TAMPEREEN TEKNILLINEN YLIOPISTO

Koskimaa, Antti Matias

Hajautetun järjestelmän spesifiointi ja testaaminen suoritettavien tilakoneiden avulla

Diplomityö

Tarkastaja: Tommi Mikkonen
Tarkastaja ja aihe hyväksytty
Tieto- ja sähkötekniikan
tiedekuntaneuvoston kokouksessa
5. kesäkuuta 2013

TIIVISTELMÄ

TAMPEREEN TEKNILLINEN YLIOPISTO
Tietotekniikan koulutusohjelma
Koskimaa, Antti Matias: Hajautetun järjestelmän spesifiointi ja testaaminen suoritettavien tilakoneiden avulla
Diplomityö, 46 sivua
Marraskuu 2013
Pääaine: Ohjelmistotuotanto
Tarkastajat: Tommi Mikkonen
Avainsanat: hajautetut ohjelmistot, automaattinen testaus, scxml, statechart

Hajautettu järjestelmä koostuu useasta itsenäisesti toimivasta tietokonesovelluksesta, mikä tekee niiden määrittelystä ja testaamisesta haastavaa. Eräs haasteita tuottava osa on sovellusten välisen rajapinnan käyttö. XML-viesteillä kommunikoivassa järjestelmässä käytetyt viestit voidaan määritellä esimerkiksi XSD-skeemalla, mutta sillä ei voida määritellä sitä, miten viestejä tulee käyttää.

Rajapinnan käytön määrittely on usein ihmisiä varten tehty, jolloin se voi olla puutteellinen ja suurpiirteinen. Tämän takia osaa sen toiminnoista ei välttämättä voida edes toteuttaa. Vaikka ne olisikin mahdollista toteuttaa, eri sovellusten kehittäjät voivat tulkita niiden käytön eri tavalla. Sovelluksia testatessakaan ei välttämättä ole varmuutta siitä toimiiko järjestelmä oikein, jos määritelmä antaa varaa tulkinnalle. Virhetilanteissa havaittu oire voi näkyä muussa kuin virheellisesti toimivassa sovelluksessa, joten virheen paikantaminen on myös työlästä.

Tässä diplomityössä rajapintojen käyttö esitetään tilakoneina, joissa tilat kuvaavat kommunikaation sen hetkisen tilan ja tilasiirtymät kuvaavat mitä viestejä ja millä ehdoilla sovellukset saavat lähettää kussakin tilassa. Nämä tilakoneet määritellään koneluettavalla scxml-merkkauskielellä. Niiden lukemista sekä suorittamista varten toteutetaan tietokonesovellus, jonka tehtävä on valvoa sovellusten välistä viestiliikennettä ja todentaa sitä määritelmää vasten sekä raportoida virhetilanteista.

Kommunikaatioprotokollan määrittely suoritettavilla tilakoneilla osoittautui toimivaksi ratkaisuksi järjestelmän kehityksen ja testauksen tukena. Järjestelmää testatessa se auttoi huomaamaan varmemmin ja paikantamaan nopeammin virheitä. Sillä havaittiin jopa virheitä, jotka eivät aiheuttaneet oireita järjestelmässä. Määrittely tilakoneilla pakottaa määrittelemään kaikki erikoistapauksetkin protokollan käytössä, jolloin rajapinnasta tulee huolellisemmin tehty. Kun järjestelmän voi tarkastaa suoraan määrittelyä vasten, ei määrittely myöskään ole irrallinen toteutuksesta, vaan molempien kehittäminen yhdessä on luontevaa.

ABSTRACT

TAMPERE UNIVERSITY OF TECHNOLOGY
Master's Degree Programme in Information Technology
Koskimaa, Antti Matias: Specification and testing of distributed software with executable state machines
Master of Science Thesis, 46 pages
November 2013
Major: Software Engineering
Examiners: Tommi Mikkonen
Keywords: distributed software, automated testing, scxml, statechart

A distributed system consists of multiple independent computer programs, which makes specifying and testing them difficult. One challenging part is the interface between the applications. The messages used in a system communicating with XML can be specified, for example, with XSD schema, but XSD cannot specify how the messages should be used.

The specification of the interface use is usually made for humans, which means that it might be incomplete or approximate. This may cause some of the requirements to be unable to implement. Even when they can be implemented, the developers of the different applications may interpret the requirements differently. Testing of the software is also problematic, since it is difficult to point out an error, or to state that the system works correctly, if the specification is not exact. Also, a symptom observed in the tests may be seen in some other application than the failing one. This makes locating errors troublesome.

In this thesis the behavior of the interfaces is represented as state machines, in which the transitions represent which messages, and on which conditions, the applications can send in each state of the communication. These state machines are specified with the computer-readable scxml markup language. An application to read and execute them has been implemented. The purpose of the application is to monitor the communication in the system, validate it against the specification, and report messages that do not conform to the specification.

Specifying the communication protocol with executable state machines turned out to be really supportive in the development and testing phase. It helped in noticing the errors more reliably and in locating them faster. It even helped in finding errors that did not cause any visible failures in the system. Specifying the system with state machines forces one to specify all the special cases in the protcol behavior, which makes the design of the interface more thorough. When the system can be tested against the specification, the specification is not disconnected from the implementation, but they both evolve naturally supporting one another.

ALKUSANAT

Tämä diplomityö on kirjoitettu kesällä ja syksyllä 2013. Tahdon kiittää ohjaajaa Juhana Helovuota ja tarkastajaa Tommi Mikkosta oikoluvusta ja kommenteista. Kiitokset myös Atostek Oy:lle työn sponsoroinnista.

Tampereella, 15.10.2013

Antti Koskimaa

SISÄLLYS

1. Johdanto 1
2. Tilakoneiden ja testauksen teoriaa 3
2.1. Tilakone 3
2.1.1. Deterministinen äärellinen automaatti 3
2.1.2. Statechart 5
2.2. Testaus 6
2.2.1. Eri testauksen tasoja 8
2.2.2. Automaattinen ja manuaalinen testaus 8
2.2.3. Virheiden jäljitys 9
3. Hajautetun järjestelmän määrittely 11
3.1. Ihmisluettavat määrittelyt 12
3.1.1. Pelkkä viestien kieliopin määrittely 13
3.1.2. Sanallinen dokumentointi 15
3.1.3. Sekvenssikaaviot 16
3.1.4. Tilakaavio 18
3.2. Suoritettava tilakone 18
4. Kohdejärjestelmä 20
4.1. Yleiskuvaus 20
4.2. Viestiväylä 20
4.3. Järjestelmän nykytila 21
4.4. Määrittely 22
4.5. Testauksessa ilmenneitä ongelmia 22
4.6. Määrittelynmukaisuuden toteaminen manuaalisesti 24
5. Testaussovelluksen toteutus 25
5.1. Hajautetun järjestelmän kuvaaminen tilakoneena 25
5.2. Testaussovelluksen yleiskuvaus 27
5.3. Tilakoneiden määrittely 27
5.4. Skriptaus 30
5.5. Esimerkkitapaus tilakoneen määrittelystä 30

5.6. Testaussovelluksen toteutus 32
6. Arviointi 37
6.1. Havainnot 37
6.2. Tilakoneiden ylläpidettävyys 39
6.3. Virheiden jäljittäminen 40
6.4. Määrittelyn ajantasaisuus 40
6.5. Jatkokehitys 41
7. Yhteenveto 43
Lähteet 45

LYHENTEET JA TERMIT

DFA	Deterministinen äärellinen automaatti (engl. deterministic finite automaton) [1].
DNS	Domain Name System, internetin nimipalvelujärjestelmän protokolla, [2].
HTTP	HyperText Transfer Protocol, hypertekstin siirtämisen protokolla [3].
häiriö	(engl. failure) sovelluksen ulkoisessa toiminnassa näkyvä määrittelyn vastainen tapahtuma, vian oire, ks. virhe.
regressiotestaus	testausta, jossa tarkastetaan, etteivät uudet toiminnallisuudet aiheuta uusia tai toista jo korjattuja virheitä.
RFC	Request For Comments, kokoelma internetin standardeja ja käytäntöjä.
Statechart	David Harelin tilakaavio, DFA:n laajennos [4].
TCP	Transmission Control Protocol, internetin tiedonsiirron perusprotokolla [5].
TDD	Test Driven Development, testivetoinen kehitys [6].
UML	Unified Modelling Language [7].
vika	(engl. fault) virheellinen toiminta, joka aiheutuu virheellistä kohtaa suoritettaessa, ks. virhe, häiriö.

virhe	(engl. error) sovelluksessa oleva poikkeama määrittelystä, ei riipu siitä suoritetaanko virheellinen kohta.
XML	eXtensible Markup Language.
XPATH	XML Path Language, kieli, jolla voidaan viitata tiettyihin elementteihin XML-dokumentin sisällä [8].
XSD	XML Schema Definition, tapa määritellä XML-dokumentin rakenne [9].

1. JOHDANTO

Hajautettu järjestelmä koostuu useasta toistensa kanssa kommunikoivasta tietokonesovelluksesta. Järjestelmän tila on normaalia ohjelmistoa monimutkaisempi, sillä jokaisella sovelluksella on oma tilansa ja niitä on vaikea havaita samaan aikaan. Lisäksi järjestelmän osana olevilla sovelluksilla on omat määrittelynsä ja vaatimuksensa. Järjestelmän osana toimivia sovelluksia saattaa kehittää useampi eri organisaatio, joten määrittely on erityisen tarkkuutta vaativaa.

Järjestelmän testauksessa havaitut virheet voivat johtua yksittäisen sovelluksen virheistä tai ongelmista sovellusten integroinnissa, kun sovellukset olettavat toisiltansa eri asioita. Virhetilanteissa oire saattaa jopa näkyä eri sovelluksessa kuin missä virhe on, mikä vaikeuttaa virheen paikantamista.

Tässä diplomityössä kehitetään tapa määritellä hajautetun järjestelmän protokolla tietokoneluettavasti siten, että tätä määrittelyä voidaan käyttää myös testauksen apuna. Painopiste on virheissä, jotka johtuvat siitä, että sovellukset eivät toimi yhteen oikein. Tällä tarkoitetaan toistettavaa tilannetta, jossa sovellus toimii tavalla, jota toinen sovellus ei oleta. Oire voi olla esimerkiksi väärä viesti, väärän sisältöinen viesti, odottamaton viesti tai ei viestiä silloin kun sellaista odotetaan. Syynä tähän voi olla ohjelmointivirhe jommassakummassa sovelluksessa, eri lailla tulkittu määrittely tai erityistapaus, joka on jätetty määrittelemättä ja johon molemmissa sovelluksissa on tehty omat olettamuksensa.

Jotta sovellusten määrittelynvastaisuus voidaan havaita, tulee määrittely tehdä yksiselitteisesti. Tätä varten hajautetun järjestelmän protokolla mallinnetaan tässä diplomityössä tilakoneilla, joita voidaan lukea tässä toteutetulla tietokonesovelluksella. Tämä sovellus myös seuraa järjestelmän sovellusten välistä kommunikaatiota ja huomaa siinä olevat määrittelyn vastaiset viestit. Näin järjestelmässä olevat virheet kommunikaatioprotokollan käytössä voidaan havaita automaattisesti ja virhe voidaan paikantaa helpommin virheellisen viestin perusteella. Koska tilakoneiden määrittelyt ovat koneluettavia, ihmisen ei tarvitse tulkita toimiiko järjestelmä määrittelyn mukaan. Ihmisen ei tarvitse myöskään

verrata järjestelmän toimintaa määrittelyä vasten, eli oikeellisuuden todentaminen ei ole yksittäisen testaajan tulkinnoista riippuvainen.

Ensin luvussa 2 tutustutaan siihen mikä on tilakone ja määritellään ohjelmistojen testaus. Tämän jälkeen luvussa 3 käsitellään erilaisia tapoja määritellä hajautetun järjestelmän protokolla ja pohditaan ihmisluettavien määrittelyiden täsmällisyyttä. Lopuksi luvussa esitellään tapa määritellä protokolla siten, että järjestelmä voidaan verifoida sitä vasten automaattisesti. Luvussa 4 esitellään eräs kehityksen alla oleva hajautettu järjestelmä ja pohditaan sen testauksessa ilmenneitä ongelmia. Tämän järjestelmän kommunikaatioprotokollan kehityksessä havaittujen ongelmien ratkaisemiseksi luvussa 5 esitellään testaussovellus, joka käyttää luvussa 3 esiteltyjä koneluettavia tilakoneita ja validoi järjestelmän kommunikaation näitä spesifikaatioita vastaan. Lopuksi luvussa 6 analysoidaan kuinka testaussovellus soveltui järjestelmän testaamiseen ja virheiden paikantamiseen. Luvussa 7 on tämän työn yhteenveto.

2. TILAKONEIDEN JA TESTAUKSEN TEORIAA

Tässä luvussa esitellään tilakoneiden sekä testauksen teoriaa. Tilakoneilla on luontevaa kuvata järjestelmän käyttäytymistä, eli miten se reagoi erilaisiin ulkopuolisiin herätteisiin. Tässä työssä toteutettu sovellus käyttää tilakoneita hajautetun järjestelmän kommunikaatioprotokollan määritelmänmukaisuuden todentamiseen. Näissä tilakoneissa tilasiirtymät ovat sovellusten välisiä viestejä.

Toinen osa tätä työtä on järjestelmän määrittelynmukaisuuden testaaminen. Tämän vuoksi tässä luvussa käsitellään testaukseen liittyvää teoriaa siltä osin kuin se on tämän työn kannalta oleellista.

2.1. Tilakone

Tässä kohdassa käsitellään tilakoneita ensin deterministisen äärellisen automaatin (DFA, Deterministic Finite Automaton) kautta. Tämän jälkeen käsitellään statechartia, joka on laajennos DFA:sta ja jolla tässä työssä käytetyt tilakoneet määritellään.

2.1.1. Deterministinen äärellinen automaatti

DFA on matemaattinen rakenne, joka määritellään viisikkona (Q, Σ, δ , q_0, F), jossa

- Q on äärellinen joukko tiloja
- Σ on äärellinen joukko symboleita
- ($\delta : Q \times \Sigma \rightarrow$ Q) on siirtymäfunktio
- ($q_0 \in$ Q) on alkutila
- sekä (F $\subseteq$ Q) on joukko hyväksymistiloja (engl. accepting states). [1]

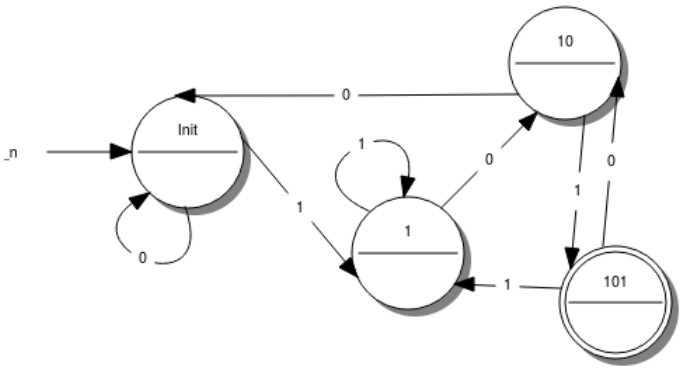
DFA:n suoritus alkaa tilasta q_0, ja aina syötteen luettuaan automaatti siirtyy seuraavaan tilaan siirtymäfunktion mukaan. Tilasiirtymä voi olla myös samaan tilaan kuin mistä lähdettiin, eli esimerkiksi tilasta q_1 syötteellä a siirrytään takaisin tilaan q_1. Syötteen loputtua automaatti hyväksyy syötteen mikäli se jää hyväksymistilaan. Joukkoa syötteitä, jonka automaatti hyväksyy, kutsutaan automaatin tunnistamaksi kieleksi.

Taulukossa 2.1 on esitetty kuvassa 2.1 näkyvän DFA:n siirtymäfunktio. Muilta osin DFA määritellään seuraavasti:

- Q = {Init, 1, 10, 101}
- Σ = {1,0}
- q_0 = Init
- F = {101}.

		Syöte	
		0	1
Tila	Init	Init	1
	1	1	10
	10	Init	101
	101	10	1

Taulukko 2.1: Siirtymäfunktio taulukkomuodossa.



