjän tarjoamia automaattisia muuttujan näkyvyyteen (esimerkiksi paikalliset ja globaalit muuttujat) perustuvia muistinvarauksia voida enää käyttää. Lisäksi, vaikka yksi komponenttia käyttänyt asiakasohjelma ei enää tarvitsisikaan komponentin palveluita, ei voida olla varmoja, ettei toinen (esimerkiksi toisella koneella sijaitseva) ohjelma tarvitsisi vielä komponenttia. Tätä varten *IUnknown*–rajapinta käyttää viitelaskuriin perustuvaa järjestelmää komponentin eliniän määrittämiseksi. [20]

Jokainen kerta, kun ohjelma haluaa käyttää komponentin palveluita, se kutsuu metodia `AddRef`, jolloin komponentin sisäistä viitelaskuria kasvatetaan yhdellä. Vastaavasti jokainen vapautus, kun palvelua ei enää tarvita tapahtuu kutsumalla metodia `Release`, vähentää viitelaskuria yhdellä. Kun komponenttiin ei ole enää yhtään viittausta, voidaan se poistaa. Toteutus on hyvin samankaltainen kuin säikeiden ja prosessien synkronoinnissa käytetyt semaforit. [23]

Järjestelmä tarjoaa myös yhteisen muistinvarauskäytännön, jota käytetään, kun luodaan esimerkiksi kopioita tiedosta. Yhteinen käytäntö onkin tarpeen, sillä komponentti varaa tarvittavan muistin, kun taas asiakasohjelma käytettyään vapauttaa sen.

Automaatio

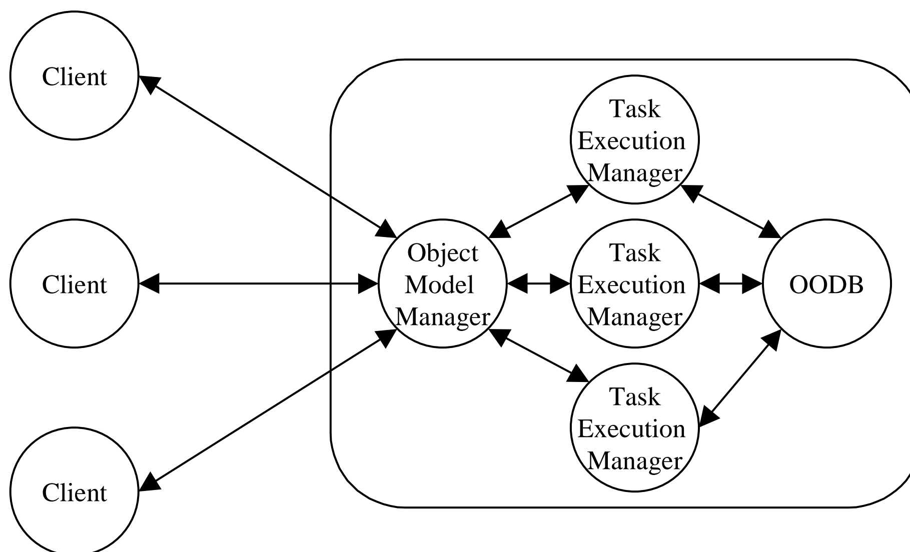
Aina ei voida käyttää staattisiin rajapintoihin perustuvaa palvelinjärjestelmää. Useat tulkittavat kielet (esimerkiksi Python, Visual Basic) toimivat dynaamisesti eivätkä siten pystyisi käyttämään hyväkseen ollenkaan COMia. Tämän vuoksi tarjotaan myös dynaaminen tapa kutsua palvelurajapintojen metodeita. Tässä järjestelmässä asiakasohjelma kysyy palvelimelta, mitä ominaisuuksia (vastaa C++-luokkien jäsenmuuttujia) ja metodeita (vastaa C++-luokkien julkisia metodeita) on tarjolla. Automaatiossa luokka toteuttaa rajapinnan `IDispatch`, joka tarjoaa metodit dynaamiselle käytölle.

Automaatio on myös riippumaton käytetystä ohjelmointikielestä. Muuttujat välitetään erityisenä tietorakenteena, joka sisältää varsinaisen informaation lisäksi myös tiedon muuttujan tyypistä. Tulkittavissa kielissä muutos "normaaleista" muuttujista tapahtuu yleensä automaattisesti, jolloin käyttäjän (eli ohjelmoijan) ei tarvitse välittää asiasta sen enempää.

Automaatiosta on tarkemmin kerrottu lähteessä [15].

3.3 Tde EDitor (TED)

TED [38] on Nokia Research Centerin kehittämä usean käyttäjän UML-notaatiota tukeva mallinnusohjelmisto. Se toimii Windows NT -ympäristössä ja tarjoaa mahdollisuuden tietojen jakamiseen helposti useiden käyttäjien kesken. Ohjelmisto koostuu itse käyttöliittymistä (client) ja palvelimesta, jossa järjestelmän tietokantakin sijaitsee (kuva 3.2). Tietokanta (*OODB*) on piilotettu kokonaan käyttöliittymiltä, joten esimerkiksi tietokantavalmistajan vaihto ei edellytä mitään toimenpiteitä asiakkailta.



Kuva 3.2. TEDin sisäinen arkkitehtuuri [38]

3.3.1 TEDin toiminta

TEDin mallinnusympäristö koostuu työpöydästä ja käyttäjien luomista työkirjoista. Työpöytä on tietokantakohtainen ja sinne tallennetaan kaikkien käyttäjien luomat mallit. Työpöytä voi sisältää myös kansioita, joita TEDissä kutsutaan nimellä työkirjat. Ympäristöä voisi verrata esimerkiksi levyn hakemistopuuhun, jossa työpöytänä toimii koko hakemisto (alkaen juuresta /) ja jokainen työkirja on oma hakemistonsa (työkirjojakin voi olla sisäkkäin). Työkirjat sisältävät esimerkiksi käyttäjän piirtämiä sekvenssikaavioita. Vastaavasti syntetisointiohjelman tuotokset voivat olla jokainen oman työkirjansa alla.

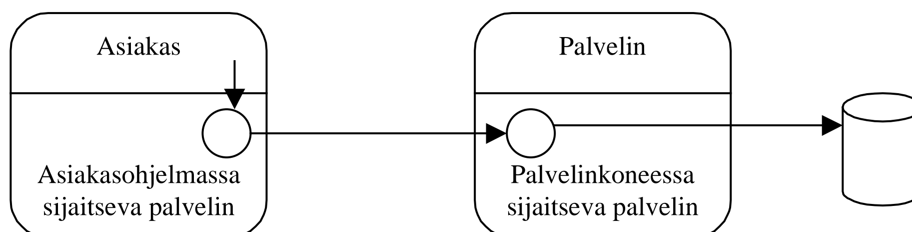
Tietokanta jakautuu malli- ja näkymäosaan (*model-view*). Malliosassa kerrotaan olion tarkempi rakenne ja semanttinen osuus, kun näkymä sisältää tiedon siitä, miltä olio näyttää. Yhdellä mallilla voi olla useampi näkymä, kun taas yhdellä näkymällä on vain yksi malli, johon näkymä viittaa. Eli sama objekti voidaan näyttää erilaisena riippuen näkymän antamasta tiedosta.

Työkirjat näkyvät myös mallipuolella hierarkkisena rakenteena. Tietty työkirja voidaan etsiä esimerkiksi hakemalla ensin juuren instanssi. Tämän jälkeen käydään rekursiivisesti kaikki juuren sisältämien työkirjojen tiedot läpi kunnes halutun kriteerin (esimerkiksi ni-

mi) täyttänyt työkirja löytyy. Käyttäen tämän työkirjan instanssia juurena, voidaan nyt hakea esimerkiksi työkirjassa sijaitsevan sekvenssikaavion tiedot.

3.3.2 Syntetisointiohjelman liittyminen TEDiin

TEDin palvelin on jaettu nopeussyistä kahteen eri osaan, joista toinen toimii asiakasohjelmiston prosessissa (*in-proc object*) ja toinen palvelinkoneessa. Käyttöliittymät keskustelevat asiakaspalvelimen kanssa käyttäen edellisessä luvussa esitettyä COM-rajapintaa, kun taas palvelinkoneessa sijaitsevan palvelimen kanssa ei ollenkaan keskustella suoraan. Nopeussyistä myös objektin tilaa koskevat muutokset käsitellään asiakasohjelman viestisilmukan avulla.



Kuva 3.3. TEDin kaksi palvelinta

Syntetisointiohjelma liittyy TEDiin käyttämällä DepUml-lisälaajennusta [14] (DepUml tulee sanoista *Design Environment Plugin for UML*). Se sisältää tarvittavat tietokantamääritykset, joita käytetään TEDissä käsiteltäessä UML-kaaviota. Syntetisoidessa UML-mallit näkyvät DepUml:n avulla ohjelmalle erilaisina COM-luokkina, jotka toteuttavat mallien käsittelyssä tarvittavat rajapinnat. Yleisesti DepUml:n luokat käsittelevät mallin semanttista puolta ja pienempi osa sen rajapinnoista on tarkoitettu näkymän käsittelyyn.

TEDiä käytetään osana syntetisointiohjelman käyttöliittymää. Käyttäjä piirtää sitä käyttäen haluamansa sekvenssikaavion ja käynnistää tämän jälkeen syntetisointiohjelman. Syntetisointiohjelma alustaa COM-yhteyden TEDin paikalliseen palvelimeen ja etsii käyttäjän antaman työkirjan. Jos työkirja löytyy, etsitään käyttäjän nimeämä sekvenssikaavion olio ja luetaan sekvenssikaavion olion viestitys ohjelmaan sisään. Yleisesti ottaen toimitaan tietokannan mallipuolella, mutta joitakin käsiteltävään kohteeseen liittyviä tietoja on pakko hakea näkymästä (esimerkiksi sekvenssikaavion viestien järjestys).

Syntetisointiohjelma käyttää TEDiä myös tulosten säilytyspaikkana. Kun syntetisointi on suoritettu, luodaan uusi työkirja, jonka sisälle rakennetaan tilakone. Tässä vaiheessa näyttöä ei pyritä mitenkään optimoimaan vaan tilat asetetaan järjestyksessä vierekkäin. Käyttäjä voi tarvittaessa siirtää tilat helpommin ymmärrettävään muotoon käyttäen ohjelmiston editoria.

4. Ohjelmiston toteutus

Koska ohjelmiston tarkoituksena on toimia lähinnä MAS-syntetisoinnin kokeilualustana oli lähtökohtana luonnollisesti helppo muokattavuus ja hyvät jäljitysominaisuudet. Lisäksi ohjelmisto pyrittiin saamaan mahdollisimman siirrettäväksi myös muihin ympäristöihin. Näiden vaatimuksien vuoksi ohjelmiston toteutuksessa käytettiin mahdollisimman paljon valmiita komponentteja, kuten *Standard Template Library* [30] (*STL*) ja *Microsoft Foundation Class Library* [21] (*MFC*).

4.1 Kehitysympäristö

Ohjelman kehitysympäristön valintaan vaikutti ratkaisevasti se seikka, että TED toimii COM-rajapinnan (katso luku 3.2) kautta ja koko TEDin ohjelmistodokumentaatio [14] on tehty C++ kieltä varten. Näin lähes ainoaksi vaihtoehdoksi jäi *Microsoft Visual C++*, josta käytettiin versiota 6.0. *Visual C++* tarjoaa lisäksi hyvän sisäänrakennetun editorin ja virheenkorjausmahdollisuuden, joten valinta on oikeastaan ihanteellinen Windows-ympäristössä tapahtuvaan raskaaseen työskentelyyn.

Versionhallintaa käytettiin vasta ohjelmistokehityksen loppuvaiheessa – lähinnä siitä johtuen, että sopivaa ohjelmistoa ei ollut löytynyt aikaisemmin. Versionhallinnaksi valittiin lopulta *ComponentSoftware RCS* [4] (myöhemmin CS-RCS), joka toimii saumattomasti sekä kääntäjän että dokumentointityökalun (*Microsoft Word 97*) kanssa. CS-RCS on lisäksi täysin yhteensopiva laajasti käytössä olevan ja vapaasti levitettävän GNU RCS –ohjelmiston kanssa. Ennen oikeaa versionhallintaohjelmistoa eri versiot tallennettiin pakattuna levykkeelle joka ilta. Näin jokainen versio oli käytössä myös myöhemminkin.

4.2 Luokka- ja moduulijako

Ohjelmiston rakennetta suunniteltaessa kävi selkeästi ilmi, että ohjelmisto jakautuu kolmeen eri osaan. Ensimmäinen osa on varsinainen syntetisointialgoritmi, joka on helppo toteuttaa laitteisto- ja ympäristöriippumattomaksi moduuliksi. Toisena osana on TEDin kanssa keskusteleva osuus, joka tietysti toimii vain TED-rajapinnan kanssa, mutta ei si-

sällä käyttöliittymää. Kolmantena osuutena on käyttöliittymä, joka keskustelee käyttäjän kanssa ja joka käyttää hyväksi ensimmäisen ja toisen osan tarjoamia palveluita.

4.3 Testaus- ja kehitysalusta

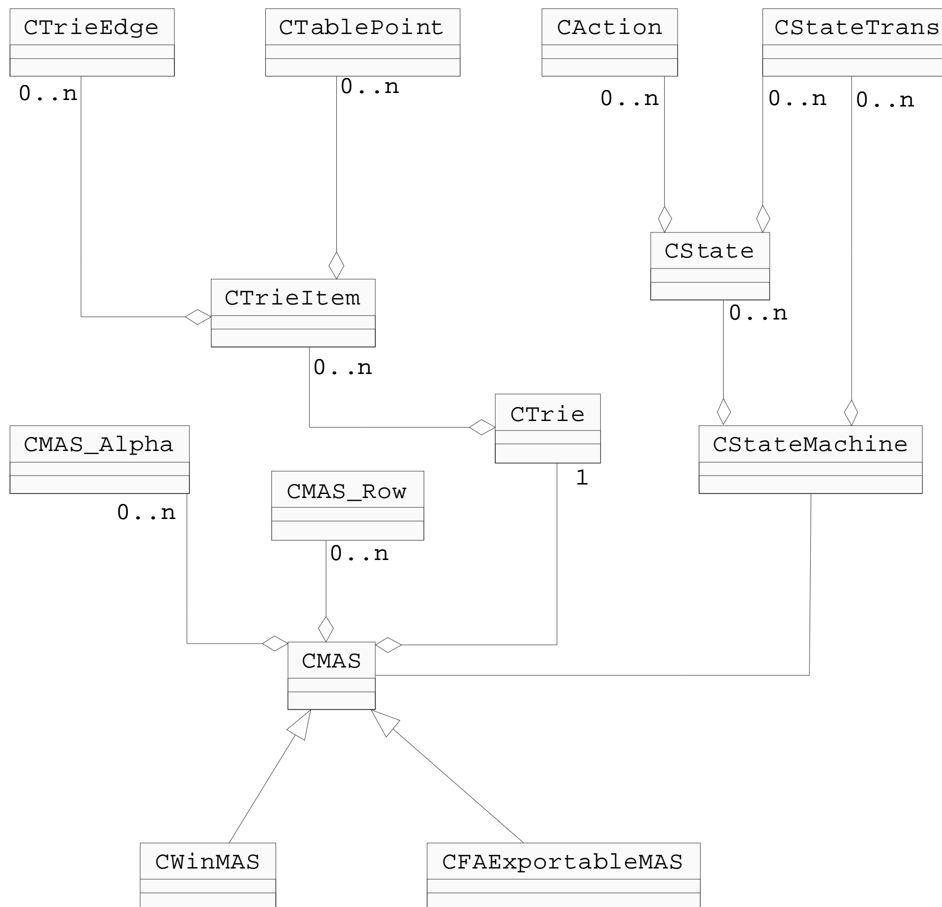
Ohjelman testauksessa käytettiin sekä sisäänrakennettua virheenkorjausympäristöä että omaa testiohjelmistoa. Lopullinen algoritmi oli suunniteltu toteutettavaksi siten, että se sisältäisi graafisen Windows-käyttöliittymän, mutta testausvaiheessa käytettiin testaajalle informatiivisempaa tekstikäyttöliittymää. Käyttöliittymän vaihto oli sinänsä erittäin helppo toimenpide johtuen luvussa 4.2 esitellystä luokkajaosta – samat algoritmitiedostot toimivat hyvin sekä graafisena että tekstinä. Itse asiassa MAS-luokka toteutettiin aluksi konsolille ja tästä perittiin luokka WinMAS, jossa oli omat dialoginsa kyselyitä varten.

Testauskäyttöön tehty tekstikäyttöliittymä tulosti joka kierroksen jälkeen observaatiotaulun ja kysyi käyttäjältä mielipidettä merkkijonosta, joka sijoitettaisiin seuraavalla iterointikierroksella sarakemuuttujaan *R*. Näin ohjelman toiminnan tarkistaminen esimerkkitaulujen avulla oli vaivatonta. Esimerkki tulosteesta on liitteessä 1.

Kehitystyön aikana käsiteltiin syötteitä ja tulosteita viestipareina. Tämä mahdollisti nopean ja tarkan kehitystyön, mutta loppukäyttäjän versiota varten menetelmä on työläs. Siksi testiversion ja lopullisen version käyttöliittymä eroaa tältä osin ratkaisevasti toisistaan.

4.4 Syntetisoijan toteutus

Syntetisoija on toteutettu moduulissa *datatypes*. Moduulin luokkakaavio on esitetty tarkemmin seuraavalla sivulla (kuva 4.1). Tässä luvussa käydään jokainen pääluokka lävitse omassa alakohdassaan.



Kuva 4.1. Moduulin datatypes luokkakaavio

4.4.1 Syntetisoijan luokat

Syntetisoija näkyy ulospäin vain yhtenä luokkana CMAS. Muut luokat määrittelevät vain syntetisoinnissa tarvittavia tietotyyppejä. Itse tieto sijaitsee kuitenkin aina syntetisoijassa.

Luokka CMAS – syntetisoija

Itse syntetisoija sijaitsee luokassa CMAS, josta peritään eri tarkoituksiin erikoistuneita luokkia. Näistä yksi on *CFAExportableMAS*, joka sisältää tarvittavat metodit [24] observaatiotaulun luomiseksi äärellisestä automaatista. Toinen peritty luokka on tällä hetkellä lopullisen käyttöliittymän sisältävä *CWinMAS*.

Luokka CMAS käyttää apunaan luokkaa CMAS_Row, joka sisältää yhden kokonaisen rivin observaatiotaulusta. Yksi rivi käsittää otsikkosarakkeen, jossa rivin nimi (esimerkiksi (*s_ct*,*VOID*)) sijaitsee. Lisäksi rivissä on taulukko, jossa eri *R*:n sarakkeiden arvot sijaitsevat. Tällä hetkellä käytetään boolean-muuttujaa, mutta jatkossa tämän muuttujan tyyppiä tarvinnee laajentaa käsittämään esimerkiksi ”en tiedä” ja ”vastaan myöhemmin” –vaihtoehdot (katso kohta 7.2.3).

Luokka CMAS_Alpha määrittelee syntetisoijan käyttämän aakkoston siten, kuin se on määritelty MAS-algoritmissa (katso tarkemmin luku 2.3). Itse aakkosto sijaitsee listana CMAS-luokan suojatussa jäsenmuuttujassa.

Luokka CTrie – syntetisoijan puu

MAS-algoritmi säilyttää jo hyväksytyt merkkijonot muistissa. MAS:in käyttämä rakenne on puurakenne (englanniksi *trie*), jota kutsutaan nimellä *W*. Tämän rakenteen tarkoituksena on eri sukupolviversioiden tehokas jäljitys (*backtracking*) ja paluu tarvittaessa edelliseen tilanteeseen. Tämä tarve onkin konkreettinen, koska on mahdollista, että käyttäjä vaihtaa mielipidettään, antaa väärän vastauksen tai ei välttämättä edes tiedä oikeaa vastausta kysymykseen (katso tarkemmin luku 7).

Jäljitettävyyden pystyy toteuttamaan eri tavoilla, kuten ottamalla kopioi observaatiotaulusta jokaisen kyselyn jälkeen tai tallentamalla jokaisen muutoksen aloitustilan jälkeen (tätä käytetäänkin hyväksi monissa tietokannoissa toteuttamaan transaktiot[1]). Tämä ei ole kovinkaan tehokasta, koska käyttäjä ei voi valikoivasti poistaa haluamiansa hyväksyttyjä merkkijonoja, vaan on tyydyttävä kokonaisvaltaiseen ”peruuta”-toimintoon. [24]

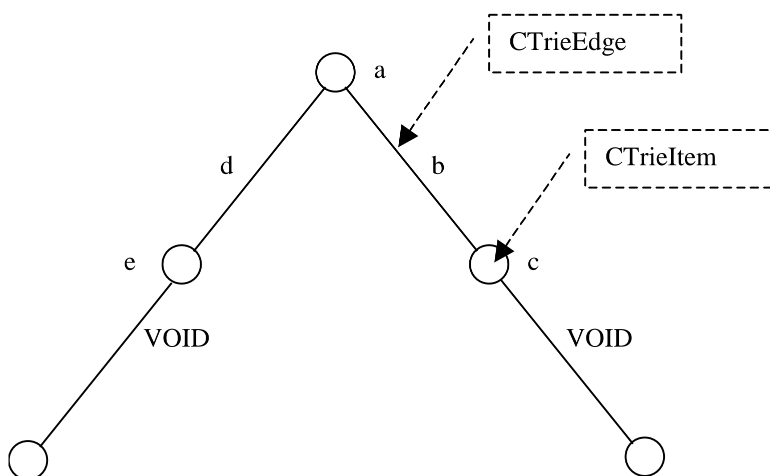
---

[1] Transaktiot ovat tapahtumia, jotka tapahtuvat joko kokonaisuudessaan tai virheen sattuessa peruutetaan takaisin virhettä edeltäneeseen tilaan. Esimerkki transaktiosta on rahan siirto tililtä toiselle: raha ei voi jäädä leijumaan eri tilien välille vaan se on joko uudella tilillä tai siirron epäonnistuttua takaisin vanhalla tilillä.

