TAMPEREEN TEKNILLINEN YLIOPISTO
TAMPERE UNIVERSITY OF TECHNOLOGY

JERI HAAPAVUO

TOIMINNALLISEN TESTAUSAUTOMAATION KEHITTÄMINEN INTEGRAATIORAJAPINNOILLE

Diplomityö

Tarkastaja: Prof. Hannu-Matti Järvinen
Tarkastaja ja aihe hyväksytty 1.3.2017

TIIVISTELMÄ

JERI HAAPAVUO: Toiminnallisen testausautomaation kehittäminen integraatiorajapinnoille
Tampereen teknillinen yliopisto
Diplomityö, 47 sivua
Elokuu 2017
Tietotekniikan koulutusohjelma
Pääaine: Ohjelmistotuotanto
Tarkastajat: Prof. Hannu-Matti Järvinen
Avainsanat: XML, mallipohjainen testaus, testausautomaatio

Laadunvarmistus on tärkeä osa ohjelmistokehitysprosessia. Ohjelmistojen testausmenetelmiä kehittämällä voidaan parantaa laadunvarmistusprosessia merkittävästi. Toteuttamalla testausautomaatio on mahdollista tuottaa rakenteisia testiskenaarioita automaattisesti ja suorittaa niitä testattavaan järjestelmään.

Tässä työssä yritykselle luodaan testausautomaatioratkaisu, joka kykenee suorittamaan ennalta määrättyjä testejä testattavaan järjestelmään ja validoimaan järjestelmän palautteet automaattisesti. Ratkaisu kykenee tulkitsemaan XML-muotoisia testisekvenssirakenteita, joissa on myös validointiehdot jokaiselle testitapaukselle.

Työn ratkaisussa yksittäisiä testitapauksia on mahdollista kirjoittaa manuaalisesti XML-muodossa tai antaa testausautomaatiotyökalun generoida testitapauksia ennalta määrätyn mallin pohjalta. Työssä käydään läpi toteutuksessa sovellettuja testausmenetelmiä, kuten mallipohjaista testausta. Lisäksi työssä esitellään toteutuksen aikana havaittuja mallipohjaisuuden etuja ja haittoja perinteisiin testausmenetelmiin verrattuna.

Työn tuloksena luotiin testausautomaatiotyökalu ja toiminnallinen testausautomaatio Atostek Oy:n eRA-järjestelmän kolmelle integraatiorajapintaversiolle. Testausautomaatiotyökalu pyrittiin luomaan modulaariseksi ja siten soveltuvaksi myös muiden järjestelmien tarpeisiin. Testausautomaatio mahdollisti testaamisen huomattavasti laajemmilla arvojoukoilla kuin manuaalinen testaus sen aiheuttaman ajansäästön ansiosta.

ABSTRACT

JERI HAAPAVUO: Implementing Functional Test Automation for Integration Interfaces
Tampere University of Technology
Master of Science Thesis, 47 pages
August 2017
Master's Degree Programme in Information Technology
Major: Pervasive Systems
Examiner: Prof. Hannu-Matti Järvinen
Keywords: XML, model-based testing, test automation

Quality assurance is an important part of the software development process. The quality assurance process can be significally improved by developing the software testing methods. Test automation allows to repeat test sequences to the system under test (SUT) but may also be able to generate the test sequences based on a predefined model.

In this Master of Science thesis, a test automation solution is created that can run predefined tests to the SUT and validate the responses of the SUT. The solution is able to parse test sequences in XML format that also contain the validation rules for each test case.

In this solution, the individual test cases can be written manually in XML format or they can be generated automatically by the test automation tool. This thesis covers some testing methods that are applied to the test automation solution, such as model-based testing (MBT). In addition, the thesis aims to clarify the pros and cons of MBT versus the traditional testing methods.

As the results of this master's thesis, a test automation tool and a functional test automation for three integration interface versions of Atostek Oy eRA system was created. The test automation tool was designed modularity in mind so it could also be used to test other systems as well. Test automation enabled testing using larger value ranges than manual testing.

ALKUSANAT

Haluan kiittää työni tarkastajaa Hannu-Matti Järvistä ja ohjaajaa Miika Järvistä rakentavasta palautteesta. Kiitän Atostek Oy:tä työni aiheen hyväksymisestä ja työni rahoittamisesta. Lisäksi kiitän Atostek Oy:n johtoa ja erityisesti eRA-projektipäällikköä Ville-Pekka Örniä joustamisesta aikataulujen suhteen diplomityöni puolesta. Kiitos myös työtovereille, jotka olette rakentaneet hyvää työilmapiiriä ja jakaneet omia kokemuksianne diplomitöihinne liittyen!

Tampereella 26.7.2017

Jeri Haapavuo

SISÄLLYS

1. Johdanto 1

2. Sovellettavat testausmenetelmät 4

2.1 Manuaalisesti kirjoitetut testiskenaariot 6

2.2 Mallipohjainen testaus 8

2.2.1 Testitapausten tuottaminen mallipohjaisesti 9

2.2.2 Täysin mallipohjainen testaus 10

2.3 Valmiit testaustyökalut ja -kehykset 13

2.3.1 GraphWalker 14

2.3.2 vREST 15

2.3.3 NModel 15

2.3.4 Robot Framework 16

3. Testattava järjestelmä: eRA 18

3.1 Web-integraatiorajapinnan tiedonsiirtokerros 19

3.2 Rajapinnan komennot ja rakenteet 20

3.3 Käyttäjän tunnistaminen 21

3.4 Järjestelmähierarkia 22

3.5 Käyttöoikeudet 23

3.6 Koodistot 24

4. Testauksen suunnittelu ja toteutus 25

4.1 Testaustyökalun runko 26

4.2 eRA-järjestelmään toteutettavat testitoiminnot 27

4.3 Menetelmä 1: Testiskenaarioiden kirjoittaminen manuaalisesti 28

4.3.1 Arvoviittaukset 31

4.3.2 Konfiguraatiorakenteet 33

4.3.3 Validointirakenteet 35

4.4 Menetelmä 2: Testitapausten mallipohjainen generointi 36

4.5 Testauksen kattavuuden mittaaminen 39

4.6 Regressiotestaus 40

4.7 Testauskontekstin vaihtaminen 40

4.8 Jatkokehitysmahdollisuudet 42

5. Yhteenveto 43

Lähteet 46

KUVALUETTELO

2.1 Kaavio toteutuksesta, jossa testioraakkeli validoi testien tulokset testattavan järjestelmän referenssitoteutuksen palautteen perusteella. . . 6

2.2 Esimerkki testausautomaation rakenteesta, jossa testisekvenssit luodaan manuaalisesti järjestelmän dokumentaation pohjalta. 7

2.3 Esimerkki testiskenaarion komentojonosta. Virheen ilmentyessä ajoa ei tarvitse keskeyttää. 10

2.4 Yksinkertaistettu kaavio mallipohjaisen testauksen työnkulusta. . . . 11

2.5 Yksinkertaistettu esimerkki eRA-järjestelmän reseptiominaisuuksien testaamiseen käytettävän tilakoneen mallista. 13

3.1 Atostek eRA tarjoaa rajapinnan sosiaali- ja terveydenhuollon tietojärjestelmien liittämiseksi Kelan tarjoamiin Kanta-palveluihin. 18

3.2 eRA-järjestelmän rajapintaversioiden komentojen ja tietorakenteiden määrä. 20

3.3 eRA-järjestelmän liittyjien hierarkia. 22

OHJELMALUETTELO

2.1 Esimerkki yksinkertaisesta testausmallista toteutettuna NModelin tarjoamia attribuutteja hyödyntäen. 16

2.2 Esimerkki Robot Frameworkille määritellystä testiskenaariosta. 17

4.1 Yksinkertaistettu esimerkki testiskenaarion rakenteesta. Esimerkissä asetetaan potilaan tiedot testiohjelman tietomalliin ja lähetetään eRA-järjestelmälle pyyntö avata potilas. eRA-järjestelmän paluuviestin HTTP-tilakoodin odotusarvo on 200 (HTTP OK). 29

4.2 Esimerkki testiskenaariosta, jossa hyödynnetään alisekvenssirakenteita sisäänkirjautumiseen ja potilaan avaamiseen. 30

4.3 Esimerkki testiskenaariosta, jossa asetetaan muuttujiin dynaamisia aikaleimoja ja viitataan kyseisiin muuttujiin testitapauksen validointivaiheessa. Näkyvissä olevaa testiskenaariota on typistetty. 32

4.4 Testaustyökalun konfiguraatiorakenne, joka sisältää testattavan palvelimen osoitteen ja listan testauksessa käytettävistä käyttäjistä, asiakasjärjestelmistä ja toimipisteistä. 34

4.5 Esimerkki testiskenaarion osana olevasta validointirakenteesta. Esimerkissä tarkistetaan eRA-järjestelmältä lähettämättömien lääkemääräysten määrä, jonka odotusarvo on 0. HTTP-tilakoodin odotusarvo on 200 (HTTP OK). 36

4.6 Esimerkki mikrobiviljelyn tyyppikoodin testauksesta mallipohjaisella rakenteella. Mikrobiviljely voi olla osa laboratoriotutkimusta, joka puolestaan on yksi monista arkistokirjaustyypeistä. 37

4.7 Esimerkki mallipohjaisuuden hyödyntämisestä manuaalisesti kirjoitetussa testiskenaariossa. Testiskenaariossa kirjaudutaan sisään, avataan potilas, luodaan potilaalle mallipohjaisesti generoituja merkintöjä, suljetaan potilas ja kirjaudutaan ulos. Näkyvissä olevaa rakennetta on typistetty. 38

4.8 Testauskontekstin vaihtaminen. 41

LYHENTEET JA MERKINNÄT

.NET-sovelluskehys	Microsoftin kehittämä ohjelmistokehys.
ASP.NET	Microsoftin kehittämä avoimen lähdekoodin ohjelmistokehys, joka on tarkoitettu verkkosovellusten kehittämiseen.
DDT	Data-driven testing. Tietovetoinen testausmenetelmä, jossa määritellään joukko syötteitä, jotka ajetaan testattavaan järjestelmään yksitellen samaa testauslogiikkaa hyödyntäen.
HL7 V3	Health Level Seven -organisaation toteuttama terveydenhuollon standardi.
HTTP	Hypertext Transfer Protocol. Tiedonsiirtoprotokolla.
IIS	Internet Information Server. Microsoftin kehittämä palvelinohjelmisto.
JSON	JavaScript Object Notation. Avoimen standardin tiedostomuoto.
Kohdejärjestelmä	Testauksen kohteena oleva järjestelmä. Katso SUT.
SHA-256	Secure Hash Algorithm. Kryptograafinen tiivistefunktio.
SUT	System Under Test. Testauksen kohteena oleva järjestelmä. Työssä myös kohdejärjestelmä.
TDD	Test-driven development. Testivetoinen kehitys. Tekniikka, jossa luodaan ensin testitapaus ja vasta sen jälkeen toteutetaan järjestelmään logiikka, joka läpäisee testitapauksen.
Testaustyökalu	Testausautomaatiotyökalu. Voi vastata mm. testien generoimisesta ja niiden ajamisesta kohdejärjestelmään.
TLS	Transport Layer Security. Salausprotokolla internet-tietoliikenteen salaamiseksi.
WPF	Windows Presentation Foundation. .NET-sovelluskehyksen osana oleva kirjasto, joka muodostaa graafisen rajapinnan Windows-käyttöjärjestelmille.
XML	Extensible Markup Language. Rakenteellinen kuvauskieli.

1. JOHDANTO

Yrityksillä on jatkuva tarve tuottaa luotettavia ja laadukkaita järjestelmiä mahdollisimman pienillä kehityskustannuksilla. Yksi suuri aikaa ja rahaa vievä osa ohjelmistokehityksessä on ohjelmistojen testaus ja laadunvarmistus. Koska testauksen kohteena voi olla kriittisiä tietoja käsitteleviä ohjelmistoja, on erityisen tärkeää, ettei niiden keskeisimmissä toiminnoissa esiinny virheitä. Tämä tutkielma on tehty yritykselle, jonka tarpeena on kehittää toteuttamiensa järjestelmien integraatiorajapintojen testausmenetelmiä. Tässä tutkielmassa toteutetaan testausautomaatio Atostek Oy:n eRA-järjestelmän XML-pohjaiselle integraatiorajapinnalle, jonka kautta eRA-järjestelmään liittyneiden asiakasjärjestelmien käyttäjien on mahdollista käyttää eRA-järjestelmän tarjoamia toimintoja, joita ovat esimerkiksi potilaan lääkemääräysten ja arkistotietojen tallentaminen Kelan tarjoamiin Kanta-palveluihin [1].

Kohdejärjestelmän (engl. SUT, System Under Test) integraatiorajapinta toimii TLS-salatun HTTP-protokollan yli, joten testauskehyksen tulee huolehtia, että tietorakenteiden sarjallistaminen ja viestien salaaminen toimivat oikein. Testaus on suoritettava mustalaatikkotestauksena, joka puolestaan tarkoittaa, ettei testattavan järjestelmän sisäinen toteutus ole tiedossa. Tuolloin testaus tehdään täysin järjestelmälle laaditun dokumentaation pohjalta, jolloin erityisen tärkeää on dokumentaation kattavuus ja oikeellisuus. Mustalaatikkotestauksen voidaan ajatella toimivan myös dokumentaation oikeellisuuden tarkasteluun.

Ennen tutkielman aloittamista Atostek Oy on soveltanut eRA-järjestelmän kehitykseen staattista testausta katselmoiden järjestelmänsä lähdekoodia ja dokumentaatiota. Staattisella testauksella tarkoitetaan testausta, jossa järjestelmää ei suoriteta lainkaan, vaan perehdytään sen määrittelyihin ja toteutukseen manuaalisesti. Staattisen testauksen tavoitteena onkin mahdollisten ongelmien ennaltaehkäisy. Dynaamista tutkivaa testausta on puolestaan käytetty integraatiorajapintojen ja toiminnallisuuksien testaukseen. Dynaamisessa testauksessa syötteitä ajetaan ajossa olevaan järjestelmään ja järjestelmän tuottamat ulostulot validoidaan tai verifioidaan [2]. Atostek Oy on ennen työn toteutusta verifioinut testauksen tulokset silmämääräisesti, jonka seurauksena on voinut aiheutua virheellisiä tulkintoja.

Manuaalisesti tehtävä testaus on hidasta ja toisteista, eikä se suuren työmäärän vuoksi aina kata riittävää määrää mahdollisista poikkeustilanteista. Järjestelmään lähetettäviä HTTP-viestejä XML-tietosisältöineen on luotu manuaalisesti yksitellen, jolloin virheitä on voinut syntyä myös testaajan tekemänä. Testauksen tueksi on kehitetty simulaattoreita, joilla on mahdollista lähettää yksittäisiä rajapintakomentoja testattavaan järjestelmään käyttöliittymästä käsin.

Automatisoidussa testauksessa validointi jää testejä ajavan ohjelmiston vastuulle, jolloin testaajasta riippuvia virheellisiä tulkintoja ei pääse syntymään. Onkin tärkeää, että validointi on toteutettu testaustyökaluun oikein tulkinnoin.

Integraatiorajapintojen tapauksessa on tärkeää, ettei niiden olemassa oleviin osiin tehdä suuria muutoksia eri kehitysvaiheissa. Jos integraatiorajapintoja muutettaisiin järjestelmäpäivitysten yhteydessä, joutuisivat kaikki liittyneet asiakasjärjestelmät tekemään vastaavat muutokset omiin integraatioihinsa. Taaksepäinyhteensopivuus onkin usein vaatimus integraatiorajapintoja suunniteltaessa. Integraatiorajapintojen muuttumattoman luonteen vuoksi testauksessa voidaan käyttää uudelleen aiemmalle järjestelmäversiolle luotuja testiskenaarioita ja laajentaa niitä dokumentaation pohjalta tarvittaessa.

Web-sovellusten osalta voidaan toiminnallisen testauksen lisäksi testata muun muassa käyttöliittymää, tietoturvaa ja kuormansietokykyä. Vaikka tässä työssä testausta lähestytään lähinnä toiminnallisen testauksen näkökulmasta, voidaan työssä läpikäytyjä menetelmiä soveltaa myös muihin testauksen osa-alueisiin.

Työn tavoitteena on toteuttaa yrityksen toteuttamien järjestelmien toiminnallisen testauksen automatisointiin soveltuva ratkaisu yleisesti tunnettuja testausmenetelmiä soveltaen. Testausautomaation seurauksena yrityksen laadunvarmistukseen kuluttama aika vähenee ja testauksen laatu paranee.

Luvussa 2 kuvataan työssä sovellettavat testausmenetelmät. Testejä ajetaan dynaamisesti ajossa olevaan järjestelmään kuvattuja menetelmiä hyödyntäen. Testausmenetelmät eivät sulje toisiaan pois, vaan niitä voidaan käyttää samanaikaisesti testauksen helpottamiseksi.

Luvussa 3 esitellään testauksen kohteena oleva eRA-järjestelmä. Kyseessä on Atostek Oy:n tuote, jonka päätehtävänä on mahdollistaa sosiaali- ja terveydenhuollon toimintayksiköiden liittyminen Kelan tarjoamiin Kanta-palveluihin.