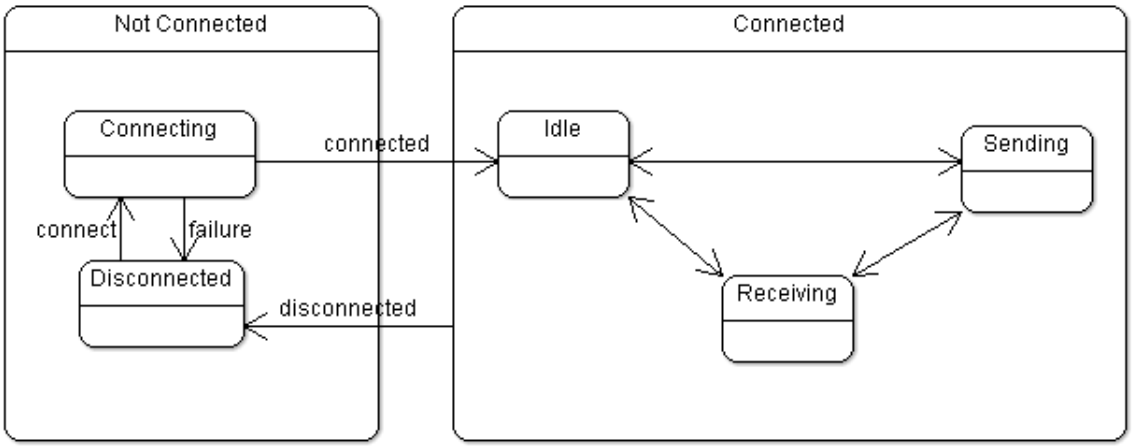
Kuva 2.1: Esimerkki deterministisestä äärellisestä automaatista.

DFA:n tilat ovat siis Init, 1, 10 ja 101. Automaatin syötteet ovat 1 ja 0. Sen alkutila on Init ja hyväksymistiloja on vain yksi, 101. Automaatti hyväksyy minkä tahansa ykkösistä ja nollista koostuvan merkkijonon, joka loppuu merkkeihin 101. DFA:n avulla voidaan siis jakaa symbolijonoja $s \in \Sigma^*$ hyväksyttyihin ja ei-hyväksyttyihin syötteisiin.

2.1.2. Statechart

Tässä työssä tilakoneella tarkoitetaan Harelin Statechart:ia [4], joka on mm. UML:sta tuttu äärellisen tila-automaatin laajennos. Tämän työn kannalta oleellinen eroavaisuus äärelliseen automaattiin verrattuna on ylä- ja alitilat. Ylätilalla voidaan ryhmitellä tiloja, joilla on yhteisiä siirtymiä. Ilman tilojen ryhmittelyä ylätilaksi jokaisesta alitilasta tulisi määritellä erikseen sama tilasiirtymä, kun taas ryhmitellyistä tiloista tarvitsee määritellä ainoastaan yksi. Ylä- ja alitilat eivät siis tuo mitään uutta äärellisen tila-automaatin teoriaan, vaan ne ovat vain syntaktista sokeria tilakaavion lukemisen ja kirjoittamisen helpottamiseksi. Tiloja ryhmittelemällä saadaan tarvittaessa korkeampi abstraktiokerros tilakaavioon. Statechartissa ei myöskään ole hyväksymistiloja, kuten DFA:ssa, vaan niiden tilalla on lopputilat, joista ei ole siirtymiä muihin tiloihin. Näiden tarkoituksena on ilmaista tilakoneen suorituksen loppuminen.

Kuvassa 2.2 on statechart, joka erään jonkin tietoliikenneyhteyden tilaa. Väylästä voidaan lukea tietoa ja siihen voidaan kirjoittaa. Statechartissa on tilat *Disconnected*, *Connecting*, *Idle*, *Sending* ja *Receiving*. Lisäksi kaaviossa on tila *Connected*, jonka alle *Idle*, *Sending* ja *Receiving* on ryhmitelty sekä *Not Connected*, jonka alle *Disconnected* ja *Connecting* on ryhmitelty. Ryhmittelyt sekä selventävät lukijalle milloin yhteys on auki ja milloin ei, että helpottavat kaavion lukemista, kun esimerkiksi tilasta *Connected* on vain yksi siirtymä tilaan *Disconnected* sen sijaan, että kaikista tämän tilan alle ryhmitellyistä tiloista olisi siirtymät tilaan *Disconnected*.



Kuva 2.2: Statechart.

Statechartit ovat ihmisille suhteellisen helppoja tulkita, minkä lisäksi niillä voidaan

esittää täsmällisesti toimintoja. Dobingin et al [10] kyselytutkimuksen mukaan niitä ei käytetä niin usein kuin muita UML:n kaavioita, mutta silloin kuin niitä käytetään, ne koetaan hyödyllisiksi ohjelmistoprojektissa. Vastaajat eivät kokeneet niiden olevan tarpeellisia useimpiin projekteihin, mutta silloin kun niitä tarvitaan, ne koetaan erittäin tarpeellisiksi, sillä ne tarjoavat usein uutta, eivätkä esitä toisteista tietoa.

Statecharteille löytyy kirjallisuudesta useita erilaisia sovellutuksia. Niitä on esimerkiksi hyödynnetty käyttötapausten validointiin. Käyttötapauksista luotiin tilakoneita, jotka yhdistämällä saatiin kaikki käyttötapaukset käsittävä yksi tilakone. Tämän tilakoneen avulla on mahdollista havaita ristiriitoja, puutteita tai virheitä määrittelyssä. [11]

Lisäksi statecharteja on myös muun muassa käytetty automaattiseen testitapausten luontiin. Kun sovelluksen käyttäytyminen kuvataan tilakaaviolla, voidaan tämän kaavion pohjalta luoda testitapauksia siten, että kaikki tilat ja siirtymät käydään läpi. [12]

2.2. Testaus

Kaikki sovellukset on määritelty toimimaan tietyllä tavalla. Vaikka tätä määrittelyä ei olisikaan kirjoitettu auki, on se kuitenkin vähintään ohjelmoijan, suunnittelijan tai tilaajan mielessä. Sovellusta tehdessä ei aina tulla ajatelleeksi kaikkia mahdollisia syötteitä, vaan sovellus saatetaan suunnitella tietyille laillisille syötteille unohtaen erikoistapaukset.

Testauksen tarkoitus on löytää sovelluksesta virheitä, eli poikkeamia määrittelystä, saada sovellus virheelliseen tilaan. Tässä työssä virheellä (error) tarkoitetaan sovelluksessa olevaa poikkeamaa määrittelystä. Kun virheellinen kohta suoritetaan, aiheutuu vika (fault, defect). Vika saattaa näkyä ohjelman ulkoisessa toiminnassa häiriönä (failure). Vika ei välttämättä näy häiriönä ollenkaan tai se voi näkyä useana häiriönä useassa kohtaa sovellusta. [13]

Listauksessa 2.1 on yksinkertainen C-kielellä kirjoitettu funktio, jonka tarkoitus on verrata kahden liukuluvun suurinpiirteistä yhtäsuurutta pyöristysvirheistä välittämättä. Lukujen on määritelty olevan yhtäsuuria, mikäli niiden erotus on pienempi kuin 0.001. Funktiossa oleva virhe on se, että x:n ja y:n erotus voi olla negatiivinen, jolloin se on aina pienempi kuin 0.001 [1]. Kun virheellinen kohta suoritetaan siten, että x on pienempi kuin y, aiheutuu vika, eli tilanne jossa luvut tulkitaan yhtäsuuriksi, vaikka ero olisi suurikin.

[1] Lisäksi lukujen erotus saattaa aiheuttaa ylivuodon, mitä ei myöskään tarkasteta. Tässä jätetään se kuitenkin yksinkertaisuuden vuoksi huomiotta.

Häiriö riippuu täysin siitä missä yhteydessä funktiota käytetään. Voi myös olla, että vika ei näy häiriönä ollenkaan. Vika ei myöskään välttämättä koskaan tapahdu, mikäli ennen funktion kutsua voidaan olla varmoja siitä, että x on suurempi tai yhtä suuri kuin y. Voi olla, että vika aiheutuu vasta kun funktiota käytetään jossain uudessa paikassa, missä tätä varmuutta ei enää ole.

```
bool roughlyEquals(float x, float y)
{
  if (x - y < 0.001)
  {
    return true;
  }
  return false;
}
```

Listaus 2.1: Funktio kahden liukuluvun yhtäsuuruusvertailuun.

Sovellusta ei voida testata kaikilla mahdollisilla syötteillä, eikä testaamisen avulla voida saada täyttä varmuutta siitä, että sovellus toimii oikein. Esimerkkifunktio ottaa parametreikseen kaksi liukulukua, joten kaikilla mahdollisilla arvoilla testaaminen vaatisi $1.8 * 10^{19}$ testitapausta olettaen, että *float* on 32-bittinen.

Funktion parametrit voidaan kuitenkin jakaa *ekvivalenssiluokkiin* ja testit voidaan suorittaa kullekin ekvivalenssiluokalle erikseen. Ekvivalenssiluokkiin jako ei ole suoraviivaista, vaan vaatii jonkinasteista luovuutta testaajalta [13]. Testaaja voi esimerkiksi päätellä, että edellä kuvatun funktion testaus syötteillä *x = 100* ja *y = 1* tuottaa saman lopputuloksen kuin testaus syötteillä *x = 200* ja *y = 1*, sillä funktio tarkastelee x:n ja y:n erotusta ja molemmissa tapauksissa x oli paljon yli 0.001 suurempi kuin y. Funktion syötteiden jako ekvivalenssiluokkiin on esitelty taulussa 2.2.

x	y	x:n ja y:n suhde	funktion paluuarvo
>0	>0	x > y && x - y > 0.001	false
>0	>0	y > x && y - x > 0.001	false
>0	>0	x > y && x - y < 0.001	true
>0	>0	y > x && y - x < 0.001	true
>0	<0	x - y > 0.001	false
>0	<0	x - y < 0.001	true
<0	>0	y - x > 0.001	false
<0	>0	y - x < 0.001	true

Taulukko 2.2: Testisyötteiden jako ekvivalenssiluokkiin.

Näiden ekvivalenssiluokkien sisällä miten tahansa valittujen parametrien arvon voidaan olettaa tuottavan sama lopputulos. Yleensä ottaen hyödyllisempää kuin valita mikä tahansa arvo ekvivalenssiluokan sisältä on tarkastella ekvivalenssiluokan reunoja. Esimerkiksi ensimmäisessä ekvivalenssiluokassa sen sijaan, että valittaisiin arvot x = 2 ja y = 1, valitaankin arvot x = 1.001 ja y = 1. [13]

Testin tuloksena on mahdotonta sanoa varmasti, että testattava sovellus toimii *oikein*. Testauksen avulla voidaan osoittaa virhe tai se, että tietyillä syötteillä ei tapahtunut virheitä, mutta kuten aina testauksessa, täydellisen oikeellisuuden osoittaminen on edelleen mahdotonta. Se, että testatessa ei löydy virheitä, ei tarkoita sitä, että sovellus toimii virheettömästi, vaan että se toimii testatuissa tilanteissa määritelmän mukaan, joka sekin voi olla virheellinen. Täydellisen varmuuden saaminen virheettömyydestä vaatii virheettömän spesifikaation sekä testauksen kaikilla mahdollisilla syötteillä, jotka ovat molemmat mahdottomia vaatimuksia. [13]

2.2.1. Eri testauksen tasoja

Yksikkötestauksella testataan tiettyä osaa sovelluksesta, esimerkiksi yksittäistä funktiota. Tarkoituksena on todeta, että yksittäinen sovelluksen osa toimii yksinään kuten pitääkin. Koska testattava osuus on pieni, yksikkötestissä havaitut virheet on suhteellisen helppo paikallistaa ja korjata. [13]

Integrointitestauksessa testataan useampaa sovelluksen komponenttia yhteenliitettyinä. Tarkoituksena on löytää virheitä komponenttien rajapinnoista tai niiden toiminnasta yhdessä, kun on ensin yksikkötestattu se, että komponentit toimivat erikseen. [13]

Järjestelmätestauksessa testataan valmista sovellusta sovelluksen spesifikaatiota vasten. [13]

2.2.2. Automaattinen ja manuaalinen testaus

Manuaalisella testauksella tarkoitetaan ihmisen suorittamaa ja analysoimaa testiä. Yleensä tämä on jo valmiin järjestelmän tai prototyypin testausta, jonka tarkoituksena on verrata sovelluksen toimintaa määrittelyä vasten.

Automaattisella testauksella tarkoitetaan menetelmää, joka testaa jotain ohjelman osaa tietyillä syötteillä ja tarkistaa tulokset ilman ihmisen tekemää työtä. Testitapaus (engl. test

case) testaa yleensä yhtä funktiota. Alustustoimenpiteiden jälkeen funktiota kutsutaan tietyillä syötteillä ja paluuarvoa verrataan oletettuun paluuarvoon. Joukko esimerkiksi samaa toiminnallisuutta eri syötteillä testaavia testitapauksia voidaan ryhmitellä yhdeksi testijoukoksi (engl. test suite). Testauskehyksen (engl. test framework) avulla voidaan ajaa automaattisesti kaikki testitapaukset. Tämä raportoi testiajon jälkeen läpi menneet testit sekä testit, joissa on havaittu häiriö. Automaattisella testauksella voidaan nopeasti testata sovellusta kattavasti ja toistuvasti ilman suurta työpanosta kehityksen aikana. Testien kirjoitus toki teettää työtä. Vaikka testeissä ei havaittaisikaan häiriöitä, testeissä voi silti olla virheitä, tai ne eivät välttämättä ole tarpeeksi kattavia. Lisäksi testitapaukset testaavat vain niissä määriteltyjä asioita. Manuaalisella testauksella ihminen olisi saattanut huomata jonkin ennalta määrittelemättömän häiriön syötteillä, mutta tietokone tarkastaa vain testeissä määritellyt asiat. Toisaalta taas ihmistestaajalta saattaa jäädä joitakin virheitä näkemättä tai testitapauksia suorittamatta esimerkiksi huolimattomuuden takia.

Yksikkötestejä saatetaan kirjoittaa jo funktion ohjelmoinnin aikana sekä ajatustyön helpottamiseksi että toimintavarmuuden saamiseksi nopeasti. Jotkut menetelmät, esimerkiksi TDD (Test Driven Development, testivetoinen kehitys), suosivat jopa testien kirjoittamista ennen ohjelmointia [6].

Automaattinen testaus on hyvä ja nopea keino suorittaa *regressiotestaus* sovellukselle pienellä työpanoksella. Regressiotestauksella tarkoitetaan testausta, jossa tarkastetaan, että uudet ominaisuudet eivät muuta olemassaolevaa muuta toiminnallisuutta. Kun kaikki testit menevät läpi ennen uuden toiminnallisuuden lisäämistä ja sen jälkeen, voidaan paremmin luottaa siihen, että uusi toiminnallisuus ei riko olemassaolevaa toiminnallisuutta. Suurella määrällä testitapauksia saadaan tätä varmuutta suremmaksi.