Jäljitettävyyden lisäksi trie:llä on myös tärkeä tehtävä MAS:in algoritmissa: jokainen kysely tapahtuu ensisijaisesti puusta ja vasta tämän jälkeen, jos oikeaa tulosta ei löydetä puusta, kysytään käyttäjältä (katso tarkemmin luku 2.3).

Luokka CTrieItem – trien solmu

Jokaisesta puun *W* solmusta tallennetaan äitisolmu, lähtevät haarat sekä viesti. Lisäksi tallennetaan observaatiotaulun koordinaatit niihin alkioihin, joiden kyselyyn on käytetty kyseistä solmua (*CTablePoint*). Esimerkkikuvassa alla näkyy *W*:n sisältö sen jälkeen, kun on hyväksytty merkkijonot (a,b)(c,VOID) ja (a,d)(e,VOID). Esimerkissä CTrieItem sisältää vanhemman ”*a*”, merkkijonon *c* ja oksana ”*VOID*”.



Kuva 4.2. Esimerkkisisältö *W*:stä

Luokka CTrieEdge – päätöspuun haara

Jokainen puun haara sisältää osoitteen sekä vanhempaan että lapseen. Lisäksi luokassa on viesti, jolla päästään kulkemaan vanhemmasta lapseen. Esimerkkikuvassa yllä (kuva 4.2) nähdään, että esimerkkinä toimiva CTrieEdge sisältää merkkijonon b (viesti), vanhemman ”*a*” ja lapsen ”*c*”. Juuresta lähtien ja esimerkkihaaraa seuraten saadaan siis merkkijono (a,b)(c,VOID).

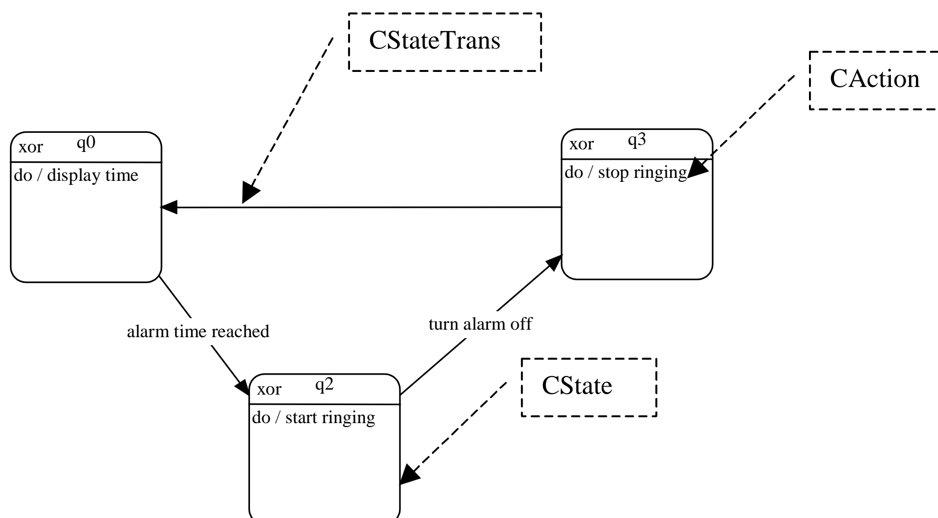
4.4.2 Tilakoneen luokat

Äärellinen tilakone luodaan konsistentista ja suljetusta observaatiotaulusta (katso tarkemmin luku 2.3 ja [24,26]). Tämän jälkeen tilakone voidaan siirtää esimerkiksi *TED*iin. Tilakoneluokka toimii myös äärellisten automaattien tuottajana.

Luokka CStateMachine – tilakone

Tilakone sijaitsee syntetisoijan tapaan yhdessä luokassa ja muut luokat määrittelevät tilakoneen tietotyypit (kuva 4.3). Tilakone alustetaan antamalla sille MAS-luokka, jolloin se käy läpi observaatiotaulun rivit ja luo niistä tilat ja tilasiirtymät. Nämä voidaan lukea tarvittaessa luokan metodeilla esimerkiksi siirrettäessä luokan tilakonetta *TED*iin.

Valmiista tilakoneesta (esimerkiksi sellaisesta, joka on haettu *TED*istä) voidaan luoda uusi MAS-luokan ilmentymä, joka on konsistentti ja suljettu ja joka määrittelee tilakoneen [24]. Tälle luodulle syntetisoijalle voidaan antaa esimerkkimerkkijono ja jatkaa syntetisointia normaalisti.



Kuva 4.3. Tilakoneen luokkajako

Luokka CStateTrans – tilakoneen tilasiirtymät

Tilakoneen (tai äärellisen automaatin) tilasiirtymä on määritelty luokassa CStateTrans. Tilasiirtymä sisältää tiedon lähtö- ja tulotilasta sekä viestin, jolla siirrytään toiseen tilaan.

Tilasiirtymät tallennetaan tilakoneen suojattuun jäsenmuuttujaan. Lisäksi jokaisella tilalla on tiedot sekä saapuvista että lähtevistä tilasiirtymistä. Lähtö- ja tulospisteet tallennetaan indeksinä tilavektoriin. Näin säästetään muistia etenkin suurissa tilakoneissa.

Luokka CState – tilakoneen tila

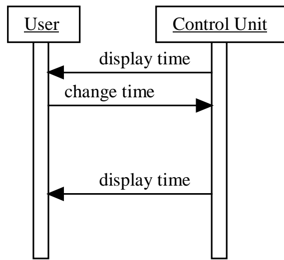
Tila sisältää tiedon lähtevistä ja tulevista tilasiirtymistä, vastaavan observaatiotaulun rivin numeron sekä toiminnot. Lisäksi ehdollisella kääntämisellä saadaan tilaan liitettyä myös *TED*in malli- ja näkymätiedot. Tilat säilytetään tilakoneen luokan suojatussa jäsenmuuttujassa.

4.4.3 Syntetisoijaluokan käyttö

Syntetisoijan käyttö omasta ohjelmasta on helppoa. Tarvittavia metodeita on vähän ja lähes kaikki toiminnot tapahtuvat luokan sisältä. Lyhyessä toteutuksessa pääsilmukaksi riittää kolme riviä. Yksi kutsuu alustusta, toinen iterointia ja kolmas tarkistaa iteroinnin paluuarvon ja luo tarvittaessa uuden CMAS-instanssin, jos vastaesimerkiksi annettiin äärellinen automaatti.

Syntetisoijan alustus

Syntetisoija alustetaan kutsumalla sen metodia `InitObservationTable.` Metodille annetaan merkkijono tai lista merkkijonoja, jonka luotava automaatti pitäisi hyväksyä. Kuva 4.4 sisältää esimerkkisyötteen, jolla halutaan alustaan syntetisoija. Tällöin kutsutaan luokan `InitObservationTable`-metodia parametrilla ”*(display time, change time)(display time, VOID)*”, jolloin metodi alustaa käyttämänsä tietorakenteet (puun, aakkoston ja observaatiotaulun).



Kuva 4.4. Esimerkkisyöte syntetisoijalle

Syntetisointi

Syntetisointi aloitetaan kutsumalla metodia `DoIteration`, jolle voidaan antaa parametriksi esimerkiksi lokitulostuksen tietovirta. Oletuksena tämä on `cout`, jolloin käytetään konsoli-ikkunaa.

Syntetisoija käy läpi MAS:in algoritmia (katso luku 2.3), kunnes joku seuraavista seikoista on tosi:

- Käyttäjä keskeyttää iteroinnin
- Käyttäjä hyväksyy luodun tilakoneen
- Käyttäjä antaa äärellisen automaatin vastaesimerkiksi.

Kahdessa ensimmäisessä tapauksessa voidaan myös ohjelman suoritus lopettaa, muuten tehdään uusi CMAS-luokan instanssi äärellisestä automaatista ja jatketaan iterointia kutsumalla sen `DoIteration`-metodia.

Kysymykset käyttäjältä

MAS-algoritmin hienoin idea on, että se osaa kysyä käyttäjältä konsultaatiota merkkijonoista, joista se itse ei osaa päätellä, kuuluvatko ne rakennettavan tilakoneen hyväksymään kieleen vai ei. Kysymykset tapahtuvat kahdessa eri tilanteessa:

- kun kysytään, kuuluuko annettu merkkijono hyväksyttyjen merkkijonojen ryhmään
- kun käyttäjältä kysytään generoidun automaatin hyväksymistä tai mahdollista vastaesimerkkiä.

Käyttäjäkysymykset on tehty käyttämällä virtuaalisia metodeja, jolloin algoritmin soveltaminen eri käyttöliittymiin helpottuu huomattavasti. Perusalgoritmia ei tarvitse muuttaa vaan käyttöliittymän mukauttamista varten määritellään uusi MAS-luokka, joka perii CMAS-luokan ja jossa käyttöliittymään liittyvät metodit on muutettu (kuten esimerkiksi *CWinMAS*).

Luokan CMAS metodi `AskForValue` palauttaa arvon `true/false`, riippuen siitä, hyväksyykö käyttäjä annetun merkkijonon. Oletustoteutuksena on yksinkertainen merkkiliittymä, jossa käyttäjälle näytetään haluttu merkkijono ja hän syöttää yksinkertaisesti 1, jos merkkijono hyväksytään tai vastaavasti 0, jos hylätään. Lopullisessa käyttöliittymässä metodi korvattiin dialogilla, joka muokkaa annetun merkkijonon sekvenssikaavioksi ja näyttää sen näytöllä.

Hyväksyminen hoidetaan luokan CMAS metodilla `AskForOK`, joka palauttaa mahdollisen vastaesimerkin tai tyhjän merkkijonon, jos käyttäjä hyväksyy luodun tilakoneen. Oletuksena käyttäjältä luetaan rivi, jossa käyttäjä antaa vastaesimerkin. Lopullisessa toteutuksessa käyttäjä piirtää vastaesimerkin TEDiin sekvenssikaaviona ja antaa ohjelmalle kaavio sijainnin. Lopulliseen käyttöliittymään lisättiin myös mahdollisuus hakea äärellinen automaatti, josta luodaan kokonaan uusi MAS (katso kohdat 4.4.2 ja 4.4.4 sekä lähde [24]).

4.4.4 Tilakoneluokan käyttö

Tilakoneluokka toimii väliasemana siirrettäessä tietoja syntetisoijasta piirto-ohjelmaan tai päinvastoin. Valmiista syntetisoijaluokan instanssista voidaan luoda helposti tilakone ja valmiista tilakoneesta on mahdollisuus luoda uusi syntetisoijaluokan instanssi.

Tilakoneen alustus

Tilakone alustetaan kutsumalla sen metodia `InitMachine`, jolle annetaan parametriksi syntetisoijan osoite. Alustusmetodi luo automaattisesti tilakoneen tilat ja siirtymät. Seuraavalla sivulla on esitetty lyhyt käyttöesimerkki, jossa aluksi on luotu syntetisoija. Tästä luodaan tilakone, joka tulostetaan konsolille. Esimerkissä ei ole annettu käyttäjälle mahdollisuutta antaa vastaesimerkkinä äärellistä automaattia.

```
CMAS mas;
CStateMachine sm;
bool ab=false;
mas.InitObservationTable("(s_ct,set)(s_at,NULL)(s_ct,reached)(buzz,off)(s_ct,VOID)");
mas.DoIteration(ab);
if (ab)
{
     cout << "Synthesizer aborted!" << endl;
     return 0;
}

sm.InitMachine(&mas);
sm.PrintMachine(cout);
```

Jos käyttäjä vastaa jäsenkyselyihin lähteessä [26] esitetyn esimerkkitaulun mukaisesti saadaan konsolille seuraava tuloste:

```
State set Q:q0=010 (row 0) q1=001 (row 1) q2=000 (row 3) q3=100 (row 5)
Initial state q0=010
Set of final states F:q3
Transition function:
q0:,(s_ct,set):q1
q0:,(s_ct,VOID):q3
q1:(s_ct,set),(s_at,NULL):q0
q2:(s_ct,set)(s_at,NULL)(s_ct,reached),(buzz,off):q0
q0:,(s_ct,reached):q2
```

Tulosteesta ilmenevät tilat (ja miltä observaatiotaulun riveiltä ne on luotu), alkutila, lopputila(t) ja tilasiirtymät.

Luokan CStateMachine käyttömahdollisuuksia

Tilakoneluokan päätarkoituksena on muuttaa observaatiotaulun sisältämä tieto sellaiseen muotoon, josta se on helppo siirtää esimerkiksi *TED*iin. Oletuksena luonnin jälkeen tilakoneen tilat ja tilasiirtymät määrittelevät alla olevan äärellisen automaatin *B*. Kuitenkin instanssin tiedot voidaan muuttaa helposti (katso luku 2 ja varsinkin kohta 2.3.2) tilakoneeksi kutsumalla luokan metodeita `RemoveExtraStates` ja `SetupActions`. Muutos ei ole peruuttamaton, sillä mitään tietoja ei hävitetä muutosprosessissa.

Toinen tilakoneen tarkoituksista on luoda äärellisestä automaatista uusi observaatiotaulu. Tämä tapahtuu alustamalla tilakoneluokka ja kutsumalla metodia `CreateMas`. Kutsu luo uuden instanssin syntetisoijaluokasta.

Alla esitetään esimerkki, miten äärellisestä automaatista tehdään uusi syntetisoija.

```
CStateMachine::STATESET stateset;
CStateMachine::TRANSITIONLIST tranlist;
CStateMachine machine;
CMAS* mas;
```

```
/* Alustetaan äärellinen automaatti, oletuksena tilasiirtymät sijaitsevat listassa
"tranlist" ja tilat joukossa "stateset". Nämä muuttujat on alustettu esimerkiksi
TEDistä saaduilla tiedoilla */

machine.SetTransitions(tranlist);
machine.SetStates(stateset);
machine.SetInitialState("q0");
machine.FindFinalState();
/* Luodaan uusi instanssi syntetisoijasta */
machine.CreateMas(&mas);
/* Tulostetaan sen observaatiotaulu */
mas->PrintTable(cout);
/* Käytetään tässä välissä syntetisoijaa muutenkin, esimerkiksi annetaan uusi
vastaesimerkki ja luodaan uusi tilakone */

/* Vapautetaan muisti, kun instanssia ei enää tarvita */
delete mas;
```

Tilakoneelta voidaan tarvittaessa hakea myös yksinkertaiset polut ja silmukat[2] [24].

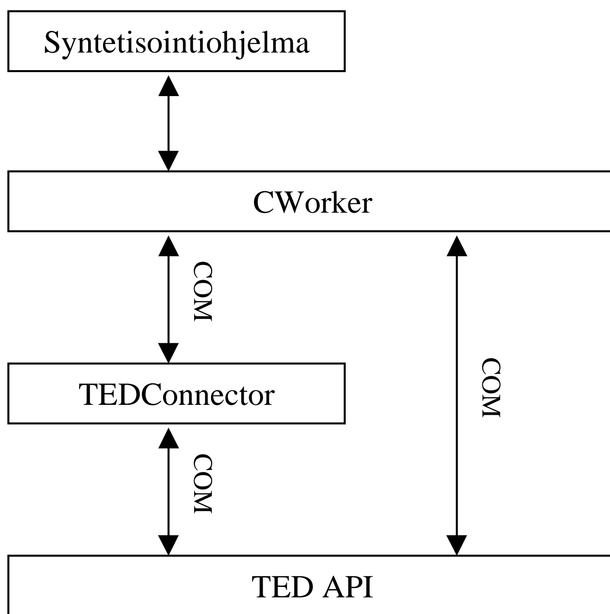
```
CStateMachine::PATHSET paths;
CStateMachine::LOOPSET loops;
paths=machine.GetSimplePaths();
loops=machine.GetSimpleLoops();
```

4.4.5 Muita luokkia

Edellä esitetyt luokat antavat perustan erilaisille MAS-syntetisointiohjelmille. Ne eivät kuitenkaan puutu mitenkään siihen, mitä kautta observaatiotaulun syöte on saatu tai mitä syntetisoinnissa saadulle tilakoneelle tehdään. Tätä tarkoitusta varten kehitettiin uusi luokka *CWorker* ja COM-palvelin *TEDConnector* (kuva 4.5).

---

[2] Yksinkertainen polku äärellisessä automaatissa on polku, joka ei käy missään tilassa useammin kuin kerran. Toisin sanoen polussa ei ole silmukoita. Yksinkertaisella silmukalla alku- ja lopputila on sama ja kaikki muut tilat ovat erilaisia toisiinsa ja alku- ja lopputilaan nähden.



Kuva 4.5. Syntetisointiohjelman TED-kerrokset

Riippumatta siitä, mitä käyttöliittymää käytetään, täytyy jokaisessa TEDiin liittyvässä syntetisointiohjelmassa tehdä samoja asioita. Nämä toimenpiteet ovat yhteyden avaaminen palvelimelle, halutun työkirjan etsiminen, sekvenssikaavion viestityksen lukeminen sekä tilakoneen käsittelyyn liittyvät asiat.

COM-palvelin TEDConnector – peruspalveluja yhteydenmuodostukseen

COM-palvelin *TEDConnector* toimii nimensä mukaisesti alhaisen tason (*low-level*) peruspalvelimena. Sen rajapintaan kuuluvat metodit yhteyden luomiseen, työkirjan osoitteen hakemiseen sekä sekvenssikaavion etsimiseen. Lisäksi palvelin hakee monessa eri tilanteissa tarvittavat malli- ja näkymäpuolen juuriosoittimet valmiiksi muiden luokkien käyttöön.

Yhteyden muodostamisen jälkeen *TEDConnector*-palvelimelta haetaan TEDin COM-palvelimen rajapinnan osoite, jonka jälkeen kaikki jatkotoimenpiteet voidaan hoitaa suoraan TEDin kanssa. Luokan, joka käyttää *TEDConnector*ia ei siis itse tarvitse huolehtia TEDin palvelimen luomisesta ja vapauttamisesta.

Luokka CWorker – raskaan työn tekijä

Luokka *CWorker* sisältää tarvittavat metodit, joilla yksinkertaistetaan huomattavasti pääohjelman työtä. Se käyttää *TEDConnector*ia läpinäkyvästi siten, että pääohjelman ei tarvitse välittää COM-rajapinnasta mitään, vaan kaikki TED-yhteydet hoidetaan luokan kutsujen kautta.

Pääohjelma aloitetaan yleensä yhteyden avaamisella. Tämä yleisesti tehty toimenpide olisi normaalisti hyvin raskas suorittaa, mutta apuluokan avulla se käy varsin kivuttomasti yhdellä rivillä.

```
CWorker worker;
hr=worker.Initialize(host, symbol);
```

Tämän jälkeen yleensä siirrytään käyttäjän valitsemaan työkirjaan käymällä rekursiivisesti työkirjoja läpi kunnes halutut ehdot täyttävä (yleensä oikean niminen) työkirja löytyy. Alla asetetaan työkirjan nimen perusteella järjestelmän oletustyökirja (vastaisi käyttöjärjestelmän käskyä *chdir*).

```
hr=worker.SetCurrentWorkbook(workbookName);
```

Tässä vaiheessa ollaan jo oikeassa paikassa, joten haetaan sekvenssikaavion tiedot muistiin ja siirretään ne merkkijonomuuttujaan *w*. Koska lähetetyt viestit ovat nimeltään yleensä aika pitkiä, pakataan samalla tieto pienempään tilaan. Sekvenssikaaviot voidaan hakea myös rekursiivisesti siten, että haetaan halutusta työkirjasta kaikki sekvenssikaaviot, joissa on annetun niminen olio sekä sen jälkeen kaikista alityökirjoista ehdot täyttävät sekvenssikaaviot.

```
hr=worker.SearchAndSetClassifierRole(participantName,true);
if (SUCCEEDED(hr)) {
     worker.PrepareItems();
     worker.CompressItems();
     w=worker.GetItemsAsString();
}
```

Koska tiedot ovat nyt muistissa, luodaan syntetisointiluokan uusi instanssi ja muodostetaan uusi tilakone.

```
worker.SetDefaultMAS();
CMAS* mas=worker.GetSynthesizer();
mas->InitObservationTable(w);
retval=mas->DoIteration(abortvalue);
/* käsitellään paluuarvo ja abortvalue... */
worker.GetStateMachine()->InitMachine(mas);
```

Lopuksi siirretään saatu tulos takaisin TEDin palvelimelle. Tämäkin monimutkainen asia hoituu pääohjelman kannalta yhdellä rivillä.

```
hr=worker.ImportToTED(&modelName,&viewName);
```

Joskus tarvitaan myös observaatiotaulun luomista äärellisestä automaatista (katso tarkemmin luku 4.4.4 ja lähde [24]). Luokka *CWorker* osaa myös luoda tilakoneen annetusta automaatista,

```
CStateMachine machine;
CMAS* mas;
hr=worker.ExportFromTed(&machine);
```

jolloin voidaan vastaava syntetisoija luoda tilakoneen perusteella helposti.

```
machine.CreateMas(&mas);
```

Kuten yllä olevista esimerkeistä huomataan, toimii apuluokka hyvin korkealla tasolla, jolloin myös ohjelmien siirto eri ympäristöihin helpottuu huomattavasti. Esimerkiksi TEDin vaihtuessa toiseen vastaavaan mallinnustyökaluun täytyy vain luokka *CWorker* ja palvelin *TEDConnector* kirjoittaa uudelleen – pääohjelmaan tai syntetisointialgoritmeihin ei tarvitse tehdä mitään muutoksia.