Luvussa 4 käydään läpi testausautomaatioon liittyvä suunnittelu ja sen toteutus kokonaisuudessaan. Toteutukseen sisältyy testauskehyksen rungon toteuttaminen,

testiskenaarioiden skriptauskielen tulkin toteuttaminen, testiskenaarioiden luominen sekä testien automatisoitu ajaminen ja validointi.

Luvussa 5 tehdään yhteenveto työn vaiheista ja tuloksista sekä vertaillaan yrityksen aiemmin soveltamia testausmenetelmiä työn tuloksena sovellettuihin menetelmiin. Lisäksi arvioidaan toteutetun testauskehyksen yleiskäyttöisyyttä erilaisten järjestelmien ja rajapintojen testauksessa.

2. SOVELLETTAVAT TESTAUSMENETELMÄT

Ohjelmistojen testausmenetelmillä (engl. software testing methods) tarkoitetaan menetelmiä, joilla verifioidaan tai validoidaan järjestelmän toimivuus sen dokumentaation, kuten vaatimus- ja määrittelydokumenttien, mukaisesti [3]. Tässä työssä käytetyistä menetelmistä keskeisin on mustalaatikkotestaus, joka tarkoittaa ettei testattavan järjestelmän lähdekoodi ole tiedossa. Tuolloin testaus laaditaan täysin järjestelmälle laaditun dokumentaation pohjalta, jolloin varmistutaan samalla myös dokumentaation oikeellisuudesta.

Testausta suunniteltaessa on oleellista määrittää testauksen tavoitteet. Tavoitteista täytyy käydä ilmi mitä testataan, miten testataan ja mistä järjestelmän osista virheitä etsitään. Jos virheitä etsitään dokumentaatiosta, testit suunnitellaan dokumentaation pohjalta, mutta ajetaan oletusarvoisesti toiminnaltaan oikeellista järjestelmää vasten. Ristiriitatilanteissa tiedetään tuolloin, että virhe on dokumentaatiossa. Tilanne ei kuitenkaan käytännössä ole koskaan näin yksiselitteinen, sillä vikoja voi olla piilossa järjestelmän toteutuksessa, vaikka sitä luultaisiinkin oikein toimivaksi.

Järjestelmää on tärkeä testata syötteillä, jotka kattavat mahdollisimman suuren arvojoukon siten, että erityisesti erilaiset rajatapaukset tulevat testattua. Rajatapauksia voivat olla mm. syötteellä aiheutettu lukualueen ylittyminen tai syötteenä annetut null-arvot. Testitapauksia, joilla pyritään rikkomaan järjestelmä tai löytämään epäjohdonmukaisuuksia järjestelmän määrittelydokumentaatiosta, kutsutaan negatiivisiksi testitapauksiksi. Positiivisia testitapauksia puolestaan ovat ne testitapaukset, joilla testataan, että järjestelmän toiminta vastaa määrittelydokumentaatiossa esitettyä toimintaa. [4]

Testattaessa havaittuja vikoja korjattaessa uudelleentestausta helpottaa, että yhtä testiskenaariota voidaan toistaa tarpeen mukaan itsenäisesti. Jotta testiskenaarioiden ajojärjestyksellä ei olisi väliä, tulee testisekvenssit suunnitella toisistaan riippumattomiksi. Optimaalisessa tilanteessa yhden testiskenaarion voidaankin ajatella olevan silmukka, joka päättyy alkutilaansa. Testattavat järjestelmät saattavat kuitenkin tehdä testattavien toimintojen seurauksena merkintöjä tietokantaan, joiden poistaminen jokaisen testiskenaarion päätteeksi ei välttämättä ole tehokkuussyistä

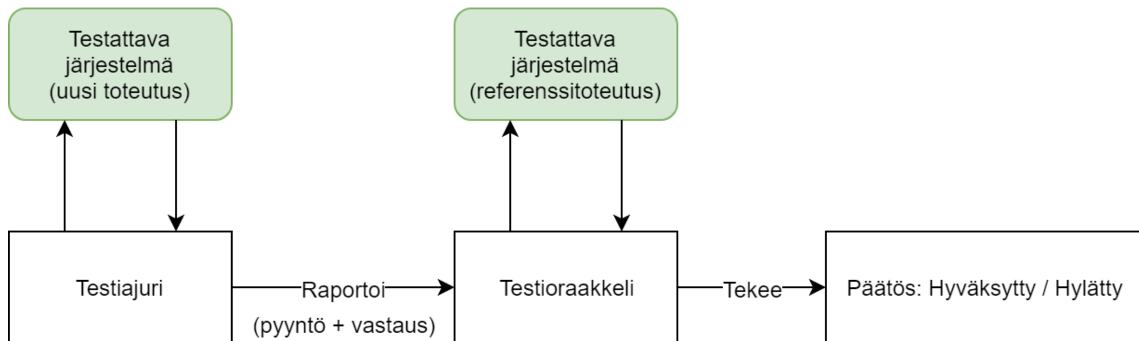
järkevää. Tuolloin tietokannan lopputila ei ole sama kuin alkutila, vaikka testattavan järjestelmän sisäinen tila olisikin muuten sama testiskenaarion alussa ja lopussa. Jos testiskenaariot suunnitellaan siten, etteivät ne riipu toisten testiskenaarioiden luomista tietokantamerkinnöistä, voidaan merkinnät jättää tietokantaan. Ratkaisu on toimiva, mutta uusia testiskenaarioita luotaessa täytyy olla erityisen tarkka, ettei uusia riippuvuussuhteita synny vanhoihin testiskenaarioihin.

Perinteisesti testausautomaatio on toteutettu toteuttamalla testiajuri, joka suorittaa sille määritellyt testisekvenssit tai testiskenaariot testattavaa järjestelmää vasten. Kun testiskenaariot määritellään manuaalisesti, tullaan samalla varmoiksi niiden sisällöstä ja niiden välisistä riippuvuussuhteista. Tämä on sekä hyvä että huono asia. Testaajan täytyy hyödyntää omaa kekseliäisyyttään saadakseen testattua kaikki tai edes valtaosan kaikista mahdollisista rajatapauksista, mutta toisaalta voidaan varmistua siitä, että ainakin mieleen tulleet rajatapaukset on katettu testeissä. Manuaalisesti tuotettuja testiskenaarioita käydään tarkemmin läpi kohdassa 2.1.

Koska testiskenaarioiden määritteleminen manuaalisesti aiheuttaa paljon työtä sekvenssien ja rakenteiden toisteisuuden vuoksi, on päädytty luomaan menetelmiä, joilla voidaan tuottaa testiskenaarioita ja yksittäisiä testitapauksia automaattisesti. Yksi tunnettu menetelmä testiskenaarioiden ja testitapausten luomiseen on mallipohjainen testaus (engl. Model-Based Testing, MBT), jossa luodaan testattavan järjestelmän toimintaa vastaava testausmalli. Tätä käsitellään kohdassa 2.2.

Osa testausautomaatiota on testien tulosten validointi. Testaustyökalun tai -kehyksen osaa, joka vastaa testien tulosten validoinnista, kutsutaan testioraakkeliksi. Se tekee yksiselitteisen päätöksen siitä, onko testin tulos oikeellinen vai virheellinen. Testioraakkeli ei kuitenkaan välttämättä aina ole oikeassa. Testioraakkelin päätösten oikeellisuus riippuu testioraakkelin sisäisestä toteutuksesta, mutta myös testausmallista, jonka perusteella testioraakkeli tekee päätökset. Testausmallia toteutettaessa on voitu tehdä virheellisiä johtopäätöksiä testattavan järjestelmän toiminnasta esimerkiksi järjestelmän määrittelydokumentteja luettaessa. Toisaalta testausmallina voi myös toimia testattavan järjestelmän referenssitoteutus, joka voi olla esimerkiksi järjestelmän oikein toimivaksi todettu vanhempi versio. Referenssitoteutuksen hyödyntäminen on toimiva validointimenetelmä erityisesti silloin, kun halutaan varmistua järjestelmän taaksepäin yhteensopivuudesta. Tätä esitellään kuvassa 2.1.

Sen sijaan, että testausmenetelmien mukaista työkalua toteutettaisiin itse, voidaan valita olemassa oleva valmis testaustyökalu. Valmiin työkalun avulla voidaan vähentää testausautomaation toteutukseen kuluvaa aikaa. Työkaluissa voi olla tuki sekä sekventiaaliselle että mallipohjaiselle testausmenetelmälle. Jos kuitenkin halutaan



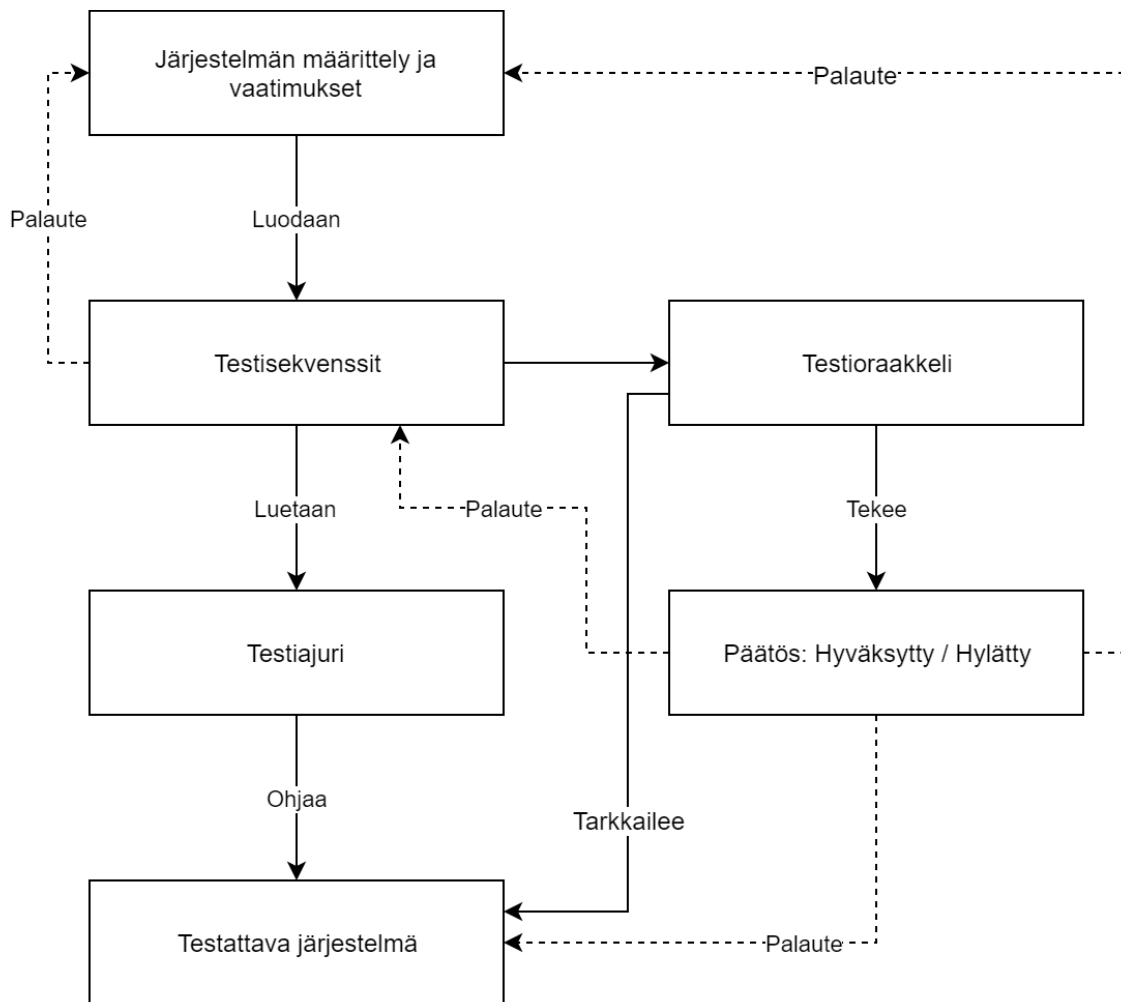
Kuva 2.1 *Kaavio toteutuksesta, jossa testioraakkeli validoi testien tulokset testattavan järjestelmän referenssitoteutuksen palautteen perusteella.*

vaikuttaa testausohjelmiston sisäiseen logiikkaan ja samalla hyödyntää mahdollisimman paljon jo olemassa olevaa ohjelmakoodia, voidaan valita valmiin testaustyökalun sijaan testauskehys osaksi itse toteutettavaa testaustyökalua. Muutamia olemassa olevia testaustyökaluja ja -kehyksiä käydään läpi kohdassa 2.3.

Testien koodikattavuutta ei ole mahdollista määrittää sovellettaessa mustalaatikkotestausta. Testauksen kattavuutta voidaan kuitenkin mitata esimerkiksi laskemalla testausmallin mukaisesti suoritetut ja suorittamattomat tilasiirtymät. Jotta mittaus antaisi oikeita tuloksia, tulee testausmalli olla toteutettu kattavasti koko järjestelmän osalta. Jos testausmallia ei ole toteutettu lainkaan, kattavuutta voi olla mahdotonta mitata tarkalla tasolla. Jos testaustyökalun tiedossa on kaikki ne testattavan järjestelmän tukemat komennot, joita halutaan testata, voidaan kattavuutta mitata vähintäänkin laskemalla, mitkä komennoista on suoritettu testiajon aikana vähintään kerran. Lisäksi voidaan pitää kirjaa komennoille syötetyistä parametreista.

2.1 Manuaalisesti kirjoitetut testiskenaariot

Yksinkertaisin tapa automatisoida testejä on kirjoittaa ennakkoon suunnitellut testiskenaariot suoritettaviksi komentosarjoiksi. Näin testiskenaarioiden testitapauksia ei tarvitse yksitellen manuaalisesti suorittaa testattavaa järjestelmää vasten, vaan voidaan hyödyntää testaustyökalun osaksi toteutettua testiajuria (engl. test driver). Testiajurin tehtävä on tulkita kirjoitettuja testiskenaarioita tai -sekvenssejä ja suorittaa niissä kuvatut testit kohdejärjestelmään. Kuvassa 2.2 on kuvattu testisekvenssejä lukevan testausautomaation rakenne. Testiajuri suorittaa järjestelmän määrittelyjen pohjalta laaditut testisekvenssit testattavaan järjestelmään ja testioraakkeli validoi kohdejärjestelmän palautteet. Testioraakkelin tekemät päätökset toimivat palautteena, josta ihmisen tulee osata tulkita mihin testausautomaation komponentteihin palaute liittyy. Epäonnistuneeseen testitapaukseen johtanut virhe voi olla missä tahansa kuvassa 2.2 esitellyistä komponenteista.



Kuva 2.2 *Esimerkki testausautomaation rakenteesta, jossa testisekvenssit luodaan manuaalisesti järjestelmän dokumentaation pohjalta.*

Manuaalisesti kirjoitettuja testiskenaarioita hyödyntävä testausautomaatio sisältää vähemmän ohjelmointityötä kuin mallipohjaista lähestymistapaa noudattava automaatio. Kun testitapausten järjestys ja testitapauksiin liittyvät validointiehdot on kirjoitettu valmiiksi testiskenaarioihin, ei tarvitse toteuttaa erillistä logiikkaa testiskenaarioiden tuottamiseen eikä validointiehtojen määrittämiseen.

Manuaalisesti kirjoitettujen testiskenaarioiden ylläpidettävyys on kuitenkin lähes aina huonompi kuin mallipohjaisessa testauksessa hyödynnetyn testausmallin ylläpidettävyys. Tämä johtuu siitä, että testiskenaarioihin kertyy usein toisteista tietoa, mutta testausmalliin tieto pyritään kirjoittamaan vain kerran. Testiskenaario voi esimerkiksi sisältää useamman testitapauksen, joissa testataan samaa komentoa useilla eri syötteillä.

2.2 Mallipohjainen testaus

Mallipohjaiseksi testaukseksi kutsutaan testausmenetelmää, jossa hyödynnetään testattavan kohdejärjestelmän määrittelydokumentaation pohjalta toteutettua järjestelmän mallia. Tuo malli on usein tilakonemainen ratkaisu, josta selviää yksiselitteisesti mihin tilaan kustakin tilasta voidaan siirtyä ja millä ehdoin.

Mallin ei kuitenkaan tarvitse toteuttaa järjestelmää täysivaltaisesti. Testaus voidaan tehdä osittain kohdassa 2.1 esiteltyjä manuaalisia menetelmiä käyttäen ja osittain mallipohjaisesti. Kohdassa 2.2.1 esitellään menetelmä, jossa testiskenaarion yksittäistä testitapausta toistetaan mallipohjaisesti, mutta skenaarion muu rakenne on määritelty manuaalisesti. Kohdassa 2.2.2 esitellään ratkaisu, jossa toteutetaan koko järjestelmän laajuinen tilakone, jolla voidaan tuottaa testiskenaarioita täysin automaattisesti.

