Tietojenkäsittelytiede **22** Joulukuu 2004 sivut 35–46
Toimittaja: Ari Korhonen

Konferenssiprotokollan automaattinen testaus

Antti Kervinen & Pablo Virolainen
Tampereen teknillinen yliopisto
Ohjelmistotekniikan laitos
antti.kervinen@tut.fi, pablo.virolainen@tut.fi

Tiivistelmä

Rinnakkaisuutta sisältävien järjestelmien testaaminen ennalta määritellyin testitapauksin on tehotonta ja hankalaa. Samoilutestaus on automaattista testausta, jossa testitapauksia ei tarvita. Niiden sijaan käytetään tilakonemuotoista kuvausta järjestelmän testattavasta käyttäytymisestä. Tässä artikkelissa raportoimme järjestelyt ja tulokset kokeesta, jossa testasimme käyttäjän näkökulmasta useaa erään konferenssiprotokollan toteutuksista koostuvaa järjestelmää samoilutestausmenetelmällä. Testikohteemme on aidosti rinnakkainen ja se voi käyttäytyä monissa tilanteissa epädeterministisesti. Se myös edustaa kokoluokkaa, jonka testaaminen tavanomaisin testitapauksin yhtä kattavasti kuin samoilutestausmenetelmällä on vaativaa — ellei mahdotonta.

1 Johdanto

Yleisesti käytössä olevissa automaattisissa testausmenetelmissä joko ihminen tai kone (mahdollisesti ihmisen avustuksella) laatii joukon testitapauksia. Kukin testitapaus kuvaa tapahtumaketjun, joka päättyy testikohteelta saaduista vasteista riippuen tulokseen *hyväksytty* (testikohde käyttäytyi niin kuin odotettiin), *hylätty* (testikohde teki jotain kiellettyä) tai *ratkaisematon*.

Tulos on ratkaisematon, jos testikohde käyttäytyy sallitulla tavalla, mutta kuitenkin siten, ettei haluttua toimintoa saada testattua. Näin käy esimerkiksi silloin, kun testitapauksessa yritetään testata viestin lähettämistä keskustelupalstalle, mutta testattava ohjelma ilmoittaakin ”yhteyttä palvelimeen ei saada”. Tällaiseen tulokseen päädyttäessä testauksessa on hukattu aikaa ja mahdollisuus jatkaa testausta kenties harvinaisestakin tilanteesta eteenpäin.

Ratkaisemattomat tulokset ovat seurausta testattavassa järjestelmässä piilevästä epädeterminismistä: samojen syötteiden antaminen voikin johtaa eri kerroilla erilaisiin lopputuloksiin. Muun muassa testattavan järjestelmän sisältämä rinnakkaisuus voi tehdä sen ulkoisesta käyttäytymisestä epädeterministisen. Tavallisesti rinnakkaisuus kasvattaa myös järjestelmän tila-avaruuden niin suureksi, että uskottavaan testikattavuuteen pystyvän testitapausjoukon laatiminen ja hallinta käyvät mahdottomiksi. Siksi rinnakkaisuutta sisältävien järjestelmien testaus tavanomaisin testitapauksin on ongelmallista.

Samoilutestaus [5] on automaattista, mallipohjaista testausta, missä testiä oh-

jaavalle tietokoneohjelmalle, *testikoneelle*, annetaan valmiiden testitapausten sijaan yksi *vaatimuskone*. Se on suunnattu graafi, joka kuvaa kaiken testattavan käyttäytymisen: mitä syötteitä voidaan testattavalle järjestelmälle milloinkin antaa ja mitä vasteita siltä odottaa. Testikoneen päätettäväksi jää, mikä syöte kaikista mahdollisista lähetetään ja milloin pysähdytään odottamaan vasteita.

Testiajon aikana testikoneen voidaan ajatella tuottavan automaattisesti yhtä loputtoman pitkää testitapausta. Testitapausten automaattista muodostamista on tutkittu paljon Chow'n 1978 julkaiseman automaattiin pohjautuvan W-menetelmän [2] jälkeen. Artikkeliin [7] on koottu monia testitapausten muodostusmenetelmiä ja niitä hyödyntäviä testaustyökaluja.

Monissa menetelmissä (W-menetelmä ja sen parannukset sekä Unique Input Output, Distinguishing sequences ja niiden johdannaiset [7]) oletetaan vaatimuskone täydelliseksi (jokaisesta solmusta lähtee kaari kaikilla tapahtumanimillä) ja/tai minimaaliseksi (kahden eri solmun jälkeiset kaikki mahdolliset tapahtumaketjut eivät saa olla täsmälleen samanlaisia).

Käytännössä menetelmät muodostavat pienistäkin vaatimuskoneista erittäin suuren määrän testitapauksia, jonka voidaan osoittaa löytävän kaikki tietyntyyppiset virheet — tietyin oletuksin (tiedossa on vaatimuskoneen solmujen ja testattavan järjestelmän tilojen määrän suhde). Laajan kattavuutensa vuoksi nämä menetelmät tukehtuvat vähänkin suurempiin vaatimuskoneisiin. Käytämmekin tämän artikkelin testiajoissa, joissa vaatimuskoneessa on lähes 200 000 tilaa, huomattavasti kevyempiä algoritmeja. Niitä voidaan pitää Transition tour -menetelmän [8] laajennoksina, ja olemme julkaisseet ne artikkelissa [6].

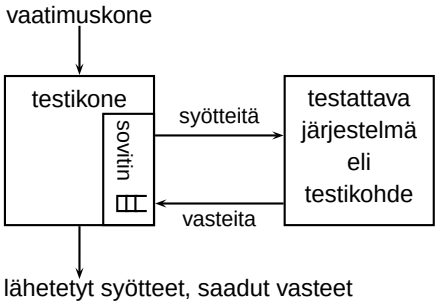
Esitämme tässä artikkelissa testaustulokset, jotka saimme testatessamme samoilutestausmenetelmällä erästä konferenssiprotokollaa palvelutasolta. Konferenssiprotokolla tarjoaa reaaliaikaisen keskustelupalvelun ("chat"). Siinä käyttäjä voi liittyä konferensseihin, lähettää niihin viestejä ja poistua konferensseista. Konferenssissa käyttäjä vastaanottaa muiden samanaikaisesti samaan konferessiin kuuluvien käyttäjien lähettämät viestit.