4.5 Käyttöliittymän toteutus

Ohjelmiston käyttöliittymä jakautuu kahteen osaan: TED-ohjelmaan ja syntetisoijan käyttöliittymään. Näistä ensimmäisessä käyttäjä piirtää sekvenssikaaviot ja sinne myös muodostetaan algoritmin tekemät tilakoneet. TED:in käyttöliittymän muokkaamiseen ei valitettavasti ole mitään työkaluja, joten MAS-syntetisoijaa täytyy käyttää kahdesta selkeästi erillisestä ohjelmasta. Tässä luvussa keskitytään syntetisoijan käyttöliittymään.

4.5.1 Käyttöliittymän eri osat

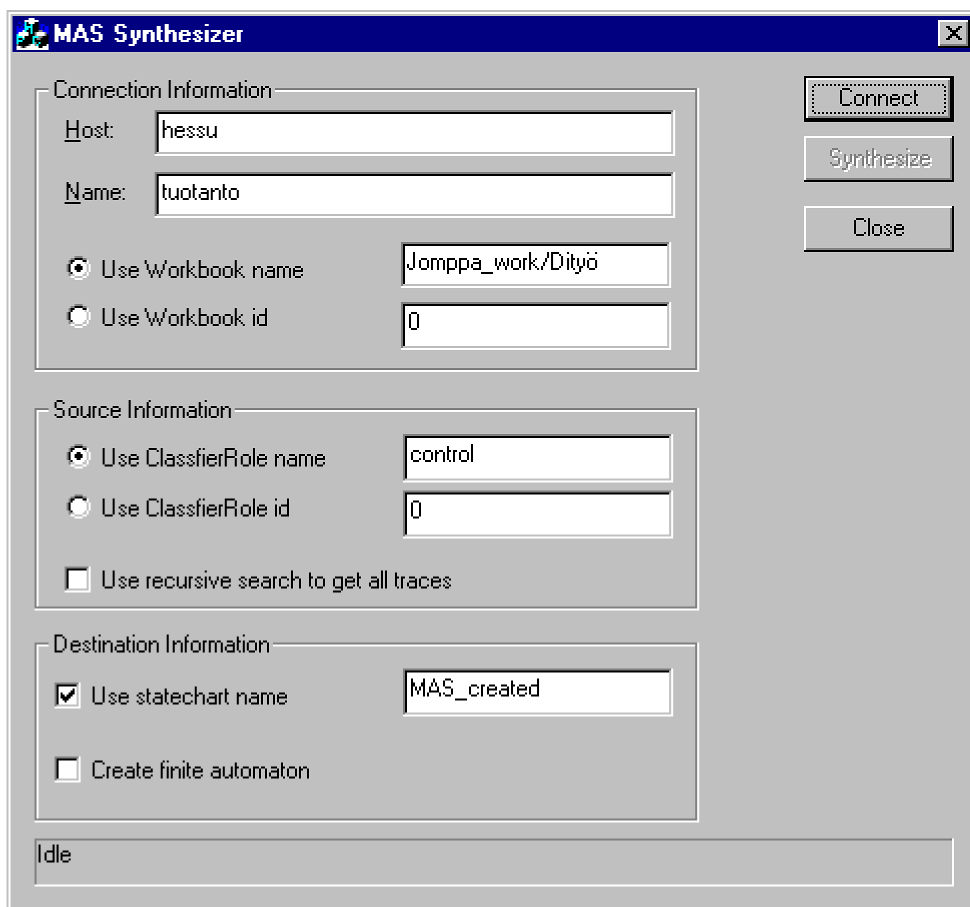
Syntetisoijan käyttöliittymä voidaan jakaa ajallisesti kolmeen osaan: Ennen suoritusta käytettävät osat, suorituksen aikana tarvittavat osat ja suorituksen jälkeiset osat. Yleisesti ottaen ennen suoritusta tarvitaan lähinnä linkki, mistä haetaan syntetisoitavat sekvenssikaaviot. Suorituksen aikana taas kysytään käyttäjältä sekvenssikaavion hyväksymisestä (eli kuuluuko merkkijono hyväksyttyihin kieliin). Suorituksen jälkeen tarvitaan linkki mahdolliseen vastaesimerkkiin tai muokattuun äärelliseen automaattiin.

4.5.2 Käyttöliittymä ennen algoritmin suoritusta

Ennen algoritmin suoritusta kysytään palvelinkoneen (*TED*) sijaintia sekä tietokannan nimeä (kuva 4.7). Käyttäjä voi valita, käytetäänkö työkirjasta nimeä vai tunnusta. Yhteys palvelimeen luodaan painamalla painiketta *Connect*. Jos yhteys epäonnistuu, ilmoitetaan siitä käyttäjälle (kuva 4.6).

Kuva 4.6. Palvelinyhteys epäonnistui

Kun yhteys on saatu, uuden yhteyden luominen estetään asettamalla *Connect*-painike harmaaksi. Painamalla nyt painiketta *Synthesize* palvelin hakee tietokannasta muutettavan sekvenssikaavion ja aloittaa syntetisointialgoritmin suorituksen. Tulos tallennetaan omaksi työkirjaksi, jonka nimi annetaan rivillä *Use statechart name*. Nimi voidaan myös olla antamatta, jolloin käytetään ohjelman tarjoamaa oletusnimeä.

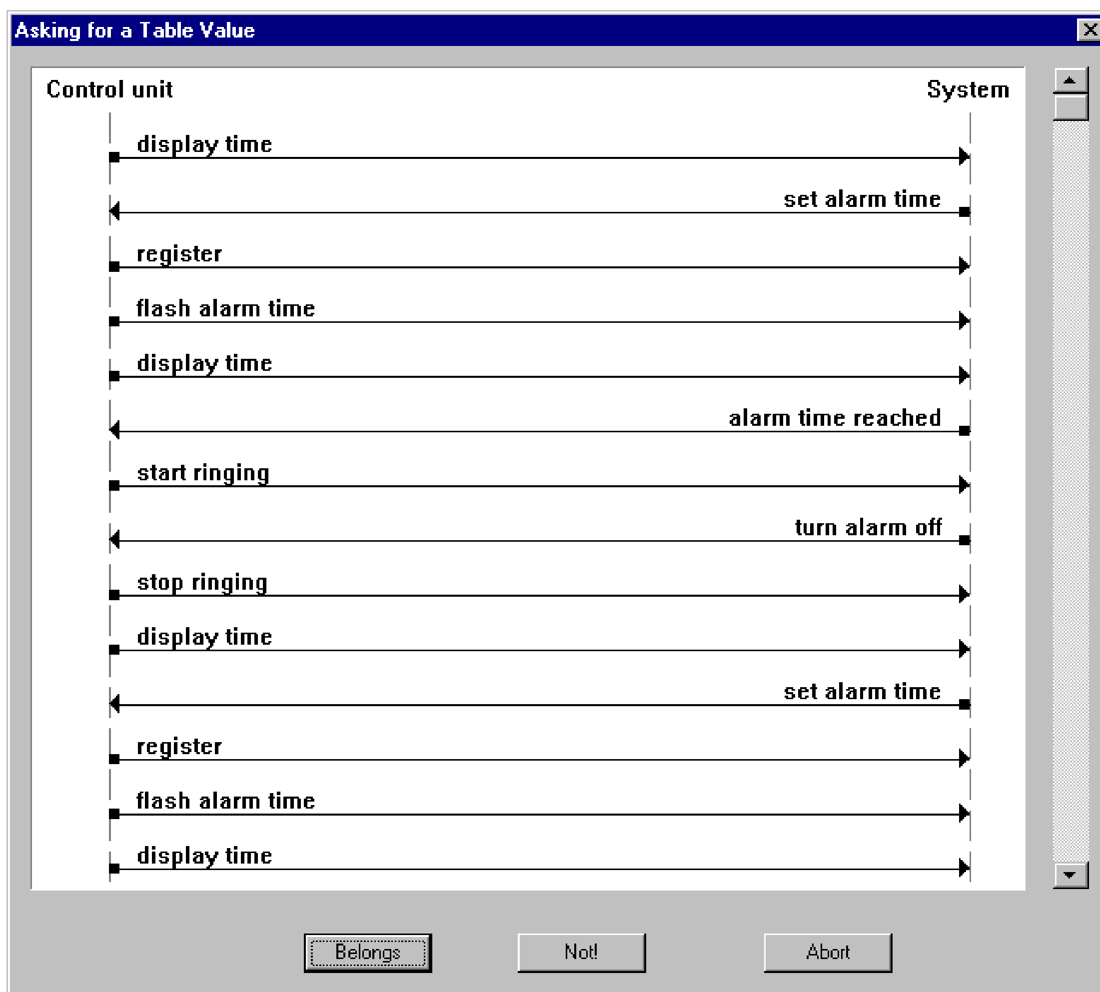


Kuva 4.7. Aloitusikkuna

4.5.3 Käyttöliittymä syntetisointialgoritmin suorituksen aikana

Ohjelma kysyy käyttäjältä syntetisointialgoritmin suorituksen aikana, pitääkö tuloksena syntyvän automaatin hyväksyä annettu sekvenssikaavio (eli kuuluuko käsiteltävä merkkijono hyväksyttyjen merkkijonojen joukkoon). Käyttäjää varten käsiteltävä merkkijono muutetaan sekvenssikaavioksi ja näytetään käyttäjälle. Sekvenssikaaviosta näytetään vain käsiteltävä olio sekä *System*, jolla kuvataan muuta järjestelmää. Näin siksi, koska syntetisoitaessa keskitytään nimenomaan yhteen olioon, eikä viestin vastaanottaja ole tärkeä. Ikkuna on esitetty alla olevassa kuvassa (kuva 4.8).

Käyttäjä hyväksyy näytetyn sekvenssin painamalla *Belongs*-painiketta. Jos taas käyttäjä on sitä mieltä, että sekvenssiä ei tulisi hyväksyä generoitavassa automaatissa, painetaan painiketta *Not!*. Käyttäjä voi peruuttaa syntetisointialgoritmin suorituksen painamalla painiketta *Abort*, jolloin palataan takaisin alkutilaan (katso kuva 4.7).

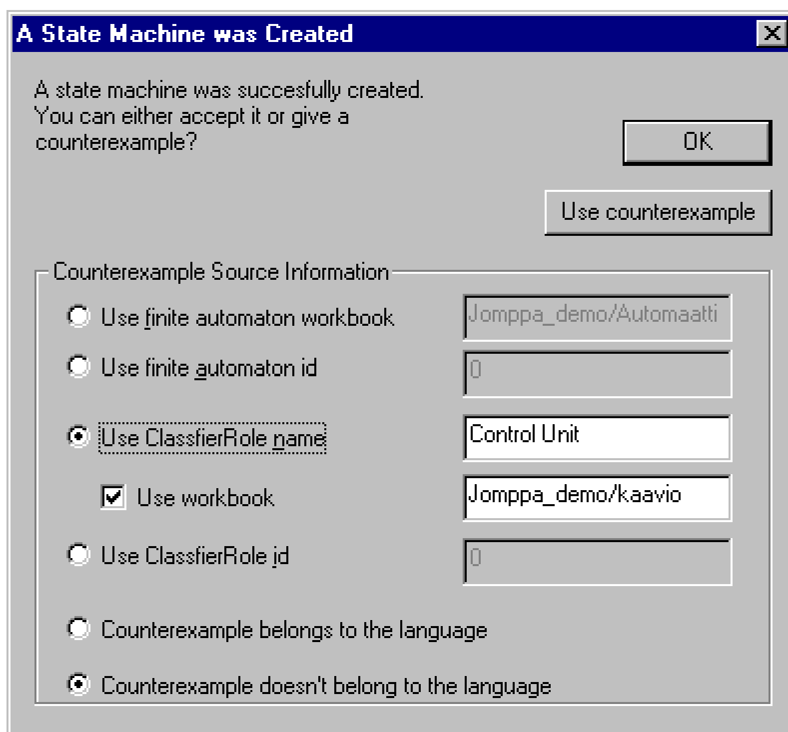


Kuva 4.8. Kyselyikkuna

4.5.4 Käyttöliittymä syntetisointialgoritmin suorituksen jälkeen

Suorituksen jälkeen käyttäjä voi hyväksyä algoritmin luoman tilakoneen (kuva 4.9). Käyttäjä voi myös antaa algoritmille linkin (vasta)esimerkkiin, joka annetaan sekvenssikaaviona, tai äärelliseen automaattiin (editoituun tilakoneeseen). Näistä ensimmäinen lisätään jo olemassa olevaan observaatiotauluun kun taas jälkimmäinen muodostaa kokonaan uuden observaatiotaulun. Uusi observaatiotaulu on automaattisesti konsistentti ja suljettu [24], joten hakemisen jälkeen täytyy antaa vastaesimerkki, jos algoritmin suoritusta halutaan jatkaa.

Käyttäjän antaessa algoritmille äärellisen automaatin ja tämän jälkeen sekvenssikaavion positiivisena vastaesimerkkinä, voidaan syntetisoida inkrementaalisesti jo olemassa olevia tilakaavioita.



Kuva 4.9. Tilakone luotu onnistuneesti

5. Kokemuksia järjestelmän toiminnasta

Tässä luvussa tarkastellaan syntetisointiohjelman toimintaa ja suorituskykyä. Kohdassa 5.1 käydään läpi vaiheittain annetun aineisto syntetisointi ja vastaesimerkkien antaminen. Kohdassa 5.2 tarkastellaan syntetisoinnin suoritusnopeutta käytännön testien pohjalta eli onko tässä esitetty järjestelmä tarpeeksi nopea tuotantokäyttöön sekä mahdolliset pullonkaulat. Lopuksi analysoidaan algoritmin teoreettinen nopeus.

5.1 Syntetisointiohjelman ajoesimerkki

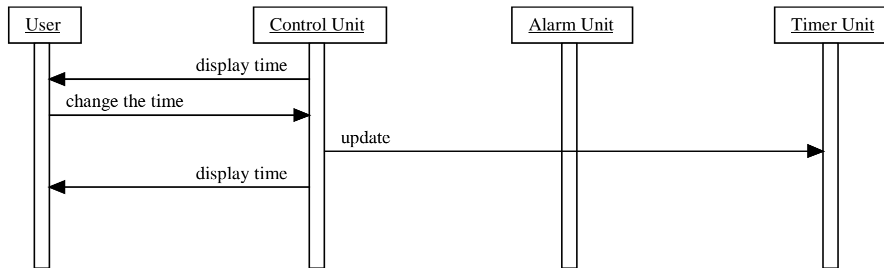
Tässä luvussa käydään vaiheittain läpi yksinkertainen kelloesimerkki [26]. Jokaisen vaiheen jälkeen näytetään algoritmin observaatiotaulu ja vastaava tilakone, joka on luotu TED-ohjelmaan.

Ensimmäinen vaihe – kello syntyy

Herätyskellon luominen aloitetaan yksinkertaisesti ajan näyttämisellä. Observaatiotaulun selkeyttä varten on taulussa käytetty seuraavia lyhenteitä:

Alkuperäinen	**Lyhenne**
display time	d_t
change the time	change
update	update

Syntetisointiohjelma käynnistettiin ja sille annettiin aloitussyötteeksi ensimmäinen sekvenssikaavio, joka kuvaa kellonajan asetusta (kuva 5.1). Tapahtumien kulku sekvenssikaaviossa on seuraava: näytetään aikaa, käyttäjä vaihtaa kellonaikaa, päivitetään kellonaika aikayksikköön (*Timer Unit)* ja lopuksi näytetään taas aikaa.



Kuva 5.1. Ensimmäinen sekvenssikaavio

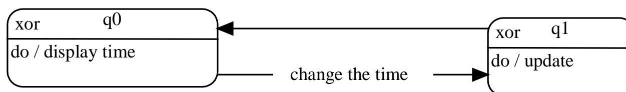
Ohjelma kysyi käyttäjältä konsultaatiota yhden kerran. Tämä on merkitty vahvennuksella observaatiotaulussa (taulukko 5.1). Kysymys vastaa tilannetta, jossa kellon sallitaan näyttää vain pelkkää aikaa, eikä käyttäjä halua muuttaa sitä.

Taulukko 5.1 Ensimmäinen observaatiotaulu.

T	r_1	r_2
λ	0	1
(d_t,change)	0	0
(d_t,change)(update,NULL)	0	1
(d_t,change)(update,NULL)(d_t,VOID)	1	0
(update,NULL)	0	0
(d_t,VOID)	**1**	0
(d_t,change)(d_t,change)	0	0
(d_t,change)(d_t,VOID)	0	0
(d_t,change)(update,NULL)(d_t,change)	0	0
(d_t,change)(update,NULL)(update,NULL)	0	0

r_1=λ, r_2=*(d_t,VOID)*

Ohjelman tuottama tilakone on nähtävissä alla (kuva 5.2). Tuotos on hyvin yksinkertainen sisältäen vain kaksi eri tilaa. Algoritmia (sivulla 17) tutkimalla voidaan havaita observaatiotaulusta tilat q_0=01 (rivi 1, on myös alkutila), q_1=00 (rivi 2) ja q_2=10 (rivi 4). Koska q_2 on lopputila, sitä ei näytetä alla olevassa kuvassa.

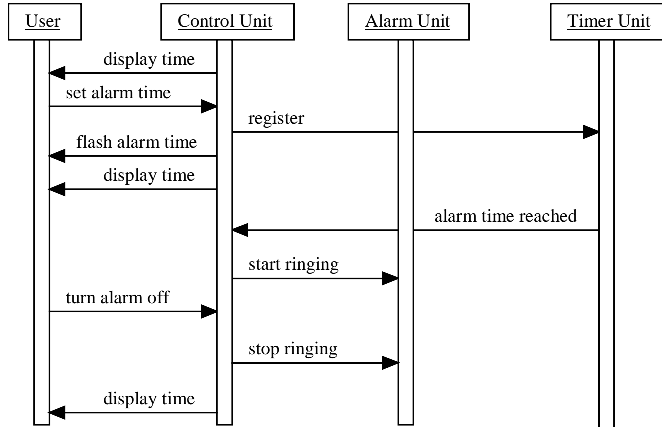


Kuva 5.2. Ensimmäinen tulos

Toinen vaihe – kellosta herätyskello

Ajoesimerkin toisessa vaiheessa tavallisesta kellosta tehtiin herätyskello. Tämä tapahtui antamalla syntetisoijalle positiivinen vastaesimerkki sekvenssikaaviona (kuva 5.3). An-

netussa kaaviossa kuvataan herätysajan asetus ja kellon tuottama herätys. Tässä kaaviossa oletetaan, että *set alarm time* asettaa samalla sekä herätyksen päälle että herätysajan.



Kuva 5.3. Kellosta herätyskello

Tällä kertaa ohjelma kysyi käyttäjältä apua kuusi kertaa. Observaatiotaulun selkeyttä varten on taulussa käytetty aikaisempien lisäksi seuraavia lyhenteitä:

Alkuperäinen	**Lyhenne**
set alarm time	set
register	reg
flash alarm time	flash
alarm time reached	reached
start ringing	ring
turn alarm off	off
stop ringing	stop

Kuten observaatiotaulusta (taulukko 5.2) huomataan, kasvaa taulun koko melko nopeasti (toisessa vaiheessa kuusi saraketta ja 82 riviä). Kuitenkin käyttäjältä kysyttävien kysymyksien määrä pysyy suhteellisen pienenä. Observaatiotaulussa huomaamme tilat q_0=010000 (rivi 1, alkutila), q_1=001000 (rivi 2), q_2=100000 (rivi 4, lopputila), q_3=000100 (rivi 7), q_4=000000 (rivi 8), q_5=000010 (rivi 10) ja q_6=000001 (rivi 11).

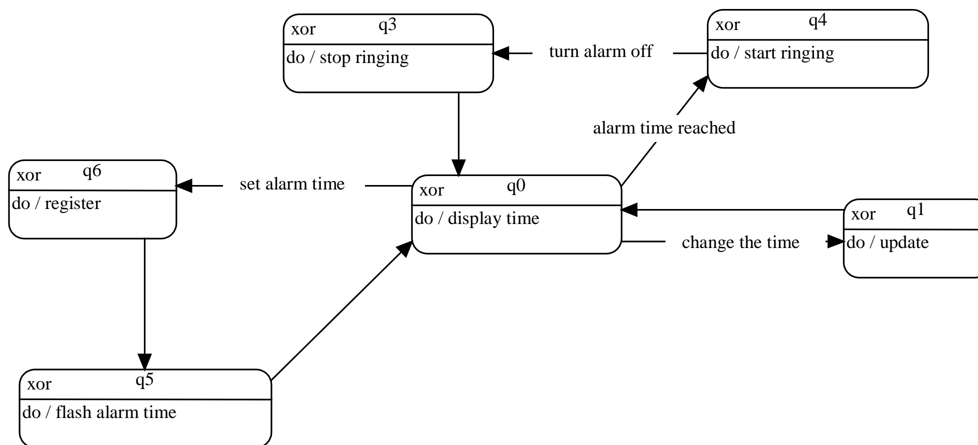
Taulukko 5.2 Observaatiotaulu herätyksen lisäyksen jälkeen.