Ennen mallipohjaisen testauksen toteuttamista on suositeltavaa tutustua testisekvenssien luomiseen manuaalisesti ja niiden suorittamiseen automatisoidusti. Näin varmistutaan, että testaajilla on tarvittava ymmärrys mallipohjaisen testausautomaation toteuttamiseksi. Samalla tullaan havainneeksi testauksessa ilmeneviä ongelmakohtia, jotka voidaan pyrkiä ratkaisemaan myöhemmin hyödyntämällä esimerkiksi mallipohjaista testausta. [5]

Mallipohjaisesti testaamalla voidaan havaita huomattava määrä dokumentaatiovirheitä verrattuna manuaaliseen testaamiseen. Oikein laadittu testausmalli mahdollistaa testaamisen suurillakin arvojoukoilla, jolloin erilaiset rajatapaukset tulevat myös suuremmalla todennäköisyydellä huomioiduiksi. Mallipohjaista testausta hyödyntämällä testaajan kekseliäisyyden merkitys testiskenaarioita mietittäessä poistuu ja tärkeämmäksi osoittautuu järjestelmän oikeellinen mallintaminen. Samalla testien ylläpidettävyys paranee, kun testiskenaarioita ei tarvitse päivittää manuaalisesti kohdejärjestelmän rajapintojen muuttuessa. [5]

Testausautomaation toteuttaminen on työläs ja aikaavievä prosessi, eikä se maksa itseään takaisin lyhyellä aikavälillä. Kuitenkin tilanteissa, joissa testejä tullaan ajamaan useita, jopa kymmeniä tai satoja kertoja uudestaan, osoittautuu testauksen automatisointi välttämättömäksi. Mitä aikaisemmassa vaiheessa testausautomaatio toteutetaan, sitä vähemmän työtä joudutaan toistamaan manuaalisesti.

Testisekvenssit voidaan tuottaa ajonaikaisesti samalla, kun testejä ajetaan järjestelmään. Tätä tilannetta, jossa mallipohjainen testaustyökalu on kytkeytyneenä ajossa olevaan järjestelmään ja tuottaa sen palautteen pohjalta lisää testejä, kutsutaan online-testaukseksi. Menetelmän hyöty sen vastakohtaan, offline-testaukseen,

on testaustyökalun kyky sopeutua testattavan järjestelmän tuottamaan palautteeseen. Online-testaus on määritelmän mukaisesti aina myös dynaamista testausta. Dynaaminen testaus tarkoittaa testien suorittamista ajossa olevaan järjestelmään, mutta ei määrittele kuinka testit tulee luoda, miten ja milloin testejä suoritetaan kohdejärjestelmää vasten tai sitä, kuinka validointi tai verifiointi tulee suorittaa. [5]

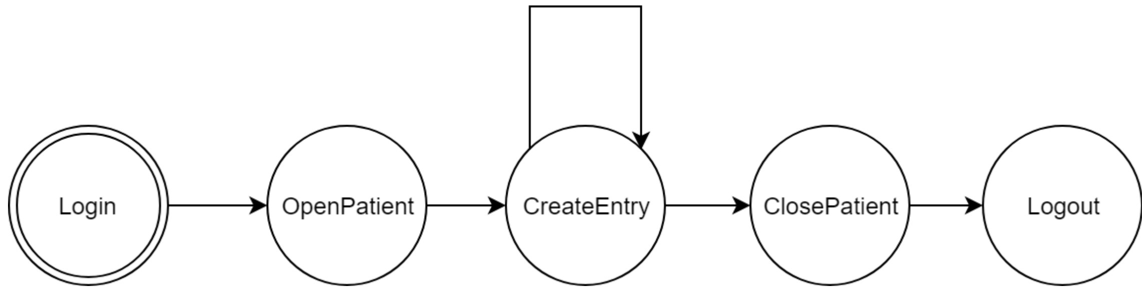
Jos testien tehokkuuden optimointi on tärkeää tai jos testisekvenssejä halutaan muokata vielä niiden tuottamisen jälkeen, voidaan testaus suorittaa offline-testauksena. Offline-testauksessa testisekvenssit tuotetaan tiedostoihin ja suoritetaan vasta myöhemmin testattavaa järjestelmää vasten. Offline-testisekvenssejä on lisäksi mahdollista suorittaa useampaan kertaan testattavaan järjestelmään esimerkiksi silloin, kun halutaan suorittaa regressiotestausta. Toisaalta mikään ei myöskään estä online-testauksessa ajettujen testisekvenssien tallentamista myöhempää toistoa varten. [5]

Testiskenaarioita tuotettaessa voidaan työkalulle antaa ohjeita ja rajoitteita, joilla voidaan varmistua haluttujen testitapausten ilmentymisestä skenaarioissa. Rajoitteiden asettaminen on erityisen hyödyllistä esimerkiksi silloin, kun arvoalueet ovat liian suuria testattavaksi järkevässä ajassa. Toisaalta voi olla tiettyjä tilasiirtymäpolkuja, joiden ilmentyminen on erittäin harvinaista. Kyseiset tilanteet on siitä huolimatta tärkeää saada testattua ja siksi niitä varten voidaankin määritellä erityisiä ohjeita testiskenaarioiden generointityökalulle.

2.2.1 Testitapausten tuottaminen mallipohjaisesti

Yksittäisen testitapauksen toistuva ajo voidaan suorittaa mallipohjaisesti eri avain-arvo-kombinaatioita tuottamalla. Tuolloin tulee olla määriteltynä testattavat arvot tai arvoalueet kaikille testatattaville kentille. Jokaiseen arvoon ja arvoalueeseen tulee myös liittyä tieto tilasta, johon järjestelmän arvon antamisen seurauksena kuuluisi päätyä. Integraatiorajapinnan tapauksessa tila voi tarkoittaa esimerkiksi paluuarvona saatavaa yksilöityä virhekoodia tai onnistunutta operaatiota testattavassa järjestelmässä. Menetelmä tunnetaan myös tietovetoisena testauksena (engl. Data-driven testing). Kuvassa 2.3 on esimerkki testiskenaariosta, jossa CreateEntry-komentoa ajetaan testattavaan järjestelmään mallipohjaisesti kaikilla määritellyillä avain-arvo-kombinaatioilla. Avain-arvo-kombinaatiot voivat aiheuttaa testattavassa järjestelmässä joko virhetilanteen tai onnistuneen operaation. Testioraakkelin tehtävänä on validoida järjestelmän reaktion oikeellisuus.

Osa kenttiin liittyvistä validointiehdoista voi riippua toisten kenttien arvoista. Kenttä voi esimerkiksi olla pakollinen vain, jos toisessa kentässä on jokin tietty arvo. Toisaalta voi olla myös kenttä, johon ei saa asettaa lainkaan arvoa, jos tiettyyn toiseen



Kuva 2.3 *Esimerkki testiskenaarion komentojonosta. Virheen ilmentyessä ajoa ei tarvitse keskeyttää.*

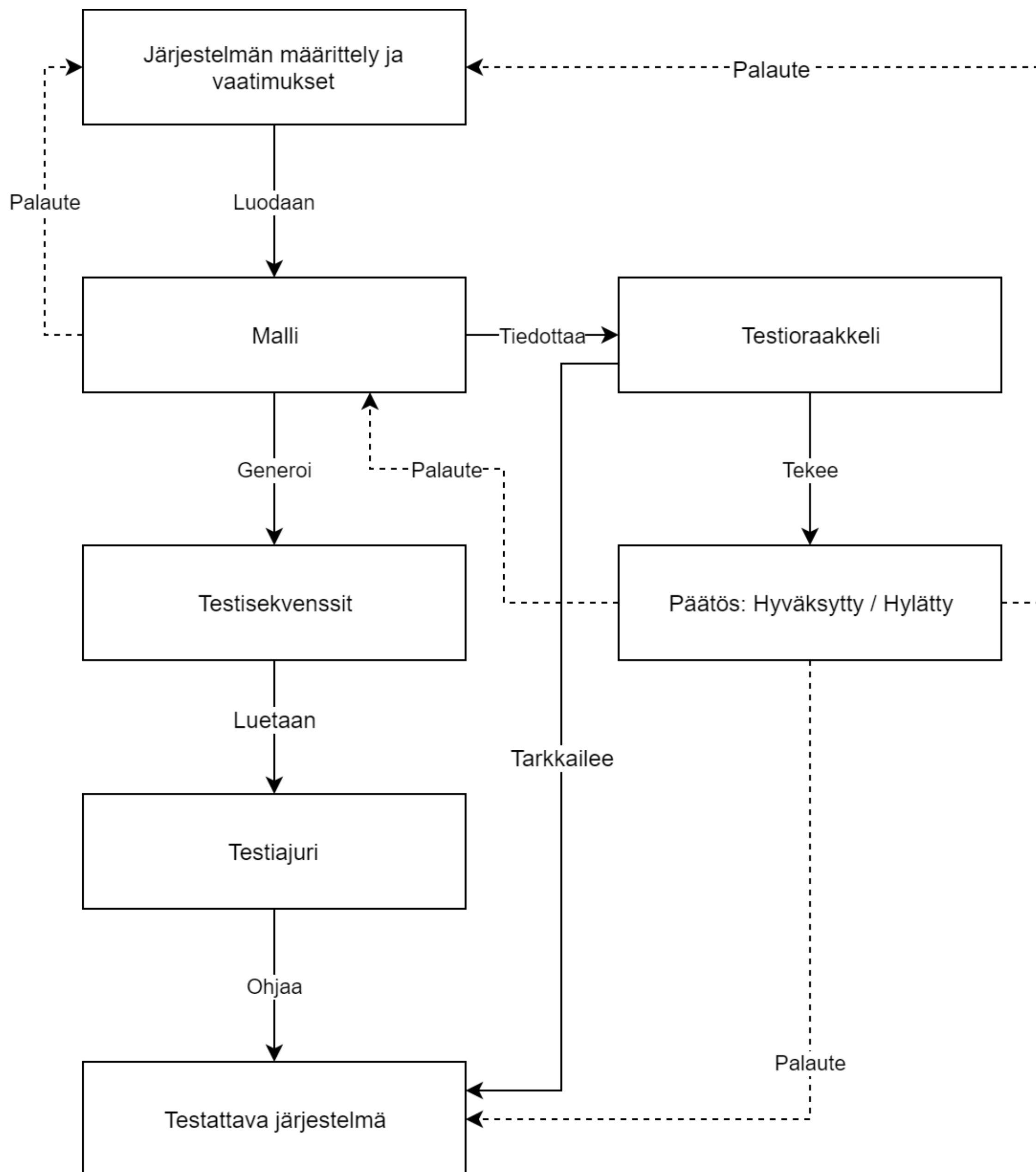
kenttään on asetettu jokin tietty arvo. Testauksessa on tärkeää huomioida kenttien väliset ehdolliset riippuvuudet, jotta testien tulosten validointi voidaan suorittaa oikein. Tämä tekeekin kaikissa tilanteissa oikein toimivan testausmallin ja -oraakkelin toteutuksesta haasteellista.

Edellä mainittujen ongelmien lisäksi järjestelmän toimintaa kuvaava malli voi olla epädeterministinen eli mallin kuvaamista tiloista voi olla mahdollista päätyä useampaan kuin yhteen tilaan. Tällaisia tilanteita voi aiheutua muun muassa silloin, kun toinen käyttäjä tai järjestelmä on lukinnut pyydetyn resurssin testiajurin ajamien komentojen välillä. Epädeterministisyydestä voidaan päästä eroon jopa kokonaan luomalla suljettu testiympäristö, johon ulkopuolisilla tekijöillä, kuten muilla käyttäjillä, ei ole pääsyä. Aina kuitenkaan epädeterministisyyden aiheuttavia tekijöitä ei ole mahdollista sulkea pois, jolloin epädeterministisyys joudutaan ottamaan huomioon testiautomaatiota suunniteltaessa.

Testausmalliin ei yksittäisten testitapausten mallipohjaista testausta varten tarvitse luoda tilakonetta testattavan järjestelmän toiminnasta kokonaisuudessaan, vaan riittää, että mallinnetaan järjestelmän toiminta komentokohtaisesti. Lisäksi tulee tunnistaa tilanteet, joissa kyseistä komentoa ei enää voida suorittaa toistuvasti järjestelmässä.

2.2.2 Täysin mallipohjainen testaus

Sen sijaan, että testiskenaariot luotaisiin manuaalisesti, voidaan ne tuottaa automaattisesti esimerkiksi järjestelmää mallintavaa tilakonetta hyödyntämällä. Mallipohjaisen testauksen työnkulkua on esitelty kuvassa 2.4. Tilakoneen täytyy tietää kohdejärjestelmän alkutila ja eri tiloissa mahdolliset tilasiirtymät seurauksineen. Riittää, että järjestelmän tiloista tiedetään ne asiat, jotka vaikuttavat testauksen kulkuun. Tämä tarkoittaa käytännössä kaikkia niitä asioita, joista järjestelmän tilasiirtymät ovat riippuvaisia. On kuitenkin huomioitavaa, että testattavan järjestel-



***Kuva 2.4** Yksinkertaistettu kaavio mallipohjaisen testauksen työnkulusta.*

män toiminta voi näkyä ulospäin epädeterministisenä. Järjestelmän epädeterministisyys testausautomaation kannalta luo tilakoneeseen haarautuvia polkuja eikä aina voida tietää missä tilassa testattava järjestelmä milläkin hetkellä on.

Testejä, jotka järjestelmän epädeterministisen toiminnan johdosta välillä onnistuvat ja välillä epäonnistuvat, kutsutaan epädeterministisiksi testeiksi. Testaajalle epädeterministien testien tulokset voivat näyttää satunnaisilta ja niistä voi olla hyvin vaikeaa tai mahdotonta yrittää päätellä käyttäytyykö testattava järjestelmä virheellisesti. Testausautomaation pääasiallinen tehtävä on nopeuttaa vikojen löytämistä testattavasta järjestelmästä, mutta epädeterministiset testit toimivat tätä tehtävää

vastoin. Siksi epädeterminististä testeistä tulee pyrkiä eroon, jotta testausautomaatiosta saatava hyöty ei kärsisi.

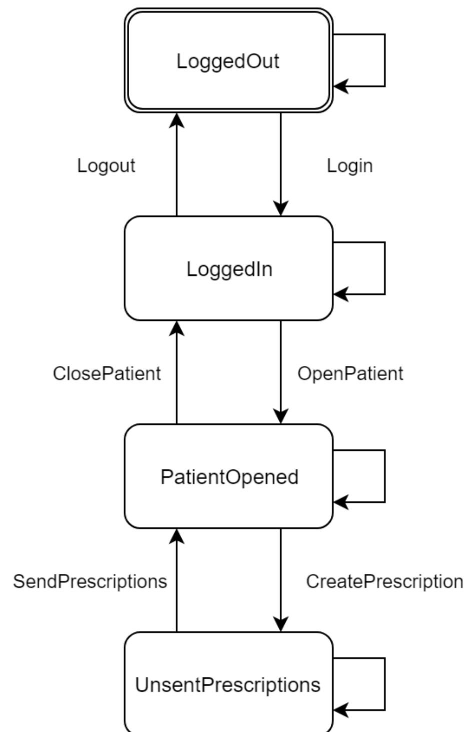
Epädeterministisen toiminnan aiheuttama tila on mahdollista selvittää kysymällä tilaa eksplisiittisesti testattavalta järjestelmältä, mikäli kysyminen on mahdollista. Jos tilatietoa ei ole saatavilla, testausmallin on tehtävä arvaus ja jatkettava testien suorittamista arvauksen pohjalta. Jos arvaus ei ole oikea, testioraakkeli voi päätyä tekemään vääriä johtopäätöksiä loppujen testien oikeellisuudesta. Yksittäisen testitapauksen epädeterministisyys voikin näin tehdä kokonaisesta testiskenaariosta epädeterministisen.[6]

Mallipohjaista testausta visualisoitaessa testausmalli voidaan esittää graafimuotoisena tilakoneena. Tilakoneesta käy ilmi mahdolliset tilasiirtymät järjestelmän tilojen välillä ja mahdollisesti myös tilasiirtymiin liittyvät ehdot. Graafia voidaan käyttää apuna testausmallin analysoinnissa, jossa voidaan löytää testausmallissa esiintyviä ongelmakohtia ja optimointimahdollisuuksia. Tilakoneita tai graafeja voi olla useita ja niiden välillä voi olla linkkejä toisiinsa. Yhden suuren tilakoneen pilkkominen useaksi pienemmäksi tilakoneeksi voi selkeyttää malliin liittyvää rakennetta ja helpottaa siinä olevien virheiden paikantamista.

Jotta testiajo saadaan mahdollisimman optimaaliseksi ja kattavaksi, voidaan hyödyntää kiinalaisen postimiehen ongelman (engl. CPP, Chinese Postman Problem) ratkaisevaa algoritmia [7]. Kiinalaisen postimiehen ongelmaksi kutsutaan optimointiongelmaa, jossa tulee kulkea graafin kaikkien kaarien läpi vähintään kerran. Tämä tarkoittaa mallipohjaisessa testauksessa tilakoneen kaikkien tilasiirtymien suorittamista vähintään kerran.

Aina ei kuitenkaan riitä, että kaikki tilasiirtymät tulee ajettua tasan kerran. Tilasiirtymäkohtaisia ehtomuuttujia voi olla useita, tai jopa useita kymmeniä. Tuolloin tilasiirtymään liittyvien ehtojen hyväksyttävien arvojen kombinaatioita voi olla satoja eikä ole järkevää tehdä tilakonetta, jossa on satoja tilasiirtymiä yksittäistä operaatiota kohden. Tällaisia tapauksia voidaan pyrkiä optimoimaan rajaamalla testattavia muuttujakohtaisia arvojoukkoja ja arvokombinaatioita. Käytännössä testausautomaatiossakin on tehtävä kompromisseja testauksen kattavuuden ja suoritusajan välillä.