2.2.3. Virheiden jäljitys

Häiriön havaitsemisen jälkeen ohjelman virhe tulee löytää ennen kuin sen voi korjata. Löytämisestä ja korjaamisesta virheen löytäminen on näistä kahdesta yleensä ylivoimaisesti työläämpää ja aikaa vievämpää. Virheen löytämiseksi on useita erilaisia tapoja. Myers [13] jakaa tavat karkeasti kahteen kategoriaan; virheen raa'alla voimalla (brute force) jäljittäminen sekä ajattelemalla jäljittäminen. Raa'alla voimalla jäljittämiseksi hän lukee muun muassa koodin sekaan viljellyt tulostukset, muistilistauksen analysoinnin se-

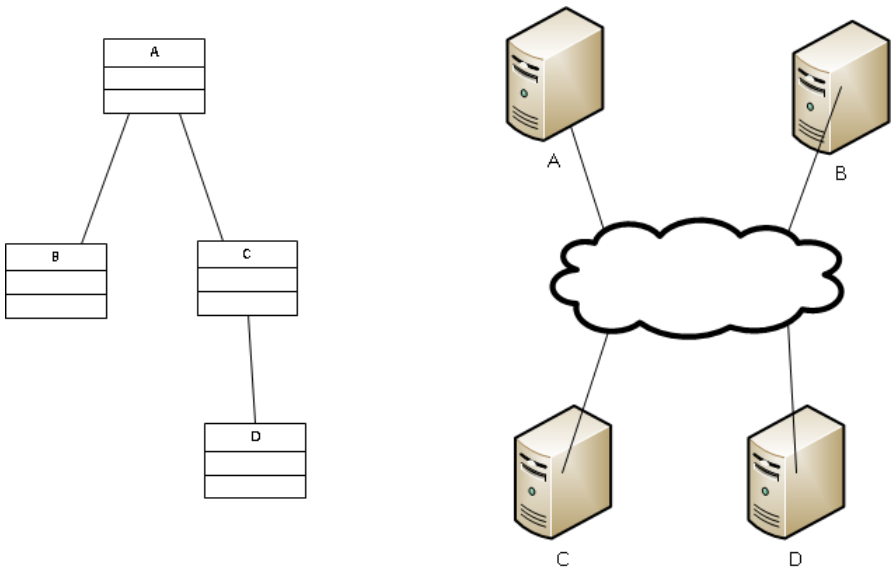
kä automaattisten virheiden jäljitystyökalujen (debugger) käyttämisen. Näitä tapoja yhdistää se, että niitä käyttäen joutuu analysoimaan isoa joukkoa mahdollisesti epäoleellista tietoa, koko ohjelman tilaa. Ohjelman tilaa katsomalla ei näe virhettä välttämättä kovin helposti, minkä lisäksi pelkästä virheellisestä tilasta ei voida päätellä mitä on tapahtunut. Pelkkä työkalujen käyttö ei auta ymmärtämään sovellusta eikä sitä, mikä siinä on vialla.

Toinen kategoria virheen löytämiseksi on ajattelu ja päättely. Tähän kuuluu oireen tarkempi analysointi, jossa mietitään tarkemmin esimerkiksi syötteitä, joilla vika ilmenee, sekä mahdollisia osia ohjelmasta, jotka sen voisivat aiheuttaa. Näiden pohjalta voidaan tehdä hypoteeseja ja todeta ne oikeiksi tai vääriksi. Ohjelmakoodia voi käydä myös lävitse pelkästään omassa päässään turvautumatta esimerkiksi kehitysympäristön debuggeriin. Näitä tapoja yhdistää se, että niiden tarkoituksena on ymmärtää sovelluksen toiminta sen sijaan, että ainoastaan sen muuttujien tilaa tarkasteltaisiin. Myersin mukaan virheiden jäljitys on ongelmanratkaisua, joka vaatii tarkkaa analyysia, eikä välttämättä ollenkaan itse sovelluksen ajoa. Jäljitystyökaluja pitäisi käyttää vain ajatustyöskentelyn ohessa tai viimeisenä keinona.

3. HAJAUTETUN JÄRJESTELMÄN MÄÄRITTELY

Hajautetulla järjestelmällä tarkoitetaan järjestelmää, jonka suoritus on hajautettu useampaan prosessiin, sovellukseen tai fyysiseen tietokoneeseen. Tässä työssä termillä käsitetään useamman eri sovelluksen muodostama kokonaisuus, jossa sovellukset suorittavat omia toimintojansa ja kommunikoivat toistensa kanssa esimerkiksi antaen toiselle sovellukselle syötteitä.

Kuvassa 3.1 on kuvattuna rinnakkain ei-hajautettu järjestelmä, jossa eri toiminnallisuuden toteuttavat modulit käyttävät toisiaan sovelluksen sisällä. Tämän vierellä on sama sovellus hajautettuna, jossa eri toiminnallisuudet ovat omissa sovelluksissaan ja ne kommunikoivat toistensa kanssa jollain tavalla.



Kuva 3.1: Hajautettu järjestelmä.

Tässä työssä oletetaan kommunikaatiokanavaksi viestiväylä, johon sovellukset lähettävät XML-viestejä. Esitettyjä asioita voidaan soveltaa myös muunlaisiin järjestelmiin, joissa sovellukset voivat kommunikoida muillakin tavoilla kuin XML-viesteillä. Koska useampi sovellus kommunikoi keskenään ja olettaa tiettyjä asioita muista sovelluksista, oleellista hajautetussa järjestelmässä on yksittäisten sovellusten määrittelyn lisäksi myös

määritellä kokonaisuus. Hajautetun järjestelmän tapauksessa tämä tarkoittaa sovellusten välillä käytäviä kommunikaatioita, eli viestienvaihtoja. Tähän kuuluu sekä viestien rakenteen, että myös niiden tarkoituksen, semantiikan, määrittely. Esimerkiksi mihin tarkoitukseen viestit on tarkoitettu, miten niitä tulee käyttää, miten järjestelmien tulee tulkita viestit missäkin vaiheessa ja miten niihin kuuluu reagoida. Sallittujen tilanteiden määrittelyn lisäksi tulee määritellä myös laittomat tilanteet. Laittomilla tilanteilla tarkoitetaan tässä esimerkiksi tilannetta, jossa viestiin A pitäisi vastata viestillä B, mutta siihen vastataankin viestillä C. Vastaanottava sovellus odottaa viestiä B, mutta sen tulee reagoida jollain tavalla myös siihen, jos vastaus onkin jokin muu ja koko järjestelmän tulee toipua tästä.

Aluksi tässä luvussa esitellään erilaisia ihmisluettavia tapoja määritellä hajautetun järjestelmän protokolla. Esitetyt tavat ovat epätarkkoja ja virhealttiita. Kaikissa on suurimpana ongelmana se, että niillä on helppo kuvata yleisluontoisia asioita, mutta tarkkojen yksityiskohtien kuvaaminen on työläämpää. Protokollan määrittelyn tulee olla yksityiskohtaista ja täsmällistä, jotta kaikilla sovelluksilla on samanlainen käsitys sen toiminnasta, eikä varaa tulkinnoille jää.

Lopuksi tässä luvussa esitetään tapa määritellä protokolla käyttäen konesuoritettavia tilakoneita. Sen lisäksi, että ihmiset voivat lukea näin tehtyä määrittelyä, myös tietokone voi todeta järjestelmän käyttäytyvän tämän määrittelyn mukaisesti.

3.1. Ihmisluettavat määrittelyt

Ihmisluettavilla määrittelyillä tarkoitetaan tekstiä ja kuvia, joiden avulla määritellään järjestelmän toiminta. Ihmisluettavat määrittelyt ovat usein yksinkertaistettuja luettavuuden helpottamiseksi. Tällöin ne eivät määrittele tarkasti järjestelmän toimintaa erilaisissa tilanteissa, vaan jättävät varaa tulkinnoille. Täsmällisten määrittelyjen tekeminen taas on erittäin työlästä ja kallista. Sen lukeminen, ymmärtäminen ja soveltaminen järjestelmän kehittämiseen sekä määrittelynmukaisuuden todentamiseen on myös työlästä. Vaatimusten muuttuessa muutokset tulee toteuttaa sekä dokumentaatioon että toteutukseen erikseen.

3.1.1. Pelkkä viestien kieliopin määrittely

Yksinkertaisin tapa määritellä protokolla on määritellä ainoastaan kommunikoinnin syntaksi, XML-viestien tapauksessa yleinen määrittelytapa on XSD-skeema (XML Schema Definition) [9]. Mikäli viestien elementtien nimet ovat selkeää luonnollista kieltä, niiden käyttötarkoitus saattaa olla helppo tulkita. Viestien rakenteista ja niiden sisältämistä nimistä saattaa päätellä mitä kenttiä kuuluu täyttää milloinkin sekä miten viestejä tulee tulkita. Viestien käyttöä kommunikoinnissa, eli mitä viestejä voidaan lähettää missäkin vaiheessa ja miten niihin tulee vastata, taas on vaikeampi päätellä.

Tässä tavassa on ilmiselviä ongelmia. Eri kehittäjät ja eri järjestelmien kehittäjät tekevät erilaisia tulkintoja, joista ainoastaan yksi voi olla oikea. Edes skeeman kehittäjällä ei välttämättä ole tarkkaa käsitystä siitä, kuinka skeemaa tulee joissakin erityistapauksissa käyttää. Spesifioidessa saattaa helposti ajatella vain yksinkertaisia sekä ongelmattomia tapauksia tai vain yleisimpiä ongelmia.

Erilaisista tulkinnoista ja olettamuksista johtuen järjestelmään syntyy vaikeasti toistettavia sekä jäljitettäviä virheitä. Virheiden oireet saattavat helposti olla jossain muualla kuin virheellisesti toimivassa järjestelmässä ja varsinaisen syyn löytäminen on työlästä. Virheen löytymisen jälkeen sovellukset tulee korjata siten, että ne toimivat yhteen, mutta muutoksista voi edelleen olla erilaisia tulkintoja, joten ne eivät välttämättä ratkaise alkuperäistä ongelmaa ja saattavat luoda uusia.

Listauksessa 3.1 on eräs XSD-skeema. Tässä skeemassa on viestit *NewJob*, *CancelJob* ja *JobProgress*. *NewJob*-viestillä voidaan kertoa mitä tuotetta, mistä materiaalista ja kuinka monta kappaletta halutaan valmistettavan. Tälle työlle voidaan raportoida *JobProgress*-viestillä tilatietoina kuinka monta kappaletta tuotteita on valmiina sekä työskenteleekö laite, onko se jo valmis tai että se ei voi suorittaa työtä, koska se on epäkunnossa tai raaka-aineita ei ole tarpeeksi.

Skeemasta voidaan suhteellisen helposti päätellä, että viesteillä voidaan luoda ja perua töitä sekä raportoida töiden tilatietoja. Tästä yksinkertaisesta esimerkistä viestien tarkoitus on pääteltävissä, mutta oikeassa järjestelmässä voi olla useita viestejä eri toimintoja varten. Tällöin saattaa olla vaikeampi päätellä mitkä viestit liittyvät mihinkäkin toimintoon. Esimerkkiskeemasta voidaan suurinpiirtein päätellä kuinka sitä tulee käyttää yleisessä tapauksessa, silloin kun kaikki menee suunnitelmien mukaan ja työ tulee tehdyksi.

```
<?xml version="1.0" encoding="Windows-1252"?>
<xs:schema elementFormDefault="qualified"
           xmlns:xs="http://www.w3.org/2001/XMLSchema">
    <xs:simpleType name="JobStatusCode">
        <xs:restriction base="xs:string">
            <xs:enumeration value="Working" />
            <xs:enumeration value="Ready" />
            <xs:enumeration value="NotEnoughMaterials" />
            <xs:enumeration value="MachineOutOfOrder" />
        </xs:restriction>
    </xs:simpleType>
    <xs:element name="NewJob">
        <xs:complexType>
            <xs:sequence>
                <xs:element name="Material" type="xs:string" />
                <xs:element name="Amount" type="xs:positiveInteger" />
                <xs:element name="Product" type="xs:string" />
                <xs:element name="ID" type="xs:string" />
            </xs:sequence>
        </xs:complexType>
    </xs:element>
    <xs:element name="CancelJob">
        <xs:complexType>
            <xs:sequence>
                <xs:element name="ID" type="xs:string" />
            </xs:sequence>
        </xs:complexType>
    </xs:element>
    <xs:element name="JobProgress">
        <xs:complexType>
            <xs:sequence>
                <xs:element name="ID" type="xs:string" />
                <xs:element name="ProductsReady" type="
   xs:positiveInteger" />
                <xs:element name="Status" type="JobStatusCode" />
            </xs:sequence>
        </xs:complexType>
    </xs:element>
</xs:schema>
```

Listaus 3.1: XSD-skeema.

Siitä on kuitenkin mahdoton päätellä sitä kuinka erikoistilanteissa toimitaan. Esimerkiksi työn statukseksi voidaan raportoida, että materiaaleja ei ole tarpeeksi tai laite on epäkunnossa. Skeema yksinään ei kuitenkaan määrittele mitä työlle tapahtuu tällöin. Jatkuuko sen suoritus normaalisti kun laite on taas kunnossa tai materiaaleja on riittävästi, oletetaanko, että työ täytyy perua kun sitä ei voida suorittaa vai perutaanko se automaattisesti.

Skeemasta ei myöskään voida tulkita aukottomasti kuinka työn suorituksesta tulee raportoida. Viestissä voidaan kertoa yksittäisen tuotteen valmistumisesta, mutta on kuitenkin toteuttajan tulkinnasta kiinni raportoidaanko työn tilatietoja esimerkiksi minuutin vä-

lein, aina uuden tuotteen valmistuttua vai vain silloin kun työ on valmis tai sen suoritus on pysähtynyt.

3.1.2. Sanallinen dokumentointi

Protokollan määrittelyä voidaan laajentaa XSD-skeeman lisäksi myös sanallisella dokumentaatiolla. Tämä dokumentaatio voi olla esimerkiksi skeemassa mukana, XSD:n tapauksessa documentation-tagin sisällä, tai sitten erillisenä dokumenttina. Sanallinen dokumentaatio on kuitenkin työlästä kirjoittaa sekä ylläpitää. Vähänkään monimutkaisemmissa tilanteissa tilanteen selittäminen sanallisesti ja sen ymmärtäminen on sekä haastavaa että virhealtista. Skeeman mukana oleva dokumentaatio taas pysyy helpommin ajan tasalla ja sillä on luontevaa selittää yksittäisen viestin tai elementin tarkoitus, mutta ei suurempia kokonaisuuksia, kuten viestienvaihtoa.

Epämuodollisen dokumentaation virheettömyyttä ei voida todistaa loogisesti. Virheellisyydenkin voi todistaa vain osoittamalla virheen. Myöskään kattavuutta ei voida todistaa. Tämän takia lukija (tai kirjoittaja) ei voi tietää onko dokumentissa kaikki tarvittava tieto. Katselmoinnilla voidaan huomata virheellisiä tilanteita tai määrittelemättömiä erikoistapauksia, mutta silläkään ei voida todeta dokumentaation virheettömyyttä ja kattavuutta. Skeeman tai sen käyttötarkoituksen muuttuessa sanallisen dokumentaation läpikäyminen ja muuttuneiden kohtien korjaaminen on erityisen virhealtista, koska jotain saattaa jäädä määrittelemättä tai uusi ominaisuus saattaa tuoda virheitä tai ristiriitoja määrittelyyn. Lisäksi luonnollisella kielellä on erittäin vaikeaa määritellä asiota täsmällisesti ja helposti ymmärrettävästi samaan aikaan.

Lethbridge et al. [14] mukaan dokumentaatio kuvaa harvoin oikeaa järjestelmää, sillä sitä ei päivitetä tarpeeksi usein. Dokumentaatiota yleensä on liikaa ja hyödyllisen tiedon löytäminen on vaikeaa, eikä dokumentaatioon välttämättä voi luottaa. Koodin seassa olevat kommentit sekä rajapintadokumentit koetaan sen sijaan hyödyllisiksi sekä luotettaviksi, mutta erillisiä määrittelyjä ei. Tämän lisäksi korkeamman tason dokumentaatio, kuten arkkitehtuuridokumentaatio koetaan hyödylliseksi. Vaikka dokumentti olisikin vanhentunut, korkeamman abstraktiotason ratkaisut ovat edelleen hyödyllisiä.

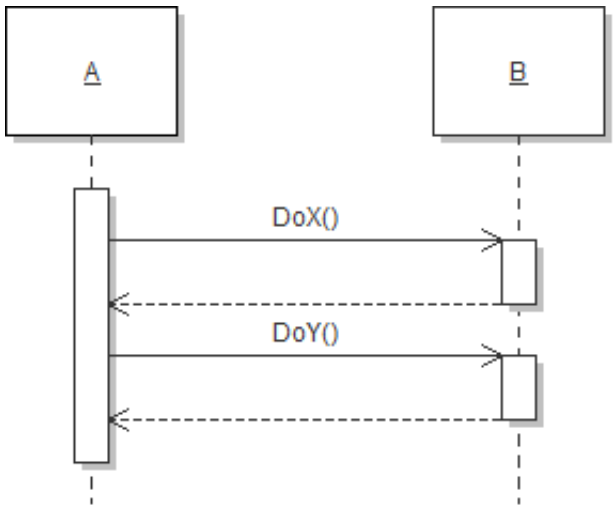
Protokollan määrittely on kuitenkin lähellä toteutusta. Irrallisena dokumenttina se jää helposti päivittämättä, kun toteutus muuttuu ja sen päivittäminen on työlästä. Yhden asian

muuttuminen saattaa vaikuttaa useampaan kohtaan, joista osa saattaa jäädä päivittämättä. Jokaisen päivityksen yhteydessä tulisi varmistaa dokumentin sisäinen eheys ja oikeellisuus ja vaikuttaako muutos johonkin toiseen toimintoon tai viestiin.