T	r_1	r_2	r_3	r_4	r_5	r_6
λ	0	1	0	0	0	0
(d_t,change)	0	0	1	0	0	0
(d_t,change)(update,NULL)	0	1	0	0	0	0
(d_t,change)(update,NULL)(d_t,VOID)	1	0	0	0	0	0
(d_t,set)(reg,NULL)(flash,NULL)(d_t,reached)(ring,off)(stop,NULL)(d_t,VOID)	1	0	0	0	0	0
(d_t,set)(reg,NULL)(flash,NULL)(d_t,reached)(ring,off)(stop,NULL)	0	1	0	0	0	0
(d_t,set)(reg,NULL)(flash,NULL)(d_t,reached)(ring,off)	0	0	0	1	0	0
(d_t,set)(reg,NULL)(flash,NULL)(d_t,reached)	0	0	0	0	0	0
(d_t,set)(reg,NULL)(flash,NULL)	0	1	0	0	0	0
(d_t,set)(reg,NULL)	0	0	0	0	1	0
(d_t,set)	0	0	0	0	0	1
(update,NULL)	0	0	0	0	0	0
(d_t,VOID)	1	0	0	0	0	0
(d_t,change)(d_t,change)	0	0	0	0	0	0
(d_t,change)(d_t,VOID)	0	0	0	0	0	0
(d_t,change)(update,NULL)(d_t,change)	0	0	**1**	0	0	0
(d_t,change)(update,NULL)(update,NULL)	0	0	0	0	0	0
(reg,NULL)	0	0	0	0	0	0
(flash,NULL)	0	0	0	0	0	0
(d_t,reached)	0	0	0	0	0	0
(ring,off)	0	0	0	0	0	0
(stop,NULL)	0	0	0	0	0	0
(d_t,change)(d_t,set)	0	0	0	0	0	0
(d_t,change)(reg,NULL)	0	0	0	0	0	0
(d_t,change)(flash,NULL)	0	0	0	0	0	0
(d_t,change)(d_t,reached)	0	0	0	0	0	0
(d_t,change)(ring,off)	0	0	0	0	0	0
(d_t,change)(stop,NULL)	0	0	0	0	0	0
(d_t,change)(update,NULL)(d_t,set)	0	0	0	0	0	**1**
(d_t,change)(update,NULL)(reg,NULL)	0	0	0	0	0	0
(d_t,change)(update,NULL)(flash,NULL)	0	0	0	0	0	0
(d_t,change)(update,NULL)(d_t,reached)	0	0	0	0	0	0
(d_t,change)(update,NULL)(ring,off)	0	0	0	0	0	0
(d_t,change)(update,NULL)(stop,NULL)	0	0	0	0	0	0
(d_t,set)(reg,NULL)(flash,NULL)(d_t,reached)(ring,off)(stop,NULL)(d_t,change)	0	0	**1**	0	0	0
(d_t,set)(reg,NULL)(flash,NULL)(d_t,reached)(ring,off)(stop,NULL)(update,NULL)	0	0	0	0	0	0
(d_t,set)(reg,NULL)(flash,NULL)(d_t,reached)(ring,off)(stop,NULL)(d_t,set)	0	0	0	0	0	**1**
(d_t,set)(reg,NULL)(flash,NULL)(d_t,reached)(ring,off)(stop,NULL)(reg,NULL)	0	0	0	0	0	0
(d_t,set)(reg,NULL)(flash,NULL)(d_t,reached)(ring,off)(stop,NULL)(flash,NULL)	0	0	0	0	0	0
(d_t,set)(reg,NULL)(flash,NULL)(d_t,reached)(ring,off)(stop,NULL)(d_t,reached)	0	0	0	0	0	0
(d_t,set)(reg,NULL)(flash,NULL)(d_t,reached)(ring,off)(stop,NULL)(ring,off)	0	0	0	0	0	0
(d_t,set)(reg,NULL)(flash,NULL)(d_t,reached)(ring,off)(stop,NULL)(stop,NULL)	0	0	0	0	0	0
(d_t,set)(reg,NULL)(flash,NULL)(d_t,reached)(ring,off)(d_t,change)	0	0	0	0	0	0
(d_t,set)(reg,NULL)(flash,NULL)(d_t,reached)(ring,off)(update,NULL)	0	0	0	0	0	0
(d_t,set)(reg,NULL)(flash,NULL)(d_t,reached)(ring,off)(d_t,VOID)	0	0	0	0	0	0
(d_t,set)(reg,NULL)(flash,NULL)(d_t,reached)(ring,off)(d_t,set)	0	0	0	0	0	0
(d_t,set)(reg,NULL)(flash,NULL)(d_t,reached)(ring,off)(reg,NULL)	0	0	0	0	0	0
(d_t,set)(reg,NULL)(flash,NULL)(d_t,reached)(ring,off)(flash,NULL)	0	0	0	0	0	0
(d_t,set)(reg,NULL)(flash,NULL)(d_t,reached)(ring,off)(d_t,reached)	0	0	0	0	0	0
(d_t,set)(reg,NULL)(flash,NULL)(d_t,reached)(ring,off)(ring,off)	0	0	0	0	0	0
(d_t,set)(reg,NULL)(flash,NULL)(d_t,reached)(d_t,change)	0	0	0	0	0	0
(d_t,set)(reg,NULL)(flash,NULL)(d_t,reached)(update,NULL)	0	0	0	0	0	0
(d_t,set)(reg,NULL)(flash,NULL)(d_t,reached)(d_t,VOID)	0	0	0	0	0	0
(d_t,set)(reg,NULL)(flash,NULL)(d_t,reached)(d_t,set)	0	0	0	0	0	0
(d_t,set)(reg,NULL)(flash,NULL)(d_t,reached)(reg,NULL)	0	0	0	0	0	0
(d_t,set)(reg,NULL)(flash,NULL)(d_t,reached)(flash,NULL)	0	0	0	0	0	0
(d_t,set)(reg,NULL)(flash,NULL)(d_t,reached)(d_t,reached)	0	0	0	0	0	0
(d_t,set)(reg,NULL)(flash,NULL)(d_t,reached)(stop,NULL)	0	0	0	0	0	0
(d_t,set)(reg,NULL)(flash,NULL)(d_t,change)	0	0	**1**	0	0	0
(d_t,set)(reg,NULL)(flash,NULL)(update,NULL)	0	0	0	0	0	0
(d_t,set)(reg,NULL)(flash,NULL)(d_t,VOID)	1	0	0	0	0	0
(d_t,set)(reg,NULL)(flash,NULL)(d_t,set)	0	0	0	0	0	**1**
(d_t,set)(reg,NULL)(flash,NULL)(reg,NULL)	0	0	0	0	0	0
(d_t,set)(reg,NULL)(flash,NULL)(flash,NULL)	0	0	0	0	0	0
(d_t,set)(reg,NULL)(flash,NULL)(ring,off)	0	0	0	0	0	0
(d_t,set)(reg,NULL)(flash,NULL)(stop,NULL)	0	0	0	0	0	0
(d_t,set)(reg,NULL)(d_t,change)	0	0	0	0	0	0
(d_t,set)(reg,NULL)(update,NULL)	0	0	0	0	0	0

(d_t,set)(reg,NULL)(d_t,VOID)	0	0	0	0	0	0
(d_t,set)(reg,NULL)(d_t,set)	0	0	0	0	0	0
(d_t,set)(reg,NULL)(reg,NULL)	0	0	0	0	0	0
(d_t,set)(reg,NULL)(d_t,reached)	0	0	0	0	0	0
(d_t,set)(reg,NULL)(ring,off)	0	0	0	0	0	0
(d_t,set)(reg,NULL)(stop,NULL)	0	0	0	0	0	0
(d_t,set)(d_t,change)	0	0	0	0	0	0
(d_t,set)(update,NULL)	0	0	0	0	0	0
(d_t,set)(d_t,VOID)	0	0	0	0	0	0
(d_t,set)(d_t,set)	0	0	0	0	0	0
(d_t,set)(flash,NULL)	0	0	0	0	0	0
(d_t,set)(d_t,reached)	0	0	0	0	0	0
(d_t,set)(ring,off)	0	0	0	0	0	0
(d_t,set)(stop,NULL)	0	0	0	0	0	0

$r_1=\lambda$, r_2=*(d_t,VOID)*, r_3=*(update,NULL)(d_t,VOID)*, r_4=*(stop,NULL)(d_t,VOID)*,
r_5=*(flash,NULL)(d_t,VOID)*, r_6=*(reg,NULL)(flash,NULL)(d_t,VOID)*.

Algoritmin tuottama toinen yritys (kuva 5.4) vastaa lähes haluttua tilakonetta. Tarkemmin katsottaessa huomataan kuitenkin, että kello voi soida, vaikka herätystä ei olekaan asetettu (toisin sanoen q_0->q_4 voi tapahtua ennen kuin q_0->q_6->q_5->q_0).



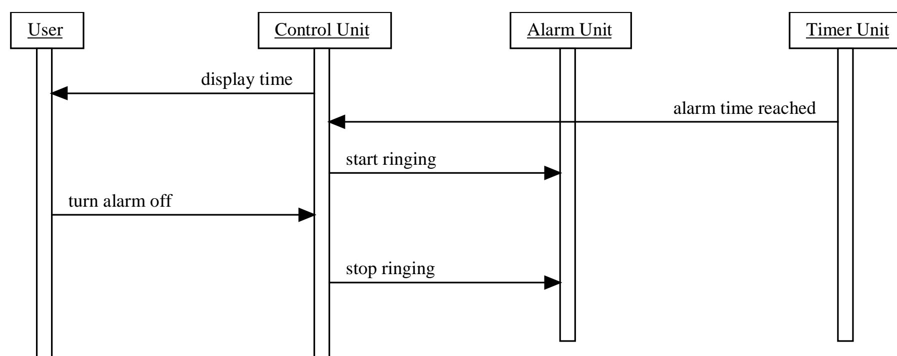
Kuva 5.4. Toinen yritys: kello herätyksellä

Viimeinen vaihe – herätys vasta asetuksen jälkeen

Edellisessä vaiheessa huomattiin, että lopputulos (kuva 5.4) ei vastannut aivan sitä, mitä herätyskellolta vaaditaan. Tämän korjaamiseksi annamme negatiivisen vastaesimerkin (kuva 5.5), jossa kerromme, että pelkkä herätys ilman herätyksen asetusta ei ole sallittua. Yleensä vain positiiviset esimerkit annetaan sekvenssikaaviona ja vastaavasti negatiiviset muutokset hoidetaan editoimalla syntynyttä tilakaaviota. Tässä tapauksessa sekvenssikaavio on kuitenkin kätevämpi tapa ilmaista haluttu asia.

Jos käytössä olisi mahdollisuus asettaa kiellettyjä merkkijonoja tai oletuksia (katso luku 7.2), voitaisiin käyttäjäkysymyksiä vähentää kertomalla syntetisoijalle, että herätys (*alarm*

time reached) ei saa tapahtua ilman herätyksen asetusta. Tällöin ei tätä kolmatta vaihetta tarvitsisi suorittaa ollenkaan, vaan herätyksen järjestys olisi hoidettu automaattisesti jo toisessa vaiheessa.



Kuva 5.5. Viimeinen sekvenssikaavio

Viimeisen vastaesimerkin jälkeen ohjelma kysyy vielä viisi kertaa (taulukko 5.3). Observaatiotaulun rivien määrä on noussut jo suureksi (kahdeksan saraketta ja 112 riviä), mutta koko ajon aikana ohjelma on kysynyt käyttäjältä vain 12 kertaa. Käyttäjän konsultaatioprosentti[3] oli siis noin 1%. Tämä ei ole todellakaan paljon, ja jos suhdeluku säilyy samana, voidaan syntetisointia käyttää suurienkin observaatiotaulujen kanssa.

[3] Observaatiotaulussa olevien käyttäjältä kysyttyjen (”Pitääkö annettu sekvenssikaavio hyväksyä, kyllä/ei?”) arvojen (0/1) määrä verrattuna kaikkiin taulun arvoihin.

Taulukko 5.3. Viimeinen taulu.

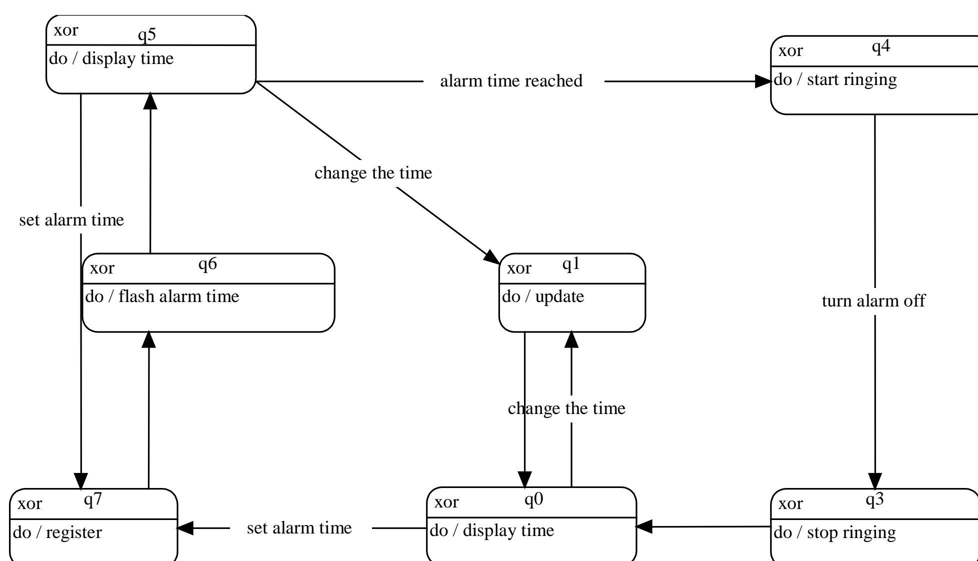
T	r_1	r_2	r_3	r_4	r_5	r_6	r_7	r_8
λ	0	1	0	0	0	0	0	0
(d_t,change)	0	0	1	0	0	0	0	0
(d_t,change)(update,NULL)	0	1	0	0	0	0	0	0
(d_t,change)(update,NULL)(d_t,VOID)	1	0	0	0	0	0	0	0
(d_t,set)(reg,NULL)(flash,NULL)(d_t,reached)(ring,off)(stop,NULL)(d_t,VOID)	1	0	0	0	0	0	0	0
(d_t,set)(reg,NULL)(flash,NULL)(d_t,reached)(ring,off)(stop,NULL)	0	1	0	0	0	0	0	0
(d_t,set)(reg,NULL)(flash,NULL)(d_t,reached)(ring,off)	0	0	0	1	0	0	0	0
(d_t,set)(reg,NULL)(flash,NULL)(d_t,reached)	0	0	0	0	0	0	1	0
(d_t,set)(reg,NULL)(flash,NULL)	0	1	0	0	0	0	0	1
(d_t,set)(reg,NULL)	0	0	0	0	1	0	0	0
(d_t,set)	0	0	0	0	0	1	0	0
(d_t,reached)(ring,off)(stop,VOID)	0	0	0	0	0	0	0	0
(d_t,reached)(ring,off)	0	0	0	**0**	0	0	0	0
(d_t,reached)	0	0	0	0	0	0	0	0
(update,NULL)	0	0	0	0	0	0	0	0
(d_t,VOID)	1	0	0	0	0	0	0	0
(d_t,change)(d_t,change)	0	0	0	0	0	0	0	0
(d_t,change)(d_t,VOID)	0	0	0	0	0	0	0	0
(d_t,change)(update,NULL)(d_t,change)	0	0	1	0	0	0	0	0
(d_t,change)(update,NULL)(update,NULL)	0	0	0	0	0	0	0	0
(reg,NULL)	0	0	0	0	0	0	0	0
(flash,NULL)	0	0	0	0	0	0	0	0
(d_t,reached)	0	0	0	0	0	0	0	0
(ring,off)	0	0	0	0	0	0	0	0
(stop,NULL)	0	0	0	0	0	0	0	0
(d_t,change)(d_t,set)	0	0	0	0	0	0	0	0
(d_t,change)(reg,NULL)	0	0	0	0	0	0	0	0
(d_t,change)(flash,NULL)	0	0	0	0	0	0	0	0
(d_t,change)(d_t,reached)	0	0	0	0	0	0	0	0
(d_t,change)(ring,off)	0	0	0	0	0	0	0	0
(d_t,change)(stop,NULL)	0	0	0	0	0	0	0	0
(d_t,change)(update,NULL)(d_t,set)	0	0	0	0	0	1	0	0
(d_t,change)(update,NULL)(reg,NULL)	0	0	0	0	0	0	0	0
(d_t,change)(update,NULL)(flash,NULL)	0	0	0	0	0	0	0	0
(d_t,change)(update,NULL)(d_t,reached)	0	0	0	0	0	0	**0**	0
(d_t,change)(update,NULL)(ring,off)	0	0	0	0	0	0	0	0
(d_t,change)(update,NULL)(stop,NULL)	0	0	0	0	0	0	0	0
(d_t,set)(reg,NULL)(flash,NULL)(d_t,reached)(ring,off)(stop,NULL)(d_t,change)	0	0	1	0	0	0	0	0
(d_t,set)(reg,NULL)(flash,NULL)(d_t,reached)(ring,off)(stop,NULL)(update,NULL)	0	0	0	0	0	0	0	0
(d_t,set)(reg,NULL)(flash,NULL)(d_t,reached)(ring,off)(stop,NULL)(d_t,set)	0	0	0	0	0	1	0	0
(d_t,set)(reg,NULL)(flash,NULL)(d_t,reached)(ring,off)(stop,NULL)(reg,NULL)	0	0	0	0	0	0	0	0
(d_t,set)(reg,NULL)(flash,NULL)(d_t,reached)(ring,off)(stop,NULL)(flash,NULL)	0	0	0	0	0	0	0	0
(d_t,set)(reg,NULL)(flash,NULL)(d_t,reached)(ring,off)(stop,NULL)(d_t,reached)	0	0	0	0	0	0	**0**	0
(d_t,set)(reg,NULL)(flash,NULL)(d_t,reached)(ring,off)(stop,NULL)(ring,off)	0	0	0	0	0	0	0	0
(d_t,set)(reg,NULL)(flash,NULL)(d_t,reached)(ring,off)(stop,NULL)(stop,NULL)	0	0	0	0	0	0	0	0
(d_t,set)(reg,NULL)(flash,NULL)(d_t,reached)(ring,off)(d_t,change)	0	0	0	0	0	0	0	0
(d_t,set)(reg,NULL)(flash,NULL)(d_t,reached)(ring,off)(update,NULL)	0	0	0	0	0	0	0	0
(d_t,set)(reg,NULL)(flash,NULL)(d_t,reached)(ring,off)(d_t,VOID)	0	0	0	0	0	0	0	0
(d_t,set)(reg,NULL)(flash,NULL)(d_t,reached)(ring,off)(d_t,set)	0	0	0	0	0	0	0	0
(d_t,set)(reg,NULL)(flash,NULL)(d_t,reached)(ring,off)(reg,NULL)	0	0	0	0	0	0	0	0
(d_t,set)(reg,NULL)(flash,NULL)(d_t,reached)(ring,off)(flash,NULL)	0	0	0	0	0	0	0	0
(d_t,set)(reg,NULL)(flash,NULL)(d_t,reached)(ring,off)(d_t,reached)	0	0	0	0	0	0	0	0
(d_t,set)(reg,NULL)(flash,NULL)(d_t,reached)(ring,off)(ring,off)	0	0	0	0	0	0	0	0
(d_t,set)(reg,NULL)(flash,NULL)(d_t,reached)(d_t,change)	0	0	0	0	0	0	0	0
(d_t,set)(reg,NULL)(flash,NULL)(d_t,reached)(update,NULL)	0	0	0	0	0	0	0	0
(d_t,set)(reg,NULL)(flash,NULL)(d_t,reached)(d_t,VOID)	0	0	0	0	0	0	0	0
(d_t,set)(reg,NULL)(flash,NULL)(d_t,reached)(d_t,set)	0	0	0	0	0	0	0	0
(d_t,set)(reg,NULL)(flash,NULL)(d_t,reached)(reg,NULL)	0	0	0	0	0	0	0	0
(d_t,set)(reg,NULL)(flash,NULL)(d_t,reached)(flash,NULL)	0	0	0	0	0	0	0	0
(d_t,set)(reg,NULL)(flash,NULL)(d_t,reached)(d_t,reached)	0	0	0	0	0	0	0	0
(d_t,set)(reg,NULL)(flash,NULL)(d_t,reached)(stop,NULL)	0	0	0	0	0	0	0	0
(d_t,set)(reg,NULL)(flash,NULL)(d_t,change)	0	0	1	0	0	0	0	0
(d_t,set)(reg,NULL)(flash,NULL)(update,NULL)	0	0	0	0	0	0	0	0
(d_t,set)(reg,NULL)(flash,NULL)(d_t,VOID)	1	0	0	0	0	0	0	0
(d_t,set)(reg,NULL)(flash,NULL)(d_t,set)	0	0	0	0	0	1	0	0
(d_t,set)(reg,NULL)(flash,NULL)(reg,NULL)	0	0	0	0	0	0	0	0
(d_t,set)(reg,NULL)(flash,NULL)(flash,NULL)	0	0	0	0	0	0	0	0
(d_t,set)(reg,NULL)(flash,NULL)(ring,off)	0	0	0	0	0	0	0	0