Kuvassa 2.5 on esitelty eRA-järjestelmän reseptiominaisuuksien testaamiseen tarkoitettu testausmalli. Kuvasta on jätetty pois istunnon lukitukseen liittyvät asiat ja ne komennot, jotka eivät muuta mallin tilaa, jotta kuva pysyy selkeänä. Kuvasta ei käy ilmi mitä parametreja komennoille on mahdollista syöttää eikä millä parametriarvoilla testaus suoritetaan. Siitä ei myöskään selviä, kuinka virhetilanteet



Kuva 2.5 *Yksinkertaistettu esimerkki eRA-järjestelmän reseptiominaisuuksien testaamiseen käytettävän tilakoneen mallista.*

käsitellään. Jos virhetilanteista on mahdollista palautua, olisi palautuminen hyvä suorittaa, jotta testausta ei tarvitse keskeyttää.

Tilanteet, joissa samat tilasiirtymät joudutaan ajamaan useammin kuin kerran, voidaan ratkaista ajamalla samaa mallin pohjalta tuotettua testisekvenssiä useamman kerran eri testidatalla. Kaikkien määriteltyjen tilasiirtymien ja avain-arvo-kombinaatioiden testaaminen on käytännössä usein mahdotonta testaukseen kuluvan ajan vuoksi. Testisekvenssien generointityökalulle voikin usein antaa ohjeita ja rajoitteita, joilla voidaan saada varmuus määriteltyjen rajatapausten ilmentymisestä testisekvenssissä. Ohjeena voivat toimia esimerkiksi ennaltamäärätyt testattavat arvot, kuten tyhjät merkkijonot tai osittaiset testisekvenssit, jotka halutaan välttämättä saada suoritettua testiajon aikana.

2.3 Valmiit testaustyökalut ja -kehykset

Testausta varten on olemassa valmiita kehyksiä ja työkaluja, jotka vähentävät testauksen automatisoimiseksi vaadittua ohjelmointityötä. Osa työkaluista on kokonaisia sovelluksia tai palveluita, joiden tehtävänä on hallita testien tuottamista ja suorittamista kokonaisuudessaan. Toiset työkaluista on kehitetty toimimaan testaajan toteuttaman testisovelluksen rinnalla, jolloin testaajan vastuulle jää luoda kommu-

nikaatio testattavan sovelluksen ja testaustyökalun välillä.

Testauskehyksillä tarkoitetaan kirjastoja, jotka voidaan ottaa osaksi testaajan toteuttamaa testisovellusta. Testauskehyksen tehtävänä voi olla esimerkiksi visualisoida, analysoida, tuottaa ja suorittaa testejä.

Tässä luvussa tutkitaan lyhyesti muutamien testaustyökalujen ja -kehyksien tarjoamia ominaisuuksia ja rakenteita. Valmiita ratkaisuja tutkimalla pyritään löytämään niille yhteisiä piirteitä ja ominaisuuksia, jotka voisivat olla hyödyllisiä osana tässä työssä toteutettavaa testaustyökalua.

2.3.1 GraphWalker

GraphWalker [8] on Java-ohjelmointikielellä toteutettu avoimen lähdekoodin työkalu mallipohjaisen testauksen automatisointiin. Sen tarkoituksena on helpottaa testien suunnittelua graafien avulla. GraphWalkerissa malli on kokoelma solmuja ja suunnattuja kaaria, joista muodostuu yhdessä suunnattu graafi.

Graafissa olevat kaaret esittävät toimintoja (engl. action) ja solmut tarkistuksia (engl. verification). Toiminto voi olla esimerkiksi "Kirjaudu sisään"ja tarkistus "Käyttäjä on kirjautuneena sisään". Solmusta voi lähteä yhdestä äärettömään kaarta ja kaaret voivat päätyä takaisin samaan solmuun.

GraphWalkeriin on toteutettu algoritmi, joka ratkaisee kiinalaisen postimiehen ongelman [9], eli kulkee jokaista graafin kaarta pitkin vähintään kerran mahdollisimman optimaalisesti. Suoritusta on mahdollista ohjata erilaisin rajoittein ja käskyin, jotta testausautomaatiota voidaan nopeuttaa siten, että halutut rajatapaukset tulevat kuitenkin testattua.

GraphWalkeria on mahdollista suorittaa REST-tilassa, jolloin se käynnistää HTTP-palvelimen, johon voi lähettää testien ajamiseen liittyviä ohjauspyyntöjä JSON-muodossa. Pyynnöt lähetetään GraphWalkerille testaajan toteuttamasta sovelluksesta HTTP-protokollan yli. Testaajan vastuulle jää toteuttaa sovellukseen logiikka, jolla kohdesovellusta testataan, ja virheenkäsittely, jonka perusteella ohjataan testauksen etenemistä.

Mallit voidaan määritellä GraphWalkeria varten millä tahansa GraphML-formaattia tukevalla sovelluksella. GraphML on XML-pohjainen graafien määrittelyyn käytetty formaatti, jolle on olemassa visuaalisia editoreita, kuten yEd [10]. GraphML-mallit voidaan lähettää HTTP-rajapinnan ylitse GraphWalkerille tai ladata ne GraphWalkeriin suoraan komentoriviltä käynnistyksen yhteydessä.

2.3.2 vREST

vREST [11] on kaupallinen työkalu, jolla on mahdollista määritellä sekventiaalisesti ajettavia testejä REST-rajapinnoille. Hinnoittelu on tilauspohjainen, joka tarkoittaa, että palvelusta maksetaan vuosi- tai kuukausimaksua, kunnes tilaus päätetään.

vREST ei työn kirjoitushetkellä tue mallipohjaista testausta. Tästä syystä kaikki testisekvenssit on määriteltävä yksitellen manuaalisesti tai nauhoitettava selainlaajennoksen avulla. vREST soveltuukin hyvin REST-rajapintaa käyttävien web-käyttöliittymien toimintojen testaamiseen, jolloin testisekvenssit voidaan nauhoittaa testattavan järjestelmän web-käyttöliittymää käyttämällä.

2.3.3 NModel

NModel [12] on ilmainen testikehys .NET-ympäristöön. Se pitää sisällään kirjaston testausmallien ohjelmoimiseksi C#-ohjelmointikielellä. Lisäksi kehyksen mukana tulee työkaluja mallien visualisointiin ja analysointiin sekä testien tuottamiseen ja suorittamiseen.

NModelissa testausmallin tilasiirtymät määritellään Action-attribuuteilla, jotka liitetään void-paluuarvollisiin metodeihin. Tilasiirtymälle on mahdollista määritellä ehtoja toteuttamalla erillinen boolean-paluuarvollinen metodi, joka palauttaa arvon tosi, mikäli tilasiirtymä on mahdollinen nykyisestä tilasta. Ohjelmassa 2.1 on yksinkertainen NModelia hyödyntäen toteutettu testausmallin laskuria mallintava osa.

```
[Feature]
static class TimeUpdate
{
        internal static int time = 0;
        internal static bool tickEnabled = false;
        [Action]
        static void Tick(int i)
        {
                time = time + i;
                tickEnabled = false;
        }
        static bool TickEnabled()
        {
                return tickEnabled;
        }
}
```

Ohjelma 2.1 *Esimerkki yksinkertaisesta testausmallista toteutettuna NModelin tarjoamia attribuutteja hyödyntäen.*

Kun malli on luotu, siitä on mahdollista tuottaa myöhempää ajoa varten tallennettavia testisekvenssejä (engl. test suite) NModelin otg-työkalulla, tai kesken tuottamisen ajettavia testisekvenssejä ct-työkalulla. NModelin otg-työkalu on siis tarkoitettu offline-testaukseen ja ct-työkalu online-testaukseen.

2.3.4 Robot Framework

Robot Framework [13] on modulaarinen testiautomaatiokehys, joka on suunniteltu ensisijaisesti hyväksymistestaukseen ja hyväksymistestivetoiseen kehitykseen (engl. Acceptance test-driven development, ATTD). Robot Frameworkia on mahdollista laajentaa toteuttamalla siihen integroituvia kirjastoja joko Java- tai Python-ohjelmointikielellä. Robot Frameworkiin on saatavilla useita ilmaisia kirjastoja, jotka mahdollistavat erilaisten alustojen ja rajapintojen testaamisen. Yksi esimerkki on Selenium2Library, joka kytkee Robot Frameworkin suoraan Selenium 2 -testauskehykseen, joka on tarkoitettu web-sovellusten testaamiseen. Robot Frameworkiin on saatavilla myös kirjastoja, jotka mahdollistavat HTTP-pyyntöjen lähettämisen testattavaan järjestelmään ilman varsinaista web-selainta.

```
*** Settings ***
Suite Setup        Open Browser To Login Page
Suite Teardown     Close Browser
Test Setup         Go To Login Page
Test Template      Login With Invalid Credentials Should Fail
Resource           resource.txt

*** Test Cases ***              User Name        Password
Invalid Username                invalid          ${VALID PASSWORD}
Invalid Password                ${VALID USER}    invalid
Invalid Username And Password   invalid          whatever
Empty Username                  ${EMPTY}         ${VALID PASSWORD}
Empty Password                  ${VALID USER}    ${EMPTY}
Empty Username And Password     ${EMPTY}         ${EMPTY}

*** Keywords ***
Login With Invalid Credentials Should Fail
    [Arguments]    ${username}    ${password}
    Input Username    ${username}
    Input Password    ${password}
    Submit Credentials
    Login Should Have Failed

Login Should Have Failed
    Location Should Be    ${ERROR URL}
    Title Should Be    Error Page
```

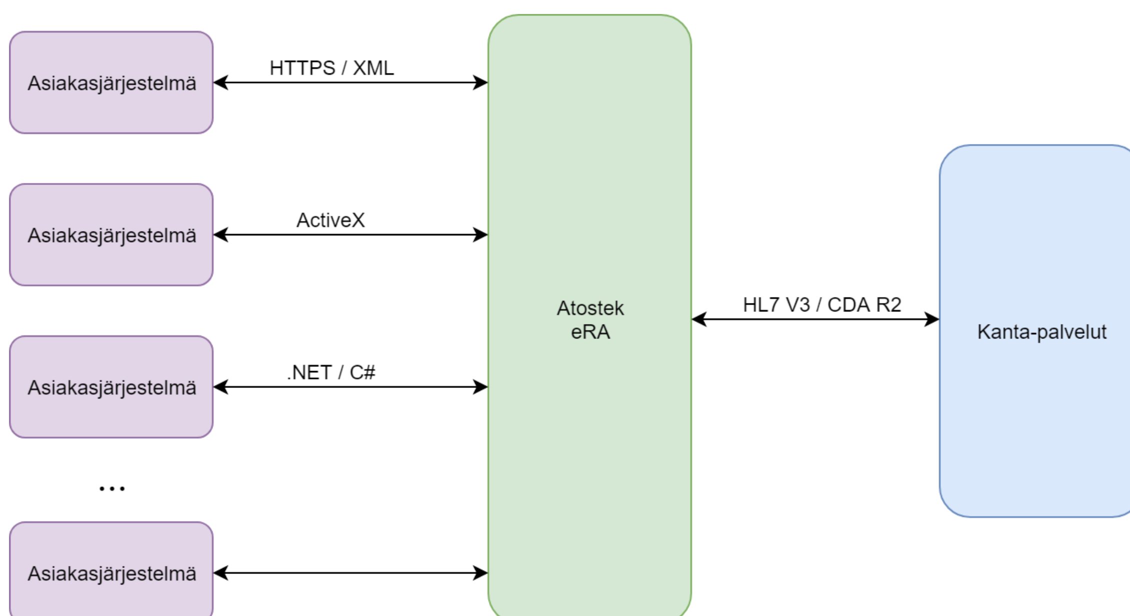
Ohjelma 2.2 *Esimerkki Robot Frameworkille määritellystä testiskenaariosta.*

Robot Frameworkin testiskenaariot pyritään kirjoittamaan selväkielisinä, kuten ohjelmassa 2.2 on kuvattu. Testiskenaarioiden määrittelyssä voidaan käyttää tietovetoista testausmenetelmää (engl. Data-driven testing, DDT). Tietovetoisella testauksella tarkoitetaan menetelmää, jossa testiautomaatiokehykselle annetaan ennaltamääritelty joukko syötteitä, jotka ajetaan yksitellen testattavaan järjestelmään hyödyntäen samaa testilogiikkaa. Ohjelmassa 2.2 määritellään useampi erilainen syöte, joista jokainen ajetaan testattavaan järjestelmään erillisenä testitapauksena.

Testiskenaarion lisäksi tulee toteuttaa testattavaa järjestelmää ohjaava kirjasto joko Python- tai Java-ohjelmointikielellä, jossa testiskenaarioissa hyödynnettävät avainsanat on määritelty. Kirjastossa määritellään kaikki muukin testiskenaarioiden hyödyntämä logiikka, kuten testattavan järjestelmän palautteiden validointi. Kirjastot voivat hyödyntää mitä tahansa Python- ja Java-ympäristöihin toteutettuja kirjastoja ja ne voivat toisaalta kytkeytyä myös muille ympäristöille toteutettuihin kirjastoihin.

3. TESTATTAVA JÄRJESTELMÄ: eRA

Tutkielman kohdejärjestelmäksi valittiin yrityksen oma tuote eRA, joka on palvelu sosiaali- ja terveydenhuollon tietojärjestelmien liittämiseksi Kelan tarjoamiin Kansallisen Terveysarkiston palveluihin eli Kanta-palveluihin [14]. eRA-järjestelmällä on mahdollista siirtää potilastietoja Kanta-palveluiden ja eRA-järjestelmään liittyneiden potilastietojärjestelmien välillä. Järjestelmähierarkiaa on esitelty kuvassa 3.1. Kanta-palvelut tarjoavat HL7 V3 -standardin [15] mukaiset sanomarajapinnat, joita ei käydä läpi tässä työssä, sillä niiden yksityiskohdat eivät ole oleellisia tämän työn kannalta.



Kuva 3.1 *Atostek eRA tarjoaa rajapinnan sosiaali- ja terveydenhuollon tietojärjestelmien liittämiseksi Kelan tarjoamiin Kanta-palveluihin.*

Potilastietojärjestelmä on mahdollista integroida eRA-järjestelmään sen tarjoamien integraatiorajapintojen kautta. eRA-järjestelmään on toteutettuna web-, ActiveX- ja .NET-rajapinnat. Web-rajapinta käsittelee XML-muotoista tietoa. ActiveX- ja .NET-rajapinnat käyttävät web-rajapinnan komentoja, mutta tarjoavat järjestelmä-

toimittajille erilliset kirjastot helpottamaan integraatiota. Integraatiorajapinnasta on laadittu järjestelmätoimittajille toimitettava integraatio-opas, joka sisältää kuvaukset integraatiorajapinnan metodeista, tietorakenteista ja virhekoodeista. Myös testausautomaatio tullaan toteuttamaan web-integraatiorajapinnalle kyseisen oppaan pohjalta luvussa 4.

eRA-järjestelmästä oli työn alussa tuotannossa rajapintaversiot 1.0 ja 1.1. Myöhemmin työn edetessä tuotantoon vietiin myös osa 2.0-rajapintaversion ominaisuuksista. Rajapintaversio 1.1 mahdollistaa lääkemääräysten ja potilaan tietojen noutamisen eRA-järjestelmältä potilastietojärjestelmään. Rajapintaversio 2.0 on huomattavasti laajempi, sisältäen tuen mm. Potilastiedon arkiston ominaisuuksille. Tuki kaikille rajapintaversioille säilytetään toistaiseksi, sillä osa eRA-järjestelmään liittyneistä asiakkaista käyttää yhä rajapintaversioita 1.0 ja 1.1. Osa tietorakenteista eroaa toisistaan rajapintaversioiden välillä toiminnallisista syistä, joten on tärkeää, että kaikki rajapintaversiot testataan erikseen.

eRA-järjestelmään kohdistuvia komentoja on mahdollista käyttää vain tietyssä tilassa ja tietyin käyttöoikeuksin. eRA-järjestelmän vastuulla on ylläpitää istuntojen sisäistä tilaa ja validoida käyttäjien käyttöoikeudet. Yksittäisen käyttäjän käyttöoikeudet määräytyvät käyttäjän roolin ja asiakasjärjestelmälle sallittujen ominaisuuksien perusteella.