Sanallisen dokumentaation ylläpitäminen ei kuitenkaan ole mahdotonta. Esimerkiksi RFC -dokumenteissa on esimerkkejä toimivista ja laajassa käytössä olevista täysin sanallisista dokumenteista. RFC:illä määritellään internetin standardeja, kuten TCP [5], HTTP [3] ja DNS [2]. Dokumentit ovat eri pituisia ja niihin on käytetty eri määrä työtä. Joihinkin on käytetty vähemmän aikaa, eivätkä ne ole niin laajalti käytössä, kun taas jotkut ovat pitkään käytössä olleita ja moneen kertaan päivitettyjä standardeja. Isompien dokumenttien tekemiseen on käytetty usean asiantuntijan iso työpanos. Esimerkiksi HTTP-protokollan version 1.1 (HTTP/1.1) määrittely, RFC 2616, on 176 sivua pitkä ja sen tekijöiksi on lueteltu seitsemän ihmistä [3]. Nämä seitsemän ihmistä eivät toki ole ainoita standardin kehittäjiä tai dokumentin tekoon osallistuneita ihmisiä, eikä tuotettu dokumentti ole ainoa työ, jota standardin eteen on tehty. Dokumentti ei myöskään ole ensimmäinen HTTP-protokollan määrittelevä dokumentti. HTTP/1.1 määriteltiin aiemmin RFC:ssä 2068 [15], mutta siinä olevien puutteiden takia määrittelystä tehtiin uusi versio. Vaikka standardeihin käytetään paljon aikaa, on niistä mahdoton saada *täysin* virheettömiä. Niistä saadaan kuitenkin tarpeeksi virheettömiä ja kattavia, mutta lopputuloksena on erittäin suuri dokumentti, jonka tekemiseen on käytetty erittäin paljon työtunteja. Tällaisten dokumenttien kirjoittamiseen ja ymmärtämiseen ei normaalissa ohjelmistoprojektissa ole aikaa eikä rahaa. Tämän takia dokumentaatio helposti jääkin sivuosaan.

3.1.3. Sekvenssikaaviot

UML määrittelee sekvenssikaaviot (Sequence Diagram), joiden avulla voidaan kuvata järjestelmän osien käymiä keskusteluja esimerkkiskenaarioiden avulla [7]. Niillä kuvataan kommunikaatiossa mukana olevat sovellukset (tai oliot tai sovelluskomponentit) ja niiden välittämät viestit toisillensa sekä vastaukset näihin. Sekvenssikaaviot tekevät dokumentaatiosta havainnollisempaa määrittelemällä sanojen sijaan kuvan avulla miten viestejä käytetään ja minkälaisia keskusteluja eri komponentit käyvät keskenään. Esimerkiksi kuvassa 3.2 on kaksi komponenttia, A ja B. Ensin A kutsuu B:n *DoX*-funktiota (tai palvelua) ja jää odottamaan paluuarvoa, minkä jälkeen A kutsuu B:n funktiota *DoY*.



Kuva 3.2: Sekvenssikaavio.

Sekvenssikaavioiden luonteeseen kuuluu, että niillä kuvataan aina yksittäinen tilanne ja ne näyttävät mitä viestejä tai (funktio)kutsuja ko. tilanteeseen liittyy. Tämä tilanne voi olla toivottu tilanne, tai sillä voidaan esittää mahdollinen virhetilanne sekä siitä toipuminen. Se esittää kuitenkin vain yhden tapauksen kerrallaan, eikä siitä voi päätellä mitenkään onko se ainoa mahdollinen tapa. Kaikkia mahdollisia skenarioita on monissa tapauksissa mahdoton kuvata. Esimerkiksi kaaviosta ei mitenkään voida päätellä tuleeko *DoX*:n jälkeen kutsua *DoY* aina vai vain esimerkkitapauksessa, vai riittääkö, että kutsutaan molempia järjestyksestä riippumatta.

Sekvenssikaaviot kertovat yleensä mitä järjestelmässä voi tapahtua, sillä niiden tarkoitus on kuvata tilanteeseen liittyvät kutsut ja osapuolet. Ne eivät kuitenkaan voi mitenkään kertoa mitä järjestelmässä ei saa tapahtua. Pelkästä joukosta esimerkinomaisia tilanteita ei voi päätellä mitkä tilanteet ovat laittomia tai mahdottomia.

Sekvenssikaavioiden tulkinnassa on samoja ongelmia kuin pelkän skeeman tulkinnassa, mitä käsiteltiin aiemmin. Pelkkä esimerkkiskenario ei kerro skeeman käytöstä kuin pienen osan.

Yksittäisistä esimerkeistä ei myöskään voida päätellä viestien ja parametrien semantiikkaa. Vaikka sekvenssikaaviot voivatkin olla hyvä apu järjestelmän kehityksessä sekä dokumentaation tukena, ne selittävät rajapinnan käytön vain tietyissä tapauksissa. Semantiikan määrittely tarvitsee sanallisen dokumentaation sekvenssikaavioiden tueksi. Kuten sanallinen dokumentaatiokin, myös sekvenssikaaviot saattavat vanhentua toteutuksen muuttuessa.

3.1.4. Tilakaavio

Kommunikaation tilaa on luontevaa kuvata tilakaaviona, jossa viestit ovat tilasiirtymiä. Tilakaaviolla voidaan kuvata yksiselitteisesti mitkä viestit ovat sallittuja missäkin tilanteissa sekä miten ne vaikuttavat keskusteluun. Siitä nähdään kaikki mahdolliset lopputulokset, joihin kommunikaation tuloksena voidaan päätyä. Tilakoneen avulla nähdään koko ajan kokonaiskuva, jolloin määrittelijän tai lukijan ei tarvitse koostaa kokonaisuutta päässään erillisistä dokumenteista.

Aiemmin käsitellyt sanallinen dokumentaatio sekä sekvenssikaaviot ovat toimiva osa dokumentaatiota, mutta eivät yksinään muodosta täsmällistä määrittelyä. Toisin kuin sekvenssikaavioilla, tilakaavioilla voidaan kuvata kaikki mahdolliset viestienvaihdot, minkä vuoksi siitä nähdään myös ei-sallitut tilanteet. Tilakoneet saattavat kuitenkin lisäksi tarvita dokumentaatiota viestien merkityksistä. Se kertoo vain, että mitä viestejä voidaan lähettää missäkin tilassa, mutta ei kerro niiden tulkinnasta mitään.

Vaikka tilakaavioiden avulla voidaan määritellä protokolla täsmällisesti, ovat tilakaaviopiirustukset koodista irrallisia dokumentteja, jotka saattavat vanhentua tai joissa voi olla huomaamatta jääneitä virheitä. Järjestelmän määrittelynmukaisuuden toteaminen on samalla tavalla manuaalista työtä kuin sanallista dokumentaatiotakin vasten tarkistaminen. Lisäksi koko kommunikaatiota kuvaava tilakaavio on vähänkään monimutkaisissa tapauksissa todella iso.

3.2. Suoritettava tilakone

Edellisissä kohdissa kuvatut menetelmät ovat hyviä työkaluja ihmisluettavan, korkeamman abstraktiotason dokumentaation tekemiseen. Protokollan määrittely on kuitenkin lähellä toteutusta, jolloin se vanhenee toteutuksen muuttuessa. Sen on myös tärkeää olla täsmällinen väärien tulkintojen välttämiseksi sekä kehityksessä että testauksessa.

Ratkaisuna tässä ehdotetaan koneluettavaa ja -suoritettavaa tilakonetta, joka reagoi oikealta järjestelmältä saatuihin ärsykkeisiin. Tämä toimii sekä tarkkana määrittelynä kommunikaatioprotokollan toiminnasta, että myös apuna kehityksessä ja testauksessa. Pelkästään paperilla olevaa spesifikaatiota on hankala lukea ja verrata sovelluksen tai koko järjestelmän toimintaan. Suoritettavassa tilakoneessa taas tietokone seuraa testattavan järjestelmän suoritusta. Tämän tulisi havaita määrittelyn vastaiset viestit ja ilmoittaa niistä

jollain tavalla.

Suoritettava tilakone toimii spesifikaationa sekä järjestelmän yksittäisten sovellusten kehityksessä, että myös testauksessa. Lisäksi se auttaa sovellus- ja integrointitestauksessa, sillä sen avulla voidaan tarkemmin paikantaa määrittelyn vastaiset toiminnot virheellisen viestin perusteella. Tällöin se, että toimiiko järjestelmä määrittelyn mukaisesti, ei jää ihmisen tulkittavaksi.

Tärkeä tavoite testauksessa on löytää virheiden syyt. Väärin toimivasta järjestelmästä on vaikea löytää alkuperäistä syytä pelkän määrittelyn ja omien tulkintojen avulla. Tähän tarkoitukseen edellä esitetty suoritettava tilakone tarjoaa välineet, sillä sen avulla saadaan selville määritelmän vastaiset viestit, mikä helpottaa varsinaisen vian etsimistä.

4. KOHDEJÄRJESTELMÄ

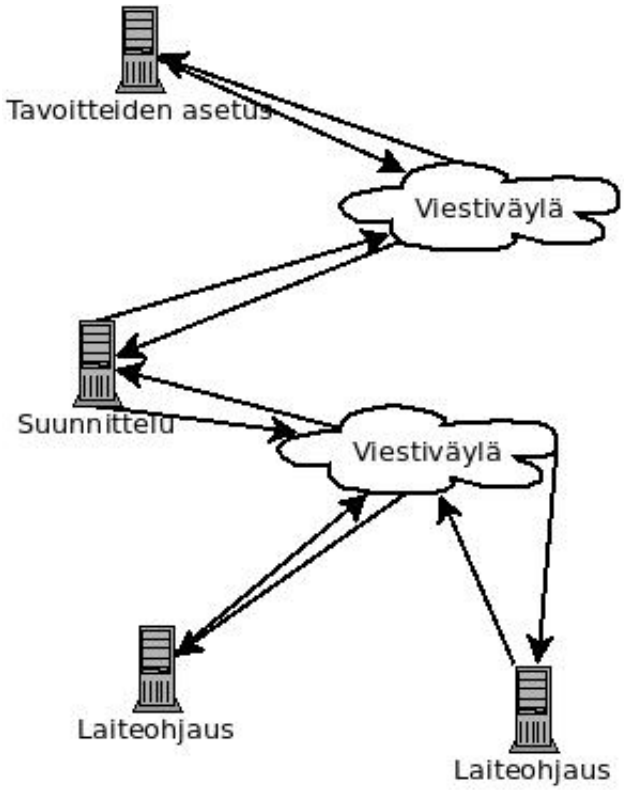
Tässä luvussa esitellään eräs hajautettu järjestelmä ja tämän järjestelmän testaamiseen sekä kehittämiseen liittyneitä ongelmia. Tässä työssä toteutettu menetelmä järjestelmän kommunikaatioprotokollan määrittelyyn ja määritelmänmukaisuuden tarkastamiseen on tehty näiden ongelmien pohjalta.

4.1. Yleiskuvaus

Kohdejärjestelmä ohjaa laiteryhmää aina korkean tason tavoitteiden asetuksesta matalan tason laiteohjaukseen. Järjestelmä on esitelty kuvassa 4.1. Se koostuu kolmesta komponentista. Alimmalla tasolla ovat mekaanisia laitteita ohjaavat sovellukset. Näille sovelluksille annetaan suoria käskyjä siitä mitä milläkin laitteella tulee tehdä ja ne raportoivat ylöspäin tietoja töiden sekä laitteiden tilasta. Näitä sovelluksia voi olla järjestelmässä useita erityyppisille laitteille. Keskimmäisellä tasolla oleva sovellus antaa työkäskyjä alemmille sovelluksille. Se suunnittelee töiden suoritusjärjestykset sekä laitevalinnat. Tämä sovellus saa korkeammalla tasolla olevalta sovellukselta töiden tavoitteita, joita se jalostaa työkäskyiksi laitteita ohjaaville sovelluksille. Korkeimmalla tasolla olevan sovelluksen tehtävä on tehdä päätöksiä halutusta lopputuloksesta asiakkailta saatujen tilausten mukaan ja antaa näitä vaatimuksia töiden suunnittelijalle.

4.2. Viestiväylä

Kaikki sovellukset kommunikoivat viestiväylän kautta. Jokaisella sovellustasolla on oma rajapintansa, XML-skeema, jolla määritellään viestit, jotka sovellukset hyväksyvät. Skeemat ovat osittain päällekkäisiä, mutta niissä on joitakin eroja, joten viestiväylän tehtävänä on tarvittaessa muuntaa viesti skeemasta toiseen. Viestiväylä myös tietää lähettäjän sekä viestin tyypin tai sisällön perusteella mille järjestelmille viesti tulee ohjata. Väylän tarkoituksena on toimia myös integrointialustana, jolloin ohjelmistoja voidaan korvata toisilla



Kuva 4.1: Kohdejärjestelmän kuvaus.

ilman, että muita osia tarvitsee muokata.

4.3. Järjestelmän nykytila

Järjestelmä on tälläkin hetkellä kehityksen alla, ja sovellusten rajapinnat muuttuvat jatkuvasti. Viestit muuttuvat useasta eri syystä, jotka ovat tuttuja ohjelmistokehityksessä. Uusien ominaisuuksien tai vastuujakojen muutosten myötä viestejä lisätään tai poistetaan. Usein suunnitteluratkaisut eivät myöskään toimi oletetulla tavalla. Jonkin huomioimatta jääneen asian takia jokin toiminto joudutaan tekemään uudella tavalla, mikä tarkoittaa muutoksia protokollaan. Joskus myös järjestelmän käyttö oikeassa ympäristössä, oikeiden mekaanisten laitteiden kanssa tuottaa uusia ongelmia, jotka saattavat heijastua koko järjestelmään, eli myös protokollaan. Muutokset ovat normaaleja sovelluksen elinkaaren aikana, mutta hajautetussa järjestelmässä muutokset heijastuvat useampaan sovellukseen.

Korkealla tasolla oleva tavoitteiden asettaja voi olla eri ympäristöissä kokonaan eri sovellus, joka integroidaan osaksi järjestelmää. Vaikka rajapinta onkin valmiina ja integraatioalustan avulla selvitään joistakin eroista, voi kokonaan uusi sovellus tuoda kuitenkin uusia vaatimuksia, joita integraatioalusta ei ratkaise. Uusi sovellus saattaa toteuttaa tietyt

asiat hieman eri lailla tai tehdä erilaisia olettamuksia ympäristöstänsä, joten jopa rajapintaa saatetaan joutua muuttamaan integrointia varten.

Alimmalla tasolla olevat laitteita ohjaavat sovellukset olivat olemassa ennen kuin sovelluksia alettiin integroimaan yhdeksi hajautetuksi järjestelmäksi. Keskellä olevan töiden suunnittelijan tekeminen aloitettiin samaan aikaan integrointityön aloituksen kanssa ja sen suunnittelussa jouduttiin sopeutumaan joiltain osin muiden sovellusten ominaisuuksiin.

4.4. Määrittely

Töiden suunnittelijan tekeminen on aloitettu siten, että laitteita ohjaavat järjestelmät ovat olleet olemassa ja XML-skeemasta on ollut alustava versio. Tätä XML-skeemaa on päivitetty kun sille on nähty tarvetta, eli uusien ominaisuuksien, muuttuneiden vaatimusten tai skeeman toimimattomuuden takia. Nämä muutokset on tehty yleensä juuri tiettyä tarvetta varten. Lähtökohtana on ollut juuri kyseisen ongelman ratkaiseminen kokonaisvaltaisen spesifioinnin sijaan. Muutenkin projektilta on puuttunut henkilö, jonka vastuulla olisi ollut koko järjestelmän integraatio. XML-skeeman lisäksi on tehty sanallista dokumentaatiota tai kuvaajia toiminnoista, jotka ovat olleet erityisen monimutkaisia tai ongelmallisia.