Protokolla on toteutettu Twenten yliopistossa [3] testaustutkimusta varten. Sen lähdekoodia voidaan käännösaikana parametrisoida siten, että tuloksena on erilaisia virheitä sisältäviä mutantteja. Tätä toteutusta ja sen mutantteja on testattu aiemminkin [1, 9]. Aiemmissa testeissä testikohteena on kuitenkin ollut yksittäinen protokollan prosessi, ei useiden prosessien yhdessä tuottama palvelu. Valitsimme palvelunäkökulman saadaksemme testikohteeksi todellista rinnakkaisuutta sisältävän järjestelmän. Sellaisen järjestelmän testaaminen tavanomaisin testausmenetelmin on ongelmallista, mutta samoilutestaukselle sen pitäisi sopia hyvin.

Luvussa 2 kerrotaan tarkemmin, mitä samoilutestaus on. Konferenssiprotokollan toiminta selitetään luvussa 3 ja sen pohjalta rakennetun vaatimuskoneen tekeminen luvussa 4. Luvussa 5 esitetään testijärjestelyt ja saadut tulokset.

2 Samoilutestaus

Samoilutestauksessa *testikoneelle* annetaan syötteeksi *vaatimuskone* (katso kuva 1), joka kuvaa testattavalta järjestelmältä, *testikohteelta*, odotetun käyttäytymisen. Viestintä testikoneen ja testikohteen välillä hoidetaan erityisen testikohdekohtaisen *sovittimen* avulla.

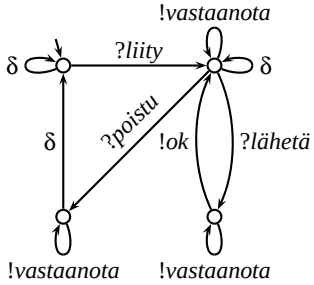


Kuva 1: Samoilutestauksen arkkitehtuuri

2.1 Vaatimuskone

Vaatimuskoneella on kaksi roolia. Ensinnäkin siitä voidaan lukea, mitä syötteitä testikoneelle voidaan milloinkin lähettää. Toisaalta se toimii oraakkelina, jolta voidaan tarkastaa, vastasiko testikohde oikein.

Kuvassa 2 esitetään yksinkertainen vaatimuskone. Se kuvaa konferenssiprotokollan käyttäytymistä käyttäjän näkökulmasta. Alussa käyttäjä voi liittyä konferenssiin ja sen jälkeen vastaanottaa muiden käyttäjien lähettämiä viestejä. Konferenssiin kuuluessaan hän voi myös lähettää itse viestejä tai poistua konferenssista.



Kuva 2: Yksinkertainen vaatimuskone

Vaatimuskone on graafi, jonka solmuista yksi on erityinen *alkutila* (kuvassa vasemmalla ylhäällä). Graafin suunnatut kaaret on nimetty tapahtumia kuvaavin nimin. Tapahtumat jakaantuvat kahteen tyyppiin: testikohteelle lähetettäviin syötteisiin (kuvassa kysymysmerkillä alkavat tapahtumanimet) ja testikohteelta saatuihin vasteisiin (huutomerkillä alkavat tapahtumanimet). Lisäksi käytössä on kaksi erityistä tapahtumanimeä: δ, joka tarkoittaa hiljaisuutta (vastetta kuunneltaessa sallitaan testikohteen olevan hiljaa) ja ρ, joka kuvaa testikohteen uudelleenkäynnistämistä. δ:aa voidaan pitää erityisenä vasteena ja ρ:ta erityisenä syötteenä. Kuvassa 2 ei esiinny ρ-kaaria, mutta sen erityisen merkityksen vuoksi (”ota sähköjohto hetkeksi irti seinästä”) voidaan ajatella, että jokaisesta vaatimuskoneen solmusta vie ρ-kaari alkutilaan.

Puhumme vaatimuskoneesta suunnattuna graafina, emme tilakoneena, korostaaksemme eroa graafin solmujen ja testattavan järjestelmän tilojen välillä. Koska vaatimuskone saattaa sisältää vain pienen osan testattavan järjestelmän käyttäytymisestä, ja koska sen jokaista solmua voi vastata moni todellisen järjestelmän tila (ja toisin päin), vaatimuskoneen pitäminen järjestelmän tiloja mallintavana tilakoneena johtaa harhaan. Esimerkiksi mallintarkastukseen perustuvaa verifiointia haittaava tilaräjähdys [11] ei estä samoilutestauksen käyttämistä monimutkaistenkin järjestelmien testauksessa.

2.2 Testikone

Testiajon aikana testikone pitää kirjaa siitä vaatimuskoneen solmusta, jossa suoritus on menossa, alkutilasta aloittaen. Solmusta lähtevät kaaret määräävät, mitä testikone voi seuraavaksi tehdä. Vaihtoehtoja ovat jonkin lähtevän kaaren nimeä vastaavan syötteen lähettäminen testikohteelle; testikohteen vasteen kuunteleminen, jos jokin lähtevien kaarien nimistä on vaste; testikohteen uudelleenkäynnistäminen ja

testin aloitus alkutilasta; ja testauksen lopettaminen.