3.1 Web-integraatiorajapinnan tiedonsiirtokerros

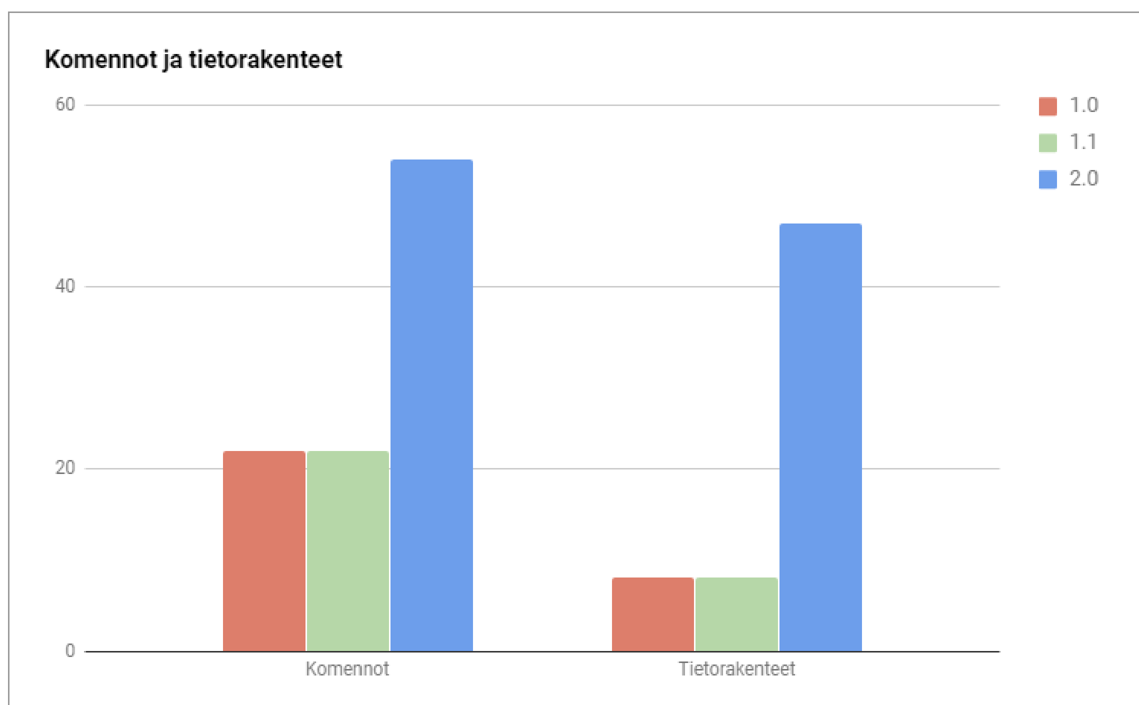
eRA-järjestelmä on ASP.NET-tekniikalla toteutettu WWW-sovelluspalvelu (engl. web service), joka toimii Microsoftin kehittämän IIS-palvelinohjelmiston (Internet Information Server) päällä. Kaikki tietoliikenne integroivien asiakasjärjestelmien ja eRA-järjestelmän välillä ohjataan käyttämään HTTPS-protokollaa, jossa on käytössä salausmenetelmänä TLS 1.0, 1.1 tai 1.2. Salauksessa käytetään varmentajan eRA-järjestelmälle myöntämää SHA-256-allekirjoitettua varmennetta. Kaikissa integraatio-rajapintaan kohdistuvissa pyynnöissä tulee käyttää HTTP POST -metodia.

Kaikissa rajapintakomennoissa välitetään pakkaamattomia XML-rakenteita asiakasjärjestelmän ja eRA-järjestelmän välillä. Rakenteissa voidaan rajapintakomennosta riippuen kuljettaa mukana potilastietoja ja käyttäjän istuntoon liittyviä autentikointitietoja.

3.2 Rajapinnan komennot ja rakenteet

eRA-järjestelmän 1.0- ja 1.1-rajapintaversiossa on yhteensä 22 dokumentoitua komentoa. eRA-järjestelmän 2.0-rajapintaversioon on toteutettuna ja dokumentoituna työn kirjoitushetkellä yhteensä 54 rajapintakomentoa, joilla integroituneiden asiakasjärjestelmien on mahdollista ohjata järjestelmän toimintaa. eRA-järjestelmän kehitys jatkuu myös työn julkaisun jälkeen, eikä kaikkia rajapintakomentoja ole vielä työn kirjoitushetkellä määriteltynä.

eRA-järjestelmän 1.0- ja 1.1-integraatiorajapintojen tietorakenteet ovat suppeita, eikä niiden sisältämien kenttien välillä ole kuin muutamia ehdollisia riippuvuussuhteita. Sen sijaan järjestelmän 2.0-rajapintaversio sisältää muun muassa arkistokirjauksien tietomallien rajapintarakenteet, jotka voivat olla jopa useiden kymmenien kenttien laajuisia ja sisältää useita ehdollisia riippuvuussuhteita. Tietorakenteita 2.0-rajapintaversiossa on kirjoitushetkellä dokumentoituna yhteensä 47 kappaletta. Integraatiorajapinnan versioiden komentojen ja tietorakenteiden määrää on verrattu kuvassa 3.2.



Kuva 3.2 *eRA-järjestelmän rajapintaversioiden komentojen ja tietorakenteiden määrä.*

Ehdollisilla riippuvuussuhteilla tarkoitetaan tietorakenteiden kenttiin annettujen arvojen vaikutusta muiden kenttien sallittuihin arvoalueisiin. Esimerkiksi potilastietoja avattaessa tulee määrittää potilaan tunnistustapa. Jos tavaksi on valittu pysyvä henkilötunnus, tulee henkilötunnus syöttää, mutta tuolloin potilaan syntymäaika ei

ole pakollinen tieto. Jos taas tunnistustavaksi on valittu sukupuoli ja syntymäaika, tulee henkilötunnus jättää rakenteessa tyhjäksi. Tietorakenteiden sisältämät ehdolliset riippuvuussuhteet vaikeuttavat testausautomaation toteuttamista, joka havaitaan luvussa 4.

eRA-järjestelmän tarjoamat rajapintakomennot ovat riippuvaisia käyttäjän istunnon tilasta. Esimerkiksi potilaan avauksella tarkoitetaan eRA-järjestelmässä suoritettavia teknisiä toimenpiteitä, joiden jälkeen käyttäjän on mahdollista tehdä kirjauksia kyseiselle potilaalle. Potilaan avaus onnistuu vain, jos potilasta ei ole vielä avattuna, ja potilaan sulkeminen vain, jos potilas on jo avattuna.

Eri rajapintakomentojen tilavaatimukset on dokumentoitu eRA-järjestelmän integraatio-ohjeeseen [16], joka on salassapitosopimuksen alaista tietoa, joten sen tarkempia yksityiskohtia ei tuoda esiin tässä työssä. Integraatio-oppaan tietoja täytyy hyödyntää testausautomaatiota luotaessa, jotta tiedetään mitkä komennot ovat sallittuja missäkin tilassa. Integraatio-opas määrittelee rajapintakomentoihin liittyvät tilavaatimukset. Jokaiselle komennolle on määritelty, mitkä seuraavista ehdoista tulee olla voimassa, jotta komentoa voidaan kutsua:

- Istunto täytyy olla avattuna.
- Potilas täytyy olla avattuna.
- Palvelutapahtuma täytyy olla avattuna.

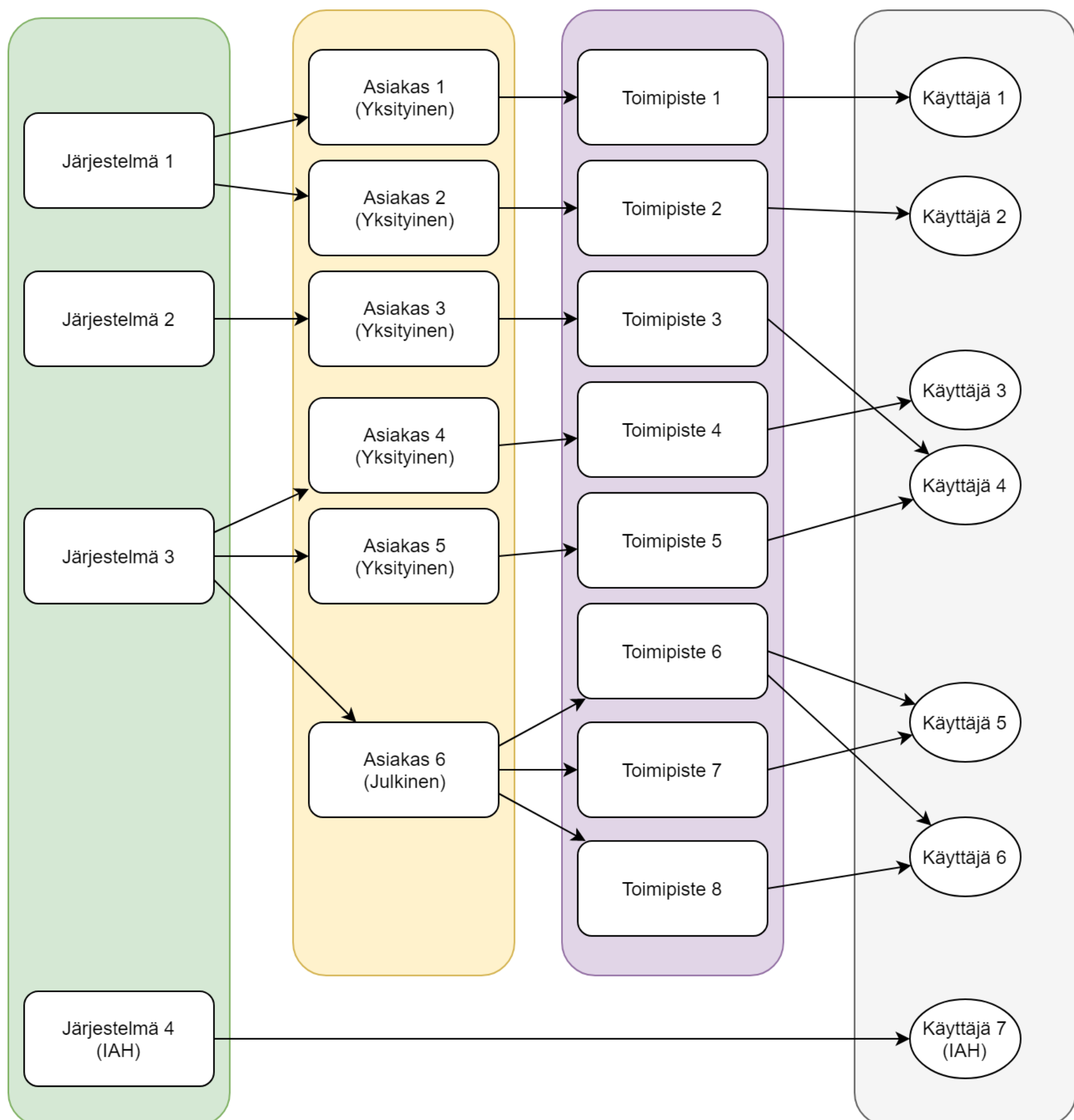
Yllä listattujen yleisten tilavaatimuksien lisäksi oppaaseen on kirjattu muutamia komentokohtaisia vaatimuksia. Jokaista komentoa kohden on koottu lista virhekoodeista, joita kyseisen komennon kutsuminen voi palauttaa.

3.3 Käyttäjän tunnistaminen

eRA-järjestelmä hyödyntää käyttäjän tunnistamiseen Väestörekisterikeskuksen myöntämiä toimikortteja. Toimikorteille on tallennettuna terveydenhuollon sähköinen varmenne, jonka perusteella käyttäjän ammattioikeustiedot tarkistetaan Valviran ylläpitämästä Terveydenhuollon rooli- ja attribuuttitietopalvelusta [17]. Tunnistamisen yhteydessä toimikortilla suoritetaan sähköinen allekirjoitus, joka on suojattu kortin haltijan asettamalla PIN-koodilla. Käyttäjän tunnistaminen ja allekirjoitus tehdään eRA-järjestelmään sisäänkirjauduttaessa ja lääkemääräyksen, lausuntojen sekä todistusten lähettämisen yhteydessä. eRA-järjestelmään toteutettu käyttäjän tunnistamisprosessi on salassapitosopimuksen alaista tietoa, eikä sitä kuvata tarkemmin tässä työssä.

3.4 Järjestelmähierarkia

eRA-järjestelmässä on neljä eri hierarkiatasoa, joilla kuvataan liittyneitä asiakkaita. Ylimmällä tasolla ovat määriteltynä asiakasjärjestelmät, joita ovat esimerkiksi potilastietojärjestelmät. Seuraavalla tasolla ovat asiakkaat, jotka vastaavat yrityksiä, joilla tietty potilastietojärjestelmä on käytössä. Kolmannella tasolla ovat asiakkaiden toimipisteet eli yksittäiset toimipaikat, jossa yrityksen työntekijät käyttävät järjestelmää. Käyttäjät ovat hierarkian neljäs eli alin taso. Sama käyttäjä voi käytännössä kuulua useampaan toimipisteeseen ja jopa useampaan asiakasjärjestelmään.



Kuva 3.3 *eRA-järjestelmän liittyjien hierarkia.*

Kuvassa 3.3 näkyvän rakenteen lisäksi käyttäjä voi toimia itsenäisenä ammatinharjoittajana (IAH) ilman varsinaista organisaatiota. Itsenäinen ammatinharjoittaja ei normaalisti kuulu yhteenkään toimipisteeseen eikä minkään asiakkaan alaisuuteen. On kuitenkin olemassa myös vuokralla toimivia itsenäisiä ammatinharjoittajia, jotka toimivat yhden tai useamman asiakkaan tiloissa. Itsenäisen ammatinharjoittajan liittäminen eRA-järjestelmään eroaakin teknisessä mielessä hieman muista käyttäjätyypeistä ja on siksi tärkeä testauksen kohde.

Lisäksi asiakkaat voidaan luokitella erikseen julkisen ja yksityisen puolen asiakkaiksi. Myös näiden välillä on THL:n (Terveyden ja hyvinvoinnin laitos) ja Kelan määrittelemiä eroja Kanta-palveluihin liityttäessä [18], mitkä on huomioitu eRA-järjestelmän toteutuksessa. Siksi sekä yksityisen että julkisen puolen asiakkaalle konfiguroidulla käyttäjällä on tärkeää suorittaa rajapintatestaus kokonaisuudessaan.

3.5 Käyttöoikeudet

Jokaiselle eRA-järjestelmän käyttäjälle on määritelty roolit, joiden mukaan määräytyvät käyttöoikeudet eri eRA-järjestelmän toiminnallisuuksiin. Käyttäjärooleja voivat olla muun muassa lääkäri, sairaanhoitaja tai uusimispyyntöjen käsittelijä. Roolit ovat määriteltävissä asiakasjärjestelmäkohtaisesti.

Toinen käyttöoikeuksia rajoittava tekijä on asiakasjärjestelmille valitut käytössä olevat ominaisuudet. Ominaisuudet on määritelty niihin liittyvän toiminnon ja kohteen perusteella. Toimintoja ovat esimerkiksi luominen, poistaminen, peruminen, muokkaaminen ja lukeminen, kun taas kohteita ovat esimerkiksi uusimispyynnöt, arkistotiedot ja palvelutapahtumat. Asiakasjärjestelmän käyttäjille voi siis esimerkiksi antaa oikeuden luoda palvelutapahtumia ja muokata arkistotietoja, mikäli käyttäjäkohtainen rooli ei sitä erikseen estä.

Potilastiedon arkistossa tieto ryhmitellään tietokokonaisuuksia tai asiayhteyttä kuvaaviin näkymiin. Määrittelyt arkistotietojen rakenteista ja näkymistä ovat osa Kanta.fi-sivustolta löytyvää Potilastiedon arkiston määrittelyä [18], eikä niiden läpikäynti ole osa tätä työtä. Arkistotietojen katselu- ja muokkausoikeudet on rajattu eRA-järjestelmässä arkistotietonäkymien ja -rakenteiden mukaan. Asiakasjärjestelmille voidaan määritellä luku- ja / tai kirjoitusoikeus vain tiettyihin arkistotietonäkymiin. Arkistotietonäkymiä ovat muun muassa HAM- ja SUU-näkymät, joihin kirjataan suun terveydenhuoltoon liittyviä merkintöjä.

3.6 Koodistot

eRA-järjestelmä hyödyntää toimiessaan Kelan tarjoamaa Kansallista koodistopalvelua [19]. Koodistot ovat ladattavissa XML-muotoisina Kelan palvelimelta. Koodistoihin on määritelty muun muassa arkistotietonäkymät, SOTE-organisaatiot ja itsenäiset ammatinharjoittajat. eRA-järjestelmä hyödyntää näiden koodistojen lisäksi monia muita koodistopalvelusta löytyviä koodistoja.

Koodistot ja niiden sisältö ovat osa Kelan Kanta-palvelukokonaisuutta. Koodistoissa olevien koodien hyödyntäminen on välttämätöntä, jotta eRA-järjestelmä voi välittää tietoja Kanta-palveluihin.

4. TESTAUKSEN SUUNNITTELU JA TOTEUTUS

Testausautomaatio päätettiin heti työn alussa jakaa kolmeen osaan: eRA-järjestelmän integraatiorajapintaversioiden 1.0, 1.1 ja 2.0 erilliseen testaukseen. Tämä siitä syystä, että vaikka rajapinnat muistuttavat paljolti toisiaan, niissä on muutamia olennaisia eroja toiminnan kannalta. Työtä aloitettaessa 2.0-rajapintaversion määrittely ei ollut vielä täysin selvillä. Osa eRA-järjestelmään liittyneistä asiakkaista käytti työn kirjoitushetkellä rajapintaversiota 1.0, joten eRA-järjestelmän oli oltava taaksepäin yhteensopiva versioon 1.0 asti. Oikeiden eRA-järjestelmän integraatiorajapinnan rakenteiden sijaan työssä esitetään selkeyden vuoksi hieman niitä vastaavia, suppeampia rakenteita.