(d_t,set)(reg,NULL)(flash,NULL)(stop,NULL)	0	0	0	0	0	0	0	0
(d_t,set)(reg,NULL)(d_t,change)	0	0	0	0	0	0	0	0
(d_t,set)(reg,NULL)(update,NULL)	0	0	0	0	0	0	0	0
(d_t,set)(reg,NULL)(d_t,VOID)	0	0	0	0	0	0	0	0
(d_t,set)(reg,NULL)(d_t,set)	0	0	0	0	0	0	0	0
(d_t,set)(reg,NULL)(reg,NULL)	0	0	0	0	0	0	0	0
(d_t,set)(reg,NULL)(d_t,reached)	0	0	0	0	0	0	0	0
(d_t,set)(reg,NULL)(ring,off)	0	0	0	0	0	0	0	0
(d_t,set)(reg,NULL)(stop,NULL)	0	0	0	0	0	0	0	0
(d_t,set)(d_t,change)	0	0	0	0	0	0	0	0
(d_t,set)(update,NULL)	0	0	0	0	0	0	0	0
(d_t,set)(d_t,VOID)	0	0	0	0	0	0	0	0
(d_t,set)(d_t,set)	0	0	0	0	0	0	0	0
(d_t,set)(flash,NULL)	0	0	0	0	0	0	0	0
(d_t,set)(d_t,reached)	0	0	0	0	0	0	0	0
(d_t,set)(ring,off)	0	0	0	0	0	0	0	0
(d_t,set)(stop,NULL)	0	0	0	0	0	0	0	0
(stop,VOID)	0	0	0	0	0	0	0	0
(d_t,change)(stop,VOID)	0	0	0	0	0	0	0	0
(d_t,change)(update,NULL)(stop,VOID)	0	0	0	0	0	0	0	0
(d_t,set)(reg,NULL)(flash,NULL)(d_t,reached)(ring,off)(stop,NULL)(stop,VOID)	0	0	0	0	0	0	0	0
(d_t,set)(reg,NULL)(flash,NULL)(d_t,reached)(ring,off)(stop,VOID)	**1**	0	0	0	0	0	0	0
(d_t,set)(reg,NULL)(flash,NULL)(d_t,reached)(stop,VOID)	0	0	0	0	0	0	0	0
(d_t,set)(reg,NULL)(flash,NULL)(stop,VOID)	0	0	0	0	0	0	0	0
(d_t,set)(reg,NULL)(stop,VOID)	0	0	0	0	0	0	0	0
(d_t,set)(stop,VOID)	0	0	0	0	0	0	0	0
(d_t,reached)(ring,off)(d_t,change)	0	0	0	0	0	0	0	0
(d_t,reached)(ring,off)(update,NULL)	0	0	0	0	0	0	0	0
(d_t,reached)(ring,off)(d_t,VOID)	0	0	0	0	0	0	0	0
(d_t,reached)(ring,off)(d_t,set)	0	0	0	0	0	0	0	0
(d_t,reached)(ring,off)(reg,NULL)	0	0	0	0	0	0	0	0
(d_t,reached)(ring,off)(flash,NULL)	0	0	0	0	0	0	0	0
(d_t,reached)(ring,off)(d_t,reached)	0	0	0	0	0	0	0	0
(d_t,reached)(ring,off)(ring,off)	0	0	0	0	0	0	0	0
(d_t,reached)(ring,off)(stop,NULL)	0	0	0	0	0	0	0	**0**
(d_t,reached)(d_t,change)	0	0	0	0	0	0	0	0
(d_t,reached)(update,NULL)	0	0	0	0	0	0	0	0
(d_t,reached)(d_t,VOID)	0	0	0	0	0	0	0	0
(d_t,reached)(d_t,set)	0	0	0	0	0	0	0	0
(d_t,reached)(reg,NULL)	0	0	0	0	0	0	0	0
(d_t,reached)(flash,NULL)	0	0	0	0	0	0	0	0
(d_t,reached)(d_t,reached)	0	0	0	0	0	0	0	0
(d_t,reached)(stop,NULL)	0	0	0	0	0	0	0	0
(d_t,reached)(stop,VOID)	0	0	0	0	0	0	0	0

$r_1=\lambda$, $r_2=$*(d_t,VOID)*, $r_3=$*(update,NULL)(d_t,VOID)*, $r_4=$*(stop,NULL)(d_t,VOID)*, $r_5=$*(flash,NULL)(d_t,VOID)*, $r_6=$*(reg,NULL)(flash,NULL)(d_t,VOID)*, $r_7=$*(ring,off)(stop,NULL)(d_t,VOID)*, $r_8=$*(d_t,reached)(ring,off)(stop,NULL)(d_t,VOID)*.

Observaatiotaulusta nähdään myös tilat q_0=01000000 (rivi 1), q_1=00100000 (rivi 2), q_2=10000000 (rivi 4), q_3=00010000 (rivi 7), q_4=00000010 (rivi 8), q_5=01000001 (rivi 9), q_6=00001000 (rivi 10), q_7=00000100 (rivi 11) ja q_8=00000000 (rivi 12).

Lopullinen tilakone (kuva 5.6) koostuu seitsemästä eri tilasta ja kymmenestä tilasiirtymästä. Lopputuloksesta voidaan nähdä, että syntetisointialgoritmi toimii hyvin ja on käyttökelpoinen suunnittelun apuväline. Samalla huomataan, kuinka annettu vastaesimerkki (kuva 5.5) muutti tilakonetta luoden uuden välitilan q_5 ja muuttaen tilasiirtymiä siten, ettei herätys (*alarm time reached*) voi tapahtua ennen kuin herätysaika on asetettu (*set alarm time*).



Kuva 5.6. Lopputuloksena herätyskello

Esimerkkien järjestyksellä on myös eroa. Jos aloitamme herätysajasta (kuva 5.3) ja lisäämme lopuksi ajan muutoksen (kuva 5.1) saamme tilakoneen, jossa ajanmuutos on mahdollista vain, kun herätysaikaa ei ole asetettu (toisin sanoen siitä puuttuu siirtymä q_5->q_1). Tällainen tilakone on alkuperäisessä esimerkissä [26].

5.2 Suorituskyky käytännön aineistolla

Syntetisointiohjelman suorituskykyyn vaikuttaa toisistaan selvästi eroavat seikat. Ensimmäisenä on algoritmin monimutkaisuus ja sen suorittamien käskyjen lukumäärä. Toisena on ohjelman liittyminen ulkoisiin ohjelmiin ja sitä kautta tulevat viiveet esimerkiksi liikennöinnissä tietokannan kanssa. Kolmas hidastava seikka on ohjelman sisäisestä rakenteesta johtuvat viiveet.

Syntetisointialgoritmin nopeutta ja monimutkaisuutta käsitellään tarkemmin myöhemmin, kun taas tässä tarkastellaan lähinnä toista ja kolmatta viivettä aiheuttavaa kohtaa. Testiaineiston ajossa käytettiin Celeron-prosessorilla varustettua konetta 128 MB muistilla. Prosessorin kellotaajuus oli 500 MHz. Palvelin sijaitsi LAN-verkon päässä, joten verkkotuloksiin vaikuttivat myös muu liikenne samassa verkkosegmentissä. Palvelimena toimi aivan normaali NT-työasema, joten palvelimen suorituskykyyn vaikuttivat myös muut koneella ajossa olevat ohjelmat.

Ajettaessa testiaineistoa kävi selvästi ilmi, että itse syntetisoinnin nopeus on testeissä käytetyllä koneella aivan riittävä. Suurin hidastava tekijä oli käyttäjä, koska ohjelma yksinkertaisesti odotti suurimman osan ajasta käyttäjän vastausta ohjelman esittämään kysymykseen. Mitään observaatiotaulun päivityksestä johtuvia viiveitä ei esiintynyt. Toisaalta testiaineiston suurin observaatiotaulun koko oli vain 49 riviä ja 11 saraketta, joten hyvä suorituskyky ei tullut yllätyksenä.

Testauksessa käytetty ohjelma on esitetty liitteessä 3.

5.2.1 Nopeuden pullonkaulat

Suurimmaksi viiveeksi, jos käyttäjän vaikutus jätetään huomiotta, nousi testeissä ehdottomasti palvelinyhteys: välillä pelkkä yhteyden avaaminen kesti jopa minuutin, vaikkakin yleensä aikaa kului alle 15 sekuntia. Lisäksi tilakoneen vienti tietokantaa vei aikaa noin sekunnin jokaiselta viedyltä tilalta. Tämä viive vaikuttaa haitallisesti jo ohjelman käytettävyyteen suuria tilakoneita luotaessa. Tietokannasta palvelimelle päin tapahtuva liikenne oli kaksoispalvelinrakenteen (katso tarkemmin alakohdasta 3.3.1) ja jonkinlaisen välimuistin ansiosta kiitettävän nopeaa.

Taulukko 5.4 näyttää selvästi välimuistin toiminnan - siinä verrataan saman testin suoritusaikoja ensimmäisellä ja viidennellä ajokerralla, jolloin huomataan, että suoritusajat tippuvat dramaattisesti muissa kohdissa paitsi tilakoneen viennissä.

Testinä käytettiin yksinkertaista ohjelmaa, joka haki muistiin valitusta työkirjasta lyhyen sekvenssikaavion. Tämän jälkeen vaihdettiin työkirjaa ja luotiin uusi observaatiotaulu hakemalla työkirjasta äärellinen automaatti. Sekvenssikaaviosta saatu merkkijono annettiin

uudelle taululle esimerkkinä. Lopuksi vietiin saatu tulos tietokantaan. Testissä observaatiotaulun luonti äärellisestä automaatista kesti vain 0,01 sekuntia, haetun esimerkin lisääminen tauluun ja iterointi 0,02 sekuntia sekä tilakoneen muodostaminen observaatiotaulusta vähemmän[4] kuin 0,001 sekuntia. Syntetisoinnin ajat ovat siis mitättömän pieniä verrattuna yhteyden aiheuttamiin viiveisiin.

Taulukko 5.4. Välimuistin vaikutus suoritusnopeuteen

	Suoritusajat 1.kerralla	**Suoritusajat 5. kerralla**
Yhteyden luonti	12,2 s	0,4 s
Työkirjaan siirtyminen	1,9 s	0,2 s
Sekvenssikaavion haku	9 s	0,9 s
Äärellisen automaatin haku	0,6 s	0,1 s
Tilakoneen vienti	4,3 s	5,5 s

Nopeustesteissä tuli selvästi esille, että nopeudet vaihtelivat huomattavasti testiajosta toiseen. Tähän on syynä palvelimena toimivan työaseman käyttö muidenkin ohjelmien suorittamiseen. Välillä esimerkiksi työkirjan etsimiseen meni aikaa reilusti yli 20 sekuntia, kun edellisellä kerralla saman työ tekeminen kesti alle sekunnin. Myös tilakoneen viennissä vaihteluväli oli noin sekunnista yli kymmeneen sekuntiin. Toisaalta useamman prosessin ajaminen samanaikaisesti ei hidastuttanut olennaisesti järjestelmää. Suorituskykyvertailun lähdeaineisto on liitteessä 2.

---

[4] Tulosmittauksessa käytettiin Windowsin multimedia-ajastinta, jonka teoreettinen tarkkuus oli asetettu arvoon 1 ms. Koska käytössä oli kuitenkin moniajojärjestelmä, antoi nopeusmittaus joko ajan 0 tai 10 ms. Vaihtelu johtuu tehtävänvaihdon sijoittumisesta keskelle tilakoneen luomista.

5.3 Syntetisoinnin teoreettinen nopeus

Tässä käsitellään syntetisointialgoritmin teoreettista nopeutta ja unohdetaan jo edellisessä luvussa käsitellyt verkko- ja toteutustapaviiveet. Kuten kohdassa 5.2 todettiin, on syntetisoinnin aiheuttama viive kertaluokkia pienempi verrattuna esimerkiksi palvelinyhteyden viiveisiin, joten syntetisoinnin nopeuden tai hitauden tarkastelu on lähinnä vain teoreettista. Käytännön merkitystä ei syntetisoinnin nopeudella testikoneella ja -aineistolla ollut vaan nopeus oli "riittävä".

Syntetisoija koostuu opettajasta (käyttäjä) ja oppilaasta (syntetisointiohjelma) (katso tarkemmin luku 2.3). Koska tässä tapauksessa opettaja on ihminen, voidaan olettaa, että vastaesimerkit ovat "hyviä" eivätkä satunnaisia. Toisaalta ei voida kohtuudella olettaa, että annetut esimerkit olisivat optimaalisia esimerkiksi pituuden suhteen. Tämä inhimillinen tekijä tekee mahdottomaksi tarkan analyysin nopeuden suhteen vaan käytännössä oikea arvo on huonoimman (satunnainen opettaja) ja parhaimman (optimaaliset vastaesimerkit) raja-arvon välimaastossa.

Syntetisointialgoritmia tutkimalla huomataan, että jokaisella iterointikierroksella jokaista riviä kohti tarvitaan yksi testi, kuuluuko annettu merkkijono kieleen (MEMBER-kysely). Tässä tapauksessa ei tehdä eroa käyttäjältä kysytyn ja päätöspuusta haetun vastauksen välillä. Lisäksi jokaisella kierroksella täytyy testata, onko observaatiotaulu T suljettu ja konsistentti.

On helppo nähdä, että syntetisointialgoritmin nopeus riippuu suoraan observaatiotaulun koosta. Observaatiotaulun koko taas riippuu rivien ja sarakkeiden (eli iterointikierrosten, jossa T ei ollut konsistentti) määrästä. Riveistä taas voidaan huomata, että niiden määrä riippuu annettujen vastaesimerkkien pituudesta[5] ja iterointikierroksista, joissa T ei ollut suljettu.

---

[5] Pituudella tarkoitetaan tässä tapauksessa viestiparien määrää eikä esimerkiksi merkkijonon merkkien määrää.

Alussa S ja R koostuvat molemmat yhdestä elementistä (λ). Jokainen kerta, kun T ei ole suljettu, lisätään yksi elementti joukkoon S. Jokaista vastaesimerkkiä (ja myös aloitusmerkkijonoa) t, jonka pituus on enintään m, kohden lisätään enintään m riviä joukkoon S. Tämä voidaan todeta algoritmin (katso algoritmi 2) rivistä *lisää t ja kaikki sen prefiksit joukkoon S*.

Vastaavasti, jos T ei ole konsistentti, lisätään uusi sarake joukkoon R (*lisää* $a \cdot r$ *joukkoon* R, algoritmi 2). Siten merkkijonojen määrä joukossa R ei voi ylittää iterointikierrosten määrää n, koska testi suoritetaan korkeintaan $n-1$ kertaa. Lisäksi merkkijonojen maksimipituus joukossa R on aluksi nolla ja kasvaa korkeintaan yhdellä jokaista lisättyä merkkijonoa kohden, joten suurin pituus R:n merkkijonoille on $n-1$.

Merkkijonojen suurin määrä joukossa S ei voi ylittää arvoa $n+m(n-1)$. Tämä arvo saadaan, koska observaatiotaulu testataan ei-suljetuksi korkeintaan $n-1$ kertaa ja iteroinnin aikana voidaan antaa korkeintaan $n-1$ kappaletta vastaesimerkkejä, joista jokainen lisää korkeintaan m riviä joukkoon S. Lisäksi merkkijonojen maksimipituus joukossa S voi olla korkeintaan $m+n-1$, koska maksimipituus kasvaa korkeintaan yhdellä jokaista suljettu-testissä lisättyä merkkijonoa kohden.

Yhdistämällä maksimipituudet saamme tuloksen myös joukon $(S \cup S \cdot A) \cdot R$ merkkijonojen maksimipituudelle, joka on $m+2n-1=O(m+n)$. Lisäksi suurin mahdollinen alkioiden lukumäärä joukolle $(S \cup S \cdot A) \cdot R$ on $O(mn^2)$, joten observaatiotaulun koko on $O(m^2n^2+mn^3)$. Näistä saamme seuraavan teoreeman, jonka myös Angluin esittää [1]:

Teoreema 5.1 Olkoon U tuntematon, säännöllinen kieli, joka esitetään algoritmilla. Olkoon lisäksi n sellaisen pienimmän äärellisen automaatin, joka hyväksyy kielen U, tilojen lukumäärä. Jos m on yläraja minkä tahansa opettajan antaman vastaesimerkin pituudelle, on ajoaika rajoitettu m:n ja n:n polynomeihin. Lisäksi tarvittavan observaatiotaulun koko on $O(m^2n^2+mn^3)$.

On huomattava, että yllä oleva teoreema antaa arvon "tyhmälle" satunnaisia esimerkkejä antavalle opettajalle[6]. Ihminen osaa kuitenkin antaa parempia ja tarkoituksenmukaisempia esimerkkejä, jolloin observaatiotaulun koko on huomattavasti pienempi ja siten myös suoritusnopeus parempi. Lisäksi koneen algoritmin suoritusnopeus on aivan eri luokkaa kuin ihmisen vastauskyky koneen esittämiin kysymyksiin, joten tärkein syntetisointinopeutta kasvattava seikka on vähentää kysymysten määrää.

---

[6] Satunnaistakin opettajaa voidaan parantaa seuraamalla Rivestin ja Schapiren [26] ohjeita. Kuitenkin näitä ohjeita ei ole käytetty MAS-algoritmissa, koska parempaan tulokseen on päästy alkuperäistä MAT-algoritmia parantamalla.

6. Syntetisointialgoritmien vertailu

Tässä luvussa käsitellään erilaisia MSC-syntetisointialgoritmeja ja verrataan niitä MAS-algoritmiin. Tässä esitetyt algoritmit ovat poikkeuksetta automaattisia eli niillä ei ole syntetisoinnin aikana vuorovaikutusta käyttäjän (tai "opettajan") kanssa.

6.1 SCED ja SMS

SCED (SCenario EDitor) pitää sisällään syntetisointialgoritmin (SMS, *state machine synthesis*), joka on esitetty lähteessä [12]. Se käyttää hyväkseen Biermannin metodia, jossa käyttäjä voi syntetisoida "ohjelman" antamalla ohjelman suoritusta kuvaavia esimerkkejä sarjana käskyjä *i* ja tiloja *m* (esimerkiksi muuttujat ja muistitilat). Käskyt voidaan ajatella joukkona *I* ja käskyjoukot voivat esiintyä useaan kertaan, jolloin ne erotellaan numerolla ($1I_1$,$2I_1$…). Erityisiä käskyjä ovat I_0 (aloituskäsky) ja I_H lopetuskäsky.

Metodin kuvaava ohjelma koostuu äärellisestä joukosta kolmikkoja (nimeltään siirtymiä, *transitions*), jotka ovat muotoa (q_i,c_k,q_j), missä q_i ja q_j ovat ohjelmakäskyjä ja c_k tila. Ohjelmat ovat deterministisiä. Täydellinen ohjelma (*complete program*) sisältää lisäksi alkukäskyn, ja jokaisen käskyn jälkeen on yksilöllinen jatkomahdollisuus.

Biermannin ohjelmaa kuvataan verkkona siten, että käskyt ovat verkon solmuja ja viivat muodostuvat siirtymistä. Verkko voidaan muodostaa helposti lineaarisena siten, että jokainen käsky on oma solmunsa (eikä haarautumisia tai silmukoita tunneta). Toisaalta järkevämpää on optimoida graafia siten, että käytetään samoja tiloja ja otetaan käyttöön silmukat ja haarautumiset, jolloin tuotettu ohjelma alkaa enemmänkin muistuttamaan tilakonetta. SMS-algoritmin päätarkoituksena onkin löytää pienin (eli vähiten tiloja sisältävä) tilakone, joka hyväksyy annetut skenaariot.

SMS käsittelee sekvenssikaaviota samalla tavalla kuin MAS eli pareina (e_i,e_j) (katso tarkemmin luvusta 2.3 sekvenssikaavion ja tilakaavion yhteys). Myös SMS sallii suorituksen jäljityksen (*backtracking*) erityisen pinon avulla. Itse algoritmi on monimutkaisempi kuin MAS-algoritmi, mutta sen tilantarve on pienempi.

Koska algoritmi perustuu pienimmän tilakoneen löytämiselle, sen tuottama lopputulos on välillä turhankin yleistetty (katso luku 2.2). SMS ei ole vuorovaikutteinen, joten sillä ei ole mahdollisuutta kysyä yleistysten kohdalla niiden oikeellisuudesta kuten MAS tekee. Toisaalta automaattisuus helpottaa SCED:issä käyttäjän siirtymistä kaaviotyypistä toiseen, koska kaikki muutokset tapahtuvat ilman, että käyttäjän tarvitsee puuttua syntetisointiin missään vaiheessa (katso myös luku 7.1). Lisäksi vuorovaikutteisuuden puute voidaan osin kiertää sillä, että SMS on inkrementaalinen eli valmista tilakonetta voidaan muokata antamalla uusia skenaariota.

Koska syntetisointialgoritmi on osa editoria, on SCED:in käyttäminen huomattavasti mukavampaa ja nopeampaa kuin TED+MAS –yhdistelmän. Mitään verkkoviiveitä ei ole havaittavissa ja syntetisointi voidaan käynnistää yksinkertaisesti valitsemalla hiirellä halutut skenaariot ja antamalla valikosta syntetisointikäsky (tätä puutetta MAS:issa käsitellään tarkemmin luvussa 7, erityisesti kohta 7.1.1).

Esimerkkiaineiston (katso luku 5) syntetisointi onnistui toiseen vaiheeseen asti helposti ja tulos oli samanlainen kuin MAS-algoritmilla syntetisoitaessa. Kuitenkin kolmannen vaiheen toteutus ei onnistunut, koska SCED ei huoli negatiivisia vastaesimerkkejä (nämähän yleensä esitetäänkin muokkaamalla tilakaaviota). Syntetisointialgoritmilla tuotettua tilakonetta pääsi kuitenkin helposti muokkaamaan. Kuitenkaan SCED ei huomauttanut mitenkään yleistyksestä vaan tilakaavion korjaus ja virheen huomaaminen jää kokonaan käyttäjän vastuulle.