Skeeman muuttuessa se katselmoidaan läpi eri sovellusten kehittäjillä puutteiden ja virheiden havaitsemiseksi. Joskus toisen sovelluksen kehittäjien muutosehdotukset saattavat rikkoa toiminnallisuuden toisen sovelluksen kehittäjien näkökulmasta. Skeemaa käydään lävitse näin iteroiden kunnes siitä ei enää löydy virheitä ja se ollaan valmiita implementoimaan sovelluksiin. Yleensä tästä huolimatta implementointivaiheessa havaitaan skeemasta puutteita, joiden takia sillä ei voida toteuttaa toimintoja, joihin se on tarkoitettu. Toisinaan nämä puutteet korjataan liian nopeasti ja ajattelematta sen tarkemmin, mikä saattaa tuottaa lisää puutteita skeemaan.

4.5. Testauksessa ilmenneitä ongelmia

Järjestelmän sovellusten jokainen julkaistu versio toimii hieman eri tavalla, eivätkä kaikki versiot toimi yhteen muuttuneista rajapinnoista sekä käytännöistä johtuen. Uusia ominaisuuksia on tarkasteltu usein vain kyseisen sovelluksen näkökulmasta, minkä lisäksi niiden tulkinnoissa on ollut muutenkin eroavaisuuksia.

Järjestelmän testaajat eivät ole samoja kuin sovellusten kehittäjät, joten hekin tekevät

testaamisen aikana omat tulkintansa toiminnallisuuksista, joista ei ole tarkkaa dokumentaatiota. Koska tarkkaa määrittelyä ei ole, osa testaajien raportoimista virheistä on toisinaan sovelluksen normaalia toimintaa. Testaajilta on myös saattanut jäädä raportoimatta virheitä, jotka eivät aiheuta mitään näkyvää häiriötä tai joista järjestelmä toipuu itsestään. Toisinaan virheraportteja kirjoitetaan sovellukselle, jossa häiriö näkyy virheellisesti käyttäytyvän sovelluksen sijaan. Koska eri sovelluksia kehittävät eri yritykset eivätkä testaajat ole mistään näistä, häiriöiden tarkempi selvittäminen on työlästä ja hidasta.

Järjestelmätestauksen aikana kaikki sovellukset suorittavat omia toimintojansa. Virheistä ja ajoituksista riippuen järjestelmä voi reagoida virheeseen usealla tavalla. Virhe saattaa olla pieni tai useampi virhe kumoaa toisensa, jolloin se ei näy ulospäin minkäänlaisena häiriönä. Yksinkertaisimmassa tapauksssa virhe näkyy selkeästi virheellisesti toimivassa sovelluksessa, eikä heijastu muihin sovelluksiin. Usein kuitenkin käy niin, että sovelluksen virhe näkyy kommunikaatiossa. Joko sovellus lähettää väärän viestin, oikean viestin väärään aikaan tai jättää lähettämättä viestin. Tällöin sovellus, jossa virhe on, näyttää toimivan oikein, mutta se saa muut sovellukset toimimaan testaajan kannalta virheellisesti, vaikka ne tekevätkin oikeita asioita nille annetun tiedon perusteella. Häiriö näkyykin virheellisesti toimivan sovelluksen sijaan muissa sovelluksissa. Tämän tyyppiset virheet ovat todella hankala jäljittää. Jäljittäminen tapahtuu kuitenkin aina samalla tavalla; etsitään sovellusten logeista kohta, missä virhe ilmeni ja katsotaan mitä viestejä järjestelmässä on liikkunut ennen sitä. Näin saadaan paikannettua virheen aiheuttanut viesti ja sovellus, jolloin vian etsintä voidaan kohdistaa oikeaan paikkaan.

Eri sovellukset käyttävät eri skeemoja, joiden välisistä konversioista viestiväylä huolehtii. Toisinaan viestien konvertoinnissa skeemasta toiseen tulee viestiväylästä johtuvia virheitä. Esimerkiksi osa viestin kentistä saattaa jäädä täyttämättä tai ne konvertoidaan väärin. Tällöin virheellisesti konvertoidun viestin vastaanottaja saattaa toimia toisin kuin oletettiin. Viestiväylässäkin voi olla virheitä siinä missä järjestelmän muissakin sovelluksissa.

Viestiväylän virheiden paikantaminen on hiukan työläämpää kuin muiden sovellusten virheiden paikantaminen, sillä virheen etsiminen alkaa sovelluksesta, jossa häiriö ilmeni ja jatkuu siitä sovellukseen, joka lähetti virheellisen toiminnan aiheuttaneen viestin. Vasta tämän jälkeen huomataan, että viestin konvertoinnissa on virhe.

4.6. Määrittelynmukaisuuden toteaminen manuaalisesti

Yksinkertaisimmissakin kehityksen aikaisissa testeissä saattaa olla noin kymmenkunta laitetta suorittamassa töitä. Yhden laitteen suorittama työ aiheuttaa noin yhden XML-viestin viestiväylään kymmenessä sekunnissa. Mikäli testaaja voisi todeta varmasti, tekemättä virheellisiä tulkintoja, järjestelmän kommunikaation määrittelynmukaisuuden viestejä seuraamalla, hänen tulisi voida lukea ja tulkita noin yksi viesti sekunnissa. Pienessä tuotantoympäristössä laitteita voi olla 20-30 ja suuremmassa jopa 150. Viestiväylään kirjoitetaan niin tiheästi viestejä, että sen seuraaminen manuaalisesti on mahdotonta. Tämän takia järjestelmän määrittelynmukaisuus on välttämätöntä testata automaattisesti.

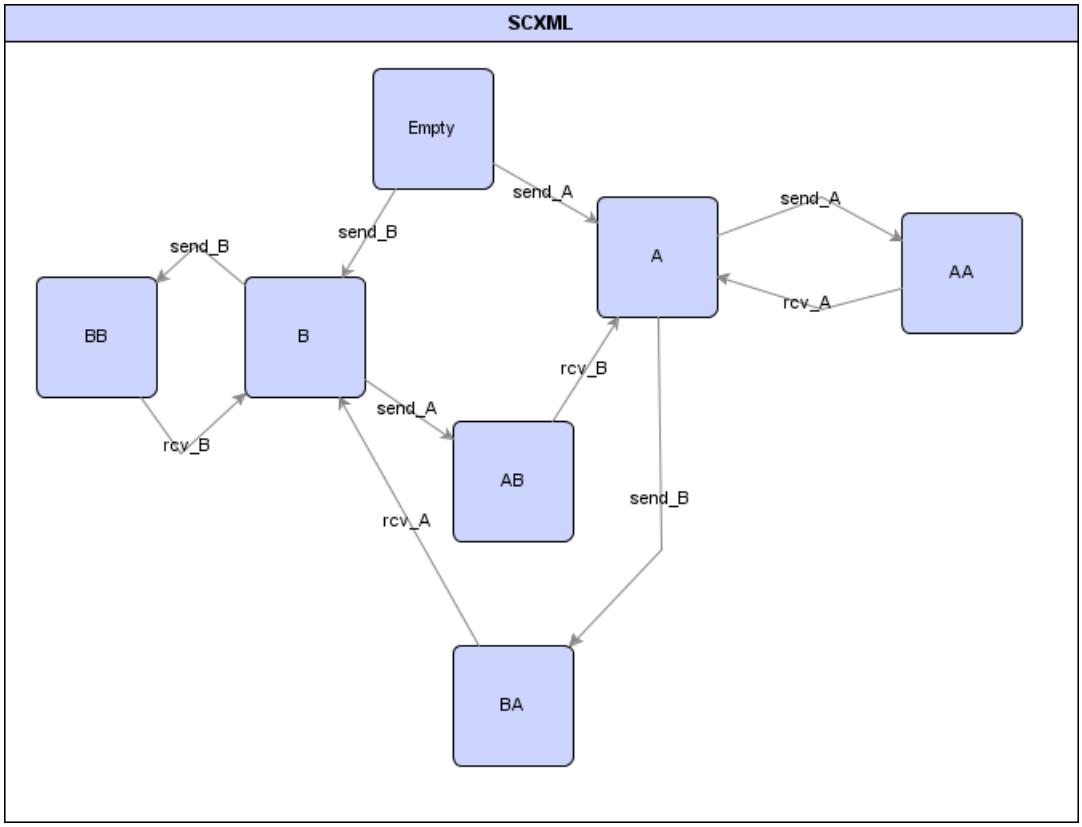
5. TESTAUSSOVELLUKSEN TOTEUTUS

Tässä luvussa käsitellään suoritettavien tilakoneiden käyttöä käytännössä. Tässä esitellään protokollan määrittely tilakoneiden avulla ja kuvaus sovelluksesta, joka lukee näitä tilakoneita ja tarkastaa kommunikaatiota näitä määrittelyjä vasten. Ensin käsitellään yleisellä tasolla testaussovellusta ja tämän jälkeen tilakoneiden määrittelyä tarkemmin. Lopuksi esitellään esimerkinomaisesti eräs tilakoneen määrittely ja käydään testaussovelluksen toimintaperiaatetta tarkemmin läpi tämän esimerkin avulla.

5.1. Hajautetun järjestelmän kuvaaminen tilakoneena

Kohdejärjestelmä koostuu useasta sovelluksesta, jotka kommunikoivat keskenään viestiväylän avulla. Järjestelmän yksittäiset sovellukset voidaan mallintaa epädeterminististen tilakoneiden avulla. Epädeterministisyys johtuu sovelluksen sisäisistä päätöksistä, joita ei voida havaita ulkoa käsin. Yksinkertaisena esimerkkinä voi toimia esimerkiksi tekstinkäsittelyohjelma. Kun tekstinkäsittelyohjelman käskee avata tiedoston, ei voida etukäteen tietää onnistuuko tiedoston avaaminen, onko tiedostoa olemassa tai onko siihen lukuoikeuksia. Lisäksi tiedostojärjestelmä voi olla rikki tai tiedosto voi sijaita verkkolevyllä, johon verkkoyhteys ei toimi. Sovelluksen monimutkaisuuden kasvaessa mahdollisten tilojen sekä epädeterministisyyden määrä kasvaa.

Myös viestiväylä voidaan ajatella tilakoneena. Yksinkertaistuksen vuoksi kuvassa 5.1 on kuvattu viestiväylä, johon voi kirjoittaa viestit A ja B. Viestiväylä toimii FIFO-periaatteella. Kun viestiväylään kirjoitetaan viesti, seuraavaksi se voi lähettää saman viestin eteenpäin tai siihen voidaan lähettää toinen viesti. Jos se vastaanottaa toisen viestin, se voi seuraavaksi lähettää ensimmäisen väylään lähetetyn viestin eteenpäin. Yksinkertaistuksen vuoksi kuvan viestiväylään mahtuu vain kaksi viestiä kerrallaan. Tilakoneen siirtymät ovat send_A ja send_B, jotka tarkoittavat sitä, että viestiväylään kirjoitetaan viestit A tai B, sekä rcv_A ja rcv_B, jotka tarkoittavat sitä, että viestiväylä lähettää kyseiset vies-



Kuva 5.1: Viestiväylän mallinnus tilakoneen avulla.

tit eteenpäin. Tilat kertovat mitä viestejä viestiväylä pitää sisällään. Esimerkiksi tila AB on tila, jossa viestiväylään on ensin tullut viesti B ja sitten A. Koska väylään mahtuu vain kaksi viestiä kerrallaan, tästä tilasta pääsee pois vain lähettämällä viestin B eteenpäin. Todelliseen viestiväylään voi lähettää useamman viestin ja mahdollisia syötteitä voi olla sekä kaikki mahdolliset järjestelmän tuntemat viestit, että myös sellaiset, joita järjestelmä ei tunne. Lisäksi viestiväylä voi tehdä muitakin operaatioita viesteille sekä sisältää muuta toiminnallisuutta.

Kaikkia järjestelmän osia voidaan ajatella tilakoneina, joilla on liityntöjä toisiinsa silloin kun ne kommunikoivat toistensa kanssa. Nämä rinnakkaiset tilakoneet voidaan yhdistää yhdeksi, jolloin koko hajautettua järjestelmää voidaan tarkastaa yhtenä todella isona epädeterministisenä tilakoneena. Tilojen käytännössä valtavan määrän takia koko järjestelmää ei ole kuitenkaan mielekästä tarkastella yksityiskohtaisena tilakoneena, vaan sopivasti abstrahoituna. Tässä työssä keskitytään yksittäisten sovellusten sijasta sovellusten väliseen kommunikaatioon, joten koko järjestelmän tilakone abstrahoidaan kuvaamaan vain sovellusten välistä viestiliikennettä. Tällöin tilasiirtymät kuvaavat yksittäisiä viestejä ja kahden tilasiirtymän välillä kukin järjestelmän osa voi toimia itsenäisesti siten, että se

ei näy kommunikaatiossa.

Koko järjestelmän kommunikaatiota ei mallinneta yksittäisenä tilakoneena vaan useampana. Esimerkiksi edellisessä luvussa kuvatussa järjestelmässä on resursseja, joita laitteet voivat käsitellä. Laitteita ohjaavat sovellukset ilmoittavat resurssin varauksesta ja vapautuksesta. Yksi laite voi käsitellä vain yhtä resurssia kerrallaan ja yksi resurssi voi olla vain yhden laitteen käsiteltävissä. Näistä rajoitteista voidaan esimerkiksi tehdä kaksi tilakonetta, joista yksi seuraa yksittäistä resurssia. Resurssi pitäisi varauksen jälkeen vapauttaa ennen kuin sen voisi varata uudestaan. Toisen tilakoneen voisi tehdä seuraamaan yksittäistä laiteitta. Tämä tilakone määrittelisi, että laite vapauttaa varaamansa resurssin ennen uuden resurssin varausta. Mikäli koko kommunikaatio esitettäsiin yhtenä tilakoneena, kasvaisi tilojen määrä erittäin suureksi, sillä tilakoneen tulisi kuvata kaikki mahdolliset ja mahdottomat keskustelut eri tavoilla lomitettuna.

5.2. Testaussovelluksen yleiskuvaus

Tätä työtä varten toteutettiin sovellus, joka lukee jatkuvasti viestiväylästä kahden järjestelmän välistä keskustelua ja tarkistaa sen laillisuuden määritellyn tilakoneen avulla. Sovellus ei itse puutu keskusteluun mitenkään, vaan ainoastaan ilmoittaa virheellisistä viesteistä. Sovellusta voidaan käyttää eri XML-skeemoilla keskustelevien järjestelmien keskustelujen verifioimiseen, sillä se ei ota kantaa järjestelmiin eikä skeemaan. Tilakoneen tulee ainoastaan sopia keskustelun verifioimiseen ja tilakoneita voidaan lisätä, poistaa sekä muokata sovelluksen muuttumatta.