Rajapintaversioiden 1.0 ja 1.1 testaus toimi valittujen testausmenetelmien soveltuvuusselvityksenä samalla, kun rajapintaversio 2.0 oli vielä kehitteillä. Järjestelmän rajapintaversio 2.0 on kuitenkin ominaisuuksiltaan jo huomattavasti laajempi kuin rajapintaversio 1.1, joten uudelleenselvitys oli paikallaan rajapintaversion 1.1 testauksen jälkeen. Testausautomaatio kaikille rajapintaversioille päätettiin toteuttaa alusta alkaen itse, jotta voitiin varmistua siitä, että testausautomaatio 2.0-rajapintaversion monimutkaisemmille ominaisuuksille on mahdollista toteuttaa saman teknologian päälle. Valmiin testaustyökalun käyttöönotto vaatisi huomattavan määrän lisäselvitystyötä eikä kattavasta selvityksestä huolimatta voitaisi olla varmoja sen soveltuvuudesta vielä keskeneräisen 2.0-rajapintaversion testaukseen.

Merkittävä eRA-järjestelmän testauksessa huomioitava asia ovat aiemmin aliluvussa 3.5 mainitut käyttäjäroolit ja asiakasjärjestelmätason käyttöoikeudet. Nämä eivät varsinaisesti ole vain integraatiorajapinnan ominaisuus, vaan eRA-järjestelmän sisäinen ominaisuus, joka vaikuttaa kaikkien rajapintaversioiden toimintaan. Rajapintaversiossa 1.1 on ainoastaan yleisiä ja reseptitoiminnallisuuksia, joten käyttöoikeuksien hallinta on sen osalta yksinkertainen. 2.0-rajapintaversioon liittyvät käyttöoikeudet ovat monimutkaisemmat: Uudet rajapintatoiminnot vaativat erillisiä käyttöoikeuksia, ja lisäksi kaikille arkistorakenteille ja -näkymille on olemassa omat luku- ja kirjoitusoikeutensa.

Rajapintaversio 2.0:n testausautomaatiota suunniteltaessa testiskenaarioiden kirjoittaminen manuaalisesti havaittiin erittäin työlääksi, joten päädyttiin hyödyntämään mallipohjaisuutta testausautomaation toteutuksessa. Esimerkiksi potilaan arkistomerkintöjen luontiin tarkoitettu komento sisälsi kirjoitushetkellä yhteensä 126 mahdollista kenttää, joiden kaikkien toiminta tuli testata mahdollisimman kattavasti. Jos toteutettaisiin manuaalisesti testiskenaario, jossa testattaisiin jokaista kenttää kohden yksi positiivinen ja yksi negatiivinen testitapaus, tulisi testitapauksia pelkästään tälle komennolle yhteensä 252 kappaletta. Tuolloin lisäksi monia testauksen kannalta oleellisia arvoja jäisi testaamatta, eikä testeissä myöskään huomioitaisi kenttien välisiä ehdollisia riippuvuussuhteita toisiinsa. Todellisuudessa jo pelkästään arkistomerkintöjen luontiin tarkoitetulle komennolle testitapauksia tulee kenttien väliset ehdolliset riippuvuussuhteet huomioiden yli tuhat. Yhden testitapauksen kirjoittamiseen kuluvan ajan ollessa keskimäärin kaksi minuuttia, kuluisi tuhannen testitapauksen luomiseen yli 30 tuntia.

4.1 Testaustyökalun runko

Testaustyökalun runko päätettiin toteuttaa Microsoftin WPF-kirjastoa (Windows Presentation Foundation) käyttäen, sillä se oli toteuttajalle entuudestaan tuttu ja se mahdollistaa olemassa olevien .NET-kirjastojen hyödyntämisen sovelluksessa. WPF on .NET Framework 3.0:n ja uudempien osana oleva kirjasto, joka mahdollistaa käyttöliittymien toteuttamisen hyödyntäen näytönohjaimen laskentaominaisuuksia [20]. Atostek Oy:ssä on havaittu, että WPF-kirjastolla on mahdollista luoda Windows-työpöytäsovelluksia nopeasti siten, että sovelluksen arkkitehtuuri pysyy selkeänä.

Ohjelmointikieleksi valittiin C#, jotta tiedon välitys käyttöliittymän ja toimintalogiikan välillä olisi mahdollisimman helppoa. Tuolloin myös tietomallien sarjallistaminen XML-muotoon ja takaisin on mahdollista .NET-kehyksen tarjoaman XmlSerializer-luokan avulla [21]. Rungon toteutukseen käytettiin ainoastaan Microsoftin tarjoamia ilmaisia kirjastoja.

Sovellukseen luotiin sisäinen tietomalli, jota käytettiin rajapinnassa liikkuvan tiedon säilömiseen muistissa. Tietomallin avulla voitiin helposti lukea ja muokata parametreja rajapintakutsujen välissä. Tietomalliin voidaan ladata konfiguraatioita suoraan testiskenaarioista, mutta sinne tallentuu automaattisesti myös rajapintakutsujen vastauksena tulevia tietoja niille varattuihin kenttiin. Tietomallin päivittäminen tapahtuu aina purkamalla XML-rakenne muistiin ja ylikirjoittamalla tietomallin osia XML-rakenteesta löytyneillä kentillä. Tietomallin sisältö on näkyvissä ajonaikaisesti testaustyökalun käyttöliittymässä olevasta puunäkymästä.

Oleellinen osa testausautomaation toteutusta on testioraakkeli, jonka vastuulla on palvelimen vastausten validointi. Validointi päätettiin toteuttaa reflektiolla, joka on .NET-kehyksen tarjoama ominaisuus, jolla on mahdollista lukea ajonaikaisesti olioiden tietosisältöjä tietämättä olioiden varsinaisia tietotyyppejä [22]. Reflektion ansiosta on mahdollista luoda geneerisiä vertailufunktioita, jolloin vältytään tietomalleille spesifien vertailufunktioiden toteuttamiselta kaikissa kehitysvaiheissa. Testauskehykseen toteutettiin työssä seuraavat vertailuoperaatiot:

- Equals (on yhtä suuri kuin)
- Contains (lista sisältää)
- NotContains (lista ei sisällä).

Kaikki vertailuoperaatiot on tarkoitettu käytettäviksi XML-muotoisten testiskenaarioiden validointirakenteissa, joista on kerrottu tarkemmin kohdassa 4.3.3. Equals-vertailuoperaatiota on mahdollista käyttää mille tahansa tietotyypille, kun taas Contains- ja NotContains-vertailuoperaatiot on tarkoitettu erityisesti rajapinnassa esiintyville listarakenteille.

Testaustyökalu kirjaa testien ajoon liittyvät tiedot automaattisesti lokiin kellonaikoineen. Lokista selviää muun muassa kaikki tietomalliin liittyvät päivitykset ja testattavalle järjestelmälle lähetetyt komennot. Testien tulosten validoinnin epäonnistuessa lokiin kirjataan tarkemmat tiedot komennon ja paluuviestin sisällöstä. Jokaiseen lokimerkintään on liitetty vakavuusaste, joka näkyy lokinäkymässä tekstimuotoisena. Merkinnät on värjätty näkymässä niiden vakavuusasteen perusteella.

Myöhemmin testaustyökaluun toteutettiin myös näkymä, josta testaaja voi nähdä testattavaan järjestelmään lähetettävien HTTP-pyyntöjen ja -vastausten XML-muotoiset tietosisällöt. Näkymä toteutettiin, jotta varsinaisten kehittäjien olisi helpompi havaita vikoja tietosisällöissä tutustumatta lainkaan testaustyökaluun. Samanaikaisesti näkymä myös helpottaa testitapausten toistamista manuaalisesti.

4.2 eRA-järjestelmään toteutettavat testitoiminnot

eRA-järjestelmään kirjautumiseen vaaditaan Väestörekisterikeskuksen myöntämä toimikortti [23]. Kortilla voidaan suorittaa allekirjoitustoimenpiteitä, joista toinen suoritetaan sisäänkirjautumisen yhteydessä ja toinen lääkemääräyksen lähettämisen yhteydessä. Molemmat allekirjoitustoimenpiteet vaativat PIN-koodin, jonka käyttäjä voi itse asettaa kortilleen.

Jotta testausautomaation aikana voitaisiin kirjautua sisään eri käyttäjillä, tarvittaisiin testejä suorittavaan tietokoneeseen useita kortinlukijoita, joissa olisi kaikkien testeissä esiintyvien käyttäjien toimikortit. Tämä ei kuitenkaan ole järkevää, sillä testejä ei välttämättä aina ajeta samalla tietokoneella eikä toimikortteja ole aina saatavilla. Lisäksi menetelmä vaatisi testaajaa syöttämään korttien PIN-koodit aina, kun eRA-järjestelmä niitä pyytää.

Testauksen automatisoinnin helpottamiseksi eRA-järjestelmään kehitettiin erilliset komennot kirjautumisen ja lääkemääräyksen lähettämisen simuloimiseksi. Näitä komentoja on mahdollista kutsua oikeiden allekirjoitusvaiheiden komentojen sijaan, jos eRA-järjestelmän testiominaisuudet ovat käytössä. Komennot eivät vaadi laitteeseen syötettävää toimikorttia eivätkä siten myöskään aiheuta PIN-koodikyselyä.

Käyttöoikeuksien toimivuuden testaamiseksi eRA-järjestelmään luotiin testikomennot myös asiakasjärjestelmäkohtaisten ominaisuuksien kysymiseen ja muuttamiseen. Testaustyökalun on mahdollista ennen testin ajoa asettaa eRA-järjestelmään käytettävälle asiakasjärjestelmälle sallitut ominaisuudet, jotta tiedetään, mikä on odotettu lopputulos testitapaukselle.

4.3 Menetelmä 1: Testiskenaarioiden kirjoittaminen manuaalisesti

Testauksen suunnittelu aloitettiin kohdejärjestelmän rajapintaversion 1.0 integraatio-opasta noudattaen. Oppaasta käy ilmi järjestelmän tarjoamat rajapintakomennot, niiden hyväksymät tietorakenteet sekä komentojen palauttamat mahdolliset virhekoodit. Näiden pohjalta luotiin tarkoitukseen sopiva Backus-Naur-muotoinen esitys testiskenaarioiden rakenteesta. Koska rajapinnan ylitse liikuva tieto on XML-muotoista, XML päätettiin valita myös testiskenaarioiden kuvauskieleksi.

Kuvaus pyrittiin pitämään mahdollisimman yksinkertaisena, jotta testiskenaarioiden luominen olisi helppoa. Testiskenaarion ajon aikana on voitava muuttaa testauskehyksen ylläpitämää tilaa, lähettää komentoja kohdejärjestelmälle ja validoida kohdejärjestelmän lähettämät paluuviestit. Lisäksi testiskenaarioille on määritelty tekstimuotoinen nimi, joka on mahdollista näyttää käyttöliittymässä testejä ajettaessa. Testiskenaarion rakenne on lyhyesti kuvattu ohjelmassa 4.1.

```
<test>
  <description>Testiskenaarion sanallinen kuvaus</description>
  <statement_list>
    <configuration>
      <assign>
        <patient>
          <name>Testi Potilas</name>
          <id>020202-0202</id>
        </patient>
      </assign>
    </configuration>
    <function_call name="OpenPatient">
      <expected_result_list>
        <expected_result>
          <status>200</status>
        </expected_result>
      </expected_result_list>
    </function_call>
  </statement_list>
</test>
```

Ohjelma 4.1 *Yksinkertaistettu esimerkki testiskenaarion rakenteesta. Esimerkissä asetetaan potilaan tiedot testiohjelman tietomalliin ja lähetetään eRA-järjestelmälle pyyntö avata potilas. eRA-järjestelmän paluuviestin HTTP-tilakoodin odotusarvo on 200 (HTTP OK).*

Jotta testiskenaarioihin tulisi mahdollisimman vähän toisteisia osia, luotiin testaustyökaluun tuki ns. alisekvenssirakenteille (engl. subsequence). Esimerkiksi sisäänkirjautuminen ja potilaan avaaminen ovat toimintoja, jotka toistuvat lähes kaikkien testiskenaarioiden alussa. Molemmista toiminnoista tehtiin omat alisekvenssinsä, jotka voitiin liittää osaksi laajempia testisekvenssejä yksinkertaisella subsequence-viittauksella. Ohjelmassa 4.2 on kuvattu esimerkin mukainen testiskenaario. Alisekvenssit koostuvat testisekvensseissä olevasta statement_list-rakenteesta, joka voi sisältää rajapintakutsujen lisäksi myös konfiguraatiorakenteita.

```
<test>
  <description>Potilaan avaus ja sulkeminen</description>
  <arrangements>
  <users>
    <user type="renewal_manager" />
    <user type="nurse" />
    <user type="nurse_with_prescription_rights" />
    <user type="doctor" />
  </users>
  <organizations>
    <organization type="private_healthcare" />
    <organization type="public_healthcare" />
  </organizations>
  <system_clients>
    <system_client type="single_user" />
    <system_client type="organization" />
  </system_clients>
  </arrangements>
  <statement_list>
    <!-- KIRJAUDUTAAN SISÄÄN -->
    <subsequence id="login" />

    <!-- AVATAAN POTILAS -->
    <subsequence id="open_patient" />

    <!-- SULJETAAN POTILAS -->
    <function_call name="PatientClose">
      <expected_result_list>
        <expected_result>
          <status>200</status>
        </expected_result>
      </expected_result_list>
    </function_call>

    <!-- KIRJAUDU ULOS -->
    <function_call name="Logout">
      <expected_result_list>
        <expected_result>
          <status>200</status>
        </expected_result>
      </expected_result_list>
    </function_call>
  </statement_list>
</test>
```

Ohjelma 4.2 *Esimerkki testiskenaariosta, jossa hyödynnetään alisekvenssirakenteita sisäänkirjautumiseen ja potilaan avaamiseen.*

Kaikille rajapinnassa kulkeville tietorakenteille luotiin omat luokkansa, jotka vastaavat eRA-järjestelmän integraatio-oppaassa kuvattuja rakenteita. Luokkien sarjallistaminen XML-muotoon toteutettiin .NET-kehyksen tarjoamalla XmlSerializer-luokalla. Testattavan järjestelmän lähettämät paluuviestit muunnettiin samaan tapaan takaisin testaustyökaluun määritellyiksi tyypeiksi, jotta niiden sisältö voidaan validoida testioraakkelia hyödyntämällä.

4.3.1 Arvoviittaukset

Testiskenaarioissa voi asettaa arvoja kenttiin, joita käytetään testattavalle järjestelmälle lähetettävissä komennoissa. Kenttiin ei kuitenkaan aina ole mahdollista kirjoittaa arvoja suoraan, vaan niihin tarvitsee sisällyttää tietoja mm. aiempien komentojen vastauksista. Testaustyökalu taltioi automaattisesti pyyntöjen tiedot muistiin niitä vastaaviin kenttiin, jotta niitä voidaan automaattisesti käyttää tulevissa komennoissa. Jos tietoja halutaan käyttää muissakin kentissä, niihin voidaan viitata asettamalla arvoksi dollarimerkillä alkavan merkkijonon, kuten esimerkiksi "$Entry.OID".

Dollarimerkkiviittaukset helpottavat testiskenaarioiden kirjoittamista huomattavasti, mutta eivät riitä kaikissa tapauksissa. Aikaleimat osoittautuivat ongelmallisiksi testauksessa, sillä joidenkin eRA-järjestelmän tietotyyppien tila määräytyy niiden aikaleiman perusteella. Jos testiskenaarioon kirjoitetaan kiinteä aikaleima, testiskenaariota tulee päivittää aika-ajoin, jotta eRA-järjestelmän asettama tilatieto voitaisiin validoida. Siksi dollarimerkkiviittauksen lisäksi kehitettiin kaksi muuta viittaustapaa: @- ja £-merkeillä alkavat viittaukset. @-merkillä alkavat merkkijonot ovat ns. dynaamisia merkkijonoja, joiden tehtävänä on luoda merkkijono kooditoteutuksen pohjalta. Testaustyökaluun toteutettiin tuki luoda aikaleimoja testeihin dynaamisesti kirjoittamalla kentän arvoksi @lastyear, @yesterday, @today, @tomorrow tai @nextyear. Dynaamisia merkkijonoja on mahdollista luoda lisää pienellä työmäärällä tarpeen vaatiessa.

```
...
<configuration>
  <store>
    <variable name="startTime">@lastyear</variable>
    <variable name="endTime">@yesterday</variable>
  </store>
</configuration>
<function_call name="GetServiceEvents">
  <expected_result_list>
    <expected_result>
      <validations>
        <validation>
          <type>Contains</type>
          <data>
            <response>
              <get_service_events>
                <service_events>
                  <service_event>
                    <start_time>£startTime</start_time>
                    <end_time>£endTime</end_time>
                  </service_event>
                </service_events>
              </get_service_events>
            </response>
          </data>
        </validation>
      </validations>
    </expected_result>
  </expected_result_list>
</function_call>
...
```

Ohjelma 4.3 *Esimerkki testiskenaariosta, jossa asetetaan muuttujiin dynaamisia aikaleimoja ja viitataan kyseisiin muuttujiin testitapauksen validointivaiheessa. Näkyvissä olevaa testiskenaariota on typistetty.*

Jotta dynaamisesti generoituihin aikaleimoihin voitaisiin viitata myöhemmin testiskenaariossa, toteutettiin työkaluun myös tuki taltioida arvoja muuttujiin. Viittaus muuttujan arvoon ilmaistaan £-merkillä merkkijonon alussa. Ohjelmassa 4.3 on esitelty osa testiskenaariota, joka toimii esimerkkinä tällaisesta tilanteesta. Testiskenaariossa luodaan palvelutapahtuma tietylle aikavälille, haetaan eRA-järjestelmästä tietyn aikavälin palvelutapahtumat ja validoidaan, että järjestelmä palauttaa ainakin yhden palvelutapahtuman, jossa on sekunnilleen sama aikaväli kuin juuri luodussa palvelutapahtumassa. Koska muuttujien käyttö ei merkittävästi hidasta testien suorittamista, voidaan niitä käyttää myös selkeyttämään testien rakennetta vähentäen

samalla mahdollisten virheiden todennäköisyyttä testiskenaarioita kirjoitettaessa.