Eräs SCED:in hienoista ominaisuuksista on, että se osaa näyttää eri skenaarioiden vaikutukset tilakoneeseen (eli valitun skenaarion aiheuttamat muutokset näytetään tilakoneessa). Tämä olisi mahdollista toteuttaa myös MAS-algoritmissa pienin muutoksin. Käyttöliittymän tekeminen tälle toiminnolle on taas hieman hankalampaa. Vastaavasti toinen ominaisuus, joka MAS:ista tällä hetkellä puuttuu, on automaattinen tilakoneen ulkoasun ja tilojen optimointi. Tämä voidaan kuitenkin tehdä erillisenä lisäohjelmana TED:iin, jolloin se toimii minkä tahansa syntetisointialgoritmin kanssa, tai liittää suoraan MAS-syntetisoijaan.

SCED käyttää OMT-notaatiota MAS:in ja TED:in UML-notaation sijasta. Molemmat notaatiot ovat kuitenkin lähes samanlaisia, joten työkalusta toiseen siirtyminen ei ole vaikeaa. Itse asiassa SMS-algoritmi on toteutettu erillisenä ohjelmana TED-ympäristöön.

SCED:iä ja sen käyttömahdollisuuksia erilaisten algoritmien kanssa on tutkittu tarkemmin lähteessä [33].

6.2 Ajoitetut skenaariot

Yleensä mallinnettaessa teknisiä laitteita tarvitaan erilaisia aikarajoja. Esimerkiksi pankkiautomaatti ei voi jäädä ikuisesti odottamaan käyttäjän syöttämää tunnuslukua, vaan pankkikortti otetaan turvaan automaattiin. UML tukee sekvenssikaavionotaatiossa erilaisia aikarajoja, mutta tällä hetkellä MAS ja TED eivät hyödynnä niitä. Tämä voidaan kuitenkin kiertää muotoilemalla sekvenssikaavioiden viestit ja käyttämällä ehtoja [24].

Somé [31,32] esittää algoritmin, jolla luodaan skenaarioista ajoitettuja automaatteja. Skenaario koostuu tässä tapauksessa alkutilasta (*pre-condition*), heräte-reaktio –pareista (*stimuli/reactions*) ja ajoituksista (*triggering delays* ja *completion delays*). Skenaariot voidaan esittää formaalisti nelikkönä, jossa on mukana skenaarion numero, alkutila, heräte-reaktio –parit ja valmistumisaika. Vastaavasti ajoitettu automaatti määritellään viisikkona, jossa on äärellinen aakkosto, alku- ja lopputilojen joukko, kellomuuttujat sekä tilasiirtymät.

Algoritmi muodostuu määritelmistä ja säännöistä, joilla skenaario kuvataan automaatiksi. Se toimii inkrementaalisti (kuten MAS) eli siten, että jo syntetisoitua automaattia voidaan täydentää uusilla skenaarioilla, jolloin algoritmi automaattisesti luo mahdollisesti tarvittavat uudet tilat ja muokkaa tilasiirtymiä.

Perusalgoritmin ongelmana on se, että annettaessa syötteenä samantapaisia skenaarioita, voi syntetisoitavan automaatin tilojen määrä kasvaa eksponentiaalisesti [31]. Tämän vuoksi Somé esittää myös parannetun algoritmin, joka käyttää ryhmiteltyjä tiloja (eli ali- ja ylitiloja). Tällöin samanlaiset tilat kootaan yhdeksi ylitilaksi ja siten estetään tilojen määrän räjähdysmäinen kasvu.

Esitetty algoritmi eroaa MAS-algoritmista lähtötietojensa (skenaariot vs. sekvenssikaaviot) ja tulosteiden osalta (ajoitetut automaatit vs. tilakoneet). Lisäksi Somén algoritmi perustuu nimenomaan ajoituksiin, kun MAS ei tue niitä laisinkaan. Lisäksi MAS tekee (ainakin tällä hetkellä) yksitasoisia tilakoneita – tiloja ei yhdistetä ali- ja ylitiloiksi, kuten Somén parannettu algoritmi tekee.

6.3 Lisätietoa syntetisointiin OCL:llä

Whittle ja Schumann [37] esittävät syntetisointialgoritmin, joka muuttaa skenaarioita (sekvenssikaavioita) tilakoneiksi. Tässä algoritmissa on vääränlainen tilakoneen yleistys pyritty välttämään antamalla algoritmille sekvenssikaavioiden lisäksi ylimääräistä tietoja mallinnettavasta järjestelmästä.

Algoritmin tarvitsemat lisätiedot annetaan OCL:llä [27] määrittelemällä sekvenssikaavion viesteille järjestelmän tilamuuttujien (*state variables*) alku- ja lopputilat (*pre-* ja *post-conditions*). Tilamuuttujien arvoista algoritmi pystyy päättelemään, miten sekvenssikaaviot voidaan yhdistää oikeaksi tilakoneeksi. Algoritmin käyttö ei välttämättä tarvitse ollenkaan lisätietoja, mutta tällöin skenaarioiden yhdistys ja siten tilakoneen muodostus ei välttämättä toimi halutulla tavalla.

Lisätietojen antaminen etukäteen algoritmille on hankalampaa kuin MAS:in käyttämät jäsenkyselyt, koska käyttäjän täytyy etukäteen tietää, mitä tilamuuttujia oikeasti tarvitaan, miten ne käyttäytyvät ja milloin algoritmilla on tarpeeksi yksikäsitteisesti määritelty OCL-tiedosto. MAS vastaavasti kysyy vain tarvittaessa käyttäjän apua helposti ymmärrettävällä sekvenssikaaviolla, eikä tarkempaa tietoa toteutuksesta tai järjestelmän eri tiloista tarvita.

6.4 Reaaliaikainen mallinnus (ROOM)

Real-time Object-Oriented Modeling eli ROOM [29] on nimensä mukaisesti tarkoitettu reaaliaikaisten tapahtumien esittämiseen. Se perustuu ITU-T suositukseen Z.120 [9], joka määrittelee kaksi MSC-notaatiota: *basic MSCs* (*bMSCs*) ja *High-Level MSCs* (*HMSCs*). Näistä ensimmäinen sisältää joukon rinnakkaisia, keskenään viestittäviä prosesseja. HMSC puolestaan tarjoaa operaatiot, joilla bMSC:t voidaan yhdistää siten, että samalla

määritellään bMSC:ien suoritusjärjestys ja -tapa. Lisäksi käytetään laajennettuja äärellisiä tilakoneita (*ROOMcharts*) määrittelemään dynaamista käyttäytymistä.

ROOM:issa prosessien välinen kommunikointi on esitetty ohjausvuokaaviolla siten, että prosessi siirtyy (paikallisesta) tilasta toiseen silloin, kun se lähettää tai vastaanottaa viestin. Siten prosessien tilat voidaan kuvata yksinkertaisesti ROOMchart:in tiloiksi. Sen sijaan tilasiirtymien kohdalla kuvaus ei olekaan helppoa, sillä tilan vaihto onnistuu ROOMchart:issa vain ajastimella tai kun viesti on vastaanotettu.

Keskeisin kysymys tilasiirtymien kuvauksessa on, kuinka monta tapahtumaa suoritetaan yhden tilasiirtymän aikana. Mahdollisuuksia on kolme: tilasiirtymä suorittaa tarkalleen yhden tapahtuman, tilan vaihto päättyy ennen seuraavaa viestin vastaanottoa (tai viimeiseen tilaan) tai tilasiirtymä suorittaa kaikki viestinlähetykset ennen kuin vastaanottaa seuraavan viestin. Leue [13] esittää automaattisia algoritmeja, joilla käsitellään kahta viimeistä vaihtoehtoa.

Verrattuna MAS:iin, Leuen esittämät algoritmit eroavat jo suoritusympäristönsä osalta. Algoritmit tukeutuvat HMSC:iin, kun taas MAS käyttää normaaleja, toisistaan riippumattomia UML-sekvenssikaavioita. ROOMcharts ovat vastaavasti erilaisia kuin UML:n tilakaaviot.

7. Jatkokehitys

Diplomityön tarkoituksena oli luoda perustoteutus MAS-algoritmille siten, että itse syntetisoinnin jatkokehittäminen olisi mahdollista. Tämän vuoksi tehty ohjelma ei pyrikään olemaan täydellinen MAS-ohjelmisto – sitähän se ei edes pysty olemaan muiden syntetisointiin liittyvien ohjelmien vajavaisuuden vuoksi. Tässä luvussa esitetään ehdotuksia siitä, miten ohjelmiston ja algoritmin kehitystä olisi syytä tulevaisuudessa jatkaa.

7.1 Käyttöliittymän jatkokehitys

Ohjelmistojen käyttöliittymän suunnittelu jo sinänsä on hyvin laaja osa ohjelmistoprojektia. Tässä työssä ohjelmalle tehty käyttöliittymä on syntynyt lähinnä käytännön pakosta antaa eri tietoja algoritmille. Alussahan koko syntetisoijaa käytettiin merkkiliittymällä antamalla ohjelmalle syötteet suoraan ohjelman käyttämässä sisäisessä muodossa. Koska normaalikäyttäjän on kuitenkin vaikea ja virhealtista ajatella sekvenssikaaviota viestipareina (e_i,e_j), annettiin käyttäjälle mahdollisuus käyttää syntetisoijaa käyttäjälle tuttujen sekvenssikaavioiden avulla graafisen käyttöliittymän kautta. Kuitenkin kehityksen pääpaino on ollut nimenomaan syntetisoijan kehityksessä, ei käyttöliittymässä.

7.1.1 Käyttöliittymän integrointi

Käyttöliittymän suurimpana ongelmana on sen kaksijakoisuus. Käyttäjä työskentelee normaalisti *TED*-ympäristössä ja halutessaan generoida tilakone luomastaan sekvenssikaaviosta täytyy hänen käynnistää täysin erillinen syntetisointiohjelma, johon sitten syötetään erikseen sekvenssikaavion sijainti ja halutun tuloksen sijainti. Tämän jälkeen palataan jälleen *TED*-ohjelmaan ja piirretään vastaesimerkki, jonka sijainti täytyy kertoa syntetisoijalle. Tätä eri ohjelmasta toiseen siirtymistä jatketaan kunnes haluttu tilakone on valmis. Toisaalta MAS:in ollessa erillisenä ohjelmana, sitä voidaan käyttää (ainakin soveltaen) hyvinkin erilaisten mallinnusohjelmien kanssa.

Kaksijakoisuuden poistamiseen on vain yksi keino: ohjelmien integrointi. Se, integroidaanko syntetisointialgoritmi piirto-ohjelmaan vai piirto-ohjelma syntetisointialgorit-

miin, riippuu käyttäjän tarpeesta. Jos piirretään UML-kaavioita ja halutaan välillä tarkistaa, tuottaako annettu sekvenssikaavio oikean lopputuloksen, täytyy syntetisointi käynnistyä ja toimia suoraan kaavion piirto-ohjelmassa (katso tähän liittyen tarkemmin alakohta 7.3.1).

Ihanteellisessa tapauksessa käyttäjä toimisi kokonaan yhdessä ohjelmassa siten, että aluksi hän valitsisi halutun objektin sekvenssikaaviosta ja antaisi komennon *Synthesize statechart*, jonka jälkeen tilakone muodostettaisiin automaattisesti esimerkiksi sekvenssikaavion viereen. Tämän jälkeen käyttäjällä olisi mahdollisuus editoida tilakonetta haluamakseen esimerkiksi poistamalla tilasiirtymiä tai lisäämällä tiloja. Nämä muutokset puolestaan vaikuttaisivat suoraan sekvenssikaavion esimerkiksi vaihtamalla kiellettyjen viestien värit punaisiksi. Näin käyttäjä voisi valita aina parhaimman esitystavan ongelmalleen.

Täydellinen integrointi vaatisi kuitenkin hieman toisenlaisen piirtotyökalun kuin käytössä oleva *TED*. Ohjelman pitäisi pystyä paremmin erottamaan eri näkymät toisistaan siten, että yllä esitetty sekvenssikaavion ja tilakoneen rinnakkainen esittäminen onnistuisi helposti. Eräs mahdollisuus olisi käyttää jaettua ikkunaa siten, että toisella puolella muokataan sekvenssikaaviota ja toisella puolella näytetään vastaava lopputulos tilakaaviona. Tilakoneen voisi myös esittää erillisessä kelluvassa ikkunassa. Ikävä kyllä käytetty syntetisointialgoritmi ei tue reaaliaikaista syntetisointia vuorovaikutteisen luonteensa (käyttäjäkysymykset) vuoksi, vaan jokainen tilakone täytyy luoda mahdollisten kysymysten kautta uudestaan. Toisaalta tarve jatkuvasti päivitettävään tilakoneeseen lienee hyvin pieni.

7.1.2 Käyttäjäkysymykset

Eräs ongelmallinen seikka on se, miten algoritmin kysymykset esitetään käyttäjälle. Kysymyksiä on kahdenlaisia: ”Pitääkö annettu sekvenssikaavio hyväksyä?” (eli ”Kuuluuko merkkijono s annettuun kieleen?”) ja ”Mikä on vastaesimerkki luodulle tilakoneelle?”. Algoritmi sinänsä haluaa vastauksen ensimmäiseen kysymykseen numerona *0* tai *1* sekä jälkimmäiseen merkkijonona, joka koostuu sekvenssikaavion viestipareista (e_i,e_j).

Käyttöliittymän suunnittelu vastaesimerkin antamiselle on helppoa, koska ilmeinen esitystapa on sekvenssikaavio, joka muunnetaan algoritmin haluamaan muotoon. Ainoa on-

gelma on se, että tällä hetkellä itse sekvenssikaavioiden piirto tapahtuu eri ohjelmassa eikä ohjelmien välillä ole mitään automaattista linkkiyhteyttä, vaan käyttäjän on syötettävä käsin piirtämänsä sekvenssikaavion sijainti.

Sekvenssikaavion hyväksyminen (eli merkkijonon kuuluminen kieleen) on vähän ongelmallisempi kysymys käyttöliittymän kannalta. Luonnollisin vaihtoehto olisi ”piilottaa” algoritmin kysymykset käyttäjältä ja esittää kuuluminen eri skenaarioina. Toisin sanoen annetaan käyttäjälle eri vastauksia vastaavia tilakoneita, joista hän sitten valitsee oikean tai sellaisen puuttuessa lähimpänä oikeata olevan. Tämän toteuttaminen käytännössä on kuitenkin hyvin hankalaa ellei jopa mahdotonta, sillä kysymysten ja siten myös skenaarioiden määrä voi kasvaa huomattavan suureksi[7], eikä valmista skenaario-tilakonetta voida luoda ennen kuin kaikkiin kysymyksiin kyseisessä skenaariossa on vastattu.

Tällä hetkellä kieleen kuuluminen on esitetty sekvenssikaaviona, joka on lienee myös toteuttamiskelpoisin vaihtoehto. Tässä käyttäjän kannalta mahdollisena ongelmana on se, että viestin vastaanottajaa ei voida yksilöidä, vaan viestit lähetetään aina yleisesti *Järjestelmälle*. Tämä vaikeuttanee hieman viestityksen oikeellisuuden tarkastelua. Lisäksi eri sekvenssiryhmät kannattaa värittää erilaisella taustavärillä, jolloin silmukoiden ja tapahtumien rajojen hahmottaminen helpottuu.

Algoritmi antaa mahdollisuuden myös luoda observaatiotaulu äärellisestä automaatista. Tämä on hyvä ominaisuus, mutta siihenkin sisältyy käyttäjän kannalta ongelmia. Käyttäjä on kiinnostunut tilakoneista, joita algoritmin tarkoituksena onkin tuottaa. Kuitenkin algoritmin kannalta kaikki editointi täytyy tehdä äärelliseen automaattiin, joka tosin myöskin saadaan generoitua algoritmilla. Eli ulospäin tuotetaan erilainen kaavio kuin algoritmin jatkokäytön kannalta olisi tarpeellista. Tilakone on muutettavissa algoritmia varten hel-

---

[7] Koska jokaiseen algoritmin kysymykseen voidaan vastata kyllä tai ei, vaihtoehtojen määrä kaksinkertaistuu jokaisen kysymyksen kohdalla, eli $m=2^n$. Toisaalta tarvittavien kysymysten määrä vaihtelee eri skenaarioissa, joten kysymysten tarkkaa kokonaismäärää ei voida etukäteen arvioida.

posti äärelliseksi automaatiksi *B*, mutta muunnoksen jälkeenkin puuttuvat algoritmin tarvitsemat alku- ja lopputilat.

7.1.3 Muita parannusehdotuksia

Algoritmi tarjoaa mahdollisuuden käyttäjälle muuttaa mielipidettään merkkijonon kuulumisesta pääteltävään kieleen. Tätä ei ole kuitenkaan huomioitu nykyisessä käyttöliittymässä mitenkään, vaan käyttäjä voi ainoastaan keskeyttää algoritmin suoritus ja aloittaa alusta. Testikäytössä tästä ei kuitenkaan ole suurempaa haittaa, sillä algoritmin lähtötiedot eivät yleensä muutu ja vastausten korjailulle jälkikäteen on vähemmän tarvetta.

Käyttäjän kannalta voisi toisaalta olla hyödyllistä tarjota ”mitä jos” –skenaarioita siten, että käyttäjä pystyy esimerkiksi sallimaan yhden aikaisemmin kiellettynä olleen merkkijonon ja katsomaan, miten generoitu tilakone muuttui aikaisemmasta.

7.2 Syntetisoinnin jatkokehitys

MASin vahvuus on siinä, että se pystyy kysymään käyttäjältä lisäinformaatiota silloin, kun se ei itse pysty päättelemään, pitäisikö jonkin merkkijono hyväksyä vain ei. Tässä esitetyt parannusehdotukset koskevat lähinnä juuri tätä ominaisuutta.

7.2.1 Kielletyt merkkijonot

Käyttäjän kannalta olisi hyödyllistä, jos ohjelmalle voisi antaa suoraan merkkijonoja, jotka eivät ainakaan ole pääteltävässä kielessä. Tämä vähentäisi merkittävästi turhien kysymysten määrää. Tätä tilannetta onkin käsitelty lähteessä [24]. MAS varustetaan kiellettyjen osamerkkijonojen puulla *F*, jota käytetään samalla tavoin kuin puuta *W*. Käyttäjä voi milloin tahansa lisätä merkkijonoja kiellettyjen merkkijonojen joukkoon. Vastaavasti kiellettyjen osamerkkijonojen poistaminen on mahdollista. Lisäämisen ja poistamisen jälkeen päivitetään automaattisesti observaatiotaulua siten, ettei se ole ristiriidassa *W*:n kanssa.

Kiellettyjen merkkijonojen lisäämisen puuhun *F* voidaan hoitaa erillisellä käyttöliittymällä tai läpinäkyvästi siten, että käyttäjän kiellettyä yhden merkkijonon, kysytään kielletäänkö automaattisesti myös kaikki ne merkkijonot, jotka sisältävät jonkin osamerkkijonon kiel-

letystä merkkijonosta. Mahdolliset poikkeukset voitaisiin hoitaa kohdassa 7.1.3 esitetyllä tavalla jälkikäteen.

7.2.2 Oletukset

Vastaavasti kuin osamerkkijonojen kieltäminen, olisi syytä antaa käyttäjän tehdä myös haluamiaan oletuksia, jotka vähentäisivät osaltaan tarvittavien kysymysten määrää. Yksi tällainen oletus on esimerkiksi silmukoiden automaattinen salliminen (sama viesti voi toistua useamman kuin yhden kerran merkkijonossa). Usein nimittäin käy niin, että tutkittava sekvenssikaavio sisältää jonkinasteisen silmukan (esimerkiksi kellon- tai herätysajan asetus useamman kerran peräkkäin, hissillä ajo ylöspäin ilman että tarvitsee välillä tulla alaspäin). Tällä hetkellä algoritmi kysyy erikseen ensin hyväksymien yhdelle kerralle ja tämän jälkeen silmukalle.

Myös muita oletuksia voitaisiin antaa, kuten esimerkiksi ”tapahtuman x täytyy tapahtua vähintään yhden kerran hyväksytyssä merkkijonossa”. Kuitenkin yleispätevien sääntöjen ja oletuksien keksiminen siten, että niistä olisi todellista hyötyä muissakin kuin erikoistapauksissa, voi olla hankalaa.

7.2.3 Epävarmat vastaukset

Syntetisointiohjelman käyttäjä ei aina osaa vastata suoraan, pitäisikö ohjelman esittämä sekvenssikaavio hyväksyä kuuluvaksi luotavaan tilakoneeseen. Tätä varten algoritmia on mahdollista kehittää hyväksymään myös epävarmat 'ehkä' –vastaukset .

Epävarmojen 'luultavasti kyllä' ja ''luultavasti ei' –vastauksien lisäksi käyttäjä voi olla täysin tietämätön vastauksesta ('En tiedä'), jolloin observaatiotaulun käsittely täytyy suorittaa eri tavalla kuin 'ehkä' -vastauksien. Myös mielipiteen muuttaminen epävarmojen vastauksien kohdalla täytyy olla mahdollista (vertaa kohta 7.1.3).

Eräs helposti toteutettava parannus on antaa käyttäjän siirtää vastaustaan myöhemmäksi, kun käyttäjällä on parempaa tietoa kuulumisesta. Toteutus onnistuu helposti asettamalla observaatiotauluun vastaavalle kohdalle merkki, jolla ilmaistaan, että arvoa ei vielä tiedetä. Kun kaikki käyttäjäkysymykset on käyty läpi, käydään observaatiotaulu uudelleen läpi

'myöhemmin' -merkittyjen osalta. Osa näistä kohdista ratkeaa jopa ilman käyttäjän apua, jos tarvittava informaation on tullut esille jossain myöhemmin esitetyssä kysymyksessä. Toisaalta 'myöhemmin' ei poista sitä tosiseikkaa, että kaikkiin kysymyksiin pitää tietää oikea vastaus.