Kuvan 2 vaatimuskoneessa testikoneella on alkutilassa kaksi vaihtoehtoa jatkaa testiä. Se voi joko kuunnella vastetta ja sallia tällöin vain hiljaisuuden, tai lähettää testikohteelle tapahtumaa ?*liity* vastaavan syötteen. Liittymisen jälkeen vastetta kuunneltaessa testikohteen sallitaan ilmoittavan vastaanotetusta viestistä tai olevan hiljaa. Samassa tilassa testikone voi valita lähettävänsä testikohteelle syötteen ”lähetä viesti” (?*lähetä*) tai ”poistu konferenssista” (?*poistu*).

Vaatimuksista poikkeava käyttäytyminen havaitaan joko testikohteen kieltäytymisenä annetusta syötteestä tai sellaisen vasteen palauttamisena, joka solmusta lähtevien kaarten mukaan ei ole sallittu. Jos vastetta ei saada, sitä pidetään δ-vasteena. Syötteestä kieltäytymisessä sovitin ei voi antaa testikoneen valitsemaa syötettä testikohteelle. Esimerkiksi pankkikorttia ei voida syöttää pankkiautomaattiin, kun luukku on suljettu.

Testiajon aikana testikoneen toteutuksemme tulostaa muun muassa jokaisen lähettämänsä syötteen ja saamansa vasteen. Jos vaatimuskoneessa on kaari, jota em. tulostettu tapahtuma vastaa (eli jos testikohteen käyttäytyminen ei poikkea vaatimuksista), myös kaaren lähtösolmu ja maalisolmu tulostetaan. Tämä tieto auttaa vaatimuksista poikkeavan käyttäytymisen syiden selvittämisessä.

Muutamia kymmeniä solmuja sisältävän vaatimuskoneen voi vielä laatia käsin. Testattaessa monimutkaista järjestelmää tämän kokoluokan vaatimuskoneet sisältävät kuitenkin vain pienen osan koko järjestelmän käyttäytymisestä. Vaatimuskoneita voikin koota, kuten todellisiakin järjestelmiä, kytkemällä pienempiä, keskenään kommunikoivia automaatteja rinnakkain (tähän teoriaan johdattaa verifiointitutkimuksen näkökulmasta artikkeli [10]). Näin saadaan aikaan vaatimuskoneita, jotka mm. sallivat testikoneen testaavan hyvin monia erilaisia tapahtumaketjujen permutaatioita.

2.3 Sovitin

Vaikka testikoneen toiminta onkin pitkälti testijärjestelyistä riippumatonta, testikone sisältää myös testikohteesta riippuvan sovittimen (katso kuva 1).

Sovitin tekee muunnokset vaatimuskoneessa esiintyvien tapahtumanimien ja testikohteen todellisten tapahtumien välillä. Syöte ?*liity* voi esimerkiksi muuntua tietyn merkkijonon kirjoittamiseksi jonkin testikohteen sisältämän prosessin lukemaan putkeen tai fyysisen laitteen painikkeen painamiseksi.

Sen lisäksi, että sovitin kääntää testikohteen vasteet jälleen tapahtumanimiksi, se myös tulkitsee vasteiden tulojärjestyksen tallentaessaan ne havaittujen vasteiden jonoon. Samoilutestaus ei tunne tapahtumien yhtaikaisuutta. Jos jotkin tapahtumat voivat olla samanaikaisia, se esitetään vaatimuskoneessa kaikkien tapahtumajärjestysten sallimisena.

Sovitin käsittelee myös erityiset ρ- ja δ-tapahtumat. Se voi sammuttaa testikohteen ja käynnistää sen uudelleen sekä päättää, kuinka kauan todellisia vasteita odotetaan, ennen kuin vaste tulkitaan hiljaisuudeksi.

2.4 Samoilun ohjaaminen

Eräs samoilutestaukseen liittyvä tutkimusongelma on, miten valitaan vaatimuskoneesta seuraava testiaskel — kuunnellako vastetta vai lähettääkö jokin syöte (mikä)? Tässä artikkelissa raportoitavissa testiajoissa käytetyt valinta-algoritmit esiteltiin tarkemmin artikkelissa [6].

Pelaaja-algoritmi (esitelty artikkelissa [6] nimellä *Adaptive player, complex step evaluation function*) laskee jokaisessa vaatimuskoneen solmussa viisi testiaskelta eteenpäin. Se valitsee seuraavan tapahtuman ensisijaisesti siten, että se saa todennäköisimmin viiden askeleen kuluessa lähetettyä ja kuunneltua mahdollisimman monta aiemmin testaamatonta syötettä ja vastetta. Toissijaisesti se pyrkii mahdollisimman moneen aiemmin käymättömään vaatimuskoneen solmuun ja seuraavaksi se pyrkii suorittamaan mahdollisimman monta sellaista vaatimuskoneen kaarta, joita ei ole aiemmin suoritettu. Viimeisimpänä se välttää suorittamasta sellaisia kaaria, jotka on suoritettu jo hyvin monta kertaa.

Ahne satunnainen algoritmi (esitelty artikkelissa [6] nimellä *Greedy random*) pyrkii valitsemaan testiaskeleen siten, että solmussa valitaan aina aiemmin suorittamaton kaari. Vasteet ovat etusijalla syötteisiin nähden. Jos se ei ole mahdollista, valinta on satunnainen.

Satunnainen algoritmi valitsee satunnaisesti yhden kaaren. Jos se oli vaste, jäädään kuuntelemaan testikohteen vastetta, muutoin lähetetään kaarta vastaava syöte. Näihin syötteisiin ei lueta mukaan erityistä ρ-tapahtumaa.

3 Konferenssiprotokolla

Testaamassamme konferenssiprotokollan toteutuksessa kaikki asemat, eli koko järjestelmän sisältämät protokollaprosessit, ovat tasavertaisia. Toisin sanoen, toteutus ei sisällä erillistä palvelinta. Sen sijaan asemiin on toteutettu multicast-rajapinta, joka TCP- tai UDP-protokollaa käyttäen lähettää sanomia muille asemille.