4.3.2 Konfiguraatiorakenteet

Testiskenaarioiden lisäksi suunniteltiin erillinen rakenne konfiguraatioille, jotka ovat yhteisiä kaikille testiskenaarioille. Konfiguraatioiden rakenne pyrittiin selkeyden vuoksi pitämään mahdollisimman lähellä testiskenaarioiden rakennetta. Tämä mahdollistaa myös konfiguraatioiden laajentamisen helposti tarpeen vaatiessa. Konfiguraatioon luotiin mahdollisuus määritellä joukko käyttäjiä, asiakasjärjestelmiä ja toimipisteitä. Rakenne on esitelty ohjelmassa 4.4.

```
<global>
  <configuration_list>
    <configuration>
      <assign>
        <server_address>https://localhost:44300/</server_address>
        <users>
          <user type="nurse_with_prescription_rights">
            <first_name>Sanni</first_name>
            <last_name>Sairaanhoitaja Tes</last_name>
            <registration_number>12345678901</registration_number>
          </user>
          ...
        </users>
        <organizations>
          <organization type="private_healthcare">
            1.2.246.10.15719974.10.1
          </organization>
          ...
          <organization type="invalid">
            1.2.345.67.99999999.89.0
          </organization>
        </organizations>
        <system_clients>
          <system_client type="single_user">
            00000000-0000-0000-0000-A00000000011
          </system_client>
          <system_client type="organization">
            00000000-0000-0000-0000-B00000000011
          </system_client>
          <system_client type="invalid">
            ABCDEFGH-1234-5678-4321-ABCDEFGHIJKL
          </system_client>
        </system_clients>
      </assign>
    </configuration>
  </configuration_list>
</global>
```

Ohjelma 4.4 *Testaustyökalun konfiguraatiorakenne, joka sisältää testattavan palvelimen osoitteen ja listan testauksessa käytettävistä käyttäjistä, asiakasjärjestelmistä ja toimipisteistä.*

Testisekvenssien alussa on mahdollista määritellä, millä kaikilla edellä mainittujen hierarkiajoukkojen alkioilla kyseinen testisekvenssi ajetaan. Testaustyökalu voi esimerkiksi automaattisesti ajaa saman testisekvenssin sekä lääkärillä että sairaanhoitajalla julkisen ja yksityisen terveydenhuollon järjestelmiä käyttäen. Testaustyökalu

muodostaa kaikki mahdolliset kombinaatiot listatuista alkioista, kun testisekvenssi ladataan työkaluun. Testaustyökalussa näitä järjestelmähierarkia-alkioiden kombinaatioita kutsutaan järjestelyiksi (engl. arrangement).

4.3.3 Validointirakenteet

Työssä toteutetussa työkalussa testitapausten tulokset validoidaan automaattisesti testiskenaarioiden XML-rakenteessa olevien validointiehtojen perusteella. Validointiehtoja voi olla useita ja jokaiselle ehdolle on määriteltävä tyyppi. Validointiehtojen tyypit ovat samat kuin testiohjelman runkoon toteutetut vertailuoperaatiot, joita ovat Equals, Contains ja NotContains. Testattavan järjestelmän paluuviestistä voidaan validoida minkä tahansa kentän arvoja edellä mainituilla ehdoilla. Ohjelmassa 4.5 validoidaan kohdejärjestelmän palauttama tilakoodi ja osa vastausviestin XML-rakenteesta löytyvästä tietosisällöstä.

Validoinneissa on mahdollista hyödyntää kohdassa 4.3.1 mainittuja arvoviittauksia. Paluuviestin arvoa voi esimerkiksi verrata testaustyökalun muistissa olevaan arvoon dollariviittausta käyttämällä (esimerkiksi "$Entry.OID"). Oraakkeli korvaa automaattisesti viittausmerkillä alkavat merkkijonot niitä vastaavilla arvoilla.

```
...
<!-- EI LÄHETTÄMÄTTÖMIÄ LÄÄKEMÄÄRÄYKSIÄ -->
<function_call name="UnsentPrescriptionCount">
  <expected_result_list>
    <expected_result>
      <status>200</status>
    </expected_result>
    <expected_result>
      <validations>
        <validation>
          <type>Equals</type>
          <data>
            <response>
              <unsent_prescription_count>
                <prescription_count>0</prescription_count>
              </unsent_prescription_count>
            </response>
          </data>
        </validation>
      </validations>
    </expected_result>
  </expected_result_list>
</function_call>
...
```

Ohjelma 4.5 *Esimerkki testiskenaarion osana olevasta validointirakenteesta. Esimerkissä tarkistetaan eRA-järjestelmältä lähettämättömien lääkemääräysten määrä, jonka odotusarvo on 0. HTTP-tilakoodin odotusarvo on 200 (HTTP OK).*

Rakenteiden validoinnin lisäksi toteutettiin tuki eRA-järjestelmän palauttamien tilakoodien validoinnille. eRA-järjestelmä lähettää paluuviestissä kaksi tilakoodia, joista toinen on HTTP-standardin mukainen tilakoodi ja toinen eRA-järjestelmän sisäisesti määrittelemä tilakoodi, joka ilmaisee järjestelmässä ilmentyneen virheen tyypin.

4.4 Menetelmä 2: Testitapausten mallipohjainen generointi

Kun testausta laajennettiin tukemaan rajapintaversiota 2.0, huomattiin pian, että testauksen laajus alkoi kasvaa eksponentiaalisesti. Uuden version tuomien uusien ominaisuuksien osalta rajapinta oli huomattavasti laajempi ja monimutkaisempi. Tässä vaiheessa ilmeni, että kenttien välille muodostuvat ehdolliset riippuvuussuhteet laajoine arvoalueineen olivat todellinen rasite testaukselle. Rajapintaversiolle 1.1 toteutetun testaustyökalun arkkitehtuuri mahdollisti kuitenkin työkalun laajentamisen 2.0-rajapintaversiolle sopivammaksi mallipohjaisuutta hyödyntäen.

Koska testiskenaariot eivät yleensä sisällä pitkiä komentosarjoja, on niiden kirjoittaminen nopeaa myös manuaalisesti. Vaivalloiseksi osoittautuikin kaikkien avain-arvo-kombinaatioiden kirjoittaminen testiskenaarioiden yksittäisiä testitapauksia eli komentoja kohden. Esimerkiksi potilaan arkistomerkintöjen luontiin tarkoitettu komento sisälsi kirjoitushetkellä yhteensä 126 mahdollista kenttää, joille jokaiselle oli määriteltävä useita testattavia arvoja. Jos kaikkien kenttien kaikki testattavaksi määritellyt arvot kirjoitettaisiin auki testiskenaarioon, tulisi skenaarioon yli tuhat testitapausta kyseiselle komennolle. Työmäärä on liian suuri tehtäväksi käsin, joten työssä päädyttiin suunnittelemaan sopivampi rakenne, jolla voidaan pienellä työmäärällä kuvata malli, jonka pohjalta testaustyökalu generoi komennolle ajettavat testitapaukset ja myös ajaa ne automaattisesti testattavaan järjestelmään. Menetelmä muistuttaa paljolti tietovetoista testausmenetelmää, jota on mahdollista käyttää esimerkiksi kohdassa 2.3.4 esitellyssä Robot Frameworkissa.

```
...
2 <field name="microbe_cultivation_result">
    <field name="code">
4     <field name="type">
        <value type="valid">microbe_nomenclature</value>
6       <value type="invalid">icd10</value>
        <value type="invalid"></value>
8     </field>
      <field name="code">
10      <!-- Mycobacterium mucogenicum -->
        <value type="valid">1000-61</value>
12      <!-- Clostridium leptum -->
        <value type="valid">1001-49</value>
14      <value type="invalid">XXXXX</value>
        <value type="invalid"></value>
16    </field>
    </field>
18 </field>
...
```

Ohjelma 4.6 *Esimerkki mikrobiviljelyn tyyppikoodin testauksesta mallipohjaisella rakenteella. Mikrobiviljely voi olla osa laboratoriotutkimusta, joka puolestaan on yksi monista arkistokirjaustyypeistä.*

Mallipohjaisuutta hyödyntävä rakenne mahdollistaa testiskenaarioiden kirjoittamisen manuaalisesti, mutta samalla mallipohjaisuuden hyödyntämisen yksittäisiä komentoja kohden. Yksi mallipohjainen rakenne riittää kuvaamaan mallin yksittäiselle komennolle, jonka pohjalta voidaan testata komentoa toistuvasti kaikilla määritel-

lyillä arvoilla. Ohjelmassa 4.6 on osa mallipohjaista testiskenaariota, jossa testataan mikrobiviljelyn tulosten kirjaamista. Rakenne sisältää listauksen komennolle mahdollisista virhekoodeista ja ehdot jokaiselle virhekoodille, joiden perusteella testioraakkeli validoi vastaanotettujen virhekoodien oikeellisuuden.

```
1 <test>
    <description>Testiskenaarion sanallinen kuvaus</description>
3   <statement_list>
      <function_call name="Login" />
5     <function_call name="OpenPatient" />
      <model_based_test name="CreateEntry">
7       <!-- VIRHEKOODIT -->
        <error_codes>
9         <error_code value="114">
            <when field="entry.service_event_oid" state="not_missing" />
11        </error_code>
          ...
13      </error_codes>
        <!-- ARVOT -->
15      <values>
          <field name="entry">
17          <field name="service_event_oid">
              <value type="invalid">TESTI</value>
19          </field>
            ...
21        </field>
        </values>
23    </model_based_test>
      <function_call name="ClosePatient" />
25    <function_call name="Logout" />
    </statement_list>
27 </test>
```

Ohjelma 4.7 *Esimerkki mallipohjaisuuden hyödyntämisestä manuaalisesti kirjoitetussa testiskenaariossa. Testiskenaariossa kirjaudutaan sisään, avataan potilas, luodaan potilaalle mallipohjaisesti generoituja merkintöjä, suljetaan potilas ja kirjaudutaan ulos. Näkyvissä olevaa rakennetta on typistetty.*

Ohjelmassa 4.7 esitelty rakenne ilmaisee, että eRA-järjestelmän pitäisi palauttaa virhekoodi 114, kun merkintään liittyvän palvelutapahtuman tunniste ei ole tyhjä. Kyseiselle kentälle testattavaksi arvoksi on listattu ainoastaan merkkijono "TESTI", joka on merkitty virheelliseksi. Kun komentoa testataan virheellisellä syötteellä, testioraakkeli tietää odottaa lopputuloksena virhettä. Kun kaikki syötteet ovat oikeita,

testioraakkeli hyväksyy ainoastaan ne testitapaukset, joissa testattava järjestelmä ei palauta lainkaan virhettä.

eRA-järjestelmän käyttöoikeuksien hallinta tekee kuitenkin poikkeuksen validointisääntöihin. Jos käyttäjällä ei ole oikeutta käyttää kyseistä komentoa, on oikein, että eRA-järjestelmä palauttaa sitä vastaavan virheen. Virhe voi tulla myös silloin, kun kaikki komentoon liittyvät syötteet ovat itsessään virheettömiä. Testioraakkelin tulee siksi käyttöoikeusvirheen saadessaan tietää onko nykyisellä käyttäjällä oikeus käyttää kyseistä komentoa. Komentokohtaiset oikeudet on kirjattu testaustyökalun osana olevaan malliin, josta löytyvät myös kaikkien kohdejärjestelmän rajapintakomentojen URL-osoitteet.

4.5 Testauksen kattavuuden mittaaminen

On tärkeää, että testaustyökalu kykenee jollain tapaa ilmaisemaan testaajalle nykyisten testiskenaarioiden kattavuuden. Testauksen kattavuutta on kuitenkin vaikeaa mitata ilman koko järjestelmää kattavaa mallia. Kun kohdejärjestelmän lähdekoodi ei ole tiedossa, ei myöskään ole mahdollista mitata koodikattavuutta.

Tässä työssä testaustyökaluun toteutettiin näkymä, josta on nähtävissä kuinka monesti kutakin rajapintakomentoa kutsutaan toteutetuissa testiskenaarioissa. Testaustyökalun mallissa on tieto kaikista rajapintakomennoista. Kattavuusnäkymässä korostetaan niitä rajapintakomentoja, joille ei ole olemassa lainkaan testitapauksia. Näin havaitaan puutteet testauksessa komentotasolla, mutta tietorakenteiden kenttien tasolla kattavuus ei ole tiedossa. Tällä menetelmällä ei voida tietää, millä arvojoukoilla komentoja on testattu.

Jotta testien kattavuus saataisiin tietoon yksittäisten kenttien tasolla, tulisi kaikkien kenttien mahdolliset arvoalueet määritellä testaustyökaluun liitetyssä mallissa. Tässä on kuitenkin mahdollista oikaista määrittämällä esimerkiksi tietyt arvot, joita testataan kaikille tietyn tyyppisille kentille. Kaikkien kenttien kohdalla testattavia arvoja voisivat olla esimerkiksi (tyhjä merkkijono), null (ei lainkaan arvoa) ja "TESTI". Vaikka kenttien arvoalueet eivät olisi tiedossa, voidaan silti laskea kenttiin asetettujen uniikkien arvojen määrä. Määrä ei yksin kerro prosentuaalista kattavuutta, mutta voi osoittaa puutteellisesti testattuja kenttiä testaajalle.

Jos toteutettuna olisi koko kohdejärjestelmän kattava tilakone, voitaisiin testauksen kattavuutta mitata testiajon aikana toteutuneiden tilasiirtymien määrää laskemalla. Tilakone voidaan hahmottaa esimerkiksi suunnattuna graafina, jossa jokainen solmu vastaa rajapintakomentoa ja jokainen kaari kohdejärjestelmän palauttamaa virhe-

koodia tai tilaa johon päädyttiin. Täydellinen kattavuus saavutettaisiin näin ollen suorittamalla testejä siten, että graafin kaikkia kaaria pitkin kuljetaan vähintään kerran.

4.6 Regressiotestaus

Regressiotestauksella tarkoitetaan testausta, jolla pyritään varmistamaan, etteivät uudet muutokset ole sivuvaikutuksillaan rikkoneet jotakin aiempaa järjestelmään luotua toiminnallisuutta [24]. Regressiotilanteiden havaitsemiseksi voidaan hyödyntää aiempien testiajojen lokitietoja. Jos havaitaan, että aiemmin onnistuneesti ajettu testiskenaario epäonnistuu uudella järjestelmäversiolla, on tapahtunut regressio.

Työssä toteutettuun testaustyökaluun toteutettiin historianäkymä, josta testaaja voi tarkastella testiskenaarioiden suoritushistorioita päivämäärineen. Näkymästä on nähtävissä jokaiseen testiskenaarioon liittyvä käyttäjärooli, toimipiste, asiakas ja asiakasjärjestelmä. Lisäksi jokaiseen epäonnistuneeseen ajoon liittyy tieto testin numerosta, jossa ajo epäonnistui, sekä eRA-järjestelmän mahdollisesti palauttama virhekoodi.

Jos testiskenaariota muokataan, ei suoritushistoria päivity uuden testiskenaarion mukaiseksi. Tästä syystä jokaisesta testiskenaariosta tallennetaan ajon jälkeen tiiviste, jolla voidaan tarkastaa myöhemmin, ajetaanko samaa testiskenaarion versiota. Jos testiskenaariota on muokattu testiajojen välissä, ei voida olla varmoja onko myöhemmän ja aiemman testiajon yhteydessä esiintyvän vian syy sama. Historiatietoja ei kuitenkaan poisteta testiskenaarion muokkauksen yhteydessä, sillä historiatiedoista voi silti olla apua uuden järjestelmässä esiintyvän vian syyn selvittämisessä.