7.3 Toteutuksen jatkokehitys

Tässä luvussa käsitellään lähinnä sitä, miten ohjelman yleistä toteutusta olisi mahdollista kehittää. Tässä ei siis puututa ohjelman ulkoiseen asuun eli käyttöliittymään tai ohjelman käyttäytymiseen eli syntetisointialgoritmiin.

7.3.1 Ohjelman ulkoinen käyttö

Tällä hetkellä syntetisoinnin käyttö toimii luomalla uusi Visual C++ -projekti, johon liitetään tarvittavat luokat ja jonka pääohjelma luo tietokantayhteyden, kutsuu syntetisointialgoritmia halutuilla parametreilla ja vie tuloksen takaisin tietokantaan. Esimerkiksi Windows-versio hakee käyttäjän syötteet dialogilla, luo yhteyden palvelimeen, hakee syötteiden perusteella tiedon syntetisointialgoritmiin ja syntetisoinnin jälkeen siirtää tiedon palvelimelle.

Ulkopuolisen ohjauksen mahdollistamiseksi täytyy luoda COM-palvelin, joka sisältää syntetisointialgoritmin ja rajapinnan, jonka avulla algoritmia käytetään. Palvelimen laajuus riippuu sovellusalueesta. TEDin käyttöliittymän kanssa toimiessaan COM-palvelimen tulisi osata hakea TEDin palvelimesta tarvittavat tiedot esimerkiksi id-numeron perusteella ja samoin luoda tietokantaan tulos. Vastaavasti, jos palvelinta käytetään jonkin muun (mallinnus)ohjelman kanssa, täytyy COM-palvelimelle syöttää vain lähtötiedot ja hakea tulos jossain muodossa.

Syntetisoinnin täydellinen automatisointi (vertaa SCED) on hankalaa käyttäjältä kysyttävien kysymysten vuoksi. Kuitenkin osittainen automaatio voidaan tehdä, sillä normaali pääohjelmahan koostuu lähinnä kolmesta kohdasta.

- Yhteyden luonti palvelimeen ja tiedon haku käyttäjän määräämästä paikasta
- Syntetisointiohjelman pääsilmukan (`DoIteration`) kutsu ja paluuarvoon reagointi (esimerkiksi vastaesimerkin hakeminen)
- Valmiin tuloksen siirtäminen takaisin TEDin palvelimelle

Näiden kolmen kohdan toteuttaminen riittää normaaliin syntetisointiohjelman käyttöön. Vastaavasti käyttäjäohjelman pääohjelma muodostuisi näistä riveistä (esimerkissä on jätetty selvyyden vuoksi käsittelemättä virhetilanteet ja funktioiden paluuarvot):

```
CComPtr<ISynthesizer> synt;
CoCreateInstance(CLSID_Synthesizer,0,CLSCTX_INPROC_SERVER,QU_ARGS(ISynthesizer,&synt));
synt->Initialize(host, symbol);
synt->SetCurrentWorkbook(workbookName);
synt->SearchAndSetClassifierRole(participantName);
synt->DoSynthesize();
synt->ImportToTED(modelName,viewName);
```

Pääohjelma voisi olla jopa lyhyempi, jos alustus, sekvenssikaavion luku, syntetisointi ja vienti yhdistetään:

```
CComPtr<ISynthesizer> synt;
CoCreateInstance(CLSID_Synthesizer,0,CLSCTX_INPROC_SERVER,QU_ARGS(ISynthesizer,&synt));
synt->Initialize(host, symbol,workbookName);
synt->DoSynthesizeAndImportResult(modelName,viewName);
```

Eri käyttöliittymät (konsoli ja graafinen) voidaan valita esimerkiksi käyttämällä eri rajapintaa syntetisointipalvelimella:

```
CComPtr<I_GUI_Synthesizer> synt;
CoCreateInstance(CLSID_Synthesizer,0,CLSCTX_INPROC_SERVER,
QU_ARGS(I_GUI_Synthesizer,&synt));
synt->Initialize(host, symbol,workbookName);
synt->DoSynthesizeAndImportResult(modelName,viewName);
```

tai luomalla metodi, jolla asetetaan rajapinta:

```
CComPtr<ISynthesizer> synt;
CoCreateInstance(CLSID_Synthesizer,0,CLSCTX_INPROC_SERVER,QU_ARGS(ISynthesizer,&synt));
synt->Initialize(host, symbol,workbookName);
synt->SetGUI(true);
synt->DoSynthesizeAndImportResult(modelName,viewName);
```

Syntetisointiohjelma voidaan myös tehdä tukemaan automaatiota (katso luku 3.2), jolloin syntetisoinnin käyttö erilaisissa makrokielissä olisi helppoa.

7.3.2 Ohjelman suoritusnopeus

Luvussa 5 tarkasteltiin ohjelmiston suorituskykyä ja kuten tuloksista huomattiin, järjestelmän pullonkaula on verkkoyhteyksissä. Syntetisointi toimi käytetyllä testiaineistolla varsin nopeasti. Nopeustestissä käytetty aineisto oli kuitenkin käytännön syistä varsin suppea (observaatiotaulussa alle 550 alkiota) eikä toteutuksen skaalautuvuutta tutkittu käyttötestissäkään kuin 900 alkioon asti.

Jatkotutkimuksena olisikin järkevää tutkia suurien observaatiotaulujen (alkioita esimerkiksi yli 10000) vaikutusta sekä jäsenkyselyiden määrään että algoritmin ja toteutuksen nopeuteen. Tarvittaessa syntetisointinopeutta voidaan kasvattaa vähentämällä iterointikierrosten lukumäärää esimerkiksi oletusten avulla (katso tarkemmin luku 5.3).

8. Yhteenveto

Ohjelmistosuunnittelussa käytetään syntetisointialgoritmeja lähinnä suunnitteluvaiheessa järjestelmän käyttäytymisen mallintamiseen. Lisäksi algoritmit nopeuttavat suunnittelua tarjoamalla (lähes) automaattisen tavan muuntaa käytettyjä kaavioita toisentyyppiseksi, jolloin eri mallien yhtäläisyys säilyy käsin tehtävää muunnosta varmemmin. Mallinnusohjelmistot tarjoavat joitakin syntetisointialgoritmeja, mutta tässä työssä esitettyä vuorovaikutteista algoritmia vastaavaa syntetisoijaa ei löytynyt.

Tämän työn päätarkoituksena oli luoda perustoteutus MAS-algoritmille. Tuloksena saatiin modulaarinen ohjelma, joka voidaan helposti muokata erilaisiin ympäristöihin. Ohjelma toimii lähinnä testausalustana esimerkiksi sille, kuinka monta käyttäjäkysymystä tarvitaan tiettyjen skenaarioiden syntetisoinnissa tilakoneeksi. Tuotantokäyttöön nykyisestä versiosta tuskin on sen käytön hankaluuden vuoksi.

Nykyinen ohjelmaversio sisältää vain perusalgoritmin, joten jatkossa siinä on useita laajennusmahdollisuuksia. Eräs laajennettava kohta on käyttäjän antamat epävarmat vastaukset. Jatkossa toteutus voisi ottaa huomioon myöskin UML-kaavioiden laajennukset: nykyinen algoritmi osaa vain perustason sekvenssikaaviot, esimerkiksi ehtoja ei ole nykyisessä toteutuksessa otettu lainkaan huomioon.

Käyttöliittymän järkevä toteutus on erillisen TED-ohjelmiston käytöstä johtuen hankalaa. Onkin syytä miettiä, kannattaisiko MAS:in ympärille luoda kokonaan oma editointiohjelmisto ja jättää TED:in käyttö syntetisoinnin apuvälineenä vähemmälle. MAS voitaisiin upottaa esimerkiksi SCED-ohjelmistoon, jolloin käyttäjä voisi valita käytettävän syntetisointialgoritmin tottumuksensa ja kehitystyönsä mukaan. Nykyisellään TED-palvelimesta aiheutuvat viiveet estävät järkevän työskentelyn tehokkaallakin koneella ja syntetisointialgoritmin ajaminen erillisenä ohjelmana hankaloittaa syöttötietojen esittämistä algoritmille (käytettävien ohjelmien välillä ei ole mitään linkkiä, jolloin käytännössä lähtötiedot joudutaan antamaan algoritmille tietokantapolkuna). Lisäksi hitaus estää reaaliaikaisten esitystapojen (esimerkiksi animaatiot) käytön.

Jatkossa kannattaa myös tutkia tarkemmin sitä, miten ohjelmisto esittää käyttäjälle syntetisoinnissa tarvittavat jäsenkysymykset. Tämä on kriittinen kohta koko järjestelmän toimivuuden kannalta. Jos käyttäjä ei helposti ymmärrä koneen esittämää sekvenssikaaviota ja sen merkitystä mallinnettavan järjestelmän osana, on syntetisointi hidasta ja epävarmaa sekä johtaa helposti vääränlaisiin lopputuloksiin. Lisäksi sekvenssikaavio (ainakaan nykyisessä muodossaan) ei välttämättä ole paras esitystapa jäsenkysymysten esittämiseen. Toisaalta algoritmin observaatiotaulu antaa omat rajoituksensa käytettävissä olevan tiedon määrälle.

Tutkimuksen arvoista on myös se, miten käyttäjä *helpoiten* pystyy antamaan järjestelmälle yleistyksiä. Esimerkiksi silmukoiden (eli sama asia tapahtuu sekvenssikaaviossa useamman kerran peräjälkeen) automaattinen salliminen voisi tapahtua jäsenkysymyksen yhteydessä siten, että käyttäjä voi pelkän myöntämisen lisäksi kertoa syntetisointialgoritmille, että kyseinen tapahtuma saa tai ei saa tapahtua useammankin kerran. Tällä hetkellä algoritmi kysyy jäsenkysymyksen ensin yhdelle kerralle ja sen jälkeen (yleensä seuraavana kysymyksenä) erikseen useammalle kerralle. Yleispätevän luvan antaminen (esimerkiksi rasti alkuruudussa) on huono idea, sillä samassa mallissa voi osa tapahtumista tapahtua vain kerran (*herätyskellon herätys ei voi tapahtua herätyksen sulkemisen jälkeen*) ja osa useamman kerran (*kellon aika voidaan asettaa useamman kerran peräkkäin*).

Toinen työn tarkoituksista oli vertailla käytettävissä olevia algoritmeja ja testata MAS:in sopivuutta käytännön työskentelyyn. Vastaavia vuorovaikutteisia syntetisointialgoritmeja ei kuitenkaan löytynyt vaan kaikki MSC-syntetisoijat tuntuvat pohjautuvat enemmänkin automaattisiin algoritmeihin. Automatiikan ongelmana on väärät yleistykset, jolloin tuloksena syntynyt tilakone vaatii vielä käyttäjältä muokkausta ennen kuin se on lopullisesti valmis.

Algoritmina MAS tuntuu toimivan oikein eikä syntetisointi tarvitse yletöntä konetehoa. Tärkein nopeus- ja soveltuvuusmittari on algoritmin vuorovaikutteisin luonteen vuoksi käyttäjältä vaadittavan tiedon määrä. Sinänsä yhtään turhaa kysymystä algoritmi ei kysy, mutta siihen voidaan lisätä "älykkyyttä" erilaisilla oletuksilla, jolloin kysymysten määrä vähenee entisestään. Täysin automaattiseksi järjestelmää ei kuitenkaan saada.

LÄHDELUETTELO

1. Angluin, D. Learning Regular Sets from Queries and Counterexamples. Information and Computation 75 (1987). s. 87-106.

2. Angluin, D., Smith, C. Inductive Inference: Theory and Methods. ACM Computing Surveys 15 (1983). s.237-269.

3. Chung, P., Huang, Y., Yajnik, S. DCOM and CORBA Side by Side, Step by Step, and Layer by Layer. [www-csag.ucsd.edu/individual/achien/cs491-f97/papers/dcom_corba.html]

4. Component Software. CS-RCS. [http://www.ComponentSoftware.com/csrcs/], 2000.

5. Glinz, M. An integrated formal model of scenarios based on statecharts. Lecture Notes in Computer Science 989 (1995). s. 254-271.

6. Gold, E. Language Identification in the Limit. Information and Control 10 (1967). s. 447-474.

7. Haikala, I., Märijärvi, J. Ohjelmistotuotanto. Espoo: Suomen ATK-kustannus Oy, 1995. 343 s.

8. Harel, D. Statecharts: a Visual Formalism for Complex Systems. Science of Computer Programming (1987). s. 231-274.

9. ITU-T. Recommendation Z.120:Message Sequence Chart (MSC). ITU-T, Geneva, 1996.

10. Khriss, I., Elkoutbi, M., Keller, R. Automating the Synthesis of UML Statechart Diagrams from Multiple Collaboration Diagrams. UML'98: Beyond the Notation, LNSC 1618, 1999. s. 132-147.

11. Koskimies, K., Mäkinen, E. Automatic Synthesis of State Machines from Trace Diagrams. Software – Practice & Experience, 24(7) (1994). s. 643-658.

12. Koskimies, K., Systä, T., Tuomi, J., Männistö, T. Automated support for modeling OO software. IEEE Software 15,1 (1998). s. 87-94.

13. Leue, S. Mehrmann, L., Rezai, M. Synthesizing Software Architecture Descriptions from Message Sequence Chart Specifications. Automated Software Engineering – ASE-98. Honolulu, USA, October 1998. s. 192-195.

14. Marton, G. DepUml 1.0 Documentation, Non-Official. Nokia Research Center, 1999. 21 s.

15. Microsoft Corporation. Automation Start Page. [http://msdn.microsoft.com/library/psdk/automat/autoportal_7l45.htm], 2000.

16. Microsoft Corporation. COM (Component Object Model). [http://msdn.microsoft.com/library/psdk/com/comportal_3qn9.htm], 2000.

17. Microsoft Corporation. COM Clients and Servers. [http://msdn.microsoft.com/library/psdk/com/comext_8p2r.htm], 2000.

18. Microsoft Corporation. Distributed Component Object Model Protocol - DCOM/1.0. [http://msdn.microsoft.com/library/specs/distributedcomponentobjectmodelprotocoldcom10.htm], 1998.

19. Microsoft Corporation. Interfaces and Pointers. [http://msdn.microsoft.com/library/psdk/com/com_37w3.htm], 2000.

20. Microsoft Corporation. Managing Object Lifetimes Through Reference Counting. [http://msdn.microsoft.com/library/psdk/com/com_63fr.htm], 2000.

21. Microsoft Corporation. Microsoft Foundation Class Library. [http://msdn.microsoft.com/library/devprods/vs6/visualc/vcmfc/mfchm.htm], 2000.

22. Microsoft Corporation. Rules for Implementing QueryInterface. [http://msdn.microsoft.com/library/psdk/com/com_1j8l.htm], 2000.

23. Microsoft Corporation. Rules for Managing Reference Counts. [http://msdn.microsoft.com/library/psdk/com/com_1vxv.htm], 2000.

24. Mäkinen, E., Systä, T. Implementing minimally adequate synthesizer. Tampere: Dept. of Computer and Information Sciences, University of Tampere, 2000. Report A-2000-9. 25 s.

25. Mäkinen, E., Systä, T. MAS – an interactive synthesizer to support behavioral modeling in UML, manuscript , 2000.

26. Mäkinen, E., Systä, T. Minimally adequate teacher designs software. Tampere: Dept. of Computer and Information Sciences, University of Tampere, 2000. Report A-2000-7. 23 s.

27. Rational Software Corporation. The Unified Modeling Language Notation Guide v.1.3. [http://www.rational.com/uml], 2000.

28. Rivest, R., Schapire, R. Inference of Finite Automata Using Homing Sequences. Information and Computation 103 (1993). s. 299-347.

29. Selic, B. Gullekson, G. Ward, P.T. Real-Time Object-Oriented Modelling. John Wiley & Sons, Inc., 1994. 500 s.

30. Silicon Graphics Computer Systems, Inc. Standard Template Library Programmer's Guide. [http://www.sgi.com/Technology/STL/], 1999

31. Somé, S., Dssouli, R. An Enhancement of Timed Automata generation from Timed Scenarios using Grouped States.Université de Montréal, DIRO Technical Report #1029, April 1996. 18 s.

32. Somé, S., Dssouli, R., Vaucher, J. From Scenarios to Timed Automata: Building Specifications from Users Requirements. Asia Pacific Software Engineering Conference APSEC 95, Brisbane, Australia, 1995.

33. Systä, T. Static and Dynamic Reverse Engineering Techniques for Java Software Systems. Tampere: Dept. of Computer and Information Sciences, University of Tampere, 2000. Report A-2000-4. 233 s.

34. Tampereen teknillinen korkeakoulu. Linkkejä Ohjelmistotuotanto-kirjaan liittyvään materiaaliin. [http://www.cs.tut.fi/~otm/kirja/], 2000.

35. The Object Management Group. The Common Object Request Broker: Architecture and Specification, Revision 2.3.1. [http://www.omg.org/technology/documents/formal/corba_2.htm], October 1999.

36. Urponen, T. 73117 Automaattiteoria - Luentokalvot syksy 1996. Tampereen teknillinen korkeakoulu, 1996.

37. Whittle, J., Schumann, J. Generating Statechart Designs From Scenarios. Proceedings of International Conference on Software Engineering (ICSE 2000), Limerick, Ireland, June 2000. s. 314-323.

38. Wikman, J. Evolution of a Distributed Repostitory-Based Architecture. NOSA'98: Proceedings of the First Nordic Workshop on Software Architecture, 1998, s.107-114.

LIITTEET

LIITE 1 Esimerkki tulostetusta observaatiotaulusta

LIITE 2 Suoritusnopeuden mittaustulokset

LIITE 3 Suorituskykytestissä käytetty testiohjelma

LIITE 4 Käytettyjen luokkien otsikkotiedostot

LIITE 1

Esimerkki testikäyttöliittymän tulostamasta observaatiotaulusta. Esimerkissä on muodostettu äärellisestä automaatista uusi observaatiotaulu [24]. Observaatiotaulu on tässä tapauksessa konsistentti ja suljettu.

```
r1='' r2='(s_ct,VOID)' r3='(s_at,NULL)(s_ct,VOID)' r4='(s_ct,set)(s_at,NULL)(s_ct,VOID)'
r5='(buzz,off)(s_ct,VOID)' r6='(s_ct,reached)(buzz,off)(s_ct,VOID)'
r7='(s_at,NULL)(s_ct,reached)(buzz,off)(s_ct,VOID)'
r8='(s_ct,set)(s_at,NULL)(s_ct,reached)(buzz,off)(s_ct,VOID)'
r9='(s_at,NULL)(s_ct,set)(s_at,NULL)(s_ct,VOID)'
r10='(s_ct,set)(s_at,NULL)(s_ct,set)(s_at,NULL)(s_ct,VOID)'
                                                            0 1 0 1 0 0 0 1 0 1
(s_ct,VOID)                                                 1 0 0 0 0 0 0 0 0 0
(s_ct,set)                                                  0 0 1 0 0 0 1 0 1 0
(s_ct,set)(s_at,NULL)                                       0 1 0 1 0 1 0 1 0 1
(s_ct,set)(s_at,NULL)(s_ct,VOID)                            1 0 0 0 0 0 0 0 0 0
(s_ct,set)(s_at,NULL)(s_ct,reached)                         0 0 0 0 1 0 0 0 0 0
(s_ct,set)(s_at,NULL)(s_ct,reached)(buzz,off)               0 1 0 1 0 0 0 1 0 1
(s_ct,set)(s_at,NULL)(s_ct,reached)(buzz,off)(s_ct,VOID)    1 0 0 0 0 0 0 0 0 0
(s_ct,set)(s_at,NULL)(s_ct,set)                             0 0 1 0 0 0 1 0 1 0
(s_ct,set)(s_at,NULL)(s_ct,set)(s_at,NULL)                  0 1 0 1 0 1 0 1 0 1
(s_ct,set)(s_at,NULL)(s_ct,set)(s_at,NULL)(s_ct,VOID)       1 0 0 0 0 0 0 0 0 0
----------------------------------------------------------------------
(s_at,NULL)                                                 0 0 0 0 0 0 0 0 0 0
(s_ct,reached)                                              0 0 0 0 0 0 0 0 0 0
(buzz,off)                                                  0 0 0 0 0 0 0 0 0 0
(s_ct,set)(s_ct,VOID)                                       0 0 0 0 0 0 0 0 0 0
(s_ct,set)(s_ct,set)                                        0 0 0 0 0 0 0 0 0 0
(s_ct,set)(s_ct,reached)                                    0 0 0 0 0 0 0 0 0 0
(s_ct,set)(buzz,off)                                        0 0 0 0 0 0 0 0 0 0
(s_ct,set)(s_at,NULL)(s_at,NULL)                            0 0 0 0 0 0 0 0 0 0
(s_ct,set)(s_at,NULL)(buzz,off)                             0 0 0 0 0 0 0 0 0 0
(s_ct,set)(s_at,NULL)(s_ct,reached)(s_ct,VOID)              0 0 0 0 0 0 0 0 0 0
(s_ct,set)(s_at,NULL)(s_ct,reached)(s_ct,set)               0 0 0 0 0 0 0 0 0 0
(s_ct,set)(s_at,NULL)(s_ct,reached)(s_at,NULL)              0 0 0 0 0 0 0 0 0 0
(s_ct,set)(s_at,NULL)(s_ct,reached)(s_ct,reached)           0 0 0 0 0 0 0 0 0 0
(s_ct,set)(s_at,NULL)(s_ct,reached)(buzz,off)(s_ct,set)     0 0 1 0 0 0 1 0 1 0
(s_ct,set)(s_at,NULL)(s_ct,reached)(buzz,off)(s_at,NULL)    0 0 0 0 0 0 0 0 0 0
(s_ct,set)(s_at,NULL)(s_ct,reached)(buzz,off)(s_ct,reached) 0 0 0 0 0 0 0 0 0 0
(s_ct,set)(s_at,NULL)(s_ct,reached)(buzz,off)(buzz,off)     0 0 0 0 0 0 0 0 0 0
(s_ct,set)(s_at,NULL)(s_ct,set)(s_ct,VOID)                  0 0 0 0 0 0 0 0 0 0
(s_ct,set)(s_at,NULL)(s_ct,set)(s_ct,set)                   0 0 0 0 0 0 0 0 0 0
(s_ct,set)(s_at,NULL)(s_ct,set)(s_ct,reached)               0 0 0 0 0 0 0 0 0 0
(s_ct,set)(s_at,NULL)(s_ct,set)(buzz,off)                   0 0 0 0 0 0 0 0 0 0
(s_ct,set)(s_at,NULL)(s_ct,set)(s_at,NULL)(s_ct,set)        0 0 1 0 0 0 1 0 1 0
(s_ct,set)(s_at,NULL)(s_ct,set)(s_at,NULL)(s_at,NULL)       0 0 0 0 0 0 0 0 0 0
(s_ct,set)(s_at,NULL)(s_ct,set)(s_at,NULL)(s_ct,reached)    0 0 0 0 1 0 0 0 0 0
(s_ct,set)(s_at,NULL)(s_ct,set)(s_at,NULL)(buzz,off)        0 0 0 0 0 0 0 0 0 0
Strings in Trie:
(s_ct,VOID)
(s_ct,set)(s_at,NULL)(s_ct,VOID)
(s_ct,set)(s_at,NULL)(s_ct,reached)(buzz,off)(s_ct,VOID)
(s_ct,set)(s_at,NULL)(s_ct,set)(s_at,NULL)(s_ct,VOID)
```

LIITE 2

Testatessa suoritusnopeutta ajettiin yksinkertainen simulointiohjelma, jonka eri kohtien suoritusaikoja mitattiin käyttöjärjestelmän tarjoamalla multimedia-ajastimella. Ajastimen tarkkuus on 1 ms, mutta koska käytössä on moniajojärjestelmä, voidaan ajanoton tarkkuudeksi ajatella 10 ms. Testit suoritettiin kahteen kertaan peräkkäin. Käyttäjän syötettä ei tarvittu missään vaiheessa suorituksen aikana, joten testiohjelman aikojen mittaaminen oli helppoa.