5.3. Tilakoneiden määrittely

Sovelluksen lukemat tilakoneet määritellään W3C:n kehittämällä SCXML-merkkauskielellä. SCXML:llä voidaan määritellä XML-muodossa Statechart:eja, joita voidaan myös suorittaa tietokoneella. SCXML on toistaiseksi työn alla oleva luonnos, mutta sille on olemassa joitakin graafisia editoreja sekä sovelluskehyksiä, joiden avulla voidaan tehdä sovellus SCXML:ää apuna käyttäen. [16]

SCXML-dokumentti koostuu tiloista, siirtymistä näiden välillä sekä tietomallista ja suoritettavasta sisällöstä. Pieni esimerkki merkkauskielestä esitellään listauksessa 5.1. Kuvassa 5.2 esitetään sama tilakone graafisessa muodossa. Tilakoneessa tilat C ja D on

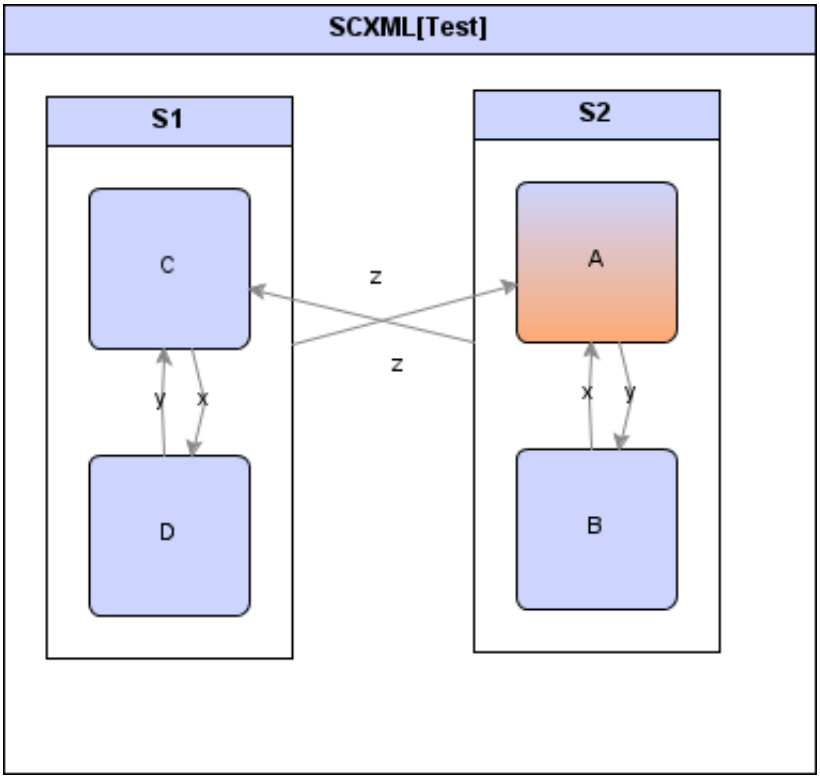
ryhmitelty ylitilan S1 alle ja tilat A ja B ylitilan S2 alle. Tapahtumalla z siirrytään tilasta S1, eli sekä tilasta C että D, tilaan A ja tilasta S2 tilaan C.

```
<scxml name="Test" version="0.9" xmlns="http://www.w3.org/2005/07/scxml
    ">
 <state id="S2" initial="A">
  <transition event="z" target="C"></transition>
  <state id="A">
   <transition event="y" target="B"></transition>
  </state>
  <state id="B">
   <transition event="x" target="A"></transition>
  </state>
 </state>
 <state id="S1">
  <transition event="z" target="A"></transition>
  <state id="C">
   <transition event="x" target="D"></transition>
  </state>
  <state id="D">
   <transition event="y" target="C"></transition>
  </state>
 </state>
</scxml>
```

Listaus 5.1: Esimerkki SCXML-merkkauskielestä.

Tässä työssä toteutetun sovelluksen SCXML:ää lukeva komponentti osaa lukea vain SCXML:n tässä tarvittavaa osajoukkoa. Tukea esimerkiksi rinnakkaisille tilakoneille ei ole. Sen sijaan tietomalli ja suoritettava sisältö ovat tärkeässä osassa. Tilakone poikkeaa alakohdassa 2.1.1. esitetystä DFA:sta siten, että tilakoneen syötteet eivät ole yksittäisiä merkkejä tai yksinkertaisia tapahtumia, vaan monimutkaisempia, paljon tietoa sisältäviä viestejä, jolloin sekä viestin tyyppi että sisältö toimii siirtymän ehtona.

Tietomalli on myös tärkeässä osassa toteutetussa sovelluksessa, sillä sovellus kuuntelee viestiliikennettä, jossa on kerrallaan useampi keskustelu kesken. Koska tilakone raportoi virheelliset tilasiirtymät, tulee siihen määritellä viestit, jotka vaikuttavat tilakoneeseen. Lisäksi samasta tilakoneesta voi olla olemassa kerrallaan useita *ilmentymiä* kuvaamassa samanlaisten, mutta eri keskustelujen tiloja. Tästä syystä tilakoneille on välttämätöntä määritellä myös tapa, joilla viestit identifioidaan tiettyyn ilmentymään liittyviksi. Tässä työssä tavaksi identifioida viestit on valittu XPATH (XML Path Language) [8]. Kaikille viesteille ei myöskään ole kaikissa tiloissa tilasiirtymää, vaan ainoastaan sillä hetkellä sallituille viesteille. Tilakoneen avulla huomataan tilasiirtymän puutteesta viestit, jotka



Kuva 5.2: Kuvakaappaus scxmlgui-editorista.

eivät ole sallittuja kyseisessä tilassa. Tilakoneen lopputila tarkoittaa sitä, että kyseisen ilmentymän kommunikaatio on päättynyt. Jos lopputilassa olevaan tilakoneeseen tulee uusi viesti, on tämä määritelmän vastainen ja raportoidaan virheenä.

Suoritettava sisältö tilakoneiden määrittelyssä on välttämätöntä, sillä pelkkä viestin tyyppi ei aina riitä siirtymän ehdoksi, vaan viestin sisällöstä saatetaan haluta lukea muitakin ehtoja. Viestistä voidaan tallentaa muuttujia, joihin voidaan viitata siirtymän ehdoissa. Muuttujat määritellään XPATH:n avulla. Lisäksi muuttujat säilyvät muistissa myöhempää käyttöä varten. Esimerkiksi jos saman keskustelun viestit on tarkoitus numeroida juoksevalla numerolla, viestistä tallennetaan muuttujaan kyseisen viestin luvun arvo viestiä käsitellessä. Tilasiirtymän ehtona voidaan pitää sitä, että tämä muuttuja on yhden suurempi kuin ilmentymälle tallennettu viimeisin arvo. Onnistuneen siirtymän myötä viestistä luettu arvo tallennetaan viimeisimmäksi arvoksi, jotta seuraavaa viestiä voidaan taas verrata tähän. Luvun ollessa jotain muuta siirtymän ehto ei täyty, jolloin tämä viesti raportoidaan virheellisenä ja tilakone pysyy samassa tilassa.

SCXML:n tuottamiseen ei ole tätä kirjoitettaessa kovin hyviä työkaluja. Jos tietomalli ei olisi niin tärkeässä osassa, SCXML:ää voisi tuottaa siihen tarkoitettujen editoreiden li-

säksi myös konvertoimalla esimerkiksi UML-editorilla tehdystä tilakaaviosta. Tätä työtä tehdessä tutustuttiin muutamiin SCXML-editoreihin, mutta ainoa näistä, jolla pystyi tuottamaan tarpeeksi hyvän lopputuloksen, oli scxmlgui [17]. Kuvassa 5.2 esitetty tilakone on tehty tällä sovelluksella. Muissa oli suurempia tai pienempiä puutteita, joista suurimpana oli tietomallin määrittely, joka on tässä työssä tärkeä. Editorin helppokäyttöisyys ei kuitenkaan ole tärkeä ominaisuus työn kannalta, sillä scxmlgui:lla pystyi tekemään kaiken mitä tarvitsee. Jos SCXML-määrittely saavuttaa standardin aseman, on mahdollista, että editorit kehittyvät tulevaisuudessa paremmiksi.

5.4. Skriptaus

Tilasiirtymien ehtoja, muuttujien tallenusta sekä muuta logiikkaa varten tilakoneet tarvitsevat skriptausympäristön. Tämän ympäristön on oltava kevyt, mutta kuitenkin tarpeeksi kattava. Sen pitäisi osata käsitellä yksinkertaisia aritmeettisia sekä loogisia operaatioita sekä tietotyyppejä, kuten kokonaislukuja, liukulukuja, totuusarvoja sekä merkkijonoja.

Aluksi skriptien tulkitsemiseen kokeiltiin Googlen V8 -javascript-moottoria, mutta sen avulla pystyi luomaan vain suhteellisen pienen määrän tilakoneilmentymiä. Jokaista ilmentymää varten luotiin oma javascript-konteksti, jotta niillä olisi oma suoritusympäristö. Ilmeisesti jokaista kontekstia varten varattiin iso muistialue, koska ilmentymien lukumäärän kasvaessa muisti loppui kesken. Tämän takia lähdettiin etsimään moottoreita, jotka on tarkoitettu vain lausekkeiden evaluoimiseen yleiskäyttöisen ohjelmoinnin sijasta.

Lopulta tässä päädyttiin kirjastoon nimeltä FLEE (Fast Lightweight Expression Evaluator). Vaikka sen kehitys onkin ilmeisesti lopetettu jo vuonna 2009, siinä on kaikki tarvittavat ominaisuudet ja lisäksi se on tehokas, eikä käytä muistia liikaa. Lisäksi sen LGPL-lisenssi sallii käytön kaupallisissakin sovelluksissa [18].

5.5. Esimerkkitapaus tilakoneen määrittelystä

Listauksessa 5.2 esitellään tilakone, jota toteutettu sovellus voisi käyttää. Seassa olevat kommentit kertovat scxmlgui-editorille mihin ja millä lailla tilakoneen solmut ja kaaret piirretään. Rivillä 1 määritellään tilakoneen nimi sekä alkutila.

Riveillä 2-13 määritellään tilakoneen tunnistamat viestit. Näitä ovat *StartMessage*, *EndMessage* ja *Event*. Kustakin viestistä identifioidaan ilmentymä XPATH:lla *//ID*.

```
1  <scxml initial="Init" name="Example" version="0.9" xmlns="http://www.w3
       .org/2005/07/scxml"><!--  node-size-and-position x=0 y=0 w=250 h
       =660  -->
2   <datamodel>
3    <data id="events">
4     <events>
5      <event id="//ID" type="StartMessage"></event>
6      <event id="//ID" type="EndMessage"></event>
7      <event id="//ID" type="Event">
8       <variable id="status" value="string //Status"></variable>
9       <variable id="counter" value="int //Counter"></variable>
10     </event>
11    </events>
12   </data>
13  </datamodel>
14 <!-- node-size-and-position x=0 y=0 w=155,61 h=638 -->
15  <state id="Init"><!--  node-size-and-position x=22,11 y=43 w=75 h=75
       --><!-- node-size-and-position x=22,11 y=43 w=75 h=75  node-size-
       and-position x=22,11 y=43 w=75 h=75  node-size-and-position x
       =22,11 y=43 w=75 h=75 -->
16   <transition event="StartMessage" target="Starting"></transition>
17  </state>
18  <state id="Starting"><!--  node-size-and-position x=22,11 y=168 w=75
       h=75  --><!-- node-size-and-position x=22,11 y=168 w=75 h=75  node
       -size-and-position x=22,11 y=168 w=75 h=75  node-size-and-position
        x=22,11 y=168 w=75 h=75  node-size-and-position x=249,73 y=230 w
       =75 h=75 -->
19   <transition cond="status = &quot;Started&quot;" event="Event" target=
       "Running">currentCounter = counter</transition>
20  </state>
21  <state id="Stopping"><!--  node-size-and-position x=22,11 y=418 w=75
       h=75  --><!-- node-size-and-position x=22,11 y=418 w=75 h=75  node
       -size-and-position x=22,11 y=418 w=75 h=75  node-size-and-position
        x=22,11 y=418 w=75 h=75 -->
22   <transition cond="status = &quot;Stopped&quot;" event="Event" target=
       "Stopped"></transition>
23  </state>
24  <final id="Stopped"><!--  node-size-and-position x=22,11 y=543 w=75 h
       =75  --></final>
25 <!-- node-size-and-position x=22,11 y=543 w=75 h=75  node-size-and-
       position x=22,11 y=543 w=75 h=75  node-size-and-position x=22,11 y
       =543 w=75 h=75 -->
26  <state id="Running"><!--  node-size-and-position x=22,11 y=293 w=75 h
       =75  --><!-- node-size-and-position x=22,11 y=293 w=75 h=75  node-
       size-and-position x=22,11 y=293 w=75 h=75  node-size-and-position
       x=22,11 y=293 w=75 h=75 -->
27   <transition event="EndMessage" target="Stopping"></transition>
28   <transition cond="counter &gt; currentCounter AND status = &quot;
       Running&quot;" event="Event">currentCounter = counter</transition>
29  </state>
30 </scxml>
```

Listaus 5.2: Esimerkki SCXML:llä määritellystä tilakoneesta, jossa käytetään datamodelia ja FLEE:tä.

Viesti *Event* myös tallentaa kaksi muuttujaa ilmentymälle. Muuttujaan *status* tallennetaan XPATH:lla *//Status* saatava merkkijono ja muuttujaan *counter* XPATH:lla "//Counter"saatava kokonaisluku.

Riveillä 15-20 määritellään tilat *Init* ja *Starting*. Tilasta *Init* voidaan siirtyä tilaan *Starting* viestillä *StartMessage*. Tilasta *Starting* taas viestillä *Event* tilaan *Running*, mikäli siirtymän ehto *status* = *Started* täyttyy. Onnistuneen tilasiirtymän myötä tilasta *Starting* tilaan *Running* suoritetaan rivillä 19 oleva lauseke *currentCounter* = *counter*. Tämä asettaa tilakoneilmentymän sisäisen currentCounter-muuttujan arvoksi viestin luvun yhteydessä tallennetun counter-muuttujan arvon. Riveillä 21-24 määritellään tilat *Stopping* ja *Stopped*, joista jälkimmäinen on lopputila.

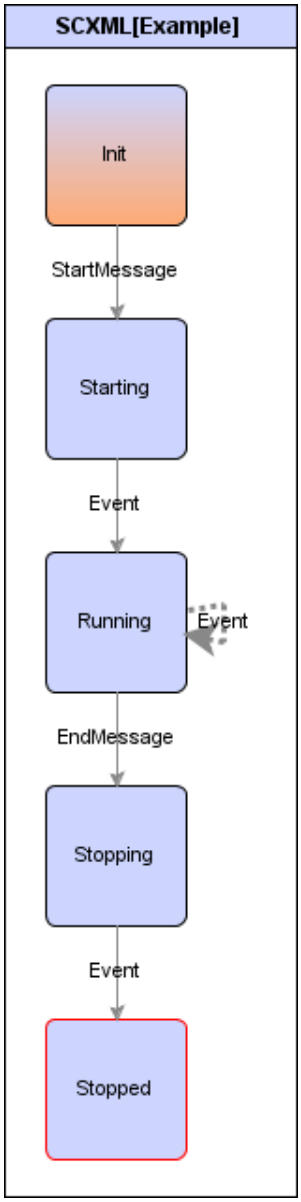
Riveillä 26-29 määritellään tila *Running*. Tässä tilassa on siirtymä itseensä viestillä *Event* siten, että muuttujan *status* arvo on *Running* ja *counter* on suurempaa kuin *currentCounter*. Onnistuneen tilasiirtymän myötä muuttujan *currentCounter* arvoksi asetetaan muuttujan *counter* arvo. Käytännössä siis tilakoneilmentymä muistaa viimeisimmän viestin *counter*-muuttujan arvon ja tarkistaa, että tämä kasvaa joka viestissä.

Kuvassa 5.3 esitellään yllä kuvattu tilakone scxmlgui-editorissa. Kuvakaappauksesta ei ole havaittavissa siirtymien ehtoja eikä skriptisisältöä, sillä ne ovat erillisen valikon takana. Kuvassa 5.4 on kuvattuna kaaren muokkausnäkymä, josta voi muokata tilasiirtymän tapahtumaa ja sen ehtoa sekä sisältöä, joka suoritetaan siirtymän jälkeen. Myös tiloille on samanlainen erillinen muokkausvalikko.

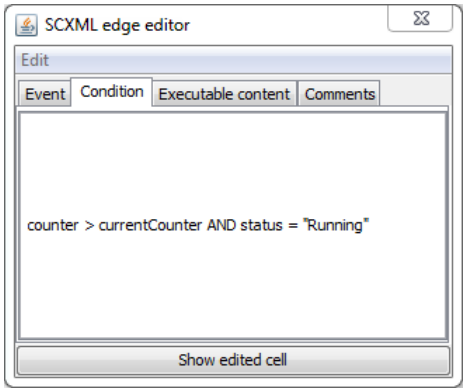
5.6. Testaussovelluksen toteutus

Sovellus alustetaan lukemalla sisään tilakoneiden määrittelyt sekä kytkeytymällä viestiväylään. Tämän jälkeen sovellus reagoi viestiväylässä kulkeviin viesteihin siirtämällä tilakoneilmentymiänsä uusiin tiloihin tai raportoimalla virheellisistä tilasiirtymistä.

Viestiväylästä luetuista viesteistä tulkitaan viestin tyypin perusteella mihin tilakoneisiin viesti vaikuttaa ja annetaan viesti tulkittavaksi näille tilakoneille. Kukin tilakone laskee erikseen omasta määrittelystänsä, kuinka kyseisestä viestistä erotetaan mille tilakoneilmentymälle viesti kuuluu. Tämän jälkeen katsotaan onko kyseisestä ilmentymää tilakoneesta vielä olemassa. Mikäli ei ole, sellainen luodaan ja alustetaan alkutilaan. Viesti annetaan tilakoneilmentymän tulkittavaksi, jolloin se katsoo, onko kyseinen siirtymä lail-



Kuva 5.3: Kuvakaappaus scxmlgui-editorista.