4.7 Testauskontekstin vaihtaminen

eRA-järjestelmässä useat samanaikaiset käyttäjät voivat toiminnallaan aiheuttaa erilaisia konfliktitilanteita. Yksi esimerkki on tilanne, jossa kaksi käyttäjää yrittää samanaikaisesti muokata samaa potilaalle annettua merkintää. Myöhempää muokkausta yrittävän käyttäjän toiminta tulee estää, jotta ei pääse syntymään tilannetta, jossa toisen käyttäjän tekemät muutokset korvataan toisen käyttäjän tekemillä muutoksilla alkuperäisiä muutoksia kuitenkaan näkemättä.

```
1 <test>
    <description>Testauskontekstin vaihtaminen</description>
3   <arrangements>
      <users>
5       <user type="doctor" />
      </users>
7     <organizations>
        <organization type="private_healthcare" />
9     </organizations>
      <system_clients>
11      <system_client type="organization" />
      </system_clients>
13  </arrangements>
    <statement_list>
15    <!-- KIRJATAAN LÄÄKÄRI SISÄÄN -->
      <subsequence id="login" />
17    <!-- VAIHDETAAN KONTEKSTIA -->
      <context name="nurse" />
19    <!-- LADATAAN SAIRAANHOITAJAN TIEDOT KONFIGURAATIOSTA -->
      <configuration>
21      <assign path="User" value="$Users.Nurse" />
      </configuration>
23    <configuration>
        <assign path="User.OrganizationUnit.Id"
25              value="$Organizations.PrivateHealthcare.OID" />
      </configuration>
27    <configuration>
        <assign path="SystemID"
29              value="$SystemClients.Organization.SystemID" />
      </configuration>
31    <!-- KIRJATAAN SAIRAANHOITAJA SISÄÄN JA ULOS -->
      <subsequence id="login" />
33    <subsequence id="logout" />
      <!-- VAIHDETAAN TAKAISIN LÄÄKÄRIN KONTEKSTIIN -->
35    <context name="default" />
      <!-- KIRJATAAN LÄÄKÄRI ULOS -->
37    <subsequence id="logout" />
    </statement_list>
39 </test>
```

Ohjelma 4.8 *Testauskontekstin vaihtaminen.*

Testaustyökaluun toteutettiin tuki vaihtaa testauskontekstia testiskenaarion aikana. Testauskonteksteille annetaan nimi, jolla niihin voidaan viitata testiskenaarioissa. Testauskontekstin vaihtaminen vaihtaa muistioliota, johon kaikki istuntoon liittyvät tiedot on mahdollista taltioida. Näin voidaan ylläpitää samanaikaisesti useampaa

kuin yhtä istuntoa. Ohjelmassa 4.8 on esimerkki testiskenaariosta, jossa kirjataan sisään ja ulos kaksi eri käyttäjää siten, että molempien istuntoja hallitaan samanaikaisesti. Testauskontekstin vaihtaminen tapahtuu context-elementtiä käyttäen, jolle syötetään attribuutiksi kontekstin nimi. Jos nimeä vastaavaa kontekstia ei ole testaustyökalun muistissa, luodaan uusi konteksti, joka perii käytössä olevan kontekstin konfiguraatiotiedot.

4.8 Jatkokehitysmahdollisuudet

Testiskenaarioiden epädeterministisyys oli suhteellisen helposti ratkaistavissa eRA-järjestelmän osalta, mutta muiden järjestelmien kohdalla tilanne ei välttämättä ole sama. Lisäksi epädeterministisyyttä on mahdoton havaita suorittamatta testejä kohdejärjestelmään useammin kuin kerran. Testaustyökaluun olisi mahdollista toteuttaa tuki epädeterminististen testitapausten havaitsemiselle tallentamalla aiempien testiajojen tulokset ja vertaamalla uusia tuloksia vanhoihin. Aiempien testiajojen tuloksia tarkastelemalla voitaisiin havaita testiskenaarioiden välisiä riippuvuuksia ja muita ongelmakohtia, jotka voivat rikkoa testausautomaation täysin. Testi voidaan todeta epädeterministiseksi, jos sen tulos poikkeaa aiemman suorituskerran tuloksesta, vaikka kohdejärjestelmän versio ja alkutila olisivatkin samoja.

Testaustyökalun mallipohjaisuus rajoittuu tällä hetkellä yksittäisten testitapausten mallipohjaiseen generointiin tietovetoisesti. Kohdejärjestelmän mallin voisi laajentaa koko järjestelmän laajuiseksi, jolloin voitaisiin generoida kokonaisia testiskenaarioita täysin automaattisesti. Tuolloin vältyttäisiin kokonaan testiskenaarioiden ylläpidolta, mutta toisaalta kohdejärjestelmän mallin ylläpito tulisi välttämättömäksi.

Työssä toteutetulle testaustyökalulle tulee nyt määritellä manuaalisesti joukko arvoja, joilla kyseistä XML-rakenteen kenttää testataan. Jos testausmalliin määriteltäisiin yksiselitteisesti virhetilojen yhteyteen ehdot, joilla virhetilan pitäisi ilmentyä, voitaisiin mallin perusteella generoida myös arvoalueeseen soveltuvia arvoja, joilla järjestelmän toimintaa testataan. Tämä tietysti vaatisi erilaisten arvoalueluokitusten määrittelemisen. Arvoalueluokituksia voisivat olla esimerkiksi aikaleimat ja erilaiset rajapintaan syötettävät koodistoarvot.

Mikäli koko kohdejärjestelmän kattava malli toteutettaisiin yksinkertaisena tilakoneena, se voitaisiin esittää testaajalle visuaalisesti graafina, jossa solmut olisivat rajapintakomentoja ja kaaret tilasiirtymiä komennosta toiseen. Tilasiirtymien rajoitteet tulisivat suoraan kohdejärjestelmän mallista ja olisivat käytännössä kohdejärjestelmän palauttamia virhekoodeja tai muita mahdollisia tilasiirtymiin vaikuttavia paluuarvoja.

5. YHTEENVETO

Tutkielman tuloksena tuotettiin testauksen automatisoiva ohjelmisto, joka kykenee lukemaan kahdella eri tavalla määriteltyjä testiskenaarioita. Ensimmäinen tapa on kirjoittaa testitapaukset testiskenaarioihin manuaalisesti, toinen tapa mahdollistaa testitapausten generoimisen automaattisesti ennaltamäärätyn mallin pohjalta. Tässä työssä ei toteutettu tukea koko kohdejärjestelmän laajuiselle mallille, mutta toteutettu työkalu on laajennettavissa tukemaan myös täysin mallipohjaista testausta.

Parhaassa tapauksessa mallipohjaisuus poistaa täysin testiskenaarioiden suunnitteluun kuluvan ajan, mutta vaatii työtä tilakoneen suunnittelussa ja toteuttamisessa. Oikein toteutettuna mallipohjaisuus toisaalta myös varmistaa, että kaikki dokumentoidut virhekoodit tulee käsiteltyä. Testauskehyksen generoidessa testiskenaarioita mallipohjaisesti ei pitäisi olla mahdollista, että järjestelmä päätyy määrittelemättömään tilaan. Jos niin käy, on virhe joko testattavassa sovelluksessa, mallissa tai testauskehyksessä itsessään. Mallissa esiintyvien virheiden paikantaminen on usein huomattavasti vaikeampaa kuin manuaalisesti kirjoitetuissa testiskenaariossa olevien virheiden paikantaminen.

Kehitysvaiheessa testioraakkelin toteutuksessa havaittiin virheitä. Oraakkeli saattoi ilmoittaa testaajalle, että järjestelmän antama palaute oli oikein, vaikka todellisuudessa palaute olikin virheellinen. Samankaltaisia virheellisiä johtopäätöksiä voi syntyä myös ihmisen tekemän testauksen tuloksena. Kun johtopäätöksiä on paljon, on virheellisiä johtopäätöksiä hankala havaita ja niiden syytä voi olla vaikea paikantaa. Testioraakkelin tuottamat virheelliset johtopäätökset onnistuttiin kuitenkin löytämään testausautomaatiota kehitettäessä ja myös korjattua testioraakkelin toteutukseen. Samalla toteutettiin yksikkötestejä, jotka testaavat testioraakkelin suorittamien validointien toimintaa.

Työmäärää olisi saattanut olla mahdollista vähentää hyödyntämällä valmiita web-rajapintatestaukseen toteutettuja ohjelmistoja oman ohjelmiston toteuttamisen sijaan. Valmiiden ohjelmistojen perehtymiseenkin kuluu kuitenkin aikaa ja käyttämällä kolmannen osapuolen tarjoamia komponentteja tullaan niistä myös riippuvaisiksi. Nyt lopputuloksena syntynyt ohjelmisto vastaa täysin yrityksen tarpeita ja sitä on mahdollista laajentaa tarpeen vaatiessa.

Taulukko 5.1 *Testaukseen käytetty aika suhteessa rajapintakutsujen määrään.*

Tyyppi	Kutsumäärä	Ajankäyttö (h)
Manuaalinen testaus	9 000	300
Manuaalinen testaus	12 000	400
Testausautomaatio	12 000	250
Testausautomaatio	15 000	280

Tässä työssä toteutettu testausautomaatiotyökalu koostuu noin 4200 rivistä ohjelmakoodia, joihin sisältyy eRA-järjestelmän rajapintatietomallien C#-toteutukset. Työkalun lisäksi on toteutettu XML-muotoisia testisekvenssejä noin 6000 riviä. Testausautomaatioon toteutettujen testisekvenssien tarkka rajapintakutsujen määrä ei ole tiedossa, mutta tiedetään, että se on lähellä 12 000 kutsua. Työkalun ja testisekvenssien toteutukseen on käytetty aikaa noin 250 työtuntia. Lisäksi testaukseen ja havaittujen vikojen kirjaamiseen on käytetty aikaa noin 20 tuntia. Tuntikirjausten perusteella voidaan arvioida, että 3 000 uuden rajapintakutsun toteuttaminen vie aikaa noin 30 tuntia. Ajankäyttö ei ole lineaarinen suhteessa rajapintakutsujen määrään, sillä testaustyökalu on suunniteltu siten, että testiskenaarioiden osia voidaan käyttää uudelleen uusissa testiskenaarioissa.

Jos XML-viestit kirjoitettaisiin ja validoitaisiin manuaalisesti, ja sovittaisiin, että yhden viestin osalta testaamiseen kuluisi aikaa keskimäärin kaksi minuuttia, testaaja saisi lähetettyä ja validoitua tunnissa 30 viestiä. Jos testaaja käyttäisi testaamiseen aikaa 300 tuntia, tulisi rajapintaa testattua noin 9 000 erilaisella syötteellä. Ajankäyttö kasvaa lineaarisesti suhteessa rajapintakutsujen määrään. Arvion perusteella voidaan siis olla varmoja siitä, että testausautomaation toteuttaminen olisi ollut eRA-järjestelmän osalta kannattavaa jo tapauksessa, jossa testit ajettaisiin kohdejärjestelmään ainoastaan kerran. Ajankäyttöä on havainnollistettu taulukossa 5.1.

eRA-järjestelmän 1.1-rajapintaa testattaessa löydettiin yksi virhe, joka oli päässyt tuotannossa olevaan järjestelmään aiemmin tehdystä manuaalisesta testauksesta huolimatta. 2.0-rajapinnassa virheitä havaittiin useampi kymmen, eikä kaikkia virheitä ole vielä työn kirjoitushetkellä mahdollista löytää, sillä testeissä jo havaitut virheet saattavat peittää osan virheistä. Testausautomaatiota tullaan kehittämään vielä tämän työn päätyttyä.

Testausautomaation toteuttaminen on työläs ja aikaa vievä prosessi, eikä se välttämättä maksa itseään takaisin lyhyellä aikavälillä. Kuitenkin tilanteissa, joissa testejä tullaan ajamaan useita, jopa kymmeniä kertoja uudestaan, osoittautuu testauksen automatisointi välttämättömäksi. Mitä aikaisemmassa vaiheessa automaatio toteu-

tetaan, sitä enemmän työmäärällistä säästöä saadaan aikaan.

Testauskehyksen tai -työkalun modulaarinen rakenne mahdollistaa sen hyödyntämisen useiden eri järjestelmien ja rajapintojen testaukseen. Modulaarisuuden säilyttäminen aiheuttaa lisää työtä testausautomaation suunnittelu- ja toteutusvaiheessa, jos käytössä ei ole valmista testauskehystä. Toistaalta testauskehykseen perehtyminen vie aikaa eikä kehys välttämättä sovellu sellaisenaan kaikkiin tarpeisiin.

Mallipohjaisuuden hyödyntäminen testausautomaatiossa paransi testien ylläpidettävyyttä, sillä sen avulla testisekvensseihin ei jouduttu kirjoittamaan manuaalisesti toisteista tietoa. Toinen tapa vähentää toisteisuutta oli luoda tuki alisekvenssirakenteille, joita on mahdollista käyttää uudelleen testiskenaarioissa. Kun testaustyökalu generoi testitapauksia automaattisesti, saatetaan tulla testanneeksi asioita, joita testaaja ei alunperin edes ajatellut. Toisaalta osa tärkeistä rajatapauksista voi jäädä täysin testaamatta, mikäli testien generointiin ei anneta lainkaan ohjeita. Mallipohjaisuus vähentää testausautomaation toteuttamiseen vaadittua työmäärää erityisesti tapauksissa, joissa halutaan testata useita eri kenttiä laajoilla arvoalueilla. Tässä työssä testausmallien avulla generoitiin tuhansia testitapauksia, jotka oltaisiin muussa tapauksessa jouduttu kirjoittamaan manuaalisesti.

Validointirakenteissa joutuu toisinaan hyödyntämään testattavan järjestelmän aiemmin palauttamia arvoja, joihin pitää pystyä viittaamaan myöhemmässä vaiheessa testejä. Palautteena tulleet arvot täytyy voida tallentaa testaustyökalun muistiin ja arvoihin viittaamiseksi tulee toteuttaa oma syntaksinsa. Työssä havaittiin tarpeelliseksi voida viitata myös muihin vasta ajonaikaisesti selvillä oleviin arvoihin, kuten aikaleimoihin. Niiden generoimiseksi toteutettiin oma syntaksinsa, jota voitiin hyödyntää testiskenaarioissa.

LÄHTEET

[1] Kansallinen Terveysarkisto (Kanta). Kanta.fi. [WWW], [Viitattu 13.4.2017], Saatavissa: http://www.kanta.fi/.

[2] Guru99. Static Testing Vs Dynamic Testing. [WWW], [Viitattu 13.4.2017], Saatavissa: http://www.guru99.com/static-dynamic-testing.html.

[3] Ron Patton. *Software Testing*. Sams, 2001.

[4] Guru99. Positive Vs Negative testing. [WWW], [Viitattu 20.5.2017], Saatavissa: http://www.guru99.com/positive-vs-negative-testing.html.

[5] M. Utting, B. Legeard. *Practical Model-Based Testing: A Tools Approach*. Morgan Kaufmann, 2010.

[6] Martin Fowler. Eradicating non-determinism in tests. [WWW], [Viitattu 27.3.2017], Saatavissa: https://martinfowler.com/articles/nonDeterminism.html, 2011.

[7] J. Takahashi, Y. Kakuda. Extended-model based testing by directed chinese postman algorithm. *Proc. IEEE*, 2002.

[8] graphwalker.org. GraphWalker. [WWW], [Viitattu 20.2.2017], Saatavissa: http://graphwalker.github.io/.

[9] G. Chaste, A. Ooms, R. Walravens. Chinese postman problem. 2014.

[10] yWorks. yEd. [WWW], [Viitattu 20.2.2017], Saatavissa: https://www.yworks.com/products/yed.

[11] Optimizory Technologies Pvt. Ltd. vREST. [WWW], [Viitattu 20.2.2017], Saatavissa: https://vrest.io/.

[12] NModel. NModel - Home. [WWW], [Viitattu 20.2.2017], Saatavissa: https://nmodel.codeplex.com/.

[13] Robot Framework Foundation. Robot Framework. [WWW], [Viitattu 11.3.2017], Saatavissa: http://robotframework.org/.

[14] Atostek Oy. Atostek eRA. [WWW], [Viitattu 27.3.2017], Saatavissa: http://era.atostek.com/.

[15] HL7 Finland ry. Rajapintakartta - hl7 v3 -sanomarajapinnat. [WWW], [Viitattu 20.6.2017], Saatavissa: http://www.hl7.fi/rpk-hl7-v3-sanomarajapinnat/.

[16] Atostek Oy. eRA Integraatio-ohje. Saatavuus: Atostek Oy.

[17] Valvira. Terveydenhuollon rooli- ja attribuuttitietopalvelu. [WWW], [Viitattu 13.4.2017], Saatavissa: http://www.valvira.fi/terveydenhuolto/rooli-_ja_attribuuttitietopalvelu.

[18] Kansallinen Terveysarkisto (Kanta). Potilastiedon arkiston määrittelyt. [WWW], [Viitattu 13.4.2017], Saatavissa: http://www.kanta.fi/web/ammattilaisille/potilastiedon-arkiston-maarittelyt.

[19] Kela. Kansallinen koodistopalvelu. [WWW], [Viitattu 13.4.2017], Saatavissa: http://91.202.112.142/codeserver/pages/classification-list-page.xhtml.

[20] Microsoft. Windows Presentation Foundation. [WWW], [Viitattu 15.2.2017], Saatavissa: https://msdn.microsoft.com/en-us/library/ms754130(v=vs.110).aspx.

[21] Microsoft. XmlSerializer Class. [WWW], [Viitattu 18.2.2017], Saatavissa: https://msdn.microsoft.com/en-us/library/system.xml.serialization.-xmlserializer(v=vs.110).aspx.

[22] Microsoft. Reflection in the .NET Framework. [WWW], [Viitattu 15.2.2017], Saatavissa: https://msdn.microsoft.com/en-us/library/f7ykdhsy(v=vs.110).aspx.

[23] Väestörekisterikeskus. Terveydenhuollon varmennepalvelut - Eevertti. [WWW], [Viitattu 13.4.2017], Saatavissa: https://eevertti.vrk.fi/terveydenhuollolle.

[24] Microsoft. Regression Testing. [WWW], [Viitattu 15.3.2017], Saatavissa: https://msdn.microsoft.com/en-us/library/aa292167(v=vs.71).aspx.