Toteutus sallii käyttäjän kuuluvan korkeintaan yhteen konferenssiin kerrallaan. Asemat palvelevat yhtä käyttäjää kerrallaan ja pitävät kirjaa *potentiaalisista osallistujista* ja *osallistujista*. Potentiaaliset osallistujat ovat joukko asemien IP- ja porttiosoitepareja, jotka luetaan käynnistyksen yhteydessä asetustiedostosta. Tämän jälkeen joukko on muuttumaton. Osallistujat, joka on osajoukko potentiaalisista osallistujista, on tyhjä, kun käyttäjä ei kuulu mihinkään konferenssiin. Muulloin tässä joukossa pidetään kirjaa käyttäjän konferenssin muista osallistujista.

Asemat lähettävät toisilleen *liity*-, *kuittaa*-, *data*- ja *poistu*-sanomia seuraavien sääntöjen mukaisesti:

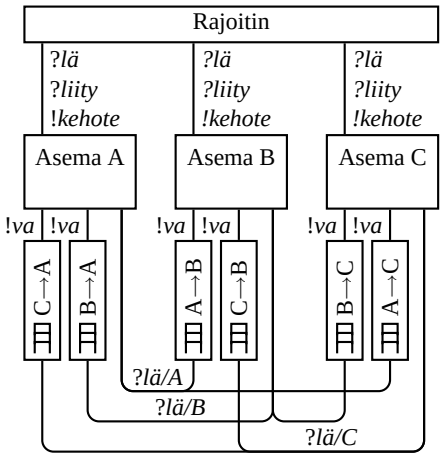
1. Kun käyttäjä A liittyy konferenssiin k, lähetetään potentiaalisille osallistujille $liity(A,k)$.
2. Jos käyttäjä A kuuluu konferenssiin k ja vastaanotetaan $liity(B,k)$, aseman A osallistujiin lisätään B ja lähettäjälle vastataan $kuittaa(A,k)$. Muutoin *liity* unohdetaan.
3. Jos käyttäjä A kuuluu konferenssiin k ja vastaanotetaan $kuittaa(B,k)$, aseman A osallistujiin lisätään B. Muutoin *kuittaa* unohdetaan.
4. Kun käyttäjä A lähettää viestin m, lähetetään osallistujille $data(A,m)$.
5. Jos [1)]käyttäjä A kuuluu konferenssiin k, [2)]vastaanotetaan $data(B,m)$ ja [3)]lähettäjä B löytyy osallistujien joukosta, kerrotaan A:lle viestin m saapuneen lähettäjältä B. Jos B:tä ei löydy osallistujista, vastataan lähettäjälle $liity(A,k)$.
6. Kun käyttäjä A poistuu konferenssista k, lähetetään $poistu(A,k)$ osallistujille ja tyhjennetään aseman A osallistujat-joukko.
7. Jos vastaanotetaan $poistu(B,k)$ poistetaan B osallistujista.

4 Vaatimuskoneen rakentaminen

Monipuolisen vaatimuskoneen aikaansaamiseksi mallinsimme TVT-verifiointityökaluin [4] konferenssiprotokollajärjestelmän, johon kuuluu kolme asemaa. Järjestelmä sisältää kaksi konferenssia, joihin asemilta voidaan liittyä.

4.1 Rinnankytkentä

Kuvassa 3 esitetään mallin arkkitehtuuri. Kuvan laatikot vastaavat mallinnettuja järjestelmän osia, joista verifiointityökalujen avulla rinnankytkemällä saatiin tulokseksi järjestelmän malli. Laatikkoja yhdistävät viivat esittävät alijärjestelmien välistä viestintää rinnankytkennässä. Niin alijärjestelmät kuin mallikin ovat vaatimuskoneen kaltaisia graafeja, joskin ilman erityisiä δ- ja ρ-tapahtumia. Rinnankytkennässä ei myöskään syötteillä ja vasteilla ole mitään eroa.



Kuva 3: Malli

Käyttämässämme rinnankytkennässä alijärjestelmät kommunikoivat keskenään synkronisesti. Rinnankytkijälle annetaan alijärjestelmien lisäksi rinnankytkentäsäännöt, jotka määräävät mitkä alijärjestelmät osallistuvat mihinkin synkroniseen suoritukseen ja millä tapahtumilla. Yksi näistä säännöistä voisi esimerkiksi määrätä, että aseman A sanoman lähetystä vastaava tapahtuma synkronoidaan aseman B sanoman vastaanottamista kuvaavan tapahtuman kanssa (kuvassa 3 ei ole tällaista synkronointia).

Synkroninen suoritus on mahdollinen täsmälleen silloin, kun jokainen siihen liittyvä alijärjestelmä voi suorittaa säännössä mainitun tapahtumansa. Suoritus näkyy lopputuloksessa yhtenä tapahtumana: siinä suoritetaan ”yhdellä rysäyksellä” kaikki synkronoidut tapahtumat.

Synkronoimattomat tapahtumat puolestaan voidaan suorittaa lopullisessa järjestelmässä aina kun ne voidaan suorittaa alijärjestelmissäkin. Niiden jokainen suoritus tuottaa yhden suorituksen lopputulokseenkin. Jos esimerkiksi asemat A ja B ovat lähettäneet toisilleen jotkin sanomat, ne voivat edelleen ilmoittaa sanoman saapumisesta käyttäjilleen kummassa järjestyksessä tahansa, kun ilmoitusta ei ole synkronoitu. Rinnankytkentää käsitellään syvällisemmin artikkelissa [10].

4.2 Asemat ja puskurit

Asemat A, B ja C (katso kuva 3) mallintavat luvussa 3 esitettyjen sääntöjen mukaisen aseman käyttäytymisen. Jos asemien välistä viestiliikennettä kuvaavat tapahtumat synkronoitaisiin suoraan, kuten rinnankytkentäsäännön esimerkissä mainittiin, asemat voisivat estää toisiaan lähettämästä viestejä kieltäytymällä vastaanottamasta niitä. Tämä rajoittaisi mallin käyttäytymistä, sillä asemat eivät voisi sekä lähettää että vastaanottaa viestejä lähes samanaikaisesti. Saadaksemme asemat toimimaan asynkronisesti sisällytimme malliin yksinkertaiset puskuriprosessit

(C→A, B→A, . . . , A→C).