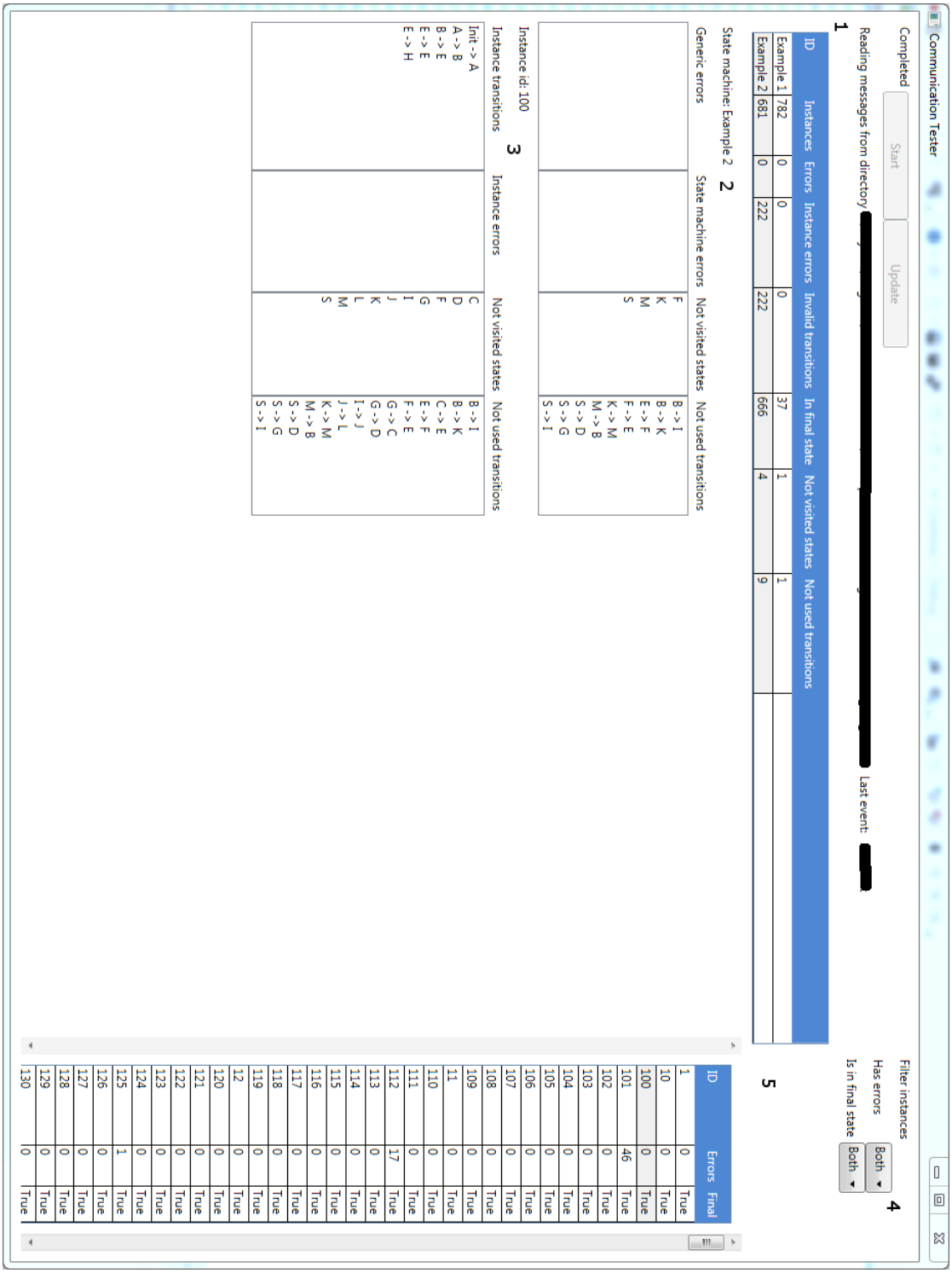
Kuva 5.4: Kuvakaappaus scxmlgui-editorin kaaren muokkausvalikosta.

linen nykyisessä tilassa. Viesteille voidaan määritellä, mitä muuttujia niistä tallennetaan ja siirtymille voidaan antaa ehtoja, jotka tarkastelevat näitä muuttujia. Tilasiirtymä voi siis olla virheellinen paitsi siksi, että annettu viesti ei ole laillinen siirtymä nykyisestä tilasta, että myös sen takia, että siirtymiseen vaaditut ehdot eivät täyttyneet.

Otetaan esimerkiksi edellisessä kohdassa määritelty tilakone. Tämä tilakone tunnistaa viestit *StartMessage*, *EndMessage* ja *Event*. Jos viestiväylästä luetaan esimerkiksi viesti *UnknownMessage*, ei tämä viesti aiheuta mitään muutosta tilakoneen tilaan, joten sovellus ei anna viestiä tilakoneen käsiteltäväksi. Kun viestiväylästä luetaan jokin sen tunnistamista viesteistä, katsotaan kunkin tilakonemäärittelyn säännöstä, miten kyseisestä viestistä luetaan id, tässä tapauksessa XPATH:lla *//ID*. Kun id on luettu, tällä id:llä etsitään olemassaoleva tilakoneilmentymä tai alustetaan uusi alkutilaan *Init*. Koska viesti vaikuttaa kyseiseen tilakoneeseen, on viestistä seurauksena joko virhe tai laillinen tilasiirtymä. Mikäli kyseisestä tilasta löytyy viestillä ehdot täyttävä tilasiirtymä, siirretään kyseinen ilmentymä uuteen tilaan. Muutoin raportoidaan tilasiirtymävirhe. Virheen tapahtuessa ilmentymän tila ei muutu, vaan se jää nykyiseen tilaansa.

Viestin vastaanottamisen ja tulkitsemisen jälkeen sovellus raportoi kuinka viesti vaikutti tilakoneisiin. Tuloksena voi olla useampi laillinen tilasiirtymä (yksi tilakonemäärittelyä kohden) sekä useampi virheellinen tilasiirtymä tai jokin muu virhe. Muut virheet voivat johtua tilakoneiden määrityksessä olleista virheistä, joita sovellus ei ole huomannut määrittelyä luettaessa tai viestistä johtuvasta virheestä. Esimerkiksi skriptin suorituksessa voi olla ajonaikainen virhe tai syntaksivirhe tai tilakoneessa on siirtymä tilaan, jota ei ole määritelty tilakoneessa. Viestistä johtuva virhe voi olla esimerkiksi se, että XPATH-säännöllä löytyy useampi id, jolloin viestiä ei voida liittää mihinkään yksittäiseen keskusteluun.

Sovelluksen käyttöliittymä on esitetty kuvassa 5.5. Sen kautta näkee yleiskuvan suorituksesta; mitä tilakonemäärittelyjä sovelluksessa on sekä kuinka monta ilmentymää ja virhettä kussakin tilakoneessa on. Yksittäisen tilakoneen valitsemalla näkee kaikki tämän tilakoneen ilmentymät ja tilakoneen suorituksessa tapahtuneet virheet sekä tietoa käymättömistä tiloista sekä käyttämättömistä tilasiirtymistä. Tilakoneilmentymän valitsemalla voi katsoa kyseisen ilmentymän tilaa, onko se lopputilassa sekä mitä tilasiirtymiä se on käynyt ja mitä virheitä siinä on tapahtunut. Kuvakaappaukseen on merkitty numeroilla



Kuva 5.5: Kuvakaappaus toteutetusta sovelluksesta.

viisi kohtaa käyttöliittymästä. Näiden tarkempi selitys alla.

1. Tilakoneet. Taulukko esittää tunnetut tilakoneet, montako ilmentymää niistä on luotu, kuinka paljon virheitä tilakoneissa sekä niistä luoduissa ilmentymissä on havaittu ja kuinka moni ilmentymä on lopputilassa. Lisäksi taulukossa näkyy kuinka moni tila on sellaisia, joissa ei ole käyty tai tilasiirtymä sellaisia, joita ei ole käytetty missään ilmentymässä.

2. Valittu tilakonemäärittely. Tämä taulukko näyttää tilakoneen virheet sekä kaikissa ilmentymissä käymättömät tilat ja käyttämättömät tilasiirtymät.

3. Valittu tilakoneilmentymä. Tämä taulukko näyttää valitun tilakoneilmentymän siirtymät, virheet sekä käymättömät tilat ja käyttämättömät tilasiirtymät.

4. Näkyvien ilmentymien suodatus. Tilakoneilmentymiä voidaan suodattaa sen mukaan onko niissä virheitä tai onko ne lopputilassa.

5. Valitun tilakoneen ilmentymät suodatuksen mukaan. Tämä taulukko näyttää kaikki valitun tilakoneen ilmentymät, niiden virheiden lukumäärän ja tiedon onko ne lopputilassa vai ei. Taulukon voi myös järjestää kunkin sarakkeen mukaan.

Sovelluksen voi käynnistää kolmessa eri tilassa; viestiväylää lukevassa, hakemistosta lukevassa tai vain viestejä tallentavassa tilassa. Viestiväylää lukeva tila sekä lukee viestejä että suorittaa tilakoneita niiden mukaan ja tallentaa viestit valittuun hakemistoon. Vain viestejä tallentava tila kuuntelee viestiväylää, mutta ei tee viesteille muuta kuin tallentaa ne. Hakemistosta lukevassa tilassa sovellus lukee edellisen kahden tilan tallentamia viestejä järjestyksessä hakemistosta ja siirtää tilakoneita näiden mukaan. Tämä on hyödyllistä sekä testatessa, että myös edellisen testin suoritusta analysoidessa. Koska kaikki viestit tallennetaan järjestyksessä, voidaan sovelluksen tila saada täsmälleen samaksi joka kerralla. Jos tilakoneiden määrityksistä löytyi virheitä testiajossa, voidaan testin jälkeen korjata määrityksen virheet ja toistaa testiajo.

6. ARVIOINTI

Tässä työssä toteutettiin testisovellus, joka tarkastaa hajautetun järjestelmän kommunikaation määritelmänmukaisuuden. Tarkastamista varten sallittu kommunikaatio tulee määritellä tilakaaviolla, joka kertoo täsmällisesti mitkä viestit ovat missäkin tilassa sallittuja. Kommunikaatiot tarkistetaan automaattisesti kahdesta syystä. Ensinnäkin ihmiset tekevät virheitä tai heiltä voi jäädä jotain huomiotta. Toisekseen, kun protokolla määritellään täsmällisesti, tullaan samalla miettineeksi ja kirjoittaneeksi auki toiminta kaikissa tilanteissa. Ihmisiä varten tehdyissä dokumenteissa kaikkea ei yleensä jakseta tai ehditä kirjoittaa auki, joten niissä osa erikoistapauksista saattaa jäädä määrittelemättä. Kun järjestelmää testataan määrittelyä vasten, huomataan määrittelemättömät asiat tilakoneiden virheinä.

Jos joitain toimintoja skeemasta jää määrittelemättä tai ne määritellään suurpiirteisesti, voi olla että niitä ei välttämättä edes voida toteuttaa skeemalla. Kommunikaatioprotokollan määrittely tilakoneiden avulla on kuitenkin yksityiskohtainen ja tarkka, joten kun sillä määritellään järjestelmän kommunikaatio, tullaan samalla todistaneeksi, että skeemalla voidaan toteuttaa kaikki toiminnot, joita siltä vaaditaan.

Sovellusta voidaan käyttää kehityksen lisäksi testauksen apuna. Sovellus ei luo testejä, mutta sitä käyttämällä testauksessa voidaan automaattisesti todeta järjestelmän määritelmänmukaisuus. Näin sovellusta voidaan käyttää luontevasti regressiotestauksen apuna. Se toimii luontevasti integraatiotestauksessa, jossa oikeat sovellukset kommunikoivat, mutta sitä voidaan käyttää myös yksittäisen sovelluksen testauksessa. Sovellusta voidaan myös käyttää niin automaattisessa testauksessa kuin manuaalisessakin.

6.1. Havainnot

Järjestelmä oli tämän työn kirjoittamisen aikana sen verran kehittynyt, että tässä työssä toteutetulle sovellukselle ei ollut samanlaista tarvetta kuin aiemmin, koska kommunikaa-

tioon liittyviä ongelmia oli vähemmän. Tämän vuoksi havainnot jäivät toistaiseksi pieniksi.

Työn aikana tuli kuitenkin testattua sovelluksen hyödyllisyys protokollan kehityksen aikana. Kehityksen aikana tallennettiin testiajo, jota voitiin toistaa erästä protokollan osaa määriteltäessä SCXML:llä. Testaussovelluksen käyttöliittymästä pystyi nopeasti havaitsemaan kaikki testiajossa sattuneet virheet, joista osa oli määrittelyn virheitä, joten niiden tarkastaminen ja määrittelyn korjaaminen oli nopeaa.

Töitä suunnittelevan sovelluksen ohjelmistopäivityksen jälkeen testatessa testisovellus havaitsi erään vian järjestelmässä: kommunikaatiossa havaittu virhe oli sellainen, että eräästä työstä oli lähetetty ainoastaan perumiskäsky, mutta ei luontikäskyä. Tämä oli siinä mielessä vaikeasti havaittava virhe, että se ei aiheuta mitään näkyvää häiriötä järjestelmässä, vaan kaikki toimii ulkoisesti kuten pitääkin. Järjestelmä toipuu tilanteesta, vaikka se ei ole määrittelyn mukainen. Töitä suorittava sovellus toteaa vain, että se ei tunne kyseistä työtä, eikä voi perua sitä.

Tämän tilanteen huomaaminen ilman toteutettua testisovellusta olisi vaatinut, että testaaja olisi seurannut viestienvaihtoa sattumalta ja huomannut tuntemattomasta työstä ilmoittavan vastauksen. Virhetilanne ei aiheuttanut mitään muuta oiretta järjestelmässä, eikä testaajakaan välttämättä huomaa viestiä muuten kuin sattumalta testitilanteessa. Järjestelmä toipui tilanteesta täysin, mutta lähetetty viesti ei kuitenkaan ole määritelmän mukainen ja saattaisi jossain muussa tapauksessa aiheuttaa muitakin oireita tai virhetilanteita.

Toinen vaihtoehto virheen löytämiseen olisi ollut jommankumman sovelluksen login lukeminen, esimerkiksi jonkun muun virheen jäljityksen takia. Molemmissa tapauksissa seuraava askel olisi ollut testattavien sovellusten login lukeminen ennen tapahtumaa ja kyseisen työn etsiminen. Mikäli työn lähetystä ei olisi löydetty, vikaa olisi etsitty työkäskyjä antavasta sovelluksesta. Periaatteessa virhe olisi voinut olla myös töitä suorittavassa sovelluksessa, mutta ainoa tapa selvittää virheellisesti toimiva sovellus oli tarkistaa kyseiseen työhön liittyvät viestit. Logien lukeminen taas on työlästä, sillä ne ovat pitkiä ja niissä on paljon muutakin sisältöä.

Testaussovelluksen kanssa virhe löytyi siten, että käyttöliittymästä näki työhön liittyvän viestienvaihdon, jossa oli yksi virheellinen tilasiirtymä. Kyseisen tilakoneiilmentymän tarkempia tietoja katsottaessa havaittiin, että siinä oleva virhe oli, että tilakoneessa

ei ole tilasiirtymää alkutilasta työn perumiskäskyllä, eli että työlle oli lähetetty ainoastaan perumiskäsky. Tällöin oli selkeää, että työkäskyjä antavassa sovelluksessa oli virhe, sillä se perui työn, jota se ei ollut antanut.

Testisovellus ei ota kantaa siihen toipuuko järjestelmä virhetilanteesta itsestään tai kuinka näkyviä virheet ovat järjestelmässä, vaan se huomaa kaikki mallin vastaiset viestit. Näin se huomasi virheen, joka muuten olisi saattanut jäädä huomaamatta. Vaikka virhetilanne ei nyt aiheuttanut mitään häiriötä järjestelmässä, olisi se saattanut aiheuttaa jatkossa, jolloin varsinaisen syyn etsiminen olisi ollut työläämpää.