Nopeusvertailussa käytetyt mittaustulokset ovat yhden ohjelman osalta alla olevasta taulukossa (taulukko 8.1). Kuten mittaustuloksista nähdään, vaihtelee palvelimena käytetyn työaseman palvelunopeus eri mittausten välillä.

Taulukko 8.1. Yhden ohjelman ajoajat

	0a	0b	$\bar{x}_1$
Initialize	691	1782	**1237**
SetCurrentWorkbook	210	1613	**912**
SearchAndSetClassifierRole	621	2433	**1527**
ExportFromTed	120	331	**226**
CreateMas	10	20	**15**
DoIteration	20	20	**20**
InitMachine	10	0	**5**
ImportToTED	12708	5057	**8883**
Yhteensä	**14390**	**11256**	**12823**

Yhden ohjelman suorituksen lisäksi testit tehtiin siten, että tällä kertaa ajettiin samanaikaisesti viisi samanlaista ohjelmaa. Tulokset ovat näkyvissä alla (taulukko 8.2). Sarakkeissa numerot tarkoittavat vastaavan ohjelman järjestysnumeroa ja kirjain testikierrosta. Eli *3b* tarkoittaa toista testiä ja ohjelmaa, joka käynnistettiin kolmantena. Viimeisessä sarakkees-

sa on keskiarvo jaettu suorituksessa olevien ohjelmien lukumäärällä, jolloin saadaan viitearvio yhden ohjelman tehokkuudelle.

Taulukko 8.2. Viiden yhtäaikaisen ohjelman suoritusajat [ms]

	1a	1b	2a	2b	3a	3b	4a	4b	5a	5b	$\bar{x}_5$	$\bar{x}_5/5$
Initialize	1732	1251	2533	1321	1702	1752	1762	1832	831	1802	**1652**	**330**
SetCurrentWork-book	711	1022	271	902	831	902	741	932	1582	922	**882**	**176**
SearchAndSet-ClassifierRole	4577	1883	3745	1933	4627	5057	4637	5027	4447	5037	**4097**	**819**
ExportFromTed	1202	320	2033	220	1342	280	1072	290	6068	210	**1304**	**261**
CreateMas	10	10	10	10	10	50	10	10	20	90	**23**	**5**
DoIteration	40	20	30	30	30	61	100	60	31	60	**46**	**9**
InitMachine	0	0	0	10	0	0	0	0	0	10	**2**	**0**
ImportToTED	7851	5838	6379	21721	3405	15342	4656	4206	13819	1272	**8449**	**1690**
Yhteensä	**16123**	**10344**	**15001**	**26147**	**11947**	**23444**	**12978**	**12357**	**26798**	**9403**	**16454**	**3291**

LIITE 3

Tässä on esitetty luvussa 5 käytetty nopeustestiohjelma.

```
bool dotest(CWorker& worker,BSTR symbol,BSTR host,BSTR workbookName,BSTR
participantName,BSTR viewName,BSTR modelName)
{
     HRESULT hr;
     string w;
     bool abortvalue=false;
     CStateMachine machine,mac3;
     CMAS* mas;
     CMAS* mas2;
     DWORD alkuaika;

     assert(timeBeginPeriod(1)==TIMERR_NOERROR);
     alkuaika=timeGetTime();
     hr = worker.Initialize(host, symbol);
     cerr<< "Initialize="<<timeGetTime()-alkuaika<<endl;
     if (SUCCEEDED(hr)) {
            alkuaika=timeGetTime();
            hr=worker.SetCurrentWorkbook(workbookName);
            cerr<< "SetCurrentWorkbook="<<timeGetTime()-alkuaika<<endl;
            alkuaika=timeGetTime();
            if (SUCCEEDED(hr))
                   hr=worker.SearchAndSetClassifierRole(participantName,true);
            cerr<< "SearchAndSetClassifierRole="<<timeGetTime()-alkuaika<<endl;
            if (SUCCEEDED(hr)) {
                   worker.PrepareItems();
                   worker.PrintItems();
                   w=worker.GetItemsAsString();

            }

            if (SUCCEEDED(hr)) {
                   CComBSTR model("Jomppa_work/Automaatti");
                   worker.SetCurrentWorkbook(model);
                   alkuaika=timeGetTime();
                   hr=worker.ExportFromTed(&machine);
                   cerr<< "ExportFromTed="<<timeGetTime()-alkuaika<<endl;
                   if (SUCCEEDED(hr)) {
                          machine.PrintMachine(cout);
                          alkuaika=timeGetTime();
                          machine.CreateMas(&mas);
                          cerr<< "CreateMas="<<timeGetTime()-alkuaika<<endl;
                          mas->PrintTable(cout);
                          alkuaika=timeGetTime();
                          mas->InsertExample(w,abortvalue);
                          mas->DoIteration(abortvalue,true);
                          cerr<< "DoIteration="<<timeGetTime()-alkuaika<<endl;
                          mas->PrintTable(cout);
                          alkuaika=timeGetTime();
                          mac3.InitMachine(mas);
                          cerr<< "InitMachine="<<timeGetTime()-alkuaika<<endl;
                          worker.SetMAS(mas);
                          worker.SetStateMachine(&mac3);
                          alkuaika=timeGetTime();
                          worker.ImportToTED(&modelName,&viewName);
                          cerr<< "ImportToTED="<<timeGetTime()-alkuaika<<endl;
                         mac3.PrintMachine(cout);
                          delete mas;
                   }
            }
     }
     timeEndPeriod(1);
     return true;
}
```

LIITE 4

Tässä liitteessä on listattu käytettyjen luokkien otsikkotiedostot.

CMAS_Alpha

```
class CMAS_Alpha
{
public:
     string GetString() const;
     string m_action;
     string m_event;
     CMAS_Alpha(){};
     CMAS_Alpha(const CMAS_Alpha& item)
     {
            m_action=item.m_action;
            m_event=item.m_event;
     }
     const CMAS_Alpha& operator =(const CMAS_Alpha& item)
     {
            m_action=item.m_action;
            m_event=item.m_event;
            return *this;
     }
};

bool operator<(const CMAS_Alpha& x,const CMAS_Alpha& y);
bool operator==(const CMAS_Alpha& x,const CMAS_Alpha& y);
```

CMAS_Row

```
class CMAS_Row
{
public:
     typedef vector<bool> T_TYPE;
     string m_str; // name (s_ct,VOID)
     T_TYPE m_T;
     bool m_bIsI;
     CMAS_Row ():m_bIsI(false){}
     CMAS_Row (const string& name):m_bIsI(false) {m_str=name;}
     CMAS_Row (const CMAS_Row & item)
     {
            m_str=item.m_str;
            m_T=item.m_T;
            m_bIsI=item.m_bIsI;
     }
     const CMAS_Row & operator =(const CMAS_Row & item)
     {
            m_str=item.m_str;
            m_T=item.m_T;
            m_bIsI=item.m_bIsI;
            return *this;
     }
};
```

CMAS

```
class CMAS
{
protected:
     typedef vector<string> R_TYPE;
     enum AFK_NEWVALUE{AFK_YES,AFK_NO,AFK_DONTKNOW,AFK_DONTCARE};
public:
     typedef list<CMAS_Alpha> ACTIONLIST;
     typedef set<CMAS_Alpha> ACTIONSET;
     typedef list<CMAS_Alpha> ALPHASET;
     typedef vector<CMAS_Row> ROWLIST;
     typedef list<string> STRINGLIST;
public:
     enum {CHANGEMAS=-1};
     enum {MAX_ITERS=10000};
public:
     static list<string> Pref(const ACTIONLIST& I);
     static list<string> Suff(const ACTIONLIST& I);
     static bool ConvertToActionlist(string w,ACTIONLIST& l);
     static bool ConvertToActionlist(string w,ACTIONSET& l);
     static int split_string(const string& source, string &key, string &message);
public:
     int GetSSize() const;
     int GetIteration() const;
     const ROWLIST* GetRows() const;
     const ALPHASET* GetAlphas() const;
     const R_TYPE* GetColumns() const;

     void AddEndMark(const string& w);
     bool CheckPartialString(const string& w);
     bool DeleteString(const string& w,bool& abortvalue);
     virtual AFK_NEWVALUE AskForOK(bool& newmas,STRINGLIST& examples) const;
     virtual string AskForR(const int no,const string& ar) const;
     virtual bool AskForValue(const string& w,bool& abortval) const;
     virtual void PrintTable(ostream& os);

     bool IsInI(const string& w) const;
     bool IsInI(const string& w,const string& w2) const;
     string GetRowName(const int i) const;

     bool IsInS(const string& w) const;
     bool IsInS(const int i) const;
     size_t GetNumberOfRows() const;
     int FindRow(const string& w) const;
     size_t InsertExample(const ACTIONLIST& w,bool& abortvalue,bool extend=true);
     size_t InsertExample(const string& w,bool& abortvalue);
     size_t InsertExample(const string& w, const bool value,bool& abortvalue);
     size_t InsertExample(const STRINGLIST& w,bool& abortvalue);
     size_t InsertExample(const STRINGLIST& w, const bool value,bool& abortvalue);
     size_t InsertExample(const ACTIONLIST& w,const bool value,bool& abortvalue,bool
extend=true);
     bool GetRowAsString(const string& row,string& str) const;
     string GetRowAsString(const int i) const;
     ROWLIST::iterator InsertToS(const string& w,bool& abortvalue);
     ROWLIST::iterator InsertToS(const string& w, const bool value,bool& abortvalue);
     int DoIteration(bool& abortvalue,bool skipsetup=false,ostream& os=cout);
     bool CheckRows(const int row1,const int row2) const;
     bool CheckRows(const string& row1,const string& row2,bool& val) const;
     bool CheckConsistent(string& ar) const;
     bool CheckClosed(string& sa) const;
     bool Check_for_Value(const string& w,bool& x) const;
     bool GetValue(const string& row1,const int r,bool& x) const;
     void AddColumnToR(const string& w);
     bool SetupTableVector(const int iter=0);
     bool MakeTrie();
     size_t InitObservationTable(const string& w);
     size_t InitObservationTable(const ACTIONLIST& w);
     size_t InitObservationTable(const STRINGLIST& w);
     size_t InitObservationTable(const list<ACTIONLIST>& w);
```

```
    CTrie* GetTrie() const;
    CMAS();
    virtual ~CMAS();
protected:
    bool extendTfromS();

protected:
    void insertwithcheck(STRINGLIST& l1,const STRINGLIST& l2) const;
    void composealphas(const ACTIONLIST& w);
    virtual bool checkstring(const string& w,bool& abortvalue) const;
    int m_iIter;
    R_TYPE m_R;
    ROWLIST m_rows;
    int m_iS;
    CTrie* m_pTrie;
    ALPHASET m_Alphas;
public:
    const CMAS operator =(const CMAS& item)
    {
            m_R=*(item.GetColumns());
            m_Alphas=*(item.GetAlphas());
            m_rows=*(item.GetRows());
            m_iIter=item.GetIteration();
            m_iS=item.GetSSize();
            if (m_pTrie!=item.GetTrie()) {
                    delete m_pTrie;
                    m_pTrie=item.GetTrie()->CreateCopy();
            }
            return *this;
    }
};
```

CStateMachine

```
class CStateMachine
{
public:
    // typedefs
    typedef int INITIALSTATETYPE;
    typedef CState STATESETTYPE;
    typedef vector<STATESETTYPE> STATESET;
    typedef vector<STATESETTYPE*> PSTATESET;
    typedef int FINALSTATETYPE;
    typedef FINALSTATETYPE FINALSTATESETTYPE;
    typedef set<FINALSTATESETTYPE> FINALSTATESET;
    typedef list<CStateTrans> TRANSITIONLIST;
    typedef set<string> PATHSET;
    typedef pair<string,int> LOOPITEM;
    typedef set<LOOPITEM> LOOPSET;
public:
    void FindFinalState();
    bool IsAcceptedString(string w);
    bool CreateMas(CMAS** mas, const PATHSET& p);
    bool CreateMas(CMAS** mas);
    PATHSET GetCompletePathStrings(const PATHSET& paths,const LOOPSET& loops);
    PATHSET GetCompletePathStrings();
    void insertwithcheck(LOOPSET& set1,const LOOPSET& set2);
    LOOPSET GetSimpleLoops();
    void SetFinalStates(const FINALSTATESET& states);
    void SetInitialState(const INITIALSTATETYPE& state);
    void SetInitialStateDirect(const INITIALSTATETYPE& state);
    void SetInitialState(const string& displayname);
    void SetTransitions(const TRANSITIONLIST& trans);
    void SetStates(const STATESET& states);
    PATHSET GetSimplePaths();
    void SetDeleteOnAllEndMarks(const bool b);
    void RemoveExtraStates();
    void SetupActions();
    TRANSITIONLIST* GetTransitions();
    void PrintMachine(ostream& os);
    bool InitMachine(CMAS* pMAS);
```

```
     STATESET* GetStates();
     CStateMachine();
     virtual ~CStateMachine();
protected:
     void CreatePseudoInitState(int init);
     void findpaths(PATHSET& pathset, INITIALSTATETYPE initialstate,FINALSTATETYPE
finalstate,string path);
     void findloops(LOOPSET& loopset, INITIALSTATETYPE initialstate, FINALSTATETYPE
finalstate,string path);
     void prepareforsearch();
     bool m_bDeleteOnAllEndMarks; // delete states on all endmarks (see
RemoveExtraStates)
     STATESET m_states;     // states in this machine
     INITIALSTATETYPE m_initial_state; // initial state (index to states)
     FINALSTATESET m_final_states; // final state (index to states)
     TRANSITIONLIST m_transitionlist; // transitions in this machine
};
```

CAction

```
class CAction
{
public:
     typedef string ACTIONTYPE;
public:
     ACTIONTYPE m_sAction;
public:
     ACTIONTYPE getName() { return m_sAction;}
     CAction(){}
     CAction(ACTIONTYPE act) {m_sAction=act;}
     CAction(const CAction& item)
     {
            m_sAction=item.m_sAction;
     }
     const CAction& operator =(const CAction& item)
     {
            m_sAction=item.m_sAction;
            return *this;
     }
     bool operator == (const CAction& item)
     {
            return m_sAction==item.m_sAction;
     }

};
bool operator<(const CAction& x,const CAction& y);
```

CStateTrans

```
class CStateTrans
{
public:
     typedef string ACTION;
     typedef string NAME;
     typedef int STATE;
public:
     STATE m_pFrom;
     STATE m_pTo;
     ACTION m_sAction; // these are same in finite automaton, action will be imported to
TED
     NAME m_sName;

public:
     string getAction() const
     {
            return m_sAction;
     }
     string getName() const
     {
            return m_sName;
     }
```

```
    CStateTrans(){}
    CStateTrans(const CStateTrans& item)
    {
            m_pFrom=item.m_pFrom;
            m_pTo=item.m_pTo;
            m_sAction=item.m_sAction;
            m_sName=item.m_sName;

    }
    const CStateTrans& operator =(const CStateTrans& item)
    {
            m_pFrom=item.m_pFrom;
            m_pTo=item.m_pTo;
            m_sAction=item.m_sAction;
            m_sName=item.m_sName;

            return *this;
    }
};
```

CState

```
class CState
{
public:
    typedef string STATE;
    typedef string NAME;
    typedef CAction ACTIONTYPE;
    typedef set<ACTIONTYPE> ACTIONLIST;
    typedef list<CStateTrans> EDGELIST;
public:
    bool IsInitial();
    bool IsFinal();
    bool m_deleted;       // is state deleted by RemoveExtraStates
    bool m_used_in_search; // is state used by findpaths
    bool m_initial; // is a final state
    bool m_final; // is a initial state;
    STATE m_state; // state name (0101001)
    NAME m_sName;  // state name ((display time,set alarm time))
    NAME m_sDisplayName; // state name (q0)
    int m_rowno;   // index to observation table
    EDGELIST m_edges_in;   // edges coming in    O<-o
    EDGELIST m_edges_out;  // edges going out      O->o
    ACTIONLIST m_actions;  // actions (do / display time)
    ACTIONLIST* GetActions();
#ifdef ISTEDCOMPATIBLE // if ted is used
    IUmlStateVertex *m_pTedModel; // pointer to model
    IDenView *m_pTedView;                 // pointer to view
    inline void setTedModel(IUmlStateVertex*s) {m_pTedModel=s;}
    inline IUmlStateVertex* getTedModel(void) { return m_pTedModel;}
    inline void setTedView(IDenView *v) {m_pTedView=v;}
    inline IDenView* getTedView(void) {return m_pTedView;}
    CState():m_pTedModel(NULL),m_pTedView(NULL),m_rowno(0),m_deleted(false),m_initial(f
alse),m_final(false){}

    CState(STATE state,NAME displayname,NAME name, int
rowno):m_state(state),m_rowno(rowno),m_deleted(false),m_initial(false),m_final(false){m
_sName=name;m_sDisplayName=displayname;}

    CState(STATE state,NAME name, int
rowno):m_pTedModel(NULL),m_pTedView(NULL),m_state(state),m_rowno(rowno),m_deleted(false
),m_initial(false),m_final(false){m_sName=m_sDisplayName=name;}

#else

    CState():m_rowno(0),m_deleted(false),m_initial(false),m_final(false){}
```

```
     CState(STATE state,NAME name, int
rowno):m_state(state),m_rowno(rowno),m_deleted(false),m_initial(false),m_final(false){m
_sName=m_sDisplayName=name;}

     CState(STATE state,NAME displayname,NAME name, int
rowno):m_state(state),m_rowno(rowno),m_deleted(false),m_initial(false),m_final(false){m
_sName=name;m_sDisplayName=displayname;}
#endif

     string getName(void) const { return m_sDisplayName;}
     CState(const CState& item)
     {
             m_state=item.m_state;
             m_rowno=item.m_rowno;
             m_sName=item.m_sName;
             m_actions=item.m_actions;
             m_deleted=item.m_deleted;
             m_used_in_search=item.m_used_in_search;
             m_sDisplayName=item.m_sDisplayName;
             m_edges_in=item.m_edges_in;
             m_edges_out=item.m_edges_out;
             m_final=item.m_final;
             m_initial=item.m_initial;

#ifdef ISTEDCOMPATIBLE
             m_pTedModel=item.m_pTedModel;
             m_pTedView=item.m_pTedView;
#endif

     }
     const CState& operator =(const CState& item)
     {
             m_state=item.m_state;
             m_rowno=item.m_rowno;
             m_sName=item.m_sName;
             m_actions=item.m_actions;
             m_deleted=item.m_deleted;
             m_used_in_search=item.m_used_in_search;
             m_sDisplayName=item.m_sDisplayName;
             m_edges_in=item.m_edges_in;
             m_edges_out=item.m_edges_out;
             m_final=item.m_final;
             m_initial=item.m_initial;

#ifdef ISTEDCOMPATIBLE
             m_pTedModel=item.m_pTedModel;
             m_pTedView=item.m_pTedView;
#endif

             return *this;
     }

};
```