Kuvassa 3 esimerkiksi tapahtuma ?*lä/A* (viestin lähetystä asemalta A kuvaava tapahtuma) synkronoidaan kahden puskurin, A→B ja A→C, vastaavien tapahtumien kanssa. Tämän jälkeen asemat B ja C voivat kummassa tahansa järjestyksessä lukea puskuristaan !*va* (viestin vastaanottamista kuvaava tapahtuma). Vastaavasti, jos asemilta A ja B lähetetään sanomat lähes samanaikaisesti, C voi vastaanottaa kumman tahansa ensin, koska C:n tapahtuma !*va* voi tällöin synkronoitua kumman tahansa puskurin (A→C ja B→C) vastaavan tapahtuman kanssa ensin.

4.3 Mallista vaatimuskoneeksi

Malliin kuuluvalle Rajoitin-alijärjestelmälle ei ole todellisessa maailmassa mitään vastinetta. Sitä käytetään mallissa poistamaan sellaiset käyttäytymiset, joita emme halua testata. Rajoitin estää esimerkiksi viestien lähettämisen sellaisiin konferensseihin, joihin joiltakin asemilta yritetään parhaillaan liittyä. Tällaisessa tilanteessa liittyvän aseman on mahdollista vastaanottaa lähetetty viesti tai olla vastaanottamatta, riippuen siitä, ehtikö viestin lähettäjän asema saada liittymisilmoituksen ja liittyjä siihen kuittauksen ennen kuin viesti lähetettiin.

Rajoittimen avulla saimme aikaan vaatimuskoneen, jossa voidaan viestin lähettämisen jälkeen olla aina varmoja siitä, keiden asiakkaiden pitää vastaanottaa viesti ja ketkä eivät saa sitä vastaanottaa. Rajoittimen käyttöä voidaan myös kritisoida, koska sen kanssa jää testaamatta, sekoaako järjestelmä jos konferenssiin yritetään liittyä ja lähettää viestiä samanaikaisesti.

Rinnankytkentään voisi lisätä muitakin tällaisia rajoittimia. Joissakin virallisten testiajojen jälkeisissä omissa kokeissamme käytimme esimerkiksi rajoitinta, joka karsi vaatimuskoneesta suuren määrän tiloja vähentämällä symmetriaa. Tämä hyvin yksinkertainen rajoitin salli asiakkaan liittyä asemalta A vain konferenssiin 1, asemalta B konferensseihin 1 ja 2, jne. Näin voidaan edelleen testata tilanteita, joissa ”jokaisessa konferenssissa on yksi asiakas” ja ”kaikki asiakkaat ovat samassa konferenssissa”, muttei enää esimerkiksi ”kaikki asiakkaat ovat konferenssissa 2”. Tämä pienensi vaatimuskonetta 63 %, alkuperäisestä n. 192 000 solmusta n. 72 000 solmuun.

Jos käyttäjät olisi mallinnettu, ne olisi synkronoitu asemien konferenssiin liittymistä, poistumista ja datan lähettämistä sekä vastaanottamista kuvaaviin tapahtumiin. Jättämällä käyttäjät pois mallista emme rajoita sitä, miten asemia käytetään. Synkronisessa rinnankytkennässähän lisäprosessit eivät saa muita prosesseja tekemään mitään, mitä ne eivät muutenkin olisi voineet tehdä.

Rinnankytkennän lopputulos syötettiin testikoneelle. Sen voidaan ajatella täydentävän vaatimuskonetta siten, että jokaisessa solmussa on mahdollista suorittaa ρ (uudelleenkäynnistys) ja siirtyä vaatimuskoneen alkutilaan. Samoilua ohjaavat heuristiikat eivät tosin koskaan valinneet ρ-tapahtumaa suoritettavaksi. Valinta tehtiin aina muiden tilasta lähtevien tilasiirtymien joukosta, eikä vaatimuskone sisältänyt sellaisia tiloja, missä tuo joukko olisi ollut tyhjä.

Testikone ei lisää vaatimuskoneeseen hiljaisuutta kuvaavia δ-kaaria automaattisesti, vaikka δ-silmukan lisääminen solmuun, josta ei lähde vasteita kuvaavia kaaria voisi tuntua luonnolliselta. Syy tähän on se, että hiljaisuuden lisäksi vastekaarien puuttuminen voi merkitä myös sitä, ettei vaatimuskoneen laatija ole määritellyt vasteen kuuntelun lopputulosta. Tämän takia testikoneen syötteessä pitäisi

olla erikseen δ-kaaret, jos niiden lähtösolmuissa haluttaisiin testata hiljaisuutta.

5 Testiajojen tulokset

Konferenssiprotokollasta tuotettiin käännösaikaisia parametreja muuttamalla 48 erilaista toteutusta, joista yksi oli toimivaksi tarkoitettu versio ja muut erilaisia mutantteja.

5.1 Tulokset

Testiajoja ajettiin rinnakkain 29 tietokoneella (Solaris 8 käyttöjärjestelmät, koneissa oli UltraSparc v9 -prosessoreja 1–2 kpl, kellotaajuuksilla 296–648 MHz). Kunkin testiajon prosessit (testikone ja kolme asemaprosessia) sijaitsivat samalla tietokoneella.

Testiajojen tulokset esitetään taulukossa 1. Taulukon riveillä on testatut mutantit: ensimmäisessä sarakkeessa on mutantin numero (käytämme samaa mutanttien numerointia kuin konferenssiprotokollan jakelupaketissakin on käytetty) ja toisessa lyhyt kuvaus mutaation vaikutuksesta.