6.2. Tilakoneiden ylläpidettävyys

Tilakoneet määriteltiin tässä työssä SCXML:llä scxmlgui-editoria käyttäen. Editorissa oli joitakin käytettävyysongelmia ja puutteita, etenkin isompien tilakaavioiden piirrossa. Automaattinen piirrosten asettelu (layout) ei esimerkiksi toiminut kunnolla. Editorin ongelmista huolimatta pienien tilakaavioiden piirtäminen oli mielestäni helppoa, mutta monimutkaisempien piirtäminen sekavaa siirtymien kaarien ja nimien mennessä päällekkäin.

Tilakaavio ei yksin välttämättä riitä dokumentaatioksi ihmiselle. Joissakin tapauksissa on luontevaa kertoa luonnollisella kielellä mihin pyritään ja miten. Joitakin toimintoja voi silti olla tarpeen kirjoittaa auki sanallisesti, vaikka niiden käytöstä onkin tarkka spesifikaatio. Tässä tapauksessa tilakaavio tulee piirtää joka tapauksessa erikseen tietokoneluettavan tilakaavion lisäksi. Koneluettava tilakaavio ei siis korvaa täysin muuta dokumentaatiota, vaan toimii muun dokumentaation tukena joko sellaisenaan tai tarpeen mukaan abstrahoituna.

Tilakaaviot piilottavat scxmlgui-editorissa tilasiirtymien ehdot ja tietomallin käsittelyn erillisen valikon taakse ja käyttöliittymässä näkyy ainoastaan tilat ja siirtymät. Tästä saa esimerkiksi PNG- tai JPEG-muotoisen kuvan, jota voidaan hyödyntää tarvittaessa korkeamman tason dokumentaatiossa. Lisäksi tilakaavion saa esimerkiksi Graphviz-graafinpiirto-ohjelman dot-kielellä tai vektorimuodossa, joten havainnollisemman kuvan voi piirtää myös muilla työkaluilla kuin scxmlguilla. Kuva toimii ihmisluettavana määrittelynä protokollasta, mutta taustalla on tarkempi, koneluettava määrittely. Näin koneluettavasta, täsmällisestä määrittelystä saadaan ihmiselle helpommin ymmärrettävä, abstraktimpi määrittely.

6.3. Virheiden jäljittäminen

Sen lisäksi että koneluettavilla tilakoneilla voidaan määritellä käytävät keskustelut ja niitä voidaan tarkastaa automaattisesti määritelmää vasten, auttavat ne myös järjestelmän virheen paikantamisessa. Tilakoneen ilmoittaessa virheellisestä tilasiirtymästä siitä nähdään suoraan mikä viesti oli virheellinen, rajoittaen vian tietyn järjestelmän tiettyyn toimintoon. Aiemmista viesteistä voidaan myös päätellä keskustelun tila ja niiden avulla voidaan yrittää toistaa ongelmaa. Menetelmä ei löydä virheitä sovelluksista, mutta auttaa niiden paikantamisessa osoittamalla määrittelyn vastaisen viestin.

Myers [13] listaa virheiden korjaukseen useita tekniikoita, joista ainakin neljään tässä työssä toteutettu sovellus tarjoaa apua. Eräs luetelluista tekniikoista on “Siellä missä on yksi bugi, on todennäköisesti useampikin”. Koneluettavien tilakoneiden avulla ei välttämättä tarvitse tai edes kannata määritellä aivan kaikkea. Sillä kannattaa sen sijaan määritellä toimintoja, joiden olettaa olevan virhealttiita tai toimintoja, joissa on jo havaittu virheitä.

Lisäksi Myers esittää tekniikat “Korjaa virhe, älä vain sen oiretta” ja “Ota huomioon mahdollisuus, että virheen korjaaminen luo uuden virheen”. Toteutettu sovellus ei takaa virheettömyyttä, mutta auttaa kehittäjää tarkastelemaan virhetilannetta kokonaisvaltaisemmin, ei vain virheellisesti toimivan sovelluksen osan, vaan koko järjestelmän kannalta. Virheen korjauksen jälkeen testi voidaan toistaa ja keskustelu verifioida uudestaan. Tällöin voidaan havaita laajemmin mitä mahdollisesti ei-toivottuja vaikutuksia virheen korjaamisella oli. Sovellus ei kuitenkaan poista suunnittelutyötä. Korjaus tulee suunnitella tarkasti, mutta sovellus auttaa korjauksen verifioinnissa. [13]

Myersin mukaan “Virheen korjauksen pitäisi saada korjaaja hetkellisesti takaisin suunnitteluvaiheeseen”. Tätä konesuoritettavat tilakoneet tukevat hyvin. Tilakone on järjestelmän spesifikaatio ja testisovellus näyttää täsmälleen miten sovellus toimi spesifikaation vastaisesti. Näin ollen spesifikaatio on luontevaa ottaa korjausprosessin avuksi. [13]

6.4. Määrittelyn ajantasaisuus

Edellä kuvatuista dokumentointitavoista mikään ei ole yksinään riittävä, vaan olisi syytä käyttää kaikkien tapojen vahvuuksia, tilakaavioita, sekvenssikaavioita tai muita UML:n tai jonkin muun notaation kuvaajia siellä missä niiden käyttäminen on selkeämpää tai

havainnollisempaa kuin sanoilla selittäminen. Vaatimusten muuttuessa muutosten päivittäminen dokumentaatioon teettää ylimääräistä työtä.

Ketterässä ohjelmistokehityksessä tällaista dokumentaatiota on vähän. Termi ketterä ohjelmistokehitys saatetaan määritellä eri tavoilla eri julkaisuissa, mutta tässä sillä tarkoitetaan mitä tahansa prosessia, joka seuraa Agile Manifeston [19] perusperiaatteita. Siinä toimiva sovellus on edistyksen mittari ja sen versioita toimitetaan asiakkaalle usein. Sovelluksen uusia tai muuttuneita vaatimuksia ollaan valmiita ottamaan vastaan missä tahansa vaiheessa kehitystä. Kaikki vaatimukset eivät ole yleensä selvillä etukäteen ja ne muuttuvat ja tarkentuvat usein. Joskus vasta toimivasta sovelluksesta nähdään, että jokin vaatimus tai suunnitteluratkaisu oli virheellinen.

Protokollan spesifiointi suoritettavilla tilakoneilla tukee ketterää ohjelmistokehitystä, sillä ne kehittyvät luontevasti koko järjestelmän kehityksen mukana. Ne eivät siis ole irrallisia, järjestelmän toiminnasta tehtyjä dokumentteja, vaan tärkeä osa järjestelmää ja apuväline sekä kehityksessä että testauksessa. Sen lisäksi, että ne määrittelevät kuinka järjestelmän tulisi toimia, niiden avulla voidaan myös todeta järjestelmän määritelmänmukaisuus ilman ylimääräistä työtä ja tulkintaa. Jos tilakoneet pääsisivät vanhenemaan, tämä havaittaisiin testatessa.

Tilakoneiden käyttö ei poista muun dokumentaation tarvetta. Tilakoneilla on tarkoitus ainoastaan määritellä yksiselitteisesti protokolla, mitä viestejä järjestelmässä liikkuu ja millä tavoilla niihin voidaan reagoida. Ne eivät kerro mitä viestit tarkoittavat millekin sovellukselle, eivätkä ota kantaa sovellusten sisäisten tilojen muutoksiin. Muilla tavoilla voidaan dokumentoida korkeamman tason suunnitteluratkaisuja ja vastuujakoja, mutta protokollan määrittely on parempi tehdä suoritettavilla tilakoneilla.

6.5. Jatkokehitys

Tilakaavioilla ei voida todentaa kaikkia järjestelmän oikeellisuuteen liittyviä toimintoja. Viestejä olisi tarpeellista tarkistaa myös staattisesti niiltä osin mitä ei skeemalla voi määritellä. Esimerkiksi tilauksen saapumispäivän pitää olla ennen käsittelypäivää, tai jokin muu ehto joka riippuu jonkin toisen elementin arvosta. Eräs vaihtoehto tällaiseen viestien staattiseen validointiin voisi olla Schematron [20].

Toteutettu sovellus ainoastaan verifioi käytyä keskustelua määrittelyä vasten osallis-

tumatta keskusteluun itse mitenkään. Sovellusta voitaisiin kehittää jatkossa siten, että se loisi testitapauksia mallin pohjalta ja validoisi sovelluksen käyttäytymistä. Tätä varten tilakoneilla määriteltyjä malleja pitäisi kuitenkin kehittää, sillä nykyisellään ne ainoastaan määrittelevät mitä viestejä väylässä saa kulkea. Niillä pitäisi lisäksi määritellä tarkemmin viestien tarkoitus ja kuinka sovellukset tulkitsevat ne, jotta testitapauksia voisi luoda automaattisesti.

Jos jokin järjestelmän osasovellus jättää vastaamatta viestiin, johon odotetaan vastausta suhteellisen nopealla aikataululla, ei sovellus nykytoteutuksella havaitse tätä. Kyseinen kommunikaatio ainoastaan jää tilakoneen suorituksessa normaaliin tilaan (ei lopputilaan). Reaaliaikajärjestelmiä varten tiloihin olisi tarpeellista määritellä myös aika, jonka kuluessa tilasta tulisi siirtyä pois. Mikäli tilassa pysytään pidempään kuin tämä aika, on järjestelmässä jokin virhe, joka estää kommunikaatiota edistymästä. Tällä hetkellä tilanteen voi havaita testisovelluksesta ainoastaan itse aktiivisesti toteamalla, että jonkin tilakoneilmentymän suoritus ei etene.

7. YHTEENVETO

Tässä työssä käytiin läpi erään hajautetun järjestelmän testaamisessa esille nousseita ongelmia. Tässä järjestelmässä sovellukset kommunikoivat viestiväylän välityksellä XML-viesteillä. Järjestelmän testaamisessa on ollut ongelmana muun muassa määrittelyn puutteellisuus ja järjestelmän monimutkaisuus. Testauksessa havaitut virheet eivät näy aina virheellisesti toimivassa sovelluksessa, vaan vaativat järjestelmän viestinvaihdon tutkimista syyn selvittämiseksi.

Yleensä kommunikaatioprotokollat määritellään sanallisesti ja mahdollisesti kuvia apuna käyttäen. Tällaiset ihmisluettavat dokumentit ovat kuitenkin työläitä ja kalliita ylläpitää ja helposti monimutkaisia tai monitulkintaisia. Vääriä tulkintoja näistä voi olla yhtä monta kuin järjestelmän kehittäjiä ja testaajia.

Ihmisluettavien dokumenttien sijaan tässä työssä esitettiin tilakonemalli, jota ihmisen lisäksi myös tietokone voi lukea. Lisäksi tässä toteutettiin testisovellus, joka lukee tämän mallin mukaisia määrittelyjä. Sovellus seuraa järjestelmän viestiliikennettä ja tarkistaa sen tilakoneita vasten sekä ilmoittaa virheellisistä viesteistä. Näin saadaan viestin tarkkuudella selville tilanteet, joissa järjestelmä toimii määritelmän vastaisesti.

Koneluettavat ja -suoritettavat tilakoneet ovat irrallisen ja monitulkintaisen dokumentin sijaan yksiselitteisiä ja luonteva osa ohjelmistokehitystä. Kun vaatimukset muuttuvat, määrittelyn muutoksen jälkeen järjestelmä voidaan automaattisesti testata uutta mallia vasten sen sijaan, että samat muutokset toteutetaan sekä järjestelmään että määrittelyyn erikseen.

Tässä työssä toteutettua testisovellusta käytettiin hyödyksi kohdejärjestelmän testauksessa ja virheiden etsinnässä myös muiden kuin allekirjoittaneen toimesta. Testisovellus otettiin erään sovelluksen toteutustiimin kesken mielellään vastaan. Sen tarve on tiedostettu jo aiemmin, ja testisovellus koettiin erittäin toimivana tarkoitukseensa. Sen avulla virheiden jäljitys on nopeampaa ja testatessa saa suuremman varmuuden järjestelmän oikeellisuudesta. Etenkin monimutkaisemmissa testeissä testisovelluksen avulla havaitaan

virheitä, jotka jäisivät muuten piiloon järjestelmän toipuessa niistä itsestään ilman näkyvää häiriötä.

Vaikka testisovellusta ei vielä ole käytetty kovin paljon, eikä sillä voida määritellä protokollan käyttäytymistä täysin, se on osoittautunut hyödylliseksi työkaluksi. Tästä syystä sitä on tarkoitus jatkokehittää ja sen asemaa vakiinnuttaa osana kehitys- ja testausprosessia. Järjestelmässä on myös lähitulevaisuudessa tiedossa melko suuri skeeman päivitys. Tämän päivityksen yhteydessä tilakoneilla tehdyt määrittelyt on luonteva ottaa kehityksen ja testauksen tueksi heti skeeman käytön alkuvaiheessa. Näin päästään määrittelemään protokollan käyttäytyminen varhaisessa vaiheessa. Oletettavasti määrittelyn avulla voidaan karsia tai ainakin havaita nopeammin virheitä, joita sovelluksissa ilmenee uusien ominaisuuksien toteutuksen jälkeen.

LÄHTEET

[1] John E. Hopcroft and Jefferey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley Publishing Company, 1979.

[2] P.V. Mockapetris. Domain names - concepts and facilities. RFC 1034 (INTERNET STANDARD), November 1987. Updated by RFCs 1101, 1183, 1348, 1876, 1982, 2065, 2181, 2308, 2535, 4033, 4034, 4035, 4343, 4035, 4592, 5936.

[3] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. RFC 2616 (Draft Standard), June 1999. Updated by RFCs 2817, 5785, 6266, 6585.

[4] David Harel. Statecharts: A visual formalism for complex systems. *Sci. Comput. Program.*, 8(3):231–274, Kesäkuu 1987.

[5] J. Postel. Transmission Control Protocol. RFC 793 (INTERNET STANDARD), September 1981. Updated by RFCs 1122, 3168, 6093, 6528.

[6] Scott Ambler. Introduction to test driven development (tdd). [WWW] `http://www.agiledata.org/essays/tdd.html`.

[7] Unified modelling language (UML) 2.4.1 superstructure specification. [WWW] `http://www.omg.org/spec/UML/2.4.1/`, 2011.

[8] Xml path language (xpath) version 1.0. [WWW] `http://www.w3.org/TR/xpath/`.

[9] Xml schema part 0: Primer second edition. [WWW] `http://www.w3.org/TR/xmlschema-0/`.

[10] Brian Dobing and Jeffrey Parsons. How uml is used. *Commun. ACM*, 49(5):109–113, May 2006.

[11] Saurabh Tiwari and Atul Gupta. Statechart-based use case requirement validation of event-driven systems. In *Proceedings of the 27th Annual ACM Symposium on Applied Computing*, SAC '12, pages 1091–1093, New York, NY, USA, 2012. ACM.

[12] Supaporn Kansomkeat and Wanchai Rivepiboon. Automated-generating test case using uml statechart diagrams. In *Proceedings of the 2003 annual research conference of the South African institute of computer scientists and information technologists on Enablement through technology*, SAICSIT '03, pages 296–300, Republic of South Africa, 2003. South African Institute for Computer Scientists and Information Technologists.

[13] Glenford J. Myers, Tom Badgett, Todd M. Thomas, and Corey Sandler. *The Art of Software Testing*. Word Association, Inc, 2. edition, 2004.

[14] Timothy C. Lethbridge, Janice Singer, and Andrew Forward. How software engineers use documentation: the state of the practice. *IEEE Software*, 20:35–39, 2003.

[15] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. RFC 2068 (Proposed Standard), January 1997. Obsoleted by RFC 2616.

[16] State chart xml (scxml): State machine notation for control abstraction (working draft 6.12.2012). `http://www.w3.org/TR/2012/WD-scxml-20121206/`, Joulukuu 2012.

[17] scxmlgui - a graphical editor for scxml finite state machines. [WWW] `https://code.google.com/p/scxmlgui/`.

[18] Fast lightweight expression evaluator. [WWW] `https://flee.codeplex.com/`.

[19] Kent Beck, Mike Beedle, Arie van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, Jon Kern, Brian Marick, Robert C. Martin, Steve Mellor, Ken Schwaber, Jeff Sutherland, and Dave Thomas. Principles behind the agile manifesto. [WWW] `http://www.agilemanifesto.org/principles.html`, 2001.

[20] Dare Obasanjo. Improving xml document validation with schematron. [WWW] `http://msdn.microsoft.com/en-us/library/aa468554.aspx`, 2004.