Kolmella seuraavalla sarakeparilla esitetään käytetyt valinta-algoritmit: Pelaaja, Ahne satunnainen ja Satunnainen algoritmi. Kukin sarakepari sisältää sarakkeen yhden mutantin (1) ja kaksi (samanlaista) mutanttia (2) sisältävälle järjestelmälle. Näissä sarakkeissa esitetään kahden testiajon pituuksien keskiarvot. Jokainen järjestelmälle lähetetty syöte ja siltä saatu vaste kasvattaa testiajon pituutta yhdellä. Testiajo päättyy joko virheen havaitsemiseen tai keskeytykseen, jos testiajon pituus ylittää 10 000 askelta. Taulukossa on viiva, jos molemmat testiajot päättyivät keskeytykseen.

Taulukossa 1 merkintä [1] tarkoittaa, että esittämämme mutaation selitys poikkeaa konferenssiprotokollan toteutuksen mukana tulleesta selityksestä. Esittämämme selitys on lähdekoodin mukainen. Merkintä [2] tarkoittaa, että molemmissa tapauksissa ”Sat.”-sarakkeen testiajoissa toinen päättyi keskeytykseen ja toinen virheen löytämiseen. Keskeytykseen päättyneen testiajon pituutta ei ole laskettu esitettyyn keskiarvoon.

Testitulokset osoittavat, että hyvin käyttäytyvien käyttäjien tapauksessa testatuista mutaatioista 18 ei todennäköisesti vaikuta protokollan tarjoamaan palveluun. Kuitenkin voi olla, että kaksi niiden joukosta valittua, eritavoin mutatoitua asemaa samassa järjestelmässä rikkovat palvelun, ja että täydellisemmällä vaatimuskoneella mutantteja paljastuisi enemmän.

Tehokas keino löytää virheitä on käyttää testikohdetta eri tavoin kuin on tarkoitettu, esimerkiksi antamalla tarkoituksella epäkelpoja syötteitä. Vaikka oikein toimiva järjestelmä mallintamalla saadaankin laaja vaatimuskone, tällaisia testitapauksia ei vaatimuskoneeseen silloin synny.

Ongelma on kuitenkin pelkästään vaatimuskoneen rakennustavassa, ei samoilutestausmenetelmässä. Odottamattomien syötteiden testaamista varten on mahdollista rakentaa toinen vaatimuskone tai virheellisesti toimiva alijärjestelmä voidaan kytkeä osaksi vaatimuskonetta.

Uskomme, että osa testin läpäisseistä mutanteista olisi löydetty, jos vaatimuskone olisi sisältänyt yhdenkin hankalan aseman, jolta olisi ollut mahdollista sanoa ”liity”, vaikka käyttäjä kuuluisi jo johonkin konferenssiin, ”poistu” vaikkei käyttäjä kuuluisikaan konferenssiin ja lähettää sanomia kuulumatta konferenssiin. Tällaista vaatimuskonetta ei ehditty tässä yhteydessä rakentaa.

Taulukko 1: Testaustulokset

		Pelaaja		Ahne sat.		Sat.	
No	Poikkeava käyttäytyminen	1	2	1	2	1	2
0	(toimivaksi tarkoitettu versio)	–	–	–	–	–	–
10	Ei käsittele käyttäjän liittymispyyntöä	26	20	20	22	72	16
11	Ei käsittele käyttäjän lähetyspyyntöä	32	26	54	37	28	18
12	Ei käsittele käyttäjän poistumispyyntöä	62	62	55	31	45	48
13	Ei käsittele vastaanotettua sanomaa *liity*	174	26	38	30	126	31
14	Ei käsittele vastaanotettua sanomaa *data*	32	26	27	18	258	90
15	Ei käsittele vastaanotettua sanomaa *kuittaa*	26	26	197	21	29	52
16	Ei käsittele vastaanotettua sanomaa *poistu*	–	–	–	–	–	–
17	Ei lähetä *kuittaa* saatuaan *liity*	146	26	118	37	157	90
18	Ei lähetä *liity* saatuaan *data*[1]	–	–	–	–	–	–
19	Ei päivitä osallistujia saatuaan *kuittaa*	26	26	50	24	99	38
20	Ei päivitä osallistujia saatuaan *poistu*	–	–	–	–	–	–
21	Ei lähetä *liity* liityttäessä	26	26	48	29	241	50
22	Ei lähetä *data* lähetyspyynnössä	32	26	60	37	51	25
23	Ei lähetä *poistu* poistuttaessa[1]	–	–	–	–	–	–
24	Ei jätä konferenssia poistuttaessa	62	56	74	62	320	129
25	Ei tyhjennä osallistujia poistuttaessa	–	–	–	–	–	–
26	Ei liity konferenssiin liittymispyynnössä	26	20	22	16	55	53
27	Ei päivitä osallistujia saatuaan *liity*	261	26	17	19	38	47
28	Ei välitä viestiä saatuaan *data*	32	26	43	40	180	25
29	Poistaa 1. osallistujan saatuaan *poistu*[1]	1784	1769	381	558	351	213
30	Käsittelee *poistu* riippumatta konferenssista	–	–	–	–	–	–
31	Ei tarkasta osallistujia saatuaan *kuittaa*	–	–	–	–	–	–
32	Ei tarkasta pot. osallistujia saatuaan *kuittaa*	–	–	–	–	–	–
33	Ei tarkasta konferenssia saatuaan *kuittaa*	–	–	–	–	–	–
34	Ei tark. osal. saatuaan *data*, ei välitä viestiä[1]	32	26	23	43	431	29
35	Ei tarkista osallistujia saatuaan *liity*	283	26	34	16	47	114
36	Ei tarkasta pot. osallistujia saatuaan *liity*	–	–	–	–	–	–
37	Ei tarkasta konferenssia saatuaan *liity*	112	110	30	33	53	32
38	Poistuu konferenssista heti liityttyään	26	20	40	64	99	17
39	Lähettää kahdesti *liity* liityttäessä	–	–	–	–	–	–
40	Lähettää kahdesti *data* viestittäessä[2]	7095	7708	–	–	8657	47
41	Lähettää kahdesti *poistu* poistuttaessa	–	–	–	–	–	–
42	Lähettää aina *liity* saatuaan *data*	–	–	–	–	–	–
43	Käsittelee käyttäjän nimen väärin	164	29	306	118	85	76
44	Asettaa saapuneen viestin pituudeksi 1500	–	–	–	–	–	–
45	Lähettää *data* takaisin lähettäjälleen	31	21	16	34	178	35
50	Lähettää kahdesti *kuittaa* saatuaan *liity*	–	–	–	–	–	–
51	Lähettää kahdesti *liity* saatuaan *data*	–	–	–	–	–	–
52	Käsittelee sanomia ilman konferenssiakin	26	20	33	19	27	37
53	Ei ymmärrä vastaanottamiaan UDP-paketteja	26	20	26	21	35	19
54	Ei ymmärrä käyttäjältä saamiaan komentoja	26	20	29	15	43	18
55	Saa käyttäjältä vain *kuittaa* komentoja	26	20	33	24	97	32
56	Ei voi kuunnella UDP-porttia (bind error)	26	20	22	24	41	42
57	Saa heti käynnistyttyään QUIT-signaalin	13	9	11	9	17	10
58	Puhdistaa aina lähetysvälimuistin	–	–	–	–	–	–
59	Komentoriviparametrein ei voi mutatoida	–	–	–	–	–	–
60	Käyttäjää ja konferenssia ei nimetä	26	20	41	15	39	21

5.2 Havaittuja virhetilanteita

Jouduimme lisäämään ahneeseen ja satunnaiseen valinta-algoritmiin ylimääräisen viiveen kahdesta syystä. Kun syötteitä annettiin testikohteelle nopeassa tahdissa, protokollan toteutuksen käyttämät UDP-kanavat alkoivat hukata sanomia. Tästä seurasi toisinaan käyttäjien toisilleen lähettämien viestien katoaminen ja siten poikkeaminen vaatimuskoneen käyttäytymisestä.

Toisena syynä oli tapaus, jossa käyttäjä *A* lähettää viestin konferenssiin *k*, johon ei kuulunut käyttäjä *B*. Tällöin vaadimme, että käyttäjä *B* ei voi vastaanottaa tätä viestiä. Kuitenkin käyttäjän *B* liittyessä konferenssiin *k* liian nopeasti toteutus mahdollistaa viestin vastaanoton. Näin käy esimerkiksi jos *A*:n protokollaprosessi ei saa ajoaikaa ennen *B*:n liittymistä konferenssiin, jolloin todellinen viestien lähetysjärjestys onkin eri kuin vaatimuskoneen olettama.

Lähes kaikki mutantit paljastuivat hyvin yksinkertaisella testillä, koska niiden takia viestejä ei joko lähetetty tai vastaanotettu. Tällaiset virheet olisi voitu löytää pienellä käsin kirjoitetulla vaatimuskoneellakin. Kuitenkin testattavana oli myös mutantti, jonka virheellisyyden paljastavaa käyttäytymistä ei välttämättä huomata sisällyttää käsin kirjoitettuun vaatimuskoneeseen tai testitapauksiin. Nimittäin mutantin 29 sisältämä virhe havaitaan vain tilanteessa, missä aluksi kaikki asemat kuuluvat samaan konferenssiin. Tämän jälkeen konferenssista poistuu asema, joka ei satu olemaan konferenssiin jäävän mutantin sisäisessä osallistujat-taulukossa ensimmäisenä. Mutaation takia kuitenkin ensimmäinen taulukon asema poistetaan. Lopuksi konferenssiin jääneelle mutantille lähetetään viesti taulukosta poistetulta, mutta konferenssiin jääneeltä asemalta. Tällöin viesti ei välity perille ja virhe havaitaan.

Mutantin 40 paljastamiseen tarvittua suurta askelmäärää yksinkertaisesta virheestä huolimatta selittää se, että protokollan toteutus lähettää sanoman kahdesti erittäin nopeasti. Vaikuttaa siltä, että useimmiten toinen kahdennetuista sanomista hukkui UDP-kanavaan. Tämä on esimerkki harvoin ilmenevästä virheestä, jota olisi tuskin löydetty ajamalla lyhyitä testitapauksia.

Tuloksissa esitetyistä virheen löytämiseen vaadituista askelmääristä ei voida tehdä yleisiä johtopäätöksiä valintaheuristiikkojen nopeudesta, koska Pelaaja-heuristiikka on deterministinen ja monet virheet aiheuttivat aina samanlaisen virheellisen käyttäytymisen. Tästä syystä samat askelmäärät esiintyvät hyvin monta kertaa Pelaaja-heuristiikan sarakkeessa.

6 Yhteenveto

Testasimme samoilutestausmenetelmällä monen käyttäjän keskustelupalvelua.

Suurin osa paljastuneista virheistä olisi voitu löytää tavanomaisin testausmenetelmin. Useimmiten paljastamiseen olisi riittänyt kahden aseman liittäminen samaan konferenssiin ja sitten viestin lähettäminen toiselta asemalta. Kuitenkin testattavana oli ainakin kaksi mutanttia, joiden virheiden löytäminen tavanomaisin menetelmin on epätodennäköistä.

Vaatimuskoneen rakentaminen mallintamalla voi olla työlästä, mutta työ palkitaan erittäin monipuolisella vaatimuskoneella. Luultavasti emme olisi havainneet kaikkia nyt löydettyjä virheitä ellemme olisi käyttäneet toimivan järjestelmän malliin pohjautuvaa vaatimuskonetta.

Toisaalta, kun vaatimuskone on tehty tällaisesta mallista, järjestelmää testataan vain niin kuin sitä on suunniteltu käytet-

täväksi. Ajamissamme testeissä yksikään asema ei esimerkiksi yrittänyt häiritä muiden toimintaa, eikä yksikään asema ottanut käyttäjältään vastaan vääriä komentoja. Kelvottomien syötteiden testaaminenkin on mahdollista, mutta niitä varten tarvitaan erilainen vaatimuskone.

Testikohteen epädeterministisyys ei haittaa testiajoja samoilutestauksessa. Sopivalla vaatimuskoneella ja testikohteella testiä voidaan ajaa keskeytyksettä riippumatta siitä, miten testikohde reagoi syötteisiin. Tilanne oli tämä myös testiajoissamme: jos virhettä ei löytynyt, testin aikana suoritettu syöte- ja vastejono oli aina 10 000 askelta pitkä.

Testisuoritusten pituus onkin eräs samoilutestauksen eduista testitapauksiin nähden: testikohdetta ei tarvitse välillä käynnistää uudelleen tai ajaa väkisin tilaan, josta seuraava testitapaus voidaan aloittaa. Näin saatetaan paljastaa virheitä, joita testitapausten suunnittelijat eivät osanneet odottaa. Pitkissä katkottomissa testiajoissa voi paljastua myös muistivuotoja ja laskurien tai pinojen yli- ja alivuotoja.

Käytimme samoilutestauksessa satunnaisen valinnan lisäksi valintaheuristiikkaa, joka pyrki suorittamaan mahdollisimman paljon aiemmin testaamattomia tapahtumia mahdollisimman aikaisessa vaiheessa (Pelaaja-heuristiikka). Tavoitteenamme oli paljastaa mahdollisimman nopeasti virheitä, joissa tietyissä vaatimuskoneen solmuissa ollessaan järjestelmä vastaa aina väärin. Käyttämämme valintaheuristiikka löysikin virheet useimmiten nopeasti ja ainoana heuristiikkana havaitsi virheen jokaisesta testikohteesta, josta pystyimme ylipäätään virheen löytämään.

Valintaheuristiikkaa muuttamalla testiajoissa voitaisiin pyrkiä toistamaan vasta läpi käytyjä lyhyitä silmukoita lukuisia kertoja ennen kuin vaatimuskoneessa edetään uusille alueille. Tällöin saatetaan edellistä helpommin havaita esimerkiksi muistivuotoja. Mikä sitten on soveltuvin valintaheuristiikka, riippuu siitä, millaisia virheitä halutaan ensisijaisesti etsiä ja millaisia asioita halutaan keskittyä testaamaan.

Myös vaatimuskoneiden matematiikka tarjoaa tehokkaan työkalun testien ohjaukseen. Tässä artikkelissa esitetyissä testiajoissamme emme tosin hyödyntäneet muuta kuin rajoitinta, joka jätti joitakin osia käyttäytymisestä testauksen ulkopuolelle. Tämän lisäksi voitaisiin esimerkiksi painottaa tiettyjen tapahtumasarjojen testausta.

Kiitokset Ari Korhoselle ja Antti Valmarille hyvistä kommenteista. Testaustyö tehtiin TEKESin, Conformiq Software Oy Ltd:n ja Nokian tutkimuskeskuksen rahoittamassa SASOKE-projektissa.

Viitteet

[1] Belinfante, A., Feenstra, J., de Vries, R. G., Tretmans, J., Goga, N., Feijs, L., Mauw, S. & Heerink, L.: ”Formal Test Automation: A Simple Experiment”. *International Workshop on Testing of Communication Systems*, Kluwer Academic, 1999, sivut 179–196.

[2] Chow, T. S.: ”Testing Software Design Modeled by Finite-state Machines”. *IEEE Transactions on Software Engineering*, SE-4(3), 1978, sivut 178–187.

[3] Conference Protocol Case Study, http://fmt.cs.utwente.nl/ConfCase/. Viimeksi päivittänyt Jan Feenstra, 19.11.1999.

[4] Virtanen, H., Hansen, H., Valmari, A., Nieminen, J. & Erkkilä, T.: ”Tampere Verification Tool”. *Proc. TACAS 2004, 10th International Conference on Tools and Algorithms for the Construction*

and Analysis of Systems, Lecture Notes in Computer Science 2988, Springer-Verlag 2004, sivut 153–157.

[5] Helovuo, J. & Leppänen, S.: ”Exploration Testing”. *Proc. ICACSD 2001, 2nd IEEE International Conference on Application of Concurrency to System Design*, IEEE Computer Society 2001, sivut 201–210.

[6] Kervinen, A. & Virolainen, P.: ”Heuristics for Faster Error Detection with Automated Black Box Testing”. *Proc. Model-Based Testing 2004* 27.–28.3. 2004 Barcelona, Espanja. Julkaistaan ENTCS-sarjassa.

[7] Lai, R.: ”A Survey of Communication Protocol Testing”. *The Journal of Systems and Software*, 62(1), Elsevier Science Inc. 2002, sivut 21–46.

[8] Naito, S. & Tsunoyama, M.: ”Fault Detection for Sequential Machines by Transition Tour”. *Proc. 11th International Symposium on Fault Tolerant Computer Systems*, IEEE, 1981, sivut 238–243.

[9] Pyhälä, T. & Heljanko, K.: ”Specification Coverage Aided Test Selection”. *Proc. ACSD’2003, Third International Conference on Application of Concurrency to System Design*, IEEE 2003, sivut 187–195.

[10] Valmari, A.: ”Composition and Abstraction”. *Modelling and Verification of Parallel Processes*, Lecture Notes in Computer Science 2067, Springer-Verlag 2001, sivut 58–99.

[11] Valmari, A.: ”The State Explosion Problem”. *Lectures on Petri Nets I: Basic Models*, Lecture Notes in Computer Science 1491, Springer-Verlag 1998, sivut 429